An Environment for Conducting Families of Software Engineering Experiments

LORIN HOCHSTEIN

University of Nebraska

TAIGA NAKAMURA

University of Maryland

FORREST SHULL

Fraunhofer Center Maryland

NICO ZAZWORKA

University of Maryland

VICTOR R. BASILI

University of Maryland, Fraunhofer Center, Maryland

MARVIN V. ZELKOWITZ

University of Maryland, Fraunhofer Center, Maryland

Abstract

The classroom is a valuable resource for conducting software engineering experiments. However, coordinating a family of experiments in classroom environments presents a number of challenges to researchers. Understanding how to run such experiments, developing procedures to collect accurate data, and collecting data that is consistent across multiple studies are major problems.

ADVANCES IN COMPUTERS, VOL. 74
ISSN: 0065-2458/DOI: 10.1016/S0065-2458(08)00605-0

This paper describes an environment, the Experiment Manager that simplifies the process of collecting, managing, and sanitizing data from classroom experiments, while minimizing disruption to natural subject behavior. We have successfully used this environment to study the impact of parallel programming languages in the high-performance computing domain on programmer productivity at multiple universities across the United States.

1. Introduction

Scientific research advances by the creation of new theories and methods, followed by an experimental paradigm to either validate those theories and methods or to offer alternative models, which may be more appropriate and accurate. Computer science is not different from other sciences, and the field has been moving to adopt such experimental approaches [25]. However, in much of computer science, and in software engineering in particular, the experimental model poses difficulties possibly unique among the sciences. Software engineering is concerned about the appropriate models applicable to the development of large software systems. As such it involves the study of numerous programmers and other professionals over long

periods of time. Thus, much of this research involves human behavior and in many ways is similar to research in psychology or the social sciences.

The major experimental approach accepted in scientific research is the replicated study. However, by being expensive to produce, software is not amenable to such studies. While a typical medical clinical trial may involve hundreds of subjects testing a drug or a new treatment, even one duplication of a software development is beyond the resources of most organizations. Although this approach is commonly used in various fields of research involving humans, such as clinical study in medical science, conducting many studies is still difficult and expensive, which is often a major obstacle for good software engineering research.

The problems with empirical studies in software engineering can be classified by two major problems: cost of such studies and accuracy of the data.

1.1 Collecting Accurate Data

1.1.1 Costs of Software Engineering Studies

Because of the cost of developing a piece of software, typically case studies of a single development are monitored, and after many such studies, general trends can be observed. As an example of this, we will look at the NASA Goddard Space Flight Center (GSFC) Software Engineering Laboratory (SEL), which from 1976 to 2001 conducted many such studies [6]. The experiences of the SEL are illustrative of the problems encountered in data collection. The data was collected, beginning in 1976, at GSFC from NASA developers and the main development contractor, Computer Sciences Corporation (CSC). The data was then manually reviewed at GSFC before being sent to the University of Maryland for entry into the project measures database using a UNIX-based Ingres system.

The naïve simplicity in which data was collected broke down by 1978 and a more rigorous set of processes was instituted. This could not be a part-time activity by faculty using undergraduate employees. In addition, the university researchers wanted a considerable amount of data, and soon realized that the GSFC programming staff did not have the time to comply with their requests. They had to compromise on the amount of data desired versus the amount of data that realistically could be collected. Data, which was collected on forms filled out by the programming staff, were shortened to allow for more complete collection.

The data collection process for the 20 projects then under study became more rigorous with this five-step approach:

1. Programmers and managers completed forms.
2. Forms were initially verified at CSC.
3. Forms were encoded for entry at GSFC.

4. Encoded data checked by validation program at GSFC.
5. Encoded data revalidated and entered into database at University (after several years, CSC took over total management of the database).

But, to obtain contractor cooperation, a 10% overhead cost to projects was allocated for data collection and processing activities. Eventually the overhead cost of collecting data was reduced, but the total cost of collecting, processing, and analyzing data continued to remain between 5% and 10%. However, on a $500K project, this still amounted to almost $50K just for data collection – an amount that few organizations are willing to invest. While the SEL believed that the payoff in improved software development justified this cost, it is beyond the scope of this chapter to prove that. Suffice it to say that most organizations consider the additional costs of data collection as unjustified expenses.

