

# Research Statement

Sriraman Tallam

<http://www.cs.arizona.edu/~tmsriram>

My dissertation research focuses on developing techniques for *efficiently collecting and storing the dynamic information (execution trace)* from program executions involving one or more threads. These execution traces are widely used in debugging and can be in the order of gigabytes even for a few seconds of execution. I have developed a representation to enable the compact storage of these traces on disk. I have also developed techniques to enable the tracing of programs that run forever, which is infeasible using conventional trace collection techniques.

I have performed research in the areas of *Program Profiling, Software Debugging, Software Testing and Embedded Systems*. I have developed a technique to enable profiling of those paths which cross loop back-edges and procedure boundaries. I have contributed to the development of a dynamic technique to locate errors that involve omitting the execution of some statements in a program. In the area of Software Testing, I have proposed a new heuristic for test-suite reduction. In the area of Embedded Systems, I have proposed a new register allocation algorithm for embedded processors supporting instruction sets to allow referencing of register bit-sections. I have also proposed a technique to conserve power on embedded devices when it is possible to execute parts of an application remotely.

## Scalable Collection and Storage of Program Traces

Execution traces have been collected and analyzed for a wide range of applications. Control and Dependence traces, in particular, have been shown to be very useful in producing reliable software through debugging and testing. These traces can be used to construct dynamic slices, which are known to be very effective in identifying the root cause of a faulty execution. However, program runs generate traces in the order of gigabytes in a few seconds. Hence, the process of collecting and storing these traces poses a significant challenge. Further, in the case of long-running programs such as server applications that run forever, even the best techniques for collecting and storing traces can exhaust available resources. In my research, I have developed techniques to address these challenges.

**Storing Control and Dependence Traces compactly on disk** [TACO 2007, PACT 2005] — Although techniques exist to compactly store control traces on disk, they are not effective for dependence traces. A few other techniques collect and store address traces instead, as the data dependences can be recovered from them easily. However, address traces too cannot be stored as compactly as control traces.

I have developed a representation called *Extended Whole Program Paths (eWPP)* to compactly store on disk the control and data dependence history of a program's execution. This representation is motivated by the observation that a significant fraction of the data dependence history can be recovered from the control flow trace. To capture the remainder, disambiguation checks are introduced in the program whose control flow signatures capture the result of the checks. The resulting extended control flow trace enables the recovery of otherwise irrecoverable data dependences. The code for the checks is designed to minimize increases in the program execution time and the extended control flow trace size when compared to directly collecting control flow and address traces. Experiments show that compressed *eWPPs* are only a quarter of the size of combined compressed control flow and address traces.

**Enabling Tracing of programs that are long-running and multithreaded** [ISSTA 2007, FSE 2006] — For programs that are multi-threaded and long-running, debugging them using trace based analyses is a very challenging problem. Since such programs are non-deterministic, reproducing the bug is non-trivial and generating and inspecting traces for executions running for days could be prohibitively expensive. Some prior techniques use checkpointing mechanisms to reduce tracing efforts by storing dynamic information since a recent

checkpoint. However, tracing is still expensive as a checkpoint interval is in the order of several minutes and traces can become very large within this interval itself. I have developed two techniques called *Execution Fast Forwarding* (EFF) and *Execution Reduction* (ER) to address these challenges.

In order to overcome the problem of bug reproducibility, in the framework, a lightweight logging technique is used to record the events during the original execution. When a bug is encountered, it is reproduced using the generated event log and a fine-grained tracing technique is employed during replay to collect the control-flow and dependence traces. In order to minimize the information collected during tracing, only the execution information from those regions of threads that are relevant to the fault is collected. This approach is highly effective because of the observation that in long-running multithreaded programs, many threads that execute are irrelevant to the fault. Hence, these threads need not be replayed and traced when trying to reproduce the bug.

