Software, Performance, or Engineering?

Daniel A. Menascé, ACM Fellow
Department of Computer Science, MS 4A5
George Mason University
Fairfax, VA 22030-4444
menasce@cs.gmu.edu

ABSTRACT

This paper discusses why Software Engineering (SE) methods often fail to produce software systems that meet their performance requirements. Five issues are raised: lack of required scientific principles and models in SE, lack of education in performance, IT workforce shortage, single-user and small database mindsets.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; D.2 [**Software**]: Software Engineering

Keywords

Software Performance Engineering, Software Engineering, Performance models.

1. INTRODUCTION

Most software systems we care about are complex to design and to build. Their requirements are divided into functional and non-functional requirements. The so-called non-functional requirements include properties that systems are supposed to exhibit with respect to security, availability, reliability, and performance (e.g., response time, throughput, fraction of rejected requests).

The need to build software systems that function correctly, adhere to non-functional requirements, and are cost-effective, gave rise to a series of techniques and methods, patterned on what engineers do, when they design systems. This discipline, called *Software Engineering (SE)*, has been around for over thirty five years and has had much success in developing methods for programming in the small and in the large [7]. Most of the success of SE has been in taming the complexity of the software development process through the use of methods and tools to develop and manage designs, requirements, tests cases, configurations, versions, and evolution.

WOSP '02, July 24-26, 2002 Rome, Italy

According to Carnegie Mellon's Software Engineering Institute (SEI), engineering is the systematic application of scientific knowledge in creating and building cost-effective solutions to practical problems in the service of mankind and software engineering is that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems [2]. So, if SE is indeed a form of engineering, it should apply scientific principles to the design of software systems in such a way that **all** of its requirements—functional and non-functional—are met.

Unfortunately, with the exception of real-time applications, performance requirements are rarely taken into account at the design stage. This would never be allowed to happen in any other form of engineering (e.g., civil, mechanical, aeronautical, etc.), which we refer to here as Conventional Engineering (CE). Would a mechanical engineer design an engine that is supposed to reach 4,000 RPM to find out when the engine is built and tested that it does not go over 1,500 RPM? Or would a civil engineer design a bridge supposed to withstand the load of sixty 3-ton vehicles at 60 mph only to find out that it collapses when fifteen vehicles go over it? Clearly not. The reason why one does not see this enormous mismatch between requirements and results is that performance, or efficiency, is an integral part of the design process in any CE modality. In other words, the mechanical engineer does not just design an engine that spins, but one that can reach a certain speed with a given fuel consumption.

The fact that performance requirements are called non-functional requirements in SE is a symptom of this problem. How can a software system function properly if some of its requirements (e.g., performance requirements) are not met? In other words, all requirements should be functional. Otherwise, they should not be requirements.

The difference in approach between CE design and SE design is illustrated in Fig. 1. Conventional engineers (bottom part of Fig. 1) design systems with workload and environment considerations in mind. For example a civil engineer takes into account the bridge workload (i.e., how many cars will cross the bridge) as well as assumptions about the environment in which the bridge will have to operate (i.e., high wind forces at the top section of the bridge). Most software engineers (top part of Fig. 1) design systems that meet some of the requirements (usually the functional ones) and then try to match the resulting system to the workload and

