DADS: Dynamic Slicing Continuously-Running Distributed Programs with Budget Constraints

Xiaoqin Fu
Washington State University, Pullman, USA
xiaoqin.fu@wsu.edu

Haipeng Cai
Washington State University, Pullman, USA
haipeng.cai@wsu.edu

ABSTRACT
Dynamic slicing is an important technique underlying much tool support for software quality assurance. Yet most existing dynamic slicers face applicability, scalability, and cost-effectiveness challenges when applied to common distributed software with continuous executions. To overcome these challenges, we present DADS, a distributed, online, scalable, and cost-effective dynamic slicer for continuously running distributed programs with respect to user-specified budget constraints. DADS is distributed by design to exploit distributed and parallel computing resources. With an online analysis, it avoids tracing hence the associated I/O time and space costs. Most importantly, DADS achieves and maintains practical scalability and cost-effectiveness according to the budget on analysis costs by continually and automatically adjusting the configuration of its analysis algorithm on the fly via reinforcement learning. We evaluated DADS against eight real-world Java distributed systems and empirically demonstrated the scalability and cost-effectiveness merits of DADS. The open-source tool package for DADS can be found here, and the demo video can be downloaded here or viewed here online.

KEYWORDS
Distributed system, dynamic slicing, configuration adjustment, reinforcement learning, scalability, cost-effectiveness

1 INTRODUCTION
Program slicing can be defined and computed on the notion of selective dependencies [13]. Moreover, a dynamic program slicer computes slices from the execution of a program [1]; the resulting slice narrows down the search space of dependencies of interest (e.g., those indicating faults). Despite these merits, applying conventional dynamic slicing to distributed systems faces major barriers. First, existing dynamic slicers require run-time tracing before computing slices offline from the traces; yet distributed systems often run continuously to provide uninterrupted services, thus the execution traces are unbounded. Second, distributed systems are commonly large-scale and complex, posing tremendous scalability and cost-effectiveness challenges to dynamic slicers against these systems.

Most dynamic slicers, working for sequential [2, 12, 15, 20] or concurrent [16, 17] programs, focus on single-process software. These slicers do not consider interprocess dependencies, thus they do not apply to distributed programs which run in multiple processes. Mohapatra et al. proposed a dynamic slicing algorithm and implemented it for distributed object-oriented programs [14]. However, the approach suffers scalability problems with large-scale distributed systems, as suggested by both the heavyweight nature of its analysis algorithms and by its original efficiency results even against simple programs (894 or fewer lines of code). In short, there is no existing dynamic slicer working with industry-scale continuously running distributed systems.

Also, existing slicers do not consider practical constraints in terms of the time budget users may be subject to. Given the limited total time allocated for a task (e.g., regression testing), users may only afford a certain amount of time for a particular step of the task [11] (e.g., using a slicer to reduce the search space of code entities that need to be regression tested). Thus, a practical dynamic slicer should respect the maximal (response) time a user can afford, offering useful results within the budget available.

To overcome these challenges, we developed DADS, a distributed, online, continuous, scalable, and cost-effective dynamic slicer for continuously running distributed programs, with respect to user-specified budget constraints on how much time can be spent. DADS itself is designed as a distributed system, with a number of analysis components each running within one of the processes of the system under analysis (SUA), so as to leverage the distributed computing resources available to the SUA. Moreover, DADS adopts an online dynamic analysis at its core—every execution event is used only once after it occurs, and is then dropped, and it answers user queries on demand. Thus, DADS avoids tracing and, accordingly, the storage and disk I/O costs incurred by offline analyzers. Most of all, DADS automatically and continually adjusts its analysis configuration on the fly to balance its analysis cost and effectiveness and to overcome potential scalability issues.

We developed DADS for Java and applied it against eight real-world Java distributed systems with continuous executions. DADS successfully worked with all these systems with different architectures, application domains, and scales. Our results revealed noticeable scalability and cost-effectiveness (65–140% higher) advantages of DADS over a similar slicer without the ability to adjust the algorithm for better cost-effectiveness balances.

DADS serves varied audiences. Distributed system developers may query dynamic slices of program points of interest to identify faults during testing and debugging, among other maintenance tasks. Tool developers may leverage DADS to build practical tools for security, performance diagnosis, and other applications. Researchers may use DADS to develop advanced client/application analyses underlying those practical tools.

