Scalable dynamic information flow tracking and its applications

Scalable Dynamic Information Flow Tracking and its Applications

Rajiv Gupta, Neelam Gupta, Xiangyu Zhang
Dennis Jeffrey, Vijay Nagarajan, Sriraman Tallam, Chen Tian
UC Riverside & Univ. of Arizona & Purdue University

Abstract

We are designing scalable dynamic information flow tracking techniques and employing them to carry out tasks related to debugging (bug location and fault avoidance), security (software attack detection), and data validation (lineage tracing of scientific data). The focus of our ongoing work is on developing online dynamic analysis techniques for long running multithreaded programs that may be executed on a single core or on multiple cores to exploit thread level parallelism.[1].

1. Introduction

Dynamic Information Flow Tracking (DIFT) is a promising technique that tracks the flow of information along dynamic dependences starting from the program inputs to the values computed by the program. Examples of applications that can benefit from a DIFT facility, include: (debugging) bug location and avoidance; (security) software attack detection and fault location; and (data validation) maintaining lineage of scientific data. The applications for which DIFT is useful are numerous; however, the overhead of dynamic *fine-grained tracing* needed by DIFT can be very high in terms of the increase in execution time of an application and substantial in terms of increased memory requirements. The actual overhead varies from one application to the next due to the complexity of information flow needed by the application. Therefore it would be beneficial to develop a general, and highly optimized DIFT framework, that can be adapted for use by developers of different applications.

In this paper we describe our experience in developing a scalable DIFT framework and employing it for the wide range of applications mentioned above. Our prior work included collection of dynamic dependence traces for single threaded programs and the performing offline processing to compactly represent them and then use them to perform dynamic slicing based fault location. Over the past year we have focused on developing a DIFT based online dependence tracking for single and multithreaded programs executing on single or multicore processors. We have used the DIFT framework in applications including: execution perturbation based fault location and avoidance, software attack detection and fault location, and data validation via lineage tracking.

[1]This research is funded by NSF grants CNS-0810906 and CNS-0751961 to Univ. of California, Riverside; NSF grant CNS-0614707 to Univ. of Arizona; and NSF grant CNS-0720516 to Purdue Univ. under the CSR Program. Contact information: Rajiv Gupta (gupta@cs.ucr.edu); Neelam Gupta (ngupta@cs.arizona.edu); Xiangyu Zhang (xyzhang@cs.purdue.edu)

2. Scalable DIFT Analysis

2.1. Single Threaded Programs: ONTRAC

In our prior work dynamic analysis was based upon a two step process [19, 5]. In the first step execution trace (address and control flow) is collected by running an instrumented version of the program and then the trace is postprocessed to build a compact representation of the dynamic dependence graph (DDG). In the second step slicing operations are performed by traversing this DDG. While previously it was thought that performing the second step in reasonable time was impossible due to the huge size of the graph even for small program runs, our work in [18] dispelled this notion by coming up with a highly compact dependence graph representation that made this step highly efficient - dynamic slices for program runs of several hundred million instructions can be computed in a few seconds. However, the generation of this compact dependence representation is still expensive. The offline post-processing had to be performed on the collected address and control flow traces to yield the compacted representation took as long as an hour even for short executions of a few seconds yielding slowdowns by factors of over 500. Since debugging is an iterative process in which the user may reexecute the program after making changes, the long time taken by DDG construction is a significant bottleneck.

In our recent work [4] we have addressed the above problem by using a DIFT framework to efficiently generate a DDG online. Our tracing system, *ONTRAC*, built on top of a dynamic binary instrumentation framework, directly computes the dynamic dependences online, thus eliminat-

ing the expensive offline post-processing step. To minimize the program slowdown, we make the design decision of not outputting the dependences to a file, instead storing them in memory in a specially allocated fixed size circular buffer. It is important to observe that the size of the buffer limits the length of the execution history that can be stored, where the execution history is a window of the most recently executed instructions. Since the dependences stored in the trace buffer pertain to the above window of executed instructions, the faulty statement can be found using dynamic slicing only if the fault is exercised within this window. Thus it is important to maximize the length of the execution history stored in the buffer. To accomplish this, we introduce a number of optimizations to eliminate the storage of most of the generated dependences, at the same time we observe from our experiments that those that are stored are sufficient to capture the bug. Besides increasing the length of the execution history that can be stored, our optimizations help limit the instrumentation overhead, because some of our optimizations identify static dependences for which dynamic instrumentation can be avoided.

