Evaluating Research on Software Design and Productivity

William Pugh
Univ. of Maryland
pugh@cs.umd.edu

October 31, 2001

Abstract

A key barrier to achieving the necessary breakthroughs in software design and productivity is that, as a community, we do not have an adequate understanding and consensus on how to evaluate research on the topic. This has hindered publication and funding of research in the area, pushing researchers to explore other topics.

Software design and productivity involves, at its core, human creativity and understanding. Rigorous experimental studies of software design and productivity are therefore difficult, expensive and rare. Requiring them as a prerequisite for research in the field would constrict the field in ways that would not serve our interests. So we need to develop benchmarks, challenge problems, frameworks and (partial) consensus to allow evaluation of useful research on software design and productivity without requiring prohibitive experiments involving programmers.

Introduction

A number of different forums and committees have recognized the important of research in software design and productivity to our national goals. However, when you look at the work that is being published in the most prestigious conferences on programming languages, you see very little work on that topic. People have long lamented the fact that the ACM SIGPLAN conference on *Programming Language Design and Implementation* contains very few publications on programming language *design*, and an abundance of papers on new language implementation techniques that improve program execution speed by 1-4%.

A key problem is that while we have a good understanding of how to evaluate papers that describe techniques to improve program execution speed, it is much more difficult to evaluate papers that aim to improve software design and productivity. Rigorous experiments with programmers are possible, but very difficult and expensive to perform; requiring them as part of all research in the field would create a prohibitive barrier.

Much of the important research in this field are likely to depend upon techniques that are not both sound and complete, but instead act as heuristics to identify possible problems in software design. Thus, we can't rely solely on a technical analysis of the formal aspects of research to guide our evaluation.

We need, as a community, to closely examine and try to come to some agreements on additional ways to evaluate research in software design and productivity. Some of this may involve a continuing effort in developing benchmarks and challenge problems. Other parts of this process will just depend on arriving at a consensus as a community about how to evaluate research on software design and productivity.

There are many different facets to this issue; in this white paper, I will discuss some of the issues I've encountered as I've tried to move from research on programming language implementation and optimization to software productivity.

Code Optimization Research

Given the fact that new compiler optimization techniques typically yield much smaller benefits than improvements in hardware performance each year, it is reasonable to question the benefit of research in code optimization. However, I think that code optimization research does have important role to play in improving software design and productivity.

Human nature being what it is, it is difficult to convince developers to use a higher level programming language with better abstractions and more safety checks if the higher level programming language provides substantially slower performance than low level programming (e.g., C programming). To get developers to switch from C to languages such as Java, you must convince them that the performance penalty for array bound checks, garbage collection and virtual method invocation will not be prohibitive.

There has been substantially research in this direction, and JVM's are now very competitive with C compilers. However, it is still very difficult to evaluate this aspect/advantage of research on code optimization. Several of the Java benchmarks have been converted from C benchmarks, and benchmarks often embody horrible designs and coding practices. Often, the codes have been hand-optimized to the point of being unmaintainable.

Getting good performance on any particular benchmark often depends on having the compiler reverse some particular bad design choice or understand some horribly complicated aspect of the code that also makes the program incomprehensible to those who have to maintain it. Many benchmarks wind up being microbenchmarks, in that getting good performance depends utterly on getting one particular detail "right".

As a case study, the Java Spec DB benchmark spends much of its time manipulating Vectors. The Vector class is designed to be thread safe, so a Vector can be concurrently accessed by multiple threads. However, the DB benchmark is single threaded, so the overhead of the synchronization is useless. There have been a number of papers on removing useless synchronizations from Java programs [ACS99, BH99, Bla99], and doing so successfully for the DB benchmark can provide a 15% improvement in execution time. However, a simple switch from Vectors to ArrayLists (introduced in Java 1.2) provides an equivalent improvement. If you actually step back and see why the DB benchmark is manipulating Vectors, you see that almost all of the time in the benchmark is spent within a hand-coded Shell sort of elements within Vectors. Replacing the 20 lines of Shell sort code with a call to the built-in merge sort reduces the execution time by 50%; using the built in Shell sort along with switching to ArrayList and a few other small changes gives an overall reduction of 65%.

There are several things we need to do to allow compiler optimization research to improve software productivity.

- We need benchmarks that reflect the way we aspire to be able to program; benchmarks that are written in a clear, modular and highly maintainable style, with a process for improving/maintaining the benchmarks as problems are discovered with it.

- We need to surmount our allergies to having people in the loop. If we care about the performance of an application, it is likely (and desirable) that we have a programmer with some understanding of the code and an interest in improving the code. A tool that tells programmers how to annotate/modify their program to get a 50% reduction in execution time is more valuable than a new optimization that gets a 15% reduction automatically, particularly if that annotation/modification also has the effect of making the program easier to understand and maintain.

