Research Statement

Xu Liu

Over the past decades, there is an increasing demand on computation capabilities in the domains of scientific computing, machine learning, big data, and others. With the exponential growth of computation and input data, existing programs can no longer solve problems in reasonable time when executed sequentially. Today, parallel applications routinely employ data centers and supercomputers to meet their computational needs. However, it is always difficult to obtain high performance. The difficulties come from two aspects: software and hardware.

On the one hand, modern applications evolve to integrating multiple components within multiple layers, which can easily bloat the source code size to millions of lines. Understanding the performance problem in each code component is challenging. Moreover, parallel programs are designed for large scales, which usually employ sophisticated programming models for scale out and up. How to choose appropriate programming models and efficiently use them for best performance is still an open question for program developers.

On the other hand, modern and emerging computation architectures are complex. First, modern supercomputers and data centers have millions of cores toward the exascale computing target. To achieve this goal, a large number of heterogeneous accelerators (e.g., GPU) are used to increase their computational power. Second, modern computer architectures employ deep and heterogeneous memory hierarchies to bridge the speed gap between CPU and memory or disk. Frequent data movement in such deep memory hierarchies not only degrades performance but also consumes energy.

Though powerful, mapping applications to architectures with these new features to get bare-metal performance is difficult. One needs performance insights to guide code design and optimization. To obtain performance insights, using a performance tool to analyze program executions is the most straightforward method. **My research interest falls into building robust, efficient performance tools that are generally applicable to large-scale applications.**

1. RESEARCH HIGHLIGHTS

We have built several open-source tools, such as Witch, CCTLib, ScaAnalyzer, SMTAnalyzer, and many others, some of which have already been deployed in DoE national laboratories and evaluated in the industrial companies. Beyond releasing these tools to benefit the community, we also contribute to common software. For example, we contribute code about fast setting up debug registers to Linux kernels, which got upstreamed. We are also involved in standardizing a set of tools API for widely used OpenMP languages, called OMPT, which got accepted by the committee. We believe our research directly benefits to not only code optimizers, but also tool developers.

2. RESEARCH DESCRIPTIONS

The research in my group mainly has three goals: (1) exploring new performance problems with new techniques; (2) reducing the overhead of analyzing existing performance problems to make tools applicable in production environments; and (3) under the guidance of our tools, optimizing

important software packages in different domains, such as well-known machine learning and graph processing frameworks.

2.1 New Techniques for New Performance Problems

Within this project, we have developed CCTLib [CGO'14], a fine-grained profiler that identifies redundant or useless operations in a program. We say this kind of performance problems is new because people previous rely on compilers to optimize them, with no insight into their existence in the fully optimized binary code. Thus, we develop CCTLib, which analyzes fully optimized binary on the fly.

CCTLib is a framework. Its functionality is to provide the calling context of each dynamic instruction instance and associate it with source code. To make CCTLib practical, we have handled signals, exceptions and multithreading, and adopted various optimization techniques to minimize its overhead. As a framework, CCTLib provides rich APIs to support client tools that identify wasteful operations and associate them with full calling contexts and source code to provide necessary insights for optimization.

We have developed three client tools: RVN [PACT'15] to identify symbolic equivalent computations, RedSpy [ASPLOS'17] to identify silent write operations, and LoadSpy [ICSE'19] to identify silent load operations. These client tools pinpoint wasteful operations due to not only poor binary code generation, but also high-level semantics, such as inefficient data structures and suboptimal algorithms.

[**HighLights**]

1. RedSpy identifies silent stores in a 6-million-line production software NWChem from Pacific Northwest National Laboratory. The optimization guided by RedSpy yields a 50% speedup. The optimization fix has been integrated into the latest NWChem.
2. RedSpy was one of ASPLOS Highlights and nominated as Best Paper in ASPLOS'17.
3. LoadSpy won Distinguished Paper Award in ICSE'19.

2.2 New Techniques for Existing Performance Problems but Low Overhead

Low overhead is critically important for a performance tool mainly due to two reasons. First, with low overhead, the tool does not disturb program execution much, which can most accurately expose the performance issues in the software and hardware interaction. Second, the low overhead is necessity in deploying the tools in the production environment. Under the umbrella of this project, we mainly target two problems that states of the art utilize the heavyweight profiling methods. One is memory profiling that mostly relies on heavyweight memory instrumentation and simulators; the other is redundancy profiling that relies on our fine-grained profiling framework CCTLib.

Lightweight memory profiling

We have developed ScaAnalyzer [SC'15] to identify memory performance issues. The basic idea of ScaAnalyzer is to use performance monitoring units (PMU) available in commodity CPU processors to monitor a program execution. ScaAnalyzer performs data-centric analysis to associate performance statistics obtained from PMUs with data objects allocated in the program. Furthermore, ScaAnalyzer performs lightweight memory access pattern analysis on each monitored data object and guides various optimizations. While ScaAnalyzer mainly targets the memory scaling bottlenecks, there are a number of tools we develop based on it to pinpoint other memory issues, such as StructSlim [CGO'16] to guide structure splitting, SMTAnalyzer

[HPDC'16] to guide locality optimization on SMT architectures, Dr-BW [IPDPS'17] to identify memory bandwidth congestion in NUMA architectures, ProfDP [ICS'18] to guide data placement in heterogeneous memory systems, Feather [PPoPP'18] to detect false across threads and processes, and RDX [HPCA'19] to collect reuse distance histograms. All the tools derived from ScaAnalyzer incur low runtime overhead, which is typically less than 5%.

