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

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

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

Li Li
Monash University, Australia
li.li@monash.edu

ABSTRACT

We present DADS, the first *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 time and space costs. Most importantly, DADS achieves and maintains practical scalability and cost-effectiveness tradeoffs according to a given budget on analysis time by continually and automatically adjusting the configuration of its analysis algorithm on the fly via reinforcement learning. Against eight real-world Java distributed systems, we empirically demonstrated the scalability and cost-effectiveness merits of DADS. The open-source tool package of DADS with a demo video is publicly available.

CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Software and its engineering → Maintaining software**.

KEYWORDS

Distributed system, dynamic slicing, reinforcement learning

ACM Reference Format:
Xiaoqin Fu, Haipeng Cai, and Li Li. 2020. DADS: Dynamic Slicing Continuously-Running Distributed Programs with Budget Constraints. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417920

1 INTRODUCTION

Dynamic slicing enables much tool support for software quality assurance. Moreover, a dynamic 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 existing dynamic slicers 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, 17, 19, 27] or concurrent [20, 22] 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 [18]. Yet the approach suffers scalability problems with large-scale systems, as suggested by the heavyweight nature of its analysis algorithms and its original efficiency results against even simple programs (≤894 lines of code). No existing dynamic slicer works 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 [14] (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.

We thus 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—any execution information is used only once after it occurs, and is then discarded, and it answers user queries on demand. Thus, DADS avoids tracing and, accordingly, the storage and disk I/O costs, which is essential for dealing with continuous executions (hence unbounded traces). Most of all, DADS automatically and continually (as opposed to manually and one-time [11, 12]) 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 balance.

DADS serves varied audiences. Distributed system developers may query dynamic slices of program points of interest to identify faults during testing and debugging. Tool developers may leverage DADS to build practical tools for security and performance diagnosis.

This work is licensed under a Creative Commons Attribution International 4.0 License.
ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3417920

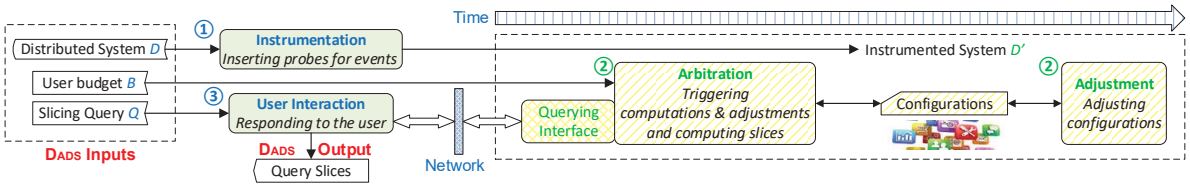

Figure 1: An overview of DADS architecture.

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 shows the DADS design, as detailed in [15]. It takes three **inputs**: a distributed system D, a slicing query Q (i.e., a method name), and a user budget B. This budget is a query response time upper bound. With these inputs, DADS continually computes dynamic dependencies and answers slice queries on demand while adjusting its configuration when necessary, in three phases:

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 continuously runs along with D', and continually arbitrates for intraprocess slice (dependence) computations and adjusts its configuration to balance the cost and effectiveness—all the intraprocess analyses, each in one of the D' processes, run in parallel hence the distributed workings of DADS. When every intraprocess slicing of Q is finished, the slice is delivered to the *querying_interface* for 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. All of the intraprocess and interprocess dependencies then 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). DADS starts with the most precise yet least efficient configuration (i.e., with all of the 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 [10] (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 the 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 computation: 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 on 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 the new configurations), DADS constructs a new static dependence graph. If *statementCoverage* is enabled, the static graph is then pruned according to the statement coverage (inferred from covered branches) 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 [8].

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 [7], 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) [26] 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 and is able to adapt. 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 discourages 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 corresponding analysis configuration 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

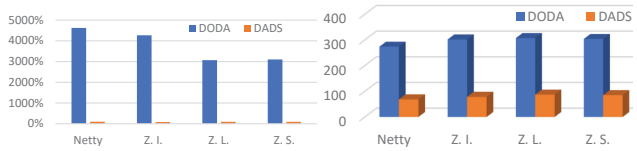

Figure 2: The overhead (y axis, left) and response time (seconds) (y axis, right) of DODA versus DADS for Netty and ZooKeeper per execution (x axis).

responses. When Q arrives at a process, if intraprocess analyses have been finished and there are already slices (dependence sets) computed, DADS would directly deliver the corresponding intraprocess slices to the *querying_interface* in the same process of D' and then to the *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 of the SUA based on the happens-before relations among corresponding events as revealed in the global partial ordering of those events, similar to DISTIA [13]. The resulting slice is the union of all these intraprocess and interprocess dependencies.