  We need techniques for evaluating code optimization research that doesn't treat benchmarks codes as sacred texts but allows the modeling of programmers interested in improving the codes.

- We need to figure out some approach to performance portability/predictability. I've seen a number of cases, in both C++ and Java, of techniques for modularizing a design that work fine on some platforms, but produce a 2-10x slowdown on other platforms.

- We need to have methods/consensus for evaluating research that propose new high-level, perhaps domain-specific, abstractions/languages and techniques for generating efficient code from them. The difficulty here is how to evaluate whether the proposed new abstraction/language does offer an advantage in software productivity, since the generated code may not be as efficient as carefully hand-tuned C code.

Neither sound nor complete

An alternative approach to experimental evaluation is to judge research based on theoretical contribution of the research. For example, a sound and complete decision procedure for array aliasing might not need much experimental evaluation of the effectiveness of the technique.

However, sound and complete techniques rarely are in practice. Typically, they apply only to a restricted domain (e.g., an assumption that integer values never exceed 32 bit representations). Very few of the published techniques for analyzing Java code adequately consider the issues of native code, reflection, or multithreaded programs with data races. Some papers on program analysis for software engineering pay little attention to whether the results obtained from the analysis are actually useful for software engineering.

More importantly, we don't want to restrict our tools to ones that are sound and complete. Automatic understanding of what software does is inherently undecidable. But we need to avail ourselves of tools that say "I think there might be a problem here". Less anthropomorphically, we need to consider tools that generate both false positives and false negatives. For such tools, how do we evaluate the false positive and false negative rate? Can we actually measure the false negative rate, since that would requiring knowing all of the errors that we would like to detect?

Some quick examples of tools that might be useful and yet hard to evaluate.

- Quadratic sort detector: we might develop a tool that looked for code that might be implementing a quadratic sorting algorithm. There are numerous stories of code containing quadratic sorting algorithms that didn't become significant until sometime much later when the code was applied to lists/arrays much larger than previously.

  Such a tools couldn't be complete, and almost certainly couldn't automatically transform the code to use an efficient sorting algorithm. How would we evaluate the usefulness of such a tool?

- Equals idiom checker: in Java, there are a number of subtle aspects to implementing the equals function. One is an invariant that two objects that compare as equal must have equal hashCodes. However, I wrote a tool that found a number of violations of that rule in Sun's Java libraries.

  In addition, one must be careful, in class A, not to define the equals function as taking an A reference as an argument. If you do so, the method will not override the definition of equals(Object) in class Object, and not provide the desired semantics. Another tool also found a number of violations of this property in Sun's Java libraries.

One thing that would be useful is the programming language equivalent of grammar checkers; tools that check code against certain known patterns and antipatterns, pointing out possible errors. While there has been some work in this area [BPS00, ECH+01, CYC+01], it is an area that needs a lot more attention.

More generally, showing that some program analysis would actually be a useful tool for software engineering is a difficult task. We must find ways to allow and encourage this.

Software Archaeology

One key resource we need to exploit is the wide variety of open source software efforts. Having a full CVS history for major software project, as well as a correlated bug database, would allow many useful experiments to be performed.

While the CVS histories for projects such as Apache are available, the bug databases are less useful. For example, searching for "memory leak" returns all bugs that have "memory leak" anywhere in the email associated with the bug, including those bugs where someone notes that the bug "is definitely not a memory leak".

We need to think about the historical records of large open source software efforts as being the genome for software design and productivity, and provide the same level of community support for indexing and accessing those records as in the genome project. There has been some work studying Linux [ECH+01, CYC+01], Mozilla and Apache [BPS00], and the AT&T 5ESS switching system [KPV96, BKPS97]. However, none of these efforts have attempted to build a common and communal infrastructure for accessing that data.

References

[ACS99] Jonathan Aldrich, Craig Chambers, and Emir Gun Sirer. Eliminating unnecessary synchronization from java programs. In *OOPSLA poster session*, October 1999.

[BH99] Jeff Bogda and Urs Hoelzle. Removing unnecessary synchronization in java. In *OOPSLA*, October 1999.

[BKPS97] Thomas Ball, Jung-Min Kim, Adam Porter, and Harvey Siy. If your version control system could talk. In *Proceedings of the Workshop on Process Modeling and Empirical Studies of Software Evolution*, May 1997.

[Bla99] Bruno Blanchet. Escape analysis for object oriented languages; application to Java. In *OOPSLA*, October 1999.

[BPS00] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience*, 30(7), 2000.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, , and Dawson Engler. An empirical study of operating systems errors. In *Symposium on Operating Systems Principles*, 2001.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, , and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, 2001.

[KPV96] A. Karr, A. Porter, and L. Votta. An empirical exploration of code evolution. In *Proceedings of International Workshop on Empirical Studies of Software Maintenance*, November 1996.