Floating Point Debugging via Basic Block Tracing

Saeed Taheri, Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, Utah
{staheri,ganesh}@cs.utah.edu

***Abstract*—Complex numerical programs are widely used in a variety of platforms and processors. Heterogeneity of compilers and architectures often cause inconsistencies in producing deterministic results, especially in complex numerical code bases. Thus root cause analysis of errors and failures of such systems is badly needed. We have been developing frameworks to efficiently obtain insight about program dynamic behavior towards failure root cause. In this paper, we introduce dynamic basic block tracing and propose ideas about analyzing the collected traces towards locating the source of floating-point bugs.**

***Index Terms*—debugging, floating point, basic block, tracing**

I. Compiler Inducing Floating Point Variability

Important high performance computing (HPC) applications are long-lived, and must be migrated across platforms and compilers, and also their optimization levels must be changed to gain performance. Unfortunately, these changes often change the computed results, and are even known to lead to unexpected deadlocks [1]. As another example, a dot product over two arrays of float numbers (listing 1) produces 16877216 when compiled with `g++ -O2` and 16877222 when compiled with `g++ -O3 -funsafe-math-optimizations -msse2`.

```
float dot(float* x, float* y, int n) {
  float sum = 0.0;
  for (int i = 0; i < n; i++) {
    sum += x[i] * y[i];
  }
  return sum;
}

main(argc, argv) {
 float x[8] {16777216.0f, 1.f ,1.f ,1.f ,1e5f,1.f,1.f,1.f};
 float y[8] {1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f};
 std::cout << dot(x, y, 8) << std::endl;
}
```

Listing 1: Dot Product

Such answer changes are often triggered by the introduction of new instructions (e.g., fused multiply add), vectorization (that can change the order of accumulation of results), or even the selection of new library bindings by a compiler. Efficiently root causing and correcting such behavioral changes is, unfortunately, an unsolved problem.

Related work has studied compiler debugging, however in the integer space [2]. Other works have supported the formal analysis of floating-point operations at the LLVM[3] level. These efforts do not directly address errors discovered upon porting or optimizing code differently. Recent work has successfully demonstrated the root-causing of floating-point variability using bisection based testing [4]. Although such technique have been shown to work in practice, it can be time consuming. Also, significant differences in the compiled floating-point library may get overlooked.

In this work, we propose a new approach for solving such challenges with the goal of finding the point at which the control flow of an application *diverges under two compilations.* The basic idea is to mine binary traces of executions using binary instrumentation, and then to analyze such traces across two different compilations. We have developed a facility to conduct *Dynamic Binary Instrumentation* and capture the sequence of executed *basic block traces*. An important requirement is to reduce overheads. In our work, the traced sequences are incrementally compressed on-the-fly, resulting in compact trace files with minimal overhead (cf. Section II). After program termination, traces are decompressed and represented in the form of control-flow graphs (CFG), with an abstraction of basic blocks as graph nodes. Each CFG is a summary representative of the floating-point behavior of the program (or a specific function). Applying Formal Concept Analysis (FCA)[5] techniques, our prior work has shown that it is possible to find the critical point of divergence in CFGs across two compilations(cf. Section III).

II. Basic Block Tracing

A natural and field-proven approach for debugging is to capture detailed execution traces and compare them against corresponding traces from a base run. Dynamic Binary Instrumentation (DBI) frameworks had been used widely in performance analyzer and debugger tools [6]. In dynamic binary instrumentation, code behavior can be monitored at runtime, making it possible to handle dynamically generated and even self-modifying code, or the specific libraries chosen.

In our previous work [7], We have introduced ParLOT, a dynamic tracing tool that efficiently collects whole-program function calls and returns. The traces are compressed incrementally on-the-fly using a novel compression scheme we have developed. Inspired by this experience, we propose the use of Intel Pin [8] to instrument the binary of target applications, and efficiently track the execution of basic blocks—in effect building *basic block traces*.

We still represent the program execution as a sequence of symbols, with each basic block summarized via a unique ID. The ParLOT approach to compression thus can be made to

Table I: MFEM Basic Blocks

		Trace Length	Instrumented Blocks	Executed Blocks
Whole Program	MFEM	1,138,962,074	13550	11426
	MFEM-X	1,155,842,436	13577	11445
`addMult-a-AAt`	MFEM	1,843,200	22	18
	MFEM-X	1,843,200	19	15
`DenseMatrix-Invert`	MFEM	855,040	96	73
	MFEM-X	880,640	91	71