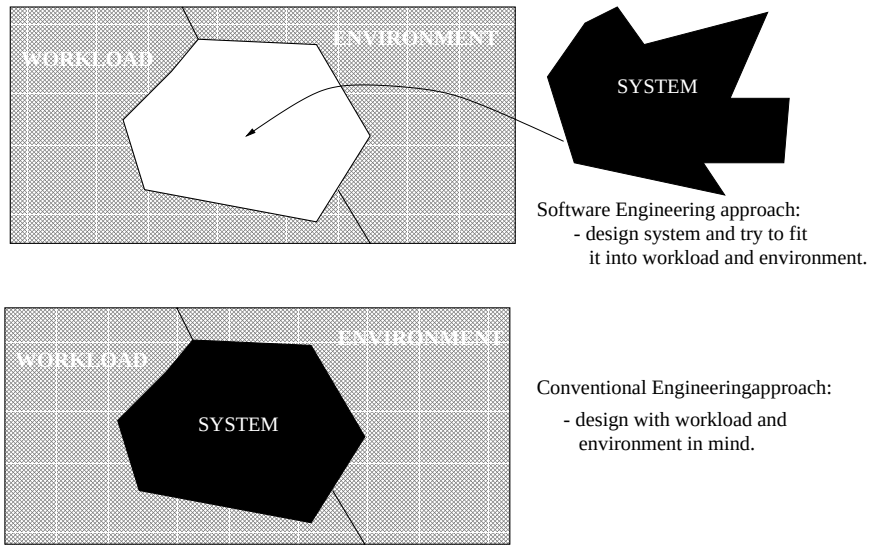

Figure 1: Conventional engineering approach vs. software engineering approach.

environment in which they are supposed to operate.

2. SOFTWARE PERFORMANCE ENGINEERING

Thirteen years after the term software engineering was introduced [7], Connie Smith coined the term *Software Performance Engineering (SPE)* in her seminal paper published in 1981 [9]. That paper brought attention to the fact that software development was carried out with the "fix-it-later" attitude when it came to performance. In other words, performance was never a design consideration, but an afterthought. Since 1981, many people contributed relevant research ideas and some software systems were developed, in industry and in academe, to support SPE processes.

A question that comes to mind is whether the term Performance in Software Performance Engineering is redundant? If SE is engineering, the design of software systems using engineering methods should produce efficient systems. Would it make sense to talk about Efficient Mechanical Engineering? Clearly not. Mechanical engineers strive to design efficient engines and mechanisms. For them, it is not enough to just do it; it has to be done efficiently.

The reason why Performance in SPE is not yet redundant is that twenty years after Smith's introduction of the concepts behind SPE, SPE has not been incorporated into the practices of Software Engineering. Therefore, it is still important to talk about SPE until the P of SPE becomes redundant, i.e., until it really blends into SE.

3. WHERE IS THE P IN SE?

This section presents some issues that may foster discussions in both the performance and software engineering communities regarding the reasons why performance does not receive proper attention during software design.

1. *Lack of scientific principles and models.* Conventional engineers must use scientific principles and models based on mathematics, physics, and computational science, to support their design processes. This allows CE to model the effects of the workload and the environment on the systems being designed. Software engineers do not need to rely on formal and quantitative models as part of the software development life cycle. In other words, software designers and developers can design and write code without using any formalism.

   There have been many developments in terms of formal models to support the software life cycle. Most of this work is centered around methodologies to manage the complexity of the process of software development, testing, maintenance, and evolution. Some of these developments have made it to tools that gained some widespread level of acceptance. However, there are no universally agreed upon formalisms and quantitative models that support the core of SE.

   The SE community has devoted most of its energies to the development of formalisms and methods that support the functional requirements aspects of SE. In fact, over 80% of the papers published in the IEEE Transactions on Software Engineering (TSE) since 1989 fall into this category. The remaining papers are performance-related. The vast majority of these deal with the performance of algorithms (e.g., concurrency control methods in databases systems, resource schedulers in operating systems and distributed systems), performance of multiprocessor systems, and even papers on performance evaluation techniques, including various forms of Stochastic Petri Nets. The percentage of papers that directly address performance problems in software systems is much smaller.

   The performance community, for the most part, has addressed issues of system performance from a resource demand point of view. For example, the input parameters of queuing network (QN) models fall into two categories: workload intensity and resource demands for each transaction type at each physical resource. The resource demand is a function of the intrinsic per-

formance characteristics of the physical resource and of the application itself. If the application is not explicitly modeled, performance models cannot be easily used to evaluate the performance impact of changes in the structure of the application. Traditional QN models have been extended by layered QNs to allow for explicit software modeling. These models are not yet as widely-known and -adopted as conventional QNs, which gained popularity due to their simplicity.

The Workshop on Software and Performance (WOSP) is an important venue to bridge the gap between software engineering people and performance people and between industry and academe [1, 8, 10].