1.1.2 Accuracy in Collected Data

Most data collected on software development projects can be generally classified as self-reported data – the technical staff fill out effort reports on hours worked, change reports on defects found and fixed, etc. The care in which such data is reported and collected greatly affects the accuracy of the process. Unfortunately, the process is not very accurate. Self-reported measures can vary over time, due to history or maturation effects [7], and the accuracy of such measures varies across individuals. This is a particular problem when the subjects have more interest in completing the task than complying with the protocols of the study. Basili et al. [2] evaluated Software Science metrics against self-reported effort data collected from GSFC software projects. There was very little correlation between this effort and metrics known to predict effort, and there was concern that poor self-reported data was distorting the results. Perry et al. [18] analyzed previous data from project notebooks and free-form programmer diaries which were originally kept for personal use. They found that the free-form diaries were too inconsistent across subjects and sometimes lacked sufficient resolution.

1.2 Classroom Studies

The overhead in collecting accurate detailed data at the SEL was too high to maintain a data collection process. This same result has been found in other data collection studies. Instead of large replications, running studies in classrooms using students as subjects have become a favorite approach for trying out new development techniques [20]. Even though a conclusion drawn from student subjects cannot always be generalized to other environments, such experiments aid in developing

approaches usable elsewhere. However, conducting larger-scale software engineering research in classroom environments can be quite complex and time consuming without proper tool support. Proper tool support is a requirement if we want to improve on the poor quality of self-reported data.

A single environment (e.g., a single class) is often insufficient for obtaining significant results. Therefore, multiple replications of a given experiment must be carried out by different faculty at different universities in order to provide sufficient data to make conclusions. This means, each experiment must be handled in a similar manner to allow for combining partial results. The complexity of providing consistent data across various experimental protocols has been overwhelming so far.

In this chapter, we describe an environment we are developing to simplify the process of conducting software engineering experiments that involve development effort and workflow and ensure consistency in data collection across experiments in classroom environments. We have used this environment to carry out research to identify the effect of parallel programming languages on novice programmer productivity in the domain of high-performance computing (e.g., MPI [12], OpenMP [11], UPC [8], and Matlab*P [10]). Although there are often issues regarding the external validity of students as subjects, this is not a major concern here since we are explicitly interested in studying student programmers.

This work was carried out in multiple universities across the United States in courses where the professors were not software engineering researchers and were, therefore, not experienced with conducting experiments that involved human subjects.

2. Classroom as Software Engineering Lab

The classroom is an appealing environment for conducting software engineering experiments, for several reasons:

- Most researchers are located at universities. Being close to your subjects is often necessary to obtain accurate results.
- Training can be integrated into the course. No extra effort is then required by the subjects since there is the assumption that the training is a valuable academic addition to the classroom syllabus.
- Required tasks can be integrated into the course.
- All subjects are performing identical programming tasks, which are not generally true in industry. This provides an easy source for replicated experiments.

In addition to the results that are obtained directly by these studies, such experiments are also useful for piloting experimental designs and protocols which can later be applied to industry subjects, an approach which has been used successfully elsewhere (e.g., [3, 4, 5]).

While there are threats to validity of such studies by using students as subjects as proxies for professional programmers (e.g., the student environment may not be representative of the ones faced by professional programmers), there are additional complexities that are specific to research in this type of environment. We encountered each of these issues when conducting research on the effects of parallel programming model on effort in high-performance computing [14]:

1. *Complexity*: Conducting an experiment in a classroom environment is a complex process that requires many different activities (e.g., planning the experimental design, identifying appropriate artifacts and treatments, enrolling students, providing for data collection, checking for process compliance, sanitizing data for privacy, and analyzing data). Each such activity identifies multiple points of failure, thus requiring a large effort to organize and run multiple studies. If the study is done at multiple universities in collaboration with other professors, these professors may have no experience in organizing and conducting such experiments.
2. *Research versus pedagogy*: When the experiment is integrated into a course, the experimentalist must take care to balance research and pedagogy [9]. Studies must have minimal interference with the course. If the students in one class are divided up into treatment groups and the task is part of an assignment, then care must be taken to ensure that the assignment is of equivalent difficulty across groups. Students who consent to participate must not have any advantage or disadvantage over students who do not consent to participate, which limits additional overhead required by the experiment. In fact, each university's Institutional Review Board (IRB), required in all United State universities performing experiments with human subjects, insists that participation (or nonparticipation) must have no effect on the student's grade in the course.
3. *Consistent replication across classes*: To build empirical knowledge with confidence, researchers replicate studies in different environments. If studies are to be replicated in different classes, then care must be taken to ensure that the artifacts and data collection protocols are consistent. This can be quite challenging because professors have their own style of giving assignments. Common projects across multiple locations often differ in crucial ways making meta-analysis of the combined results impossible [16].
4. *Participation overhead for professors*: In our experience, many professors are quite willing to integrate software engineering studies into their classroom environment. However, for professors who are unfamiliar with experimental protocols, the more effort required of them to conduct a study, the less likely it will be a success. In addition, collaborating professors who are not empirical researchers may not have the resources or the inclination to monitor the quality

of captured data to evaluate process conformance. Therefore, empirical researchers must try to minimize any additional effort required to run an empirical study in the course while ensuring that data is being captured correctly.