[**Highlights**]
1. ScaAnalyzer received the Best Paper Award from SC'15 (Supercomputing'15).
2. Feather received the Best Paper Award from PPoPP'18.

Lightweight redundancy profiling

CCTLib client tools can effectively identify redundancy in a software, but the runtime overhead is from 30x to 80x, which prevents it from practical usage. Thus, to make the redundancy profiling practical, we developed Witch, which typically incurs 3% overhead. The basic idea is that instead of analyzing the full memory trace, Witch continuously samples pairs of memory accesses that two parties in each pair are adjacent to each other to access the same memory location *M*. By investigating whether there exists any intervening loads from *M* between a pair of stores, we can know whether the first store is a dead store or not. By investigating the values of the two stores/loads written to/read from *M*, we can know whether the second store/load is silent store/load.

To sample memory access pairs, Witch uses the hardware performance monitoring units (PMU) to capture the first memory access in the pair and hardware debug registers to trap the following access to the same memory location. This scheme will need no code instrumentation. To make Witch usable, we have addressed a number challenges including the signal handling and limited debug registers. With the help of Witch, we are able to optimize a number real applications and obtain significant speedups.

[**Highlights**]
1. Witch was nominated to ACM SIGs for CACM Research Highlights.

Lightweight hardware transactional memory profiling

We have developed TxSampler [PPoPP'19] to identify performance issues in hardware transactional memory (HTM). TxSampler leverages hardware performance monitoring units available in commodity CPU processors to provide deep performance insights with only 4% overhead. Together with TxSampler, we have released the first benchmark suite based on HTM, which consists of more than 30 programs.

[**Highlights**]
1. TxSampler won the Best Paper Award at PPoPP'19.

2.3 Optimizing Important Applications Guided by Our Tools

To gain the impact of our research, we have also thoroughly optimized important software packages in various domains, under the guidance of our tools.

Locality optimization for Deep Learning system Caffe

Caffe is an important deep learning framework. With ScaAnalyzer, we have observed that Caffe incurs a large amount of remote accesses in NUMA architectures. Thus, we have specifically optimized Caffe by replicating multiple neural networks, one per socket. Our optimization largely

reduces the remote accesses and receives more than 50% speedup. The optimized Caffe is open source and our technique is documented in TACO'18.

Redundancy optimization for graph systems

With CCTLib and Witch, we have observed that some kinds of graph processing algorithms suffer from significant redundant operations. For example, when compute the single source shortest path (SSSP), the value for an intermediate vertex can be updated multiple times because from the source to target, there can be multiple paths. It is worth noting that all the intermediate values assigned to that vertex can cause the following value propagation redundant, because only the last assignment to that vertex has the minimum value. All other intermediate values will be overwritten. To optimize the useless operations, we have integrated the input graph's topology information. A vertex will bypass any value assignment before the final minimum value, which denotes the short path. This work will be published in VLDB'19.

3. FUTURE RESEARCH

In the future, I would like to continue my work in the areas of tools building but will significantly expand my exploration. The philosophy of research does not change: we build practical open-source tools with novel techniques to benefit the communities such as architecture, system, and applications. To be detail, I list some ideas as follows. Some of the ideas have been in currently exploration or documented in a research proposal.

Performance tools for new languages. In recent years, there is an exponential increase in using managed languages in various domains. Example languages include Java, Java Script, GoLang, Swift, and Python. These languages all have runtime systems to automatically manage memory. Some of them even use just-in-time compilation techniques to boost the performance. All of these features add extra layers in the code that increase the difficulties in understanding the performance inefficiencies. One of my future tasks is to build tools for these languages. More importantly, I will explore the possibility in standardizing tools API for these languages. We have successful experience in standardizing OpenMP tools API—OMPT.

Tools for debugging and security. Besides performance, functionality and security bugs or vulnerabilities are also important. Unfortunately, most existing tools are not practical due to their high overhead and limited scope in identifying different bugs. A natural extension of our tools work is to pinpoint these bugs effectively and efficiently. This proposed project will target an clear goal: how to design and implement a production tool to identify bugs. We will begin with memory-related bugs, such as data races, atomic violations, and memory-related vulnerabilities, such as buffer overflows, use after free, double fetches, or double frees. Eventually, we will extend our tools for other bugs. We will also consider to integrate the static analysis to aid our dynamic analysis.

Tools for the clouds, edges, and emerging architectures. With the rapid evolution of computer systems and architectures, there is a large demand on tools for widely used cloud systems, edge system, and specialized accelerators (especially for deep learning). These systems are significantly different from traditional clusters. For example, cloud systems employ virtualization, which introduces an extra layer between applications and hardware; edge systems use low-end processors, so both performance and energy are equality important; specialized accelerators are tailored for a small set of applications, which do not have enough hardware support for existing tools. Given these challenges, we will explore novel techniques that are suitable for the tools to be deployed in these emerging architectures.