2. *Education.* Graduates of computer science and related engineering programs are often unprepared to address the software engineering problems faced by industry. majority of undergraduate computer science and CS-related curricula do not include any required course in computer system performance evaluation and offer only minimal performance-related hours, generally in operating systems and computer network courses. The Joint IEEE Computer Society/ACM Task Force on the "Model Curricula for Computing" (CC) [5] published very recently its draft CS undergraduate curriculum. The CC2001 report divides CS into 14 areas, of which Software Engineering is one of them. The area of SE is divided into eight core areas—software design, using APIs, software tools and environments, software processes, software requirements and specifications, software validation, software evolution, and software project management—and four elective areas—component-based computing, formal methods, software reliability, and specialized systems development. No explicit mention to performance appears in the detailed description of any of the twelve subareas of SE.

   Why is computer performance not included in the list of disciplines taught to future software engineers? There are many possible reasons including 1)lack of training in performance issues by faculty who teach SE; 2) lack of universally agreed upon methods and models to be used by software engineers to address performance issues—while conventional engineers can rely on the well-established disciplines of physics and mathematics, there is no equivalent counterpart in SE; 3)faculty resistance to change, and 4)limits on the total number of credits in existing curricula. Every time something is added, something has to go.

   In order to gauge how students assess the importance of software in the performance of a computer system, I prepared a simple 5-question test that was answered by 59 students at George Mason University. Nineteen were senior students in our BS in CS program and the remaining 39 were graduate students in the MS in CS and MS in SE programs. Forty four percent of the graduate students took at least one SE course and 84% of the undergraduate ones took an SE course.

   Table 1 summarizes the results of this test. The first four questions were aimed at assessing the students understanding of very basic performance concepts: 1) Define Response Time (RTDef), 2) What units are used to indicate response time? (RTU), 3) Define throughput (XDef), and 4) What units are used to indicate throughput? (XU). The last question (RT Factors) asked students to identify possible factors that could contribute to the time taken by a Web search engine to return a reply to a browser.

   Most students were able to answer correctly questions 1 and 2 and a smaller percentage were able to correctly answer the two throughput questions. A very large percentage of students were able to identify factors that could affect the response time of a search engine. However, very few of them included software-related issues in their lists (see last column of Table 1). This survey is, admittedly, not comprehensive enough to allow one to take very general conclusions. But, I have strong reasons to believe that similar results would be obtained if this test were applied to other groups of students.

3. *IT Workforce.* U.S. Information Technology (IT) workforce estimates range from 2 to 10 million depending on the source and definition of IT worker. A more accurate estimate for "core IT workforce" that only includes computer engineers, computer system analysts and scientists, computer programmers, and computer science teachers places the number at 2.5 million people in 1999 in the U.S. [6]. The U.S. Bureau of Labor Statistics (BLS) projects that, between 1998 and 2008, IT jobs will grow slightly over 7 percent a year, far quicker than the 1.4 percent average across all jobs. This represents a need to graduate 175,000 people a year in IT. This number far exceeds the estimated 42,000 bachelor's degrees in CS and engineering awarded in the US and Canada in 2,000.

   Using BLS data, one analyst examined the educational credentials among people in four important IT professions (computer scientists, computer engineers, system analysts, and computer programmers) in 1998 [4]. The results can be summarized as follows: two thirds hold a BS or higher degree, one third (mostly programmers) had a two-year degree or only a high-school diploma, less than 50% had a bachelor's or other degree with a major or minor in CS or CS-related discipline! So, many individuals without formal training are employed in IT fields and learn on the job. Therefore, it is not surprising that many of the software systems built today exhibit performance problems.

4. *Single-user mindset.*

   Most system designers and programmers develop systems with the "single-user" mindset. This means that software developers do not consider that the code they are writing will, in most cases, be instantiated by many concurrent requests. Concurrency generates contention for physical resources (e.g., processors, storage devices, and networks) and software resources (e.g., database locks, critical sections, and software threads). Software contention can be a significant portion of the total response time as illustrated in Fig. 2, which shows the percentage of the total response time spent by a request waiting to enter a critical section as a function of the number of concurrent processes. The graph assumes that each request spends 0.2 sec at the CPU executing non-critical section code and 0.1 sec executing

Level	Number of Students	% Took System Courses	% Took SE Courses	Percent Answered Correctly					
				RTDef	RTU	XDef	XU	RT Factors	Software Factors
Senior BS in CS	19	100%	84%	100%	74%	58%	63%	84%	21%
Graduate	39	95%	44%	87%	82%	74%	54%	87%	5%

Table 1: Test Results.

critical section code. As it can be seen, as the concurrency level increases, contention for software resources dominates the response time of a process.