The required IRB approval, when attempted for the first time, seems like a formidable task. Help in understanding IRB approval would greatly aid the ability of conducting such research experiments.

5. *Participation overhead for students*: An advantage of integrating a study into a classroom environment is that the students are already required to perform the assigned task as part of the course, so the additional effort involved in participating in the study is much lower than if subjects were recruited from elsewhere. However, while the additional overhead is low, it is not zero. The motivation to conform to the data collection process is, in general, much lower than the motivation to perform the task, because process conformance cannot be graded. In addition, the study should not subvert the educational goals of the course. Putting the experiment in the context of the course syllabus is never easy.

   This can be particularly problematic when trying to collect process data from subjects (e.g., effort, activities, and defects), especially for assignments that take several weeks (e.g., we saw a reduction in process conformance over time when subjects had to fill out effort logs over the course of multiple assignments).
6. *Automatic data collection of software process*: To reduce subject overhead and increase data accuracy, it is possible to collect data automatically from the programmer's environment. Capturing data at the right level of granularity is difficult. All user-generated events can be captured (keyboard and mouse events), but this produces an enormous volume of data that may not abstract to useful information. Allowing this raw data to be used can create privacy issues, such as revealing account names, with the ability to then determine how long specific users took to build a product or how many defects they made.

   All development activities taking place within a particular development environment (e.g., Eclipse) simplifies the task of data collection, and tools exist to support such cases (e.g., Marmoset [21]). However, in many domains, development will involve a wide range of tools and possibly even multiple machines. For example, in the domain of high-performance computing, preliminary programs may be compiled on a home PC, final programs are developed on the university multiprocessor, and are ultimately run on remote supercomputers at a distant datacenter. Programmers typically use a wide variety of tools, including editors, compilers, build tools, debuggers, profilers, job submission systems, and even web browsers for viewing documentation.
7. *Data management*: Conducting multiple studies generates an enormous volume of heterogeneous data. Along with automatically collected data and

manually-reported data, additional data includes versions of the programs, pre-and post-questionnaires, and various quality outcome measures (e.g., grades, code performance, and defects). Because of privacy issues, and to conform to IRB regulations, all data must be stored with appropriate access controls, and any exported data must be appropriately sanitized. Managing this data manually is labor-intensive and error-prone, especially when conducting studies at multiple sites.

3. The Experiment Manager Framework

We evolved the Experiment Manager framework (Fig. 1) to mitigate the complexities described in the previous section. The framework is an integrated set of tools to support software engineering experiments in HPC classroom environments. While aspects of the framework have been studied by others, the integration of all features allows for a uniform environment that has been used in over 25 classroom studies over the past 4 years. The framework supports the following.

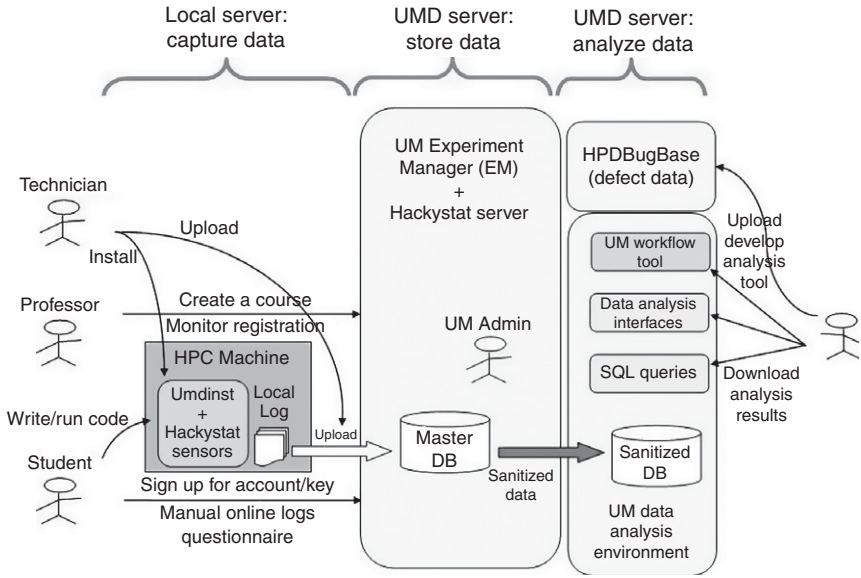

FIG. 1. Experiment Manager structure.