The optimizations that we perform to reduce the size of the dependence graph can be classified broadly into two types. While the first kind of optimizations are generic ones, based on program properties, the second kind are exclusively targeted towards debugging. Our generic optimizations are as follows. First, we eliminate the storage of dependences within a basic block that can be directly inferred by static examination of the binary. Second, we extend the same idea to traces of frequently executed code spanning several basic blocks. Third, we detect redundant loads dynamically and exclude the related dependences. Our targeted optimizations are as follows. We first provide support to safely trace only the specified parts of the program, where the programmer expects to find the bug. This is useful because the programmer sometimes has fair knowledge about the approximate location of the bug in the code. For instance, he/she might be modifying a particular function and hence may be relatively sure that the bug is in that function. It is worth noting that a naive solution where the unspecified functions are simply uninstrumented, will not work because this could potentially break the chain of dependences through the user specified functions; this can cause the backward slice to miss some statements from the specified functions that should have been included. Our second targeted optimization is based on the observation that the root cause of the bug is often in the forward slice of the inputs of the program. This observation has been verified in our prior work [1]. Thus, by computing the forward slice of the inputs dynamically, we provide support to selectively trace only those dependences that are affected by the input.

Our experiments conducted on cpu-intensive programs from the SPEC 2000 suite show that computing the dependence trace online causes the program to slowdown by a factor of 19 on an average, as opposed to 540 times slowdown caused by extensive post-processing [18]. The optimizations also ensure that we only need to store tracing information at the average rate of 0.8 bytes per executed instruction as opposed to 16 bytes per instruction without them. This enables us to store the dependence trace history for a window of 20 million executed instructions in a 16MB buffer.

Exploiting multicores. While ONTRAC executes the instrumented programs on a single core, to further reduce the execution time overhead, we also explored an approach in which an additional core is used to carry out dependence tracking [3]. This approach exploits multicores to perform DIFT transparently and efficiently. We spawn a helper thread that is scheduled on a separate core and is only responsible for performing information flow tracking operations. This entails the communication of registers and flags between the main and helper threads. We explore software (shared memory) and hardware (dedicated interconnect) approaches to enable this communication. Finally, we propose a novel application of the DIFT infrastructure where, in addition to the detection of the software attack, DIFT assists in the process of identifying the cause of the bug in the code that enabled the exploit in the first place. We conducted detailed simulations to evaluate the overhead for performing DIFT and found that to be 48% for SPEC integer programs.

2.2. Multithreaded Programs: Execution Reduction

While ONTRAC was developed for tracing single threaded applications, we have also developed a technique that scales information flow tracking so that it can be applied to *long running*, *multithreaded* programs (e.g., server programs) being executed on a single processor core. For single threaded programs that run for short durations of time, if the program fails, program can be rerun and ONTRAC can be used during the rerun to assist in debugging. However, for long running, multithreaded programs this simple strategy does not work. First, the program may fail after executing for a long time and thus rerunning the program from the start under ONTRAC may be extremely time consuming. Second, due to non-deterministic behavior of a multithreaded program, the same failure may not be encountered during the rerun.

To enable the application of DIFT to long running, multithreaded programs, we have developed the following approach that addresses the aforementioned problems by combining fine-grained tracing for dependence tracking with the *checkpointing & logging* technique as follows:

- (Logging Phase) Under normal circumstances the pro-

gram is executed with checkpointing & logging *turned on* while fine-grained tracing is *turned off*.

- (Execution Reduction Phase) When an event is encountered that raises the need for dynamic information flow tracking, then, by analyzing the replay log, the *part of execution that is relevant to the desired dynamic information flow information is identified*.

- (Replay Phase) The relevant part of execution is *replayed with fine-grained tracing turned on* so that the desired dynamic information flow trace is captured.

The above approach has two desirable consequences. First, the space overhead is reduced because the relevant part of execution to which fine-grained tracing is applied is often a small fraction of the total execution. Our experiments show that the size of the dynamic trace captured is dramatically reduced. Second, the time overhead is also significantly reduced. This is because while checkpointing and logging, which is applied to the entire execution, typically results in a slow down by a *factor of two* while fine-grained tracing which results in a slow down by two orders of magnitude is applied only to a fraction of the execution.

