Introduction

Tremendous gains in software efficiency have been achieved over the previous decades. Software has also had significant increases in complexity, which often comes from increasing layers of abstraction. Abstraction has been a key part of improving programmer productivity and enabling these ever more feature-rich (i.e., complex) programs. Yet the gains in software efficiency are diminishing. Power and energy improvements are holding against rising usage of computing resources, even while power and energy constraints are forcing increasing parallelism complicating software development.

More can be done; I have found that careful circumvention of abstractions can provide significant performance gains, which I first observed in industry work and have focused my research on exploiting this opportunity. For example, parallel programs need to use synchronization mechanisms. With enough trouble ensuring that there are no correctness issues (data races, deadlocks, etc), programmers do not have the time to select the most appropriate synchronization implementation, and instead they often use a generic mutex that works equally well (or poorly) in all cases. Programmers could also restructure data layouts to improve cache usage and avoid false sharing, yet this often concedes readability and extensibility. In these cases, I have worked with programmers to take the knowledge of their code and knowledge of the lower abstractions to find more efficient solutions.

Present Research

Manual collaboration between the programmer and performance expert is time consuming and error prone. Programmers would regularly remark that it would be nice if the compiler could do this for him or her. My early research focused on solving this class of problems. For instance, I helped develop Brainy to solve the following problem [JUNG11]: vectors, lists, and deques all have similar features, so programmers must reason very early in development about the particular needs of their implementation. Brainy solves this problem by instrumenting a program, so when it is run on a representative set of inputs; the program outputs its usage of the data structures. Brainy will then select the best implementation of the data structures given their usage, which provides the programmer an abstraction of an abstract container class.

Brainy and many other tools each rely on customized instrumentation or possibly use a framework like Pin to support collecting the required data; however, this process requires considerable effort to develop new tools and analyses for programs. Into this space, we devised a novel representation of programs that would capture the data required by the prior tools, a task graph. Contemporary task graph and task graph-like work has focused on a simplified representation of the programs; this representation is useful for runtime scheduling of programs, yet the original representation was considerably richer and has been extended by us. Our representation merges the task information of how software contexts synchronize and schedule with the architectural information of the control flow and data accesses. Furthermore, this representation can be collected efficiently by the Contech framework for a diversity of parallel programs across many dimensions, including language (C, C++, Fortran), parallel library (pthreads, OpenMP), and architecture (x86, ARM).

We first demonstrated that Contech’s task graph can support a variety of existing analyses, including branch prediction, cache simulation, coherence modeling, and data race detection. Using a Contech task graph, tools can analyze programs to find characteristics, characteristics such as the usage of data struc-
tures or the optimal synchronization implementation. Current research is exploring new characteristics that enable identifying patterns and categorizing programs based on their high level behavior.

**Future Research**

These approaches are constrained by the capability of computers to detect the characteristics of interest. Programmers could participate in this process by providing additional “insights” into the program as written. Insights enable crossing between different abstraction layers with information valuable to other components. While to a limited extent these exist [POOVEY11], only through increasing the ability for programmers to influence the circumvention of abstractions can new program efficiencies be found.

My present research into the task graph representation and its analysis has provided opportunities to mentor undergraduate as well as junior graduate students. Through these experiences I have been gaining insight into how to individually guide and motivate students, as well as the quantity of work achievable. The following are several projects on which I would like to advise students, either undergraduates or masters. First, in collecting the task graph requires high data collection rates (2+ GB/s), which would ideally be compressed before writing to disk; however, most algorithms are computationally expensive for compression and inexpensive for decompression. I am interested in investigating streaming compression approaches that might address these short comings. Second, the task graph representation is similar to traces collected for architectural simulation. We have simulated parts of the architecture, but would like to use the task graph to drive an existing simulator such as Sniper, Gem5, or Manifold. Third, the task graph representation is a rich collection of data; however, there is no current ability to directly view the data in a human friendly manner. Developing a way to visualize and manipulate the data would be a valuable addition to the framework.

**Education Research**

In working with students, I also have an interest in Computer Science Education research. This work informs my teaching, and follows two themes: improving student interest and learning through active-learning and improving student preparation for their careers through group and collaborative work. Most Computer Science Education research has focused on introductory courses, an important need that not only improves the quality of CS majors but the quality and skills of non-majors. This research has proposed a variety of active-learning techniques, yet there is only limited work to extend the usage of these techniques to upper-level courses [DONOHUE09]. By expanding the usage of these techniques to later courses, the quality of the students’ education can be further improved, yet the techniques may not be directly applicable. This past summer, in teaching an upper-level course, I used several active-learning techniques in the classroom and regularly collected feedback from the students. This feedback showed that these students also find the techniques valuable, yet have different expectations from the usage. For example, students are favorable to group activities, but would prefer without requiring specific roles to be taken by the students. I plan on continuing this research into the use of active-learning techniques in the upper-level courses and how the expectations of this classroom environment contrast with the introductory courses.
In industry, one (almost) always works in a group, yet the complement is true in school. Students would be well served by the experience of working in groups, as part of preparing for their career [RADERMACHER14]. Group projects are rarely used due to student and instructor concerns about fairness in distribution of group work and establishing an equitable grading scheme. These concerns are not present in industry, as there is a structure of management to support evaluating employees, yet this structure is not scalable in the classroom setting. Furthermore in school, a student’s time with the instructor is significantly curtailed, which limits the instructor’s knowledge about the individual’s performance as well as the ability to intervene in difficult situations. Most academic projects are limited in scope such that there is difficulty finding discrete, equally-sized work. Without this division, student assessment is for the group rather than individual contributions. Can industry management techniques assist in individual assessment? Is it possible to enlist students in upper-level courses (particularly software engineering) to serve as “managers” for other course projects? Finding an approach to group projects that properly assesses each student and addresses student concerns about group contributions and dynamics, should be an important part of the Computer Science curriculum.

References