1. *Minimal disruption of the typical programming process*: Study participants solve programming tasks under investigation using their typical work habits, spreading out programming tasks over several days. The only additional activity required is filling out some online forms. Since we do not require them to complete the task in an alien environment or work for a fixed, uninterrupted length of time, we minimize any negative impact on pedagogy or subject overhead.
2. *Consistent instruments and artifacts*: Use of the framework ensures that the same type of data will be collected and the same type of problems will be solved, which increases confidence in meta-analysis across studies at different universities.
3. *Centralized data repository with web interface*: The framework provides a simple, consistent interface to the experimental data for experimentalists, subjects, and collaborating professors. This reduces overhead for all stakeholders and ensures that data is consistently collected across studies.
4. *Sanitization of sensitive data*: The framework provides external researcher with access to the data sets that have been stripped of any information that could identify subjects, to preserve anonymity and comply with the protocols of human subject research as set out by IRBs at American universities.

3.1 Instrumentation Package

Our instrumentation package, called *UMDinst*, supports automatic collection of software process data in a Unix-based, command-line development environment, which is commonly used in high-performance computing. The package is designed to be installed in a master account on the local server and then enabled in the accounts of each subject by executing a set up script, or be installed in the account of individual subjects. The appropriate installation mode depends on the need of a specific experiment and the configuration of the machine to be instrumented. In either case, the package can be used without the intervention of system administrators.

UMDinst package instrument programs that are involved in the software development process by replacing each command that invokes a tool (e.g., compiler) with a script that first collects the desired information and then calls the original tool. It is used for instrumenting compilers, although it is also designed to support job schedulers (common in high-performance computing environments), debuggers, and profilers. For each compile, the following data is captured:

- a time stamp when the command is executed
- contents of the source file that were compiled
- contents of local header files referenced in the source file

- the command used to invoke the compiler
- the return code of the compiler
- the time to compile

The UMDinst package includes Hackystat sensors [15] to instrument supported editors such as Emacs and vi, and to capture shell commands and time stamps. The collected data is used in studies to estimate total effort as well as to infer development activities (e.g., debugging, parallelizing, and tuning). Hackystat is a system developed by Johnson at the University of Hawaii that captures low-level events from a set of instrumented tools. Thus, while UMDinst captures data at the command-line level, Hackystat captures time stamps and events from editors and related tools that have been instrumented. The pair of tools provides a complete history of user interaction in developing a program.

3.1.1 Web Portal

The heart of the Experiment Manager framework is the web portal, which serves as a front-end to the database that contains all of the raw data, along with metadata about individual experiments. Multiple stakeholders use the web interface: experimenters, subjects, and data analysts. For example, experimenters specify treatments (in our case, parallel programming models), assignment problem, participation rate, and grades. They also upload data captured automatically from UMDinst. Subjects fill in questionnaires, and report on process data such as time worked on different activities and defects. Analysts would export data of interest, such as total effort [13] for hypothesis testing, or a stream of time stamped events for workflow modeling.

3.2 Experiment Manager Roles

We have divided the functionality of the Experiment Manager into four roles. For each role, we developed several use cases that describe its functionality, thus simplifying the design of the software.

1. *Technician*: The technician sets up the environment on the local server, usually not at the University of Maryland. This will be someone at a university with access to the machine the students use for the class. Often it is the Teaching Assistant in the course the software will be used in. The tasks for the technician are to install UMDinst so that students can use it. At the end of the semester, the technician also sends the collected data to the University of Maryland server in case it was collected locally.

2. *Professor*: A database provides the professor with sample IRB questionnaires for submittal. This cannot be fully automated since each university has its own guidelines for submitting the IRB approval form. But experience with many universities over the last 4 years allows us to help in answering the most common questions on these forms.