Table II: Instruction Sets

SSE_DATA	SSE_ARITH	SSE_OTHER	SSE2_DATA
SSE2_ARITH	SSE2_OTHER	FP_DATA	FP_ARITH
FP_OTHER	AVX	AVX2	FMA

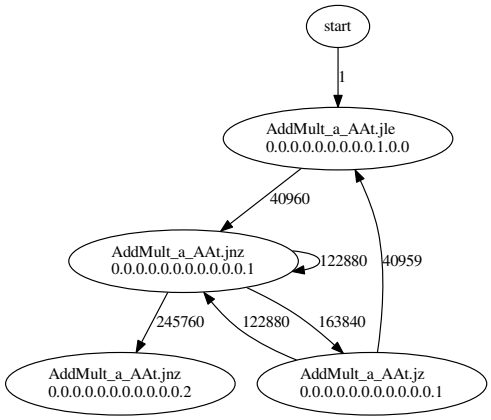
work for basic blocks. Table I presents statistics of our initial experiments on a specific test contained in MFEM, a finite element library [9]. MFEM-X refers to a build of MFEM which has been compiled with a set of flags that caused variability of the final answer. Functions `DenseMatrix-Invert` and `addMult-a-AAt` are flagged by FLiT-Bisect [4] as the root cause of answer variations. However, FLiT does not help diagnose why these functions differed in their behavior. Our proposed work will be directed at such detailed analysis.

The content of each basic block (i.e., sequence of instructions) is stored in a file at instrumentation time as *info file*. The compelete or partial CFG of basic blocks can be constructed from the trace and labeled by the information of info file. For example, figure 1 shows the CFG of basic blocks of function `addMult-a-AAt` where each node is labeled as <Function>.<Last instruction of the Basic Block>.<Floating Point instruction bitvector>. Each element of the bit vector corresponds to a set of instructions categorized based on the interesting floating point instructions. The value of each element shows the frequency of appearance of floating-point instructions within each category (table II).

III. Research Plans

In our recently accepted tool [10], we have used Formal Concept Analysis [5] techniques to narrow down the search space from thousands of concurrent traces into just a few "suspicious" traces. The approach is to convert the context of search space into a *formal concept* and inject each concept to a *concept lattice* that provides a full pair-wise similarity matrix in linear time. Within each trace, long sequences of trace entries are converted to loss-less Nested Loop Representation (NLR) for better readability. Currently, we are developing the existing FCA framework to cover basic block traces. Then by extracting *attributes* from each function, we pursue the goal of finding which function's behavior(w.r.t. floating-point operations) have been changed the most after re-compilation with different flags.

Figure 1: CFG of `addMult-a-AAt`



Also, the NLR implementation getting completely parallelized and gaining up to 20x speedup. Using NLR to summarize 1.3 billion-long traces (table I - row Whole-Program) into its minimal NLR representation would make trace analysis much feasible. Inspired by [11], we are also studying possible approaches to use tools like MonoSat[12] to infer from basic block bit-vectors.

References

[1] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Preliminary experiences with the uintah framework on intel xeon phi and stampede," in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, ser. XSEDE '13. New York, NY, USA: ACM, 2013, pp. 48:1–48:8.

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294.

[3] D. Menendez, S. Nagarakatte, and A. Gupta, "Alive-fp: Automated verification of floating point based peephole optimizations in llvm," in *International Static Analysis Symposium*. Springer, 2016, pp. 317–337.

[4] M. Bentley, I. Briggs, G. Gopalakrishnan, D. H. Ahn, I. Laguna, G. L. Lee, and H. E. Jones, "Multi-level analysis of compiler-induced variability and performance tradeoffs," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19. New York, NY, USA: ACM, 2019, pp. 61–72.

[5] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, 1st ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.

[6] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *IEEE Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[7] S. Taheri, S. Devale, G. Gopalakrishnan, and M. Burtscher, "ParLOT: Efficient whole-program call tracing for HPC applications," in *Programming and Performance Visualization Tools - International Workshops, ESPT 2017 and VPA 2017, Denver, CO, USA, November 12 and 17, 2017, and ESPT 2018 and VPA 2018, Dallas, TX, USA, November 16 and 11, 2018, Revised Selected Papers*, 2018, pp. 162–184.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[9] T. Kolev and V. Dobrev, "Mfem: Modular finite element methods library," jun 2010.

[10] S. Taheri, I. Briggs, M. Burtscher, and G. Gopalakrishnan, "Difftrace: Efficient whole-program trace analysis and diffing for debugging," in *Proceeding of IEEE Cluster Conference, 2019, Albuquerque, NM, USA, September, 2019, Under publication*, 2019. [Online]. Available: https://staheri.github.io/files/IEEE19-diffTrace.pdf

[11] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Computer Aided Verification*, D. L. Dill, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 68–80.

[12] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu, "SAT modulo monotonic theories," *CoRR*, vol. abs/1406.0043, 2014. [Online]. Available: http://arxiv.org/abs/1406.0043