6 EVALUATION

We have successfully applied DADS against eight real-world distributed Java systems, most of which are at industry scale. These systems cover different architectures, application domains, and scales, including a peer-to-peer system OpenChord [24], a popular distributed coordination service ZooKeeper [4] (used by 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) tests.

We are not aware of an existing dynamic slicer available that works with industry-scale distributed systems. Thus, we used the *online* version of a state-of-the-art dynamic analyzer D^{2}ABS [9], called DODA, as the baseline, which uses a fixated configuration. Comparing DADS and DODA then suffices for evaluating the scalability and cost-effectiveness merits of automatically adjusting analysis configurations, 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 10 random queries were answered by DADS. These queries were sent at random intervals between 5 to 15 seconds. The budget was set between 14 to 200 seconds for these subjects.

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 [16] for its simplicity and a maximum of 87 seconds on Voldemort due to its greatest complexity. For the same queries, DODA 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. In short, DADS was over 3x faster than the baseline to respond to the user. The run-time overheads (slowdown) of DADS ranged from 31% (NioEcho [23]-integration) to 226% (Voldemort-integration),

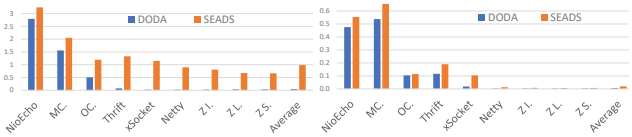

Figure 3: The cost-effectiveness expressed as ratios of precision to overhead (y axis, left) and response time (y axis, right) of DODA versus DADS per subject execution (x axis).

for 83% on average, versus those of the baseline ranging from 36% (NioEcho-integration) to 4,624% (Netty [21]-integration), for 2,355% on average over all subject executions.

Notably, the baseline DODA *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, ranging from 2MB to 200MB for 88MB on average.

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 DODA slice size was ≤30. Against the ground truth, we calculated the precision and recall of both tools. Specially, considering the changing configurations of DADS, we sent each query to it at five different times with random intervals and measured all the five returned slices.

Our results showed that, on average, DODA and DADS had 97.7% and 80% precision, respectively, while both had 100% recall with respect to the executions considered. Despite the expected higher accuracy of DODA, it is noteworthy that this accuracy may not be always applicable due to its scalability barriers: for instance, it did not scale to Voldemort; for other subjects, it was much slower.

To measure the cost-effectiveness tradeoffs of the two tools, we computed the ratios of the effectiveness (precision) to the costs (in terms of response time and overhead, separately) for each slice computed by each tool. Figure 3 shows these ratios (y axis) as a holistic cost-effectiveness metric for the subject executions (x axis) that both tools scaled to, 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. The figure shows that DADS was substantially more cost-effective than the baseline DODA (on average, 0.96 vs. 0.4 for the overhead and 0.02 vs. 0.007 for the response time), especially for real-world industry-scale distributed systems (e.g., Thrift [5], xSocket [25], Netty, and Zookeeper).

For the three smallest subjects (i.e., NioEcho, MultiChat, and OpenChord), DADS and the baseline had close cost-effectiveness because slice computations for these systems were very fast; thus, configuration adjustments hardly affect the cost-effectiveness of the dynamic dependence analyses. Yet, for other systems, the cost-effectiveness differences were significant. This indicates that the merits of DADS over the baseline were even more prominent for larger and more complex systems and executions.

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` [6]: "Update from 3.5.7 to 3.6.0 does not work". Suppose the developer has made a few code changes in the method `ZookeeperServer: main(String[] args)` to fix the bug. As a key CI requirement, the developer should quickly test the system against this change. Accordingly, the developer should 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 getSessionId(), NIOServerCnxnFactory: void run(), ZooKeeperServer: void startdata(), TxnHeader: int getCxid(), .....`}. 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

DADS is currently subject to several technical/implementation limitations. First, DADS needs to insert probes into the bytecode of a software system during the instrumentation. If the user does not allow to modify the software, DADS cannot work. Second, while DADS tries to constantly provide slicing results with a practical cost-effectiveness tradeoff with respect to a given budget constraint on response time, the cost-effectiveness it actually achieves may not be optimal. The reason is that it uses a generic learning algorithm that is not optimized for individual systems. Third, the performance of DADS can be affected by how the budget is set, while the default budget may not fit well with all systems and executions. Finally, DADS as a purely application-level analyzer does not handle native and dynamically-loaded code.