   The instructor first registers each class with the Experiment Manager to set up a classroom experiment. For each such class, the professor can assign several programming projects from our collected database of assignments or assign one of his own. During the semester, the system allows the professor to see if students have completed their assignments, but does not allow access to any of the collected data until the grades for the assignment are completed. In reality, the Teaching Assistant may be the person to actually perform this task, but conceptually is acting in the role of the professor.
3. *Student*: A student who takes part in the experiment provides data on HPC development. This requires the student to:
   1. Register with the Experiment Manager by filling out a background questionnaire on courses taken and experiences in computer science in general and HPC programming in particular. Although this registration process can take up to 15 min, it is required only once during the semester.
   2. Run the script to set up the wrappers for the commands that edit, compile and run programs. Once an assignment is underway, the data collection process is mostly automatic and data is collected mostly painlessly.
4. *Analyst*: An analyst accesses the collected data for evaluating some hypothesis about HPC development. At the present time, the analysis tools are relatively simple. Analysts can see the total effort and defects made by each student and collect workflow data.

Many tools exist in prototypes to support the various types of studies. For example, the HPC community is developing concepts of what productivity means in the HPC environment [26] and we have been looking at developing workflow models (e.g., how much time is spent in various activities, such as developing code, testing, parallelizing the code, and tuning the code for better performance). To support the study of workflows and productivity, we have developed a tool to allow the experimenter to apply various heuristics to the base data to see if we can automatically deduce the developer's programming activity, for example, testing and debugging versus development. This tool takes raw data, collected from online tools such as Hackystat and manual logs generated by students, and we have been developing algorithms for automatically inferring the workflow cycle [23]. Results of this work are described in Section 4. Current activities are looking at extending these tools.

3.3 Data Collection

The actual data collection activity was designed to present minimal complexity to the student (Figs. 2 and 3). Within the Experiment Manager, the student has two options. If data was not collected automatically, the student can enter a set of activities, with the times each activity started and ended (Fig. 3) (e.g., self-reported data, which we discussed earlier to be less reliable). However, the effort tool simplifies the process greatly. If the student clicks to start the tool (small oval near bottom of Fig. 3), then a small window opens on the top left corner of the screen (large oval in upper left in Fig. 3). Each time the student starts a different activity, the student only needs to pull down the menu in the effort tool and set the new activity type (Fig. 2). The time between clicks is recorded as the time of the previous activity. Thus, while the data is not totally automatic, we believe we have minimized the overhead of collecting such data.

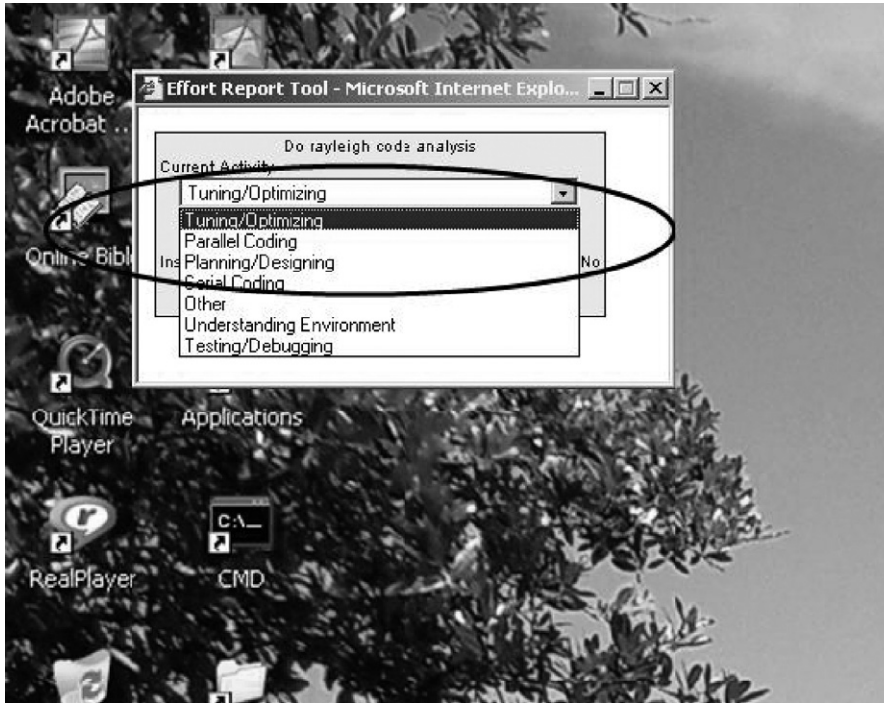

FIG. 2. Effort capture tool.

