Accepted in the Computational Science and Engineering Software Sustainability and Productivity (CSESSP) Challenges workshop, Rockville MD Oct 2015

Opportunities in Computational Science to Advance Software Engineering

Don Batory
University of Texas at Austin
batory@cs.utexas.edu

Bryan Marker
University of Texas at Austin
marker@cs.utexas.edu

Robert van de Geijn
University of Texas at Austin
rvdg@cs.utexas.edu

1. Introduction

Today's *software engineering (SE)* research develops tools and techniques to help masses of programmers to write and analyze their code. The emphasis of today's SE research is to be domain-independent, such as the following representative titles of papers presented in the 2015 *International Conference on Software Engineering (ICSE)*:

- Efficient Scalable Verification of LTL Specifications
- Safe Memory-Leak Fixing for C Programs
- Automated Modularization of GUI Test Cases
- When and Why Your Code Starts to Smell Bad

Software development in *Computational Science and Engineering (CSE)* is in a different universe. CSE is mostly mathematical computations. Tools are needed to help the few experts who write scientific libraries; these libraries are used by the CSE masses.

CSE software is broadly believed to be different from "normal" software. First, it is mathematical – there are precise (if not formal) specifications for code to be developed. Relationships among program elements are defined by mathematics – few domains in SE have such rich underpinnings. Our guess is that typical SE domains ***do*** have some of this richness, but finding such relationships is hardly mainstream SE research and certainly not practiced by or taught to typical programmers. In short, contemporary SE education and interests are misaligned for long-term CSE needs.

Second, CSE software tends to be algorithm-centric, not behavior-centric. Early SE researchers (Dijkstra, Hoare) focused on disciplined development of *algorithms*. Today's systems stress the coordination of agents (classes, subsystems) to produce particular behaviors; algorithms tend to be a tiny part of a system's overall design.

Consequently, few papers relevant to CSE software technology find their way into major SE conferences. Simply put, CSE is not on the cutting-edge or horizon of today's SE researchers. Although a discouraging but realistic observation, we see it as an opportunity in CSE to advance SE in a general way.

2. What We Think is Needed

Franchetti recently made the observation: "Just because you can write a program doesn't mean that you *understand it*". Here is our take on his statement: program development is an activity *or function or generator* $\mathcal{D}(\ldots)$ whose parameters are the particular set of needs, platform, architecture, and environment for a given task. These parameters are filled in, and $\mathcal{D}(\ldots)$ is given to a team of highly-paid experts to build *or evaluate*. Sometime later, a program $\mathtt{P} = \mathcal{D}(\ldots)$ is produced. Modify a key argument, say the architecture or language, and the task to build program $\mathtt{P}' = \mathcal{D}(\ldots')$ starts anew. Writing $\mathtt{P}'$ may not start from scratch, but it might. CSE libraries are notoriously fragile: they are highly-optimized for a particular platform or hardware architecture. Changing hardware is not a simple port: optimized algorithms that made the library efficient for architecture Δ may be quite different for Δ'.

In a panel at the 2015 Modularity Conference, it was stated that open source software repositories can become a gold mine for SE researchers to data-mine important facts to improve software design. We strongly disagreed. The most important facts in design are abstractions and the (formal) relationships among its elements: the derivation of a code base, the key decisions that were made in its development, and the alternative decisions that were discarded: all are absolutely fundamental to changing a program $\mathtt{P}$ to $\mathtt{P}'$ [1].

None of this information is present in a repository. Where is it then? It is, maybe, in the memory of programmers or it must be reconstructed. Backtracking to a certain point that is common to $\mathtt{P}$ and $\mathtt{P}'$ (so that these decisions and their code need not be revisited) is at best crudely done or impractical today. Until these "meta" decisions are captured and leveraged in a computer processable form, creating a sustainable software development process for CSE will be difficult. It is also clear that such "meta" decisions are ***not*** widely appreciated by today's SE researchers [2].

3. What We have Done

We have addressed a modest, but important, segment of CSE software development: *Dense Linear Algebra (DLA)* and Dense Tensor libraries for distributed-memory computers.

Our approach captures and encodes meta-decisions of DLA and tensor software development as graph rewrite rules [1]. Given a simple abstract dataflow graph of a computation in terms of standard CSE user-callable primitives, we apply rewrites to transform an input graph into a complex dataflow graph of low-level DLA, tensor, and communication software primitives. The transformations that we perform are automated and are isomorphic to decisions made by experts.