I have developed a novel lightweight scheme that identifies threads that are not relevant to the fault by observing all the interthread data dependences. Any thread that is irrelevant to the faulty execution is removed from the event log by pruning the corresponding events. In addition, for each thread that has to be replayed to reproduce the fault, the subset of regions that contribute to the fault is identified and selectively traced. For instance, the parent thread of a faulty thread has to be replayed but not necessarily traced. Hence, after execution reduction, the replayed execution takes lesser time to run and in combination with selective tracing produces a much smaller trace than the original execution. This reduces the cost and effort for generating and inspecting the traces. Experiments show that this technique can reduce the tracing time by three orders of magnitude and the trace sizes by two to five orders of magnitude.

**Profiling Overlapping Paths** [CGO 2004] — Path profiles have been shown to be very useful in guiding many program optimizations and in instruction scheduling. Also, efficient techniques exist to collect these profiles without incurring a huge execution overhead. However, these techniques consider only acyclic program paths. They do not collect profiles of paths that cross loop and procedure boundaries, which is referred to as *interesting paths*. The profiles of interesting paths are desirable as they are very useful in many program optimizations. There are potentially a very large number of interesting paths that makes it very hard to obtain their accurate profiles and their profile estimates from acyclic program paths are highly inaccurate, ranging from -38% to +138%.

I have developed a class of paths called *overlapping paths*, whose profiles can be used to estimate the profiles of interesting paths with a very high degree of accuracy, between -4% and 8% . The overlapping paths are longer than acyclic paths but shorter than interesting paths. Each overlapping path has a degree which determines the length of the path. Efficient techniques to obtain profiles of overlapping paths of any degree have been developed.

## Software Debugging and Testing

**Slicing of Multi-threaded Programs** — Since dynamic slices have been shown to be very effective in locating the root cause of errors in single-threaded programs, I am developing techniques to apply dynamic slicing to multi-threaded programs. In the case of single-threaded programs, the dynamic dependences that need to be tracked to construct slices to detect most bugs are *data*(RAW), *control* and *relevant* dependences. However, in multi-threaded programs, additional dependences arise. The data dependences, *WAW* and *WAR*, must be tracked to detect bugs due to data races in these programs. However, there could potentially be a large number of dynamic *WAW* and *WAR* dependences making the slices very huge. To avoid this, the set of *WAW* and *WAR* dependences is pruned using the happens-before relationship. I am also developing techniques to efficiently detect inter-thread control and relevant dependences dynamically.

**Handling Execution Omission Errors** [PLDI 2007] — In the area of Software Debugging, I have contributed to the development of a technique to identify errors in a program that manifest due to omitting the execution of

some statements. Detection of errors due to execution omissions is challenging using dynamic analysis because the omitted statements do not generate any dynamic information. For example, while dynamic slices are very effective in capturing faulty code for other types of errors, they fail to capture faulty code involving execution omission errors. A fully dynamic solution has been developed for locating execution omission errors using dynamic slices by introducing the notion of *implicit dependences*, which are otherwise invisible to dynamic slicing. Dynamic slices can then be computed and effectively pruned to produce fault candidate sets containing the execution omission errors.

**Support for Dynamic Slicing using Reverse Breakpoints in Microsoft's *cordbg* Debugger** — I am involved in a project to integrate dynamic slicing capabilities into the *cordbg* debugger. Since slicing requires collecting dependence information which can be expensive, we have provided support for reverse execution. Reverse execution can allow the user to step backwards and this feature in itself can be very useful while debugging. Also, it can help control the cost of slicing as the debugger can go back and forth in the program run and then specify a window for which he wishes to inspect the slice. Then, the slicing framework will only trace that particular window thereby limiting the cost of tracing and slicing.

**Test Suite Minimization** [PASTE 2005] — In Software Testing, test-suite minimization is a well known problem. Here, the aim is to identify a minimal cardinality subset of test cases from a huge test suite such that the subset of test cases provides complete coverage of one or more types of program entities (testing requirements). Since the problem is NP-hard in general, heuristics have been proposed and used to solve this quickly. I have developed a greedy heuristic that iteratively exploits the implications among test cases and the testing requirements in order to select the smallest subset of test cases that satisfies the testing requirements. Experiments have shown that this heuristic can generate smaller test suites than well known previously used greedy heuristics. Indeed, the heuristic computes optimal solutions most of the time.