To the best of our knowledge, DADS is the first online dynamic slicer particularly working with real-world, continuously-running Java distributed software with respect to user-specified budget constraints. It is also the first dynamic slicer in general that automatically adjusts its algorithm to maintain practical scalability and cost-effectiveness tradeoffs via reinforcement learning.
2 ARCHITECTURE

Figure 1 depicts the architecture of DADS. It takes three inputs from the user: a distributed system \(D\) under analysis, a slicing query \(Q\), and a user budget \(B\). This budget is a response time constraint for the analysis. With these inputs, DADS continually computes dynamic dependencies and answers slice queries on demand while adjusting its configuration when necessary, in three phases as follows:

In the first phase (**instrumentation**), DADS creates an instrumented version \(D'\) of \(D\), by inserting probes to monitor executed statements and the entry (i.e., program control entering a method) and returned-into (i.e., program control returning from a callee into a caller) events of each executed method. In the second phase (**arbitration & adjustment**), DADS continues to run along with \(D'\), and continually arbitrates for intraprocess slice (dependence) computations and adjusts its configuration to balance the cost and effectiveness—i.e., the load and the process cost—of the intraprocess analyses by DADS, each in one of the \(D'\) processes, run in parallel hence the distributed working of DADS. When every intraprocess slice of \(Q\) is completed, it is delivered to the querying interface of the corresponding process. In the third phase (**user interaction**), through these querying interfaces via the computer network, all relevant intraprocess slices are gathered and used to compute interprocess dependencies. These intraprocess and interprocess dependencies constitute the final dynamic slice of \(Q\) as DADS’s output presented to the user.

The key novelty of DADS lies in its ability to maintain practical cost-effectiveness tradeoffs via automatic adjustment of its (hybrid) analysis algorithm’s configuration, consisting of six parameters. Three parameters control the static analysis part of the slicing algorithm: staticGraph indicating if the static analysis is performed (to construct the static dependence graph for each component of \(D\)), while contextSensitivity and flowSensitivity indicating if the static analysis is context- and flow-sensitive, respectively. The other three parameters control the dynamic analysis part: methodEvent and statementCoverage indicating if DADS uses method execution (i.e., entry/returned-into) events and statement coverage data, respectively, while methodInstanceLevel indicating the granularity of those events (i.e., for each method, using all event instances or only the first entry and last returned-into events). Due to interdependencies among these parameters, some combinations are invalid (e.g., with staticGraph disabled, neither of context-sensitivity and flow-sensitivity should be enabled). As a result, DADS uses 26 valid configurations. It starts with the most precise yet least efficient configuration: all six parameters enabled.

3 PHASE 1: INSTRUMENTATION

In this phase, DADS inserts probes into \(D\) that will monitor covered branches and entry & returned-into events of all executed methods, to create an instrumented version \(D'\). For implementation, DADS reused a Java dynamic analyzer Diver [9] (which probes for the same events for single-process programs), and invokes it for each distributed component of \(D\). DADS works at purely application level through static instrumentation. Thus, it does not handle native code or dynamically loaded code. Instead of dealing with these common limitations of traditional slicers, DADS targets better portability—it works without any customization of the run-time platforms (e.g., JVM or OS), while focusing on addressing scalability, and cost-effectiveness challenges to existing peer tools.

4 PHASE 2: ARBITRATION & ADJUSTMENT

During this continuous execution of \(D'\), DADS performs continual arbitrations to determine when to compute slices and when to adjust the configuration of the hybrid analysis for slicing.

4.1 Arbitration

Once launched along with \(D'\), DADS continuously monitors the system execution. When a method entry or returned-into event occurs, DADS records the event and its cumulative count. In particular, upon each returned-into event, DADS checks the counter and the time that lapsed since the previous round of slice computations: when the former exceeds a threshold (e.g., 100), and latter is longer than another threshold (e.g., 1 minute), DADS triggers a new round of slice computation. If the cost of any static or dynamic analysis step, such as constructing/loading the static dependence graph or computing slices, exceeds the corresponding time constraint assigned as per the user budget \(B\), DADS would cancel the analysis and record that there is a timeout. When a timeout happens, DADS records the cost (i.e., that of the analysis at the current configuration) and triggers the configuration adjustment—choosing the next configuration via Q-learning. Then, DADS resets all relevant counters and timers, starting another iteration of arbitration.

Next, we elaborate how the slices are computed and how the next configuration is determined.

4.2 Computing Slices

When a round of slice computation is triggered (at time \(t\)), DADS computes/updates the (intraprocess) slice for every executed method (i.e., every possible query) with respect to the entire \(D'\) execution up to time \(t\). This is because DADS does not know user queries in advance and works online—the dynamic data needed are processed as they come and are not stored accumulatively.