A simple concrete example from DLA is Hermitian matrix multiplication. It is a user-callable primitive in the DLA universe. Its dataflow graph $\mathcal{G}$ is elementary: given three matrices as input, `Hemm` computes a sum of their multiplication:

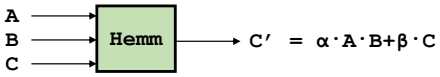

Figure 1. An Abstract Hermitian Matrix Multiplication Dataflow Graph.

Our tool, *DxTer*, with a rule base transforms this elementary graph into a complex dataflow graph $\mathcal{G}'$ containing only low-level primitive operations whose architecture-specific implementations are given to us. We can map this computed graph to code yielding a high-performance implementation of the `Hemm` operation on a particular platform/architecture.

DxTer captures insights of domain-experts by using cost-functions to estimate the efficiency of architecture-primitive operations. Given $\mathcal{G}'$, we know how to estimate its efficiency knowing the efficiency and sequence of primitives it calls.

DxTer goes further: there are colossal numbers of complex dataflow graphs to which $\mathcal{G}$ can be mapped. In seconds, DxTer quickly finds the graph $\mathcal{G}'$ that is provably optimal [5]. $\mathcal{G}'$ represents an architecture-optimized implementation of $\mathcal{G}$.

A fundamental property of our approach is *correct-by-construction (CxC)* [3]: if the initial graph is correct and each rewrite is correct, the result is correct. This means that we can build each graph in a derivation and verify/test that it is correct. We return to this property in the last section.

The input graphs to DxTer can be nontrivial. The *Coupled Cluster Single Double (CCSD)* is a commonly used method in quantum computational chemistry [4] that strikes a balance between communication cost and accuracy. It is a numerical, iterative method utilizing a set of equations to give an accurate reproduction of experimental results on electron correlation for molecules. We partition CCSD's specification into 11 dataflow graphs containing 2-15 nodes each. DxTer searches a space of (10^{16}) solution graphs of hundreds of nodes and does so in seconds to find a solution that is 40% faster and can handle problems 50% larger than an existing high-performance tensor library [6]. Figure 2 shows the performance of DxTer-generated code for CCSD on a BlueGene/Q architecture [5].

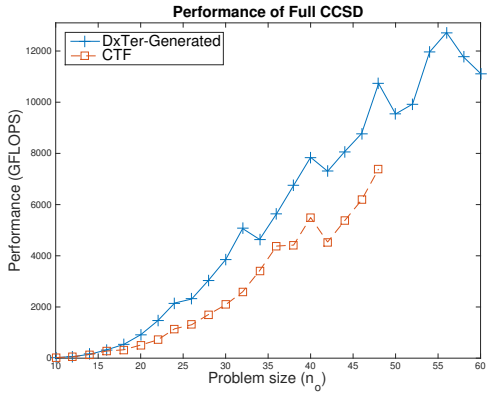

Figure 2. Performance of a single iteration of CCSD on 4,096 cores with one-quarter of peak performance at the top.

4. Closing Thoughts

The libraries that we have targeted are a small, but important, sample of the CSE software universe. We achieved sustainability in our CSE software development process by mechanizing its "software development function" $\mathcal{D}(\ldots)$. By altering parameters to $\mathcal{D}$ we not only leverage critical meta-knowledge to produce new libraries targeted to different architectures, but we do so faster, cheaper, and better via automation than can be performed manually. We therefore scale critical CSE domain-expertise from a few extraordinary individuals to CSE masses. Our work takes effort, not something unreasonable but comparable to if not less than what is done today manually for just one $\mathcal{D}(\ldots)$ instantiation.

CxC used to be a Grand Challenge in SE [3]; SE researchers gave up because it was too hard. They didn't have the right combination of ideas and examples to convince others. We have both now. We teach these ideas to our undergraduates [1], as they are not specific to CSE. With further work on CSE applications, we will have more evidence to argue that automation should be main-stream SE paradigm for a sustainable software development process.

Acknowledgements. We gratefully acknowledge support for this work by NSF grant CCF-1212683.

References

[1] D. Batory, R. Goncalves, B. Marker, and J. Siegmund. Dark knowledge and graph grammars in automated software design. In *Software Language Engineering (SLE)*. 2013.

[2] I. D. Baxter. Design Maintenance Systems. *CACM*, April 1992.

[3] C. Green et al. Report on a knowledge-based software assistant. *Kestrel Institute Technical Report KES.U.83.2*, 1983.

[4] T. J. Lee and J. E. Rice. An efficient closed-shell singles and doubles coupled-cluster method. *Chemical physics letters*, 150(6):406–415, 1988.

[5] B. Marker et al. Dxter: An extensible tool for optimal dataflow program generation. Technical Report TR-15-03, Dept of Comp. Sci. at the U. of Texas at Austin, 2015.

[6] E. Solomonik et al. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, Dec 2014.