As an illustration we describe the benefits of the approach when applied to a memory bug in the *MySql* version 3.23.56. The program execution under consideration has the following execution times under different conditions: original execution time is 14.8 seconds; execution time with only checkpointing & logging turned on is 16.8 seconds which is only slightly higher than the original execution time; and execution time with tracing turned-on is 3736 seconds which is much higher than the original execution time. After the proposed technique is applied, and relevant portion of the execution is replayed with fine-grained tracing turned on, the execution time is only 0.67 seconds. In other words, the relevant portion of the execution is only a small fraction of the total execution. As a consequence, the size of the dynamic data dependence trace generated reduces dramatically from 976 million dynamic data dependences to 3175 dynamic data dependences. Clearly the above example demonstrates that by applying the proposed approach the applicability of DIFT can be dramatically scaled making it a practical tool for debugging long running, multithreaded programs. A detailed description and evaluation of the above technique can be found in [6, 8].

Application executing on Multicores. In order to instrument programs for tracing, dynamic binary translation (DBT) tools like pin and valgrind are used. However, such dynamic analysis frameworks currently handle only sequential programs efficiently. When handling multithreaded programs, such tools often encounter racing problems and require serialization of threads for correctness. The races arise when application data and corresponding meta data stored in DBT tools are updated concurrently. To address this problem, transactional memory (TM) was recently proposed to enforce atomicity of updating application data and their corresponding meta data. However, a problem with this approach arises in the presence of synchronization operations typically present in parallel applications(e.g., barriers, locks, flag synchronization etc.). In particular, including synchronization operations inside transactions can cause livelocks, and thus degrade performance. In our recent work [9] we consider common forms of synchronizations and show how they can give rise to livelocks when TM is used to enable runtime monitoring. In order to avoid such livelocks, we present an algorithm that dynamically detects synchronizations and uses this information in performing conflict resolution. Our experiments show that our synchronization aware strategy can efficiently avoid livelocks and reduce monitoring overhead for the SPLASH benchmarks [9]. We plan to continue further work on tracing of parallel applications which are expected to become more prevalent due to the advent of multicore processors.

3. Applications of DIFT

3.1. Fault Location

In our prior work we demonstrated how execution perturbations can be employed to identify different forms of dynamic slices that capture faulty code and assist in fault location [13, 14, 1, 15, 17]. However, all forms of faults are not captured by dynamic slices computed as transitive closure of data and control dependences. In particular, *execution omission errors* are known to be difficult to locate using dynamic analysis. These errors lead to a failure at runtime because of the omission of execution of some statements that would have been executed if the program had no errors. Since dynamic analysis is typically designed to focus on dynamic information arising from executed statements, and statements whose execution is omitted do not produce dynamic information, detection of execution omission errors becomes a challenging task. In particular, while dynamic slices are very effective in capturing faulty code for other types of errors, they fail to capture faulty code in the presence of execution omission errors. To address this issue relevant slices have been defined to consider certain static dependences (called potential dependences) in addition to dynamic dependences. However, due to the conservative nature of static analysis, overly large slices are produced.

In our recent work [16], we propose a *fully dynamic* solution to locating execution omission errors using dynamic slices. We introduce the notion of *implicit dependences* which are dependences that are normally invisible to dynamic slicing due to the omission of execution of some

statements. We design a dynamic method that forces the execution of the omitted code by switching outcomes of relevant predicates such that those implicit dependences are exposed and become available for dynamic slicing. Dynamic slices can be computed and effectively pruned to produce fault candidate sets containing the execution omission errors. We solve two main problems: verifying the existence of a single implicit dependence through predicate switching, and recovering the implicit dependences in a demand driven manner such that a small number of verifications are required before the root cause is captured. Our experiments show that the proposed technique is highly effective in capturing execution omission errors [16]. Since dynamic control dependences play a critical role in slicing, we have developed an efficient online algorithm for detecting dynamic control dependences [11].