During the computation, DADS first reads the current configuration. If staticGraph is enabled and one of the static analysis parameters (i.e., context-sensitivity or flow-sensitivity) varies (between the previous and new configurations), DADS...
constructs a new static dependence graph. If statementCoverage is enabled, the static graph is then pruned according to the statement coverage for more precise slicing. Then, from the resulting static dependence graph and method events, forward dynamic dependencies are computed to form the slice of each query using the online version of DIVER [7].

If methodInstanceLevel is disabled, DADS retrieves the first entry and last return-into events from the full sequence of (i.e., instance-level) method events for faster but less precise slicing. If staticGraph is disabled, DADS uses only the method events to compute the slices based on the execution order of methods as in EAS [6], which reduces slicing cost and precision at the same time.

In this way, DADS uses different kinds of data at different granularity/sensitivity levels for the dependence analysis (which underlies the slicing) with different cost-effectiveness tradeoffs. This is also why the six parameters are chosen specifically: each of them contributes to the cost and effectiveness of the slicing in DADS in unique ways. As mentioned earlier, only intraprocess slices are computed in this phase. The interprocess dependencies are inferred from these slices at little cost during Phase 3.

4.3 Adjusting Configurations

When configuration adjustment is triggered, DADS uses a Q-learning (a type of reinforcement learning) [19] method to choose the next configuration. Other machine learning techniques, such as supervised learning, need a large training dataset, which is not available to DADS when it starts. Also, the execution dynamics may vary widely across different SUAs, thus learning from other subjects beforehand may not be effective. Reinforcement learning is not subject to such constraints. Moreover, the model of the environment (i.e., the dynamic analysis algorithm in DADS), which changes unpredictably along with the SUA execution, is unknown, so is an existing policy for configuration adjustment here. Thus, Q-learning as a model-free and off-policy reinforcement learning method is appropriate for DADS.

In DADS, the agent (the configuration adjustment module) receives a state (the current configuration) from the environment and takes an action (selecting a new configuration) according to the state while referencing either the maximal value in the Q-table or a random exploration. As a consequence, the agent receives feedback in terms of a reward computed from the action performance. We define the reward for a special configuration as 1/(the user budget \(B\)) - the slice computation time cost with the configuration) * 1000. The Q-learning algorithm encourages positive rewards and discards negative rewards. Thus, configurations with larger rewards have a greater chance to be selected. This means that when a slice computation time cost is lower than \(B\), the closer the cost is to \(B\), the more likely the analysis configuration, which corresponds to the cost, is selected.

5 PHASE 3: USER INTERACTION

DADS may have one or more user clients, called querying_client(s), to interact with the user(s). Using a querying_client, the user sends the slicing query \(Q\) to all querying_interfaces in separate processes of the instrumented system \(D'\), through the network and waits for responses. When \(Q\) arrives at a process, if intraprocess analyses have been finished and there are already slices (dependence sets), DADS would directly deliver the corresponding intraprocess slices to the querying_interface in the same process of \(D'\) and then to a querying_client. Otherwise, the querying_client must wait until all intraprocess analyses in DADS complete.

Once all the intraprocess slices are received, the querying_client computes the interprocess slice. This is done by first partially ordering the execution events of methods in the intraprocess slices according to the time stamp associated with each event. Then, it infers dynamic dependencies among methods across all the processes based on the happens-before relations among corresponding events as revealed in the global partial ordering of those events, similar to DistRA [10]. The resulting slice is the union of all these intraprocess and interprocess dependencies.

6 EVALUATION

We have successfully applied DADS against eight distributed Java systems in the real world, most of which are at industry scale. These systems cover different architectures, application domain, and scales, including a peer-to-peer system OpenChord [18], a popular distributed coordination service ZooKeeper [4] (used by Apache Hadoop and Yahoo), and a distributed key-value store Voldemort [3] (used by LinkedIn). To drive run-time executions, we used three types of (integration, load, and system) of tests that were downloaded with these subjects. Our experiments were all performed on Ubuntu 16.04 workstations with four 2.67GHz processors, 512GB DRAM, and 2TB HDD.

We are not aware of an existing dynamic slicer available that works with industry-scale continuously running distributed systems. Thus, we used the online version of a state-of-the-art dynamic analyzer D\(^2\)Abs [8], called Doda, as the baseline. Comparing DADS and Doda then suffices for evaluating the scalability and cost-effectiveness merits of automatically adjusting analysis configurations, which is right the key novelty of DADS.