## Embedded Systems

**Bitwidth Aware Register Allocation** [POPL 2003] — I have developed a technique to perform register allocation in embedded processors supporting instruction sets which allow direct referencing of bit sections within registers. This makes it possible for more than one sub-word variable to reside in a single register. The technique has two key steps. First, a combination of forward and backward data flow analyses are developed to determine the bitwidths of program variables throughout the program. Second, a novel interference graph representation is designed to enable support for a fast and highly accurate algorithm for packing of sub-word variables into a single register. On media and network processing applications which make extensive use of sub-word data, the technique can reduce register requirements by 10% to 50% when compared to a traditional scheme.

**Power Aware Program Partitioning** [JavaPDC 2004] — I have developed a program partitioning technique to save energy on an embedded device when it is possible to execute parts of a program remotely on a server through wireless communication. The technique considers the trade-off between communication and computation to decide whether a piece of the program should be executed locally or communicated to the server to be executed remotely. Experiments show that this technique can conserve energy in an embedded device from 29% to 43%.

## Future Directions

**Hardware support for tracing with many cores** — Processors with many cores are becoming the trend now. Hence, I want to investigate techniques using support from additional cores to simultaneously perform tracing while the program is running on a dedicated core. This can tremendously boost the performance of tracing and can make it possible to trace on-line. I believe that this direction is promising because with increase in the number of cores per processor, heterogeneous cores could be the trend where each core is

geared towards a specific functionality. I want to make the case for a set of cores being dedicated for tracing.

**Online Fault Avoidance Techniques** – I am interested in techniques for on-line recovery of faults in applications and have also done some work in this regard. I want to continue to pursue this direction. I have found that a major percentage of faults occurring in many applications can be avoided by suitably modifying the execution environment. Hence, when a fault occurs during execution, it is possible to re-execute the region of the application corresponding to the the fault under a modified environment thereby avoiding the fault. I am looking at techniques([TR-1]) that can be applied to capture and avoid these faults as and when they occur. Further, I am also investigating schemes to prevent faults from occurring more than once.

## References

- [1] TACO            **S.Tallam** and R. Gupta, "Unified Control Flow and Dependence Traces," *ACM Transactions on Architecture and Code Optimization*, 30 pages, to appear.
- [2] ISSTA           **S. Tallam**, C. Tian, X. Zhang, and R. Gupta, "Enabling Tracing of Long-Running Multithreaded Programs via Dynamic Execution Reduction," *International Symposium on Software Testing and Analysis*, London, UK, July 2007.
- [3] PLDI            X. Zhang, **S.Tallam**, N.Gupta, and R. Gupta, "Towards Locating Execution Omission Errors," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, June 2007.
- [4] FSE             X. Zhang, **S.Tallam**, and R. Gupta, "Dynamic Slicing Long Running Programs through Execution Fast Forwarding," *14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 81-91, Portland, Oregon, November 2006.
- [5] PACT            **S.Tallam**, R. Gupta, and X. Zhang, "Extended Whole Program Paths," *International Conference on Parallel Architectures and Compilation Techniques*, pages 17-26, Saint Louis, Missouri, September 2005.
- [6] PASTE           **S.Tallam** and N. Gupta, "A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization," *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Lisbon, Portugal, Sep. 2005.
- [7] CGO            **S.Tallam**, X. Zhang, and R. Gupta, "Extending Path Profiling across Loop Backedges and Procedure Boundaries," *Second Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 251-262, San Jose, CA, March 2004.
- [8] JavaPDC        **S.Tallam** and R. Gupta, "Profile-Guided Java Program Partitioning for Power Aware Computing," *Sixth International Workshop on Java for Parallel and Distributed Computing*, Santa Fe, NM, April 2004.
- [9] POPL           **S.Tallam** and R. Gupta, "Bitwidth Aware Global Register Allocation," *30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85-96, New Orleans, LA, January 2003.
- [10] TR-1           **S. Tallam**, C. Tian, X. Zhang, and R. Gupta, "Perturbing Program Execution For Avoiding Environmental Faults," *In Submission*.