Recently we have developed a value profile based approach for ranking program statements according to their likelihood of being faulty [2]. The key idea is to see which program statements exercised during a failing run use values that can be altered so that the execution instead produces correct output. Our approach is effective in locating statements that are either faulty or directly linked to a faulty statement. Moreover, unlike dynamic slicing which is dependence based, this value based approach can uniformly handle all errors irrespective of whether or not they are captured by dynamic slices.

We have also made significant progress in applying our dynamic analysis framework to *data race detection* in multithreaded programs. We have extended the notion of dynamic slicing to multithreaded programs in a way that incorporates write-after-read and write-after-write dependences so that data races can be detected using dynamic slicing [8]. We have also developed a dynamic synchronization aware race detection algorithm which greatly reduces the number of data races reported to the user as many benign synchronization races and infeasible races reported by other tools are filtered out and thus excluded from reporting by our algorithm [10].

3.2. Fault Avoidance

We have developed a framework to capture and recover from *environment faults* when they occur and to prevent them from occurring again [7, 8]. These faults can be either deterministic or non-deterministic and are caused by program bugs that manifest under certain environmental conditions. Also, these faults can be avoided if the environment is appropriately modified, like changing the scheduling decisions to prevent synchronization bugs. Since environment faults could be non-deterministic, our framework employs a lightweight checkpointing/logging infrastructure that records all the important events (in an *event log*) during an execution and replays the entire execution exactly when a fault occurs. Upon a faulty execution, we modify the execution environment (e.g., change scheduling decisions) by manipulating the event log and replay the altered log so that the fault is avoided. The altered log now corresponds to an execution without any faults and normal execution can be safely resumed. By analyzing the changes done to the event log, we try to detect the unsafe execution environment that resulted in the fault (e.g., scheduling at a particular point that exposed a synchronization bug). We then record the fix in a file, *environment patch*, and all future executions of this application refer to this patch to figure out the safe execution environment when executing the region of the code where the fault appeared previously. This prevents the fault from occurring again. The checking of the environment patch file, to prevent the bug in future runs, is piggybacked with the logging of events. Hence, the only overhead incurred by our framework is that of checkpointing/logging, which is low. We have looked at three different types of environment faults that can be avoided by altering the execution environment (atomicity violation, heap buffer overflow, and malformed user request) and found our system to be effective in avoiding them.

3.3. Software Attacks

Dynamic information flow tracking is a promising technique for providing security against malicious software attacks. The basic idea hinges on the fact that an important avenue through which an attacker compromises the system is through input channels. This is a direct consequence of most of the vulnerabilities being *input validation* errors. In fact, 72% of the total vulnerabilities discovered in the year 2006 are attributed to a lack of (proper) input validation. Note that most of the memory errors including buffer overflow, boundary condition and format string errors fall into this category.

We go a step further in this direction and leverage the DIFT infrastructure to assist us in the process of bug location. The basic idea is quite simple. Instead of propagating the boolean taint values, we propagate *PC values*, where PC refers to program counter. Later, we show that the multicore architecture is able to tolerate the extra taint memory overhead incurred gracefully. As usual, a zero indicates untainted data and a non-zero (PC) value represents tainted information. At any instant, the PC value corresponding to a tainted location is the PC of the *most recent* instruction that wrote to the location. When an attack is detected, the PC taint value of the tainted memory location (or register) gives us additional information, namely the most recent instruction (statement) that modified it. This information can be vital in identifying the source of the bug and our experiments confirm that in most cases this directly points to the statement that is the root cause of the bug [3].

3.4. Data Validation

We have also extended DIFT to data lineage tracing in the application of data validation. Data lineage or data provenance captures how data is generated, what tools and parameters are used, and how one data item in the output is related to the input, and hence is essential for scientific data validation. Traditionally, data lineage is computed inside a data management system through query transformations. However, it is often the case that a significant part of scientific computation is carried out as an external process to the data management system and thus tracing lineage for these external operations is beyond the capability of traditional approaches. We observe that these external operations are indeed programs and thus data lineage can be traced through monitoring their executions.