While both the subjects and DADS would run continuously in real settings, for evaluation purposes, we only ran each subject for as long as ten random queries were answered by DADS. These queries were sent at random intervals between 5 to 15 seconds.

6.1 Scalability and Efficiency

DADS took 40 seconds on average for each query over all the subject executions, with a minimum of 1.53 seconds on MultiChat due to its simplicity and a maximum of 87 seconds on Voldemort due to its greatest complexity. For the same queries, the baseline took 140 seconds by average over the seven subjects it worked with, ranging from 1.86 seconds on MultiChat to 307.17 seconds on the Zookeeper load test. In short, DADS was over 3x faster than the baseline to respond to the user. The run-time overheads

![Figure 2: The overhead (y axis, left) and response time (seconds) (y axis, right) of DODA versus DADS for Netty and ZooKeeper per execution (x axis).](image-url)
was substantially more cost-effective than the baseline (in terms of response time and overhead, separately) for each slice, with abbreviations of MC. for MultiChat, OC. for OpenChord, Z for ZooKeeper, I. for integration test, L. for load test, and S. for system test). As shown, Dads was much more scalable and efficient.

Storage costs of Dads and the baseline Doda were close, 88MB on average, ranging from 2MB to 200MB.

6.2 Cost-Effectiveness

Without ground-truth slices (nor any automated tools to compute them), we manually created the ground truth for 10 random queries for each subject execution. Due to the tedious nature of this process, we limited the queries to those whose baseline ranging from 36% (NioEcho-integration) to 4624% (Netty-integration), for 2.355% on average.

Notably, the baseline did not scale to (not finish in 12 hours for Voldemort against any of its three tests. In contrast, for each query, Dads took 47.37, 46.47, and 46.39 seconds, with 87%, 226%, and 87% run-time slowdown, for Voldemort integration, load, and system test, respectively.

Dads also scaled to all other enterprise-scale distributed systems, such as Netty and Zookeeper, with acceptable response time and overheads. To illustrate, Figure 2 shows the contrasts between our tool and the baseline for these two subjects (with abbreviations of Z for ZooKeeper, I. for integration test, L. for load test, and S. for system test). As shown, Dads was much more scalable and efficient.

Storage costs of Dads and the baseline Doda were close, 88MB on average, ranging from 2MB to 200MB.

6.3 Applying Dads

As an example use case, we illustrate the application of Dads in a maintenance task for Apache ZooKeeper [4] during continuous integration (CI). The task concerns fixing the bug ZOOKEEPER-3758 [5]: “Update from 3.5.7 to 3.6.0 does not work”. Suppose the developer has made a few code changes in the method main(String[] args) of the class ZookeeperServer when trying to fix the bug. As a key requirement of CI, the developer needs to quickly test the system against this change. Accordingly, the developer should select/prioritize regression tests or develop new tests as per the impact of the changed method in a short time.

Thus, the developer sets a budget constraint (as per the overall time budget for integrating the change) for getting the impact set (i.e., the forward dynamic slice) of the method ZookeeperServer: main(String[] args). With this constraint and slicing criteria, the developer uses Dads to obtain the slice: {NIOServerCnxn: long getIds{()}, NIO ServerCnxnFactory: void run(), ZookeeperServer: void startData(), TxnHeader: int getCmd{(), ......}. Then, the developer or an automated tool uses this slice to determine which parts (i.e., the methods in the slice) should be tested, and generate/select tests accordingly, while meeting the budget constraint in the CI process.

6.4 Limitations

The first limitation is that Dads needs to insert probes into the bytecode of a software system during the instrumentation. If the owner does not allow to modify the software, Dads cannot work. Another limitation is that Dads only provides slicing results with the best cost-effectiveness it can achieve within a response time constraint (i.e., the user budget for the average time cost to query a slice). However, the results may not always be the cost-effective ones, because of the sub-optimality of our learning algorithm. In addition, the user may not set a budget, when Dads would need to use the default budget, which may not be desirable to users. On the other hand, relying on users to set an appropriate budget may not always be practical.

7 CONCLUSION AND FUTURE WORK

We developed Dads, a distributed, online, continuous, scalable, and cost-effective dynamic slicer for common continuously running distributed systems (i.e., distributed systems). Dads itself is distributed to exploit the distributed and parallel computing resources of the system under analysis. Conventional peer approaches may face applicability challenges with trace time and storage costs. However, Dads can overcome these applicability challenges through its online analysis strategy. More notably, using a Q-learning strategy, Dads addresses scalability and
cost-effectiveness challenges via continually learning and adjusting analysis configurations on the fly.