7 CONCLUSION

We developed DADS, a distributed, online, continuous dynamic slicer for common, continuously-running Java distributed systems with respect to user-specified budget constraints. DADS features an internal decision-making module that enables it to achieve and maintain practical scalability and cost-effectiveness tradeoffs, by continually learning and adjusting its algorithmic configuration on the fly via a reinforcement learning method. Our empirical evaluation demonstrated that DADS is scalable and cost-effective, substantially outperforming a similar slicer that does not have the capability of configuration learning and adjustment. Through the dynamic slices it offers within the given budget, DADS can enable applications in maintenance, testing, and security tasks that are subject to time budget constraints.

ACKNOWLEDGMENTS

This work was supported by NSF through grant CCF-1936522.

REFERENCES

[1] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. In *ACM SIGPlan Notices*, Vol. 25. ACM, 246–256.

[2] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 246–256.

[3] Apache. 2015. Voldemort. https://github.com/voldemort/.

[4] Apache. 2015. ZooKeeper. https://zookeeper.apache.org/.

[5] Apache. 2018. Thrift. https://thrift.apache.org/.

[6] Apache. 2020. ZooKeeper / ZOOKEEPER-3758. https://issues.apache.org/jira/browse/ZOOKEEPER-3758.

[7] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2005. Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 432–441.

[8] Haipeng Cai. 2018. Hybrid Program Dependence Approximation for Effective Dynamic Impact Prediction. *IEEE Transactions on Software Engineering* 44, 4 (2018), 334–364.

[9] Haipeng Cai and Xiaoqin Fu. 2019. *D2ABS: A Framework for Dynamic Dependency Abstraction of Distributed Programs.* Technical Report. Washington State University, Pullman, WA, 99163.

[10] Haipeng Cai and Raul Santelices. 2014. DIVER: Precise Dynamic Impact Analysis Using Dependence-based Trace Pruning. In *Proceedings of International Conference on Automated Software Engineering*. 343–348.

[11] Haipeng Cai and Raul Santelices. 2015. A Framework for Cost-effective Dependence-based Dynamic Impact Analysis. In *Proceedings of International Conference on Software Analysis, Evolution, and Reengineering*. 231–240.

[12] Haipeng Cai, Raul Santelices, and Douglas Thain. 2016. DiaPro: Unifying Dynamic Impact Analyses for Improved and Variable Cost-Effectiveness. 25, 2, Article 18 (2016).

[13] Haipeng Cai and Douglas Thain. 2016. DistIA: a cost-effective dynamic impact analysis for distributed programs. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. 344–355.

[14] Cleidson RB de Souza and David F Redmiles. 2008. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the 30th international conference on Software engineering*. 241–250.

[15] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li. 2020. Seads: Scalable and Cost-Effective Dynamic Dependence Analysis of Distributed Systems via Reinforcement Learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2020).

[16] GoogleCode. 2015. MultiChat. https://code.google.com/p/multithread-chat-server/.

[17] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Information processing letters* 29, 3 (1988), 155–163.

[18] Durga P Mohapatra, Rajeev Kumar, Rajib Mall, DS Kumar, and Mayank Bhasin. 2006. Distributed dynamic slicing of Java programs. *Journal of Systems and Software* 79, 12 (2006), 1661–1678.

[19] GB Mund and Rajib Mall. 2006. An efficient interprocedural dynamic slicing method. *Journal of Systems and Software* 79, 6 (2006), 791–806.

[20] Mangala Gowri Nanda and S. Ramesh. 2006. Interprocedural Slicing of Multithreaded Programs with Applications to Java. 28, 6 (2006), 1088–1144.

[21] Netty. 2020. Netty. https://github.com/netty/netty.

[22] Venkatesh Prasad Ranganath and John Hatcliff. 2007. Slicing concurrent Java programs using Indus and Kaveri. *International Journal on Software Tools for Technology Transfer* 9, 5-6 (2007), 489–504.

[23] SourceForge. 2015. NioEcho. http://rox-xmlrpc.sourceforge.net/niotut/index.html#Thecode.

[24] Bamberg University. 2015. Open Chord. http://sourceforge.net/projects/open-chord/.

[25] Vice. 2018. xSocket. http://xsocket.org/.

[26] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

[27] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. 2004. Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 502–511.