We extend DIFT to trace the set of relevant inputs for each individual step of execution. In other words, instead of tracing a bit or a PC value, we trace a set of input values that contribute to the current executed instruction through dependences, which is essentially a genelized form of dynamic information flow tracking. The prominent challenge lies in the potentially prohibitive overhead because for each value resident in memory, we have to maintain a set; for each executed instruction, we have to perform set operations on potentially large sets. In [12], we observe that data lineage often exhibits certain characteristics that lead to efficient tracing. For example, the lineage sets for values resident in memory often have significant overlap; the input values that are in a lineage set are often clustered, i.e., if an input value is in a set, its neighboring values in the original input stream are very likely also present in the set. We exploit these characteristics using *reduced ordered binary decision diagram* (roBDD). Our experiments on a set of scientific applications show that the typical slow down factor is less than 40 when the valgrind infrastructure overhead is discounted. The memory overhead is 300% on average. Given the fact that lineage sets could be as large as thousands of elements, the extended DIFT system is capable of tracing data lineage with cost that can be tolerated in the context. Applying the system to a realistic bio-chemistry application at Purdue unversity identifies a few false positives in a real experiment, which may otherwise result in highly expensive wet-bench experiments.

4. Work in Progress

Our ongoing work is focused on online, cost effective tracing of parallel applications as well as data race detection in parallel applications. We are developing both software techniques as well as identifying hardware support that can be incorporated in multicore processors to carry out the task of tracing efficiently. In addition to employing efficient tracing to enable debugging of parallel applications, we also plan to explore its use in performing adaptive optimizations.

References

[1] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating Faulty Code Using Failure-Inducing Chops," *ASE*, 2005.

[2] D. Jeffrey, N. Gupta, and R. Gupta, "Fault Localization Using Value Replacement," *submitted*, 2008.

[3] V. Nagarajan, H-S. Kim, Y. Wu, and R. Gupta, "Dynamic Information Flow Tracking on Multicores," *INTERACT*, 2008.

[4] V. Nagarajan, D. Jeffrey, R. Gupta, and N. Gupta, "ONTRAC: A System for Efficient ONline TRACing for Debugging," *ICSM*, 2007.

[5] S. Tallam and R. Gupta "Unified Control Flow and Dependence Traces," *ACM TACO*, 4(3), 2007.

[6] S. Tallam, C. Tian, X. Zhang, and R. Gupta, "Enabling Tracing of Long-Running Multithreaded Programs via Dynamic Execution Reduction," *ISSTA*, 2007.

[7] S. Tallam, C. Tian, R. Gupta, and X. Zhang, "Perturbing Program Execution For Avoiding Environmental Faults," *submitted*, 2008.

[8] S. Tallam, "Fault Location and Avoidance in Long-Running Multithreaded Applications," PhD Thesis, U. Arizona, 2007.

[9] C. Tian, V. Nagarajan, and R. Gupta, "Synchronization Aware Conflict Resolution for Runtime Monitoring Using Transactional Memory," *submitted*, 2008.

[10] C. Tian, V. Nagarajan, and R. Gupta, "Dynamic Recognition of Synchronizations for Data Race Detection," *submitted*, 2008.

[11] B. Xin and X. Zhang, "Efficient Online Detection of Dynamic Control Dependence," *ISSTA*, 2007.

[12] M. Zhang, X. Zhang, X. Zhang, and S. Prabhakar, "Tracing Lineage Beyond Relational Operators," *VLDB*, 2007.

[13] X. Zhang, N. Gupta, and R. Gupta, "Locating Faulty Code By Multiple Points Slicing," *SP&E*, 37(9), 2007.

[14] X. Zhang, N. Gupta, and R. Gupta, "A Study of Effectiveness of Dynamic Slicing in Locating Real Faults," *Empirical Software Engineering*, 12(2), 2007.

[15] X. Zhang, R. Gupta, and N. Gupta, "Locating Faults Through Automated Predicate Switching," *ICSE*, 2006.

[16] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards Locating Execution Omission errors," *PLDI*, 2007.

[17] X. Zhang, N. Gupta, and R. Gupta, "Pruning Dynamic Slices With Confidence," *PLDI*, 2006.

[18] X. Zhang and R. Gupta, "Cost Effective Dynamic Program Slicing," *PLDI*, 2004.

[19] X. Zhang, N. Gupta, and R. Gupta, "Whole Execution Traces and Their Use in Debugging," *The Compiler Design Handbook: Optimizations and Machine Code Generation*, 2nd Edition, Chapter 4, CRC Press, 2007.