We implemented DADS for Java and applied it to eight distributed systems, with different architectures, domains, and scales, against various executions. The evaluation demonstrated promising scalability and cost-effectiveness of DADS with acceptable response time, less than 1x run-time slowdown, and almost-negligible storage costs. As future work, we plan to further optimize our tool implementation for enhanced performance. We also plan to develop practical application tools based on DADS (e.g., dynamic vulnerability detection) and validate such tools against real-world, enterprise-scale distributed systems.

REFERENCES

APPENDIX: DADS DEMONSTRATION WALKTHROUGH

In this section, we walk through a demonstration includes the complete steps for installing and using our tool in a clean environment. To run our demo, you can download the virtual machine as the DADSVM.rar file found here. Then, you can launch the virtual machine in Virtualbox, open a terminal, and go to the /home/xqfu/thrift directory. Next, you can start from Step 4 below. For more details, please refer to the README document mentioned in Step 2 below.

Note that in this demo we show that we ran the example subject to clearly illustrate the three-phase design of our tool.

1. Check prerequisites
First, please make sure the following environments are configured properly:

- Ubuntu workstation (We used Ubuntu 16.04 LTS)
- JDK 1.8 (We used java-1.8.0-openjdk-adm64)

2. Download the DADS tool
To obtain and install our tool, please follow the steps below:

- We download Dads_Material.zip file here.
- We unzip Dads_Material.zip file.
- We confirm that there is a “tool” directory that contains several jars (there are all the libraries required), and a “data” directory that contains the (calculator) application developed using Thrift.
- We check a “code” directory, which includes the source code of Dads and convenience scripts for Apache Thrift used in our evaluation study.
- Finally, we view a detailed README document, which includes all the information here in this appendix, and more.

3. Download and install an example subject
We will demonstrate the use of our tool against a subject Apache Thrift, for which the following steps can be followed

- In a terminal, we create a directory for the subject, say “thrift”, and a subdirectory “0110”.
- We download Thrift file thrift-0.11.0.tar.gz from here to the “thrift/0110” directory.
- We unzip Thrift file: tar -zxf thrift-0.11.0.tar.gz
- We copy all shell script files from the sub-directory “code/shell/Thrift” of the tool package obtained in Step 1 above to the “thrift” directory.
- We check inside each of the scripts to make sure that the “libs” related directories actually contain all the required libraries (jars) as mentioned in Step 2 above relative to your environment. In particular, set the ROOT variable to the right directory (in this case, it should be the parent directory of the “thrift” directory).
- We copy the entire “data” directory from the tool package to the “thrift” directory.

The following steps show how to run DADS against the installed Apache Thrift subject.

4. DADS phase 1: Instrumentation
In phase 1 of DADS, we execute "./DADSInstr.sh" to instrument Thrift for monitoring first/last events of all executed methods. After the execution, the instrumented program files are in the sub-folder “DADSInstrumented”, shown in Figure 4.

![Figure 4: The results of DADS phase 1: Instrumented program files](image)

5. DADS phase 2: Arbitration and Adjustment
In phase 2 of DADS, we first set milliseconds (e.g., 4,000) for a user-specified budget constraint in the file budget.txt, shown in Figure 5.

![Figure 5: The operations of DADS phase 2: The setting for the user-specified budget constraint](image)

Then, we execute "./serverDADS.sh" and "./clientDADS.sh" separately, shown in Figure 6.

![Figure 6: The operations of DADS phase 2: The initiations in the server (top) and in the client (bottom)](image)

Next, the client automatically sends computation commands and gets the computation results the server, shown in Figure 7.

We see that analysis configurations are changing during the execution according to a Q-learning strategy, shown in Figure 8.

![Figure 8: The operations of DADS phase 2: Analysis configurations changing](image)
6. **DADS phase 3: User Interaction**

In phase 3 of DADS, we start a query client to connect the Thrift server and client, and send a method name as a query, shown in Figure 9.

Then, we get the result, the method-level dependencies as the slice, shown in Figure 10. These dependencies are used as impact set as in an impact analysis supporting other tasks like regression testing. Furthermore, dynamic dependencies can be used for any other applications based on dynamic dependencies. Except for DADS, we cannot get the result from other tools with applicability, scalability, and cost-effectiveness challenges.

---

**Figure 7:** The operations of DADS phase 2: The operations in the client

**Figure 8:** The results of DADS phase 2: The changing analysis configurations

**Figure 9:** The operations of DADS phase 3: The operations in the query client

**Figure 10:** The results of DADS phase 3: The returned slice (method-level dependencies) in the query client