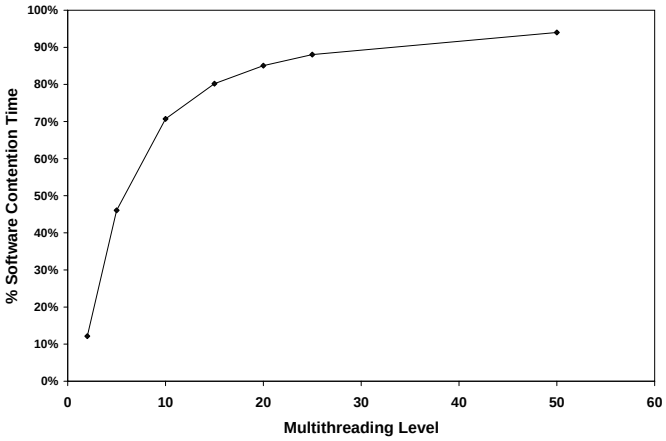

Figure 2: Percent of Software Contention Time vs. Multithreading Level

5. *Small database mindset.* The "small database" mindset means that code that accesses a database is usually written without taking into account the size of the database. The performance of an SQL call on a database with 1,000 rows is certainly different than that on a table with one million rows. One may need to use auxiliary tables and different ways to query a very large database.

4. CONCLUDING REMARKS

This paper raised five possible causes for the fact that software systems rarely meet their performance requirements: 1) lack of scientific models that must be used in software development, 2) no performance education in the vast majority of undergraduate CS curricula, 3) shortage of IT workers leading to poorly trained individuals developing software, 4) the "sngle-user" mindset of many programmers leading to a total neglect of performance problems arising from contention issues, and 5) the "small database" mindset that leads programmers to ignore the effects of queries to very large databases.

Software complexity often leads to inefficiencies. It has been observed that the most important factor in attacking complexity lies in improving the quality of programmers as opposed to the tools and techniques they use [3]. Therefore, the most effective way to generate high-performance software is through properly educating software designers and programmers on performance issues. Also, efficiency is often a matter of good design rather than good coding. Therefore, efficiency must be considered early on in the life cycle [3]. This observation implies that the benefits of being aware about the performance impacts of a software system tend to be higher at the design stage.

Acknowledgements

This work was partially supported by a National Science Foundation grant number EEC-0080379. The author would like to thank Hakan Aydin, Peter Denning, and Larry Kerschberg for applying the test reported here, and Hassan Gomaa for his comments on this paper.

5. REFERENCES

[1] S. Balsamo, P. Inverardi, and B. Selic, Proc. Third ACM Workshop on Software and Performance, Italy, Rome, July 24-27, 2002.

[2] G. Ford, 1990 SEI Report on Undergraduate Engineering Education, Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-90-TR-003.

[3] R. L. Glass, "Frequently Forgotten Fundamental Facts about Software Engineering," IEEE Software, May/June 2001, pp. 110–112.

[4] M. Hilton, "Information Technology Workers in the New Economy," Monthly Labor Review, June 2001, pp. 41–45.

[5] IEEE/ACM, "Model Curricula for Computing," Joint IEEE Computer Society/ACM Task Force, Computer Science Volume, Final Draft, Dec 15, 2001, www.computer.org/education/cc2001

[6] National Research Council, "Building a Workforce for the Information Technology," National Academy Press, Washington, DC, 2001.

[7] P. Naur and B. Randell, (eds.), Software Engineering: report of a conference sponsored by the NATO Science Committee Garmisch, Germany, 7-11 Oct. 1968, 231 pages.

[8] C. U. Smith, P. Clements, and M. Woodside, Proc. First ACM Workshop on Software and Performance, Santa Fe, NM, October 12-16, 1998.

[9] C. U. Smith, "Increasing Information Systems Productivity by Software Performance Engineering," Proc. CMG XII International Conference, December 1981.

[10] M. Woodside, H. Gomaa, and D. A. Menascé, Proc. Second ACM Workshop on Software and Performance, Ottawa, Canada, September 17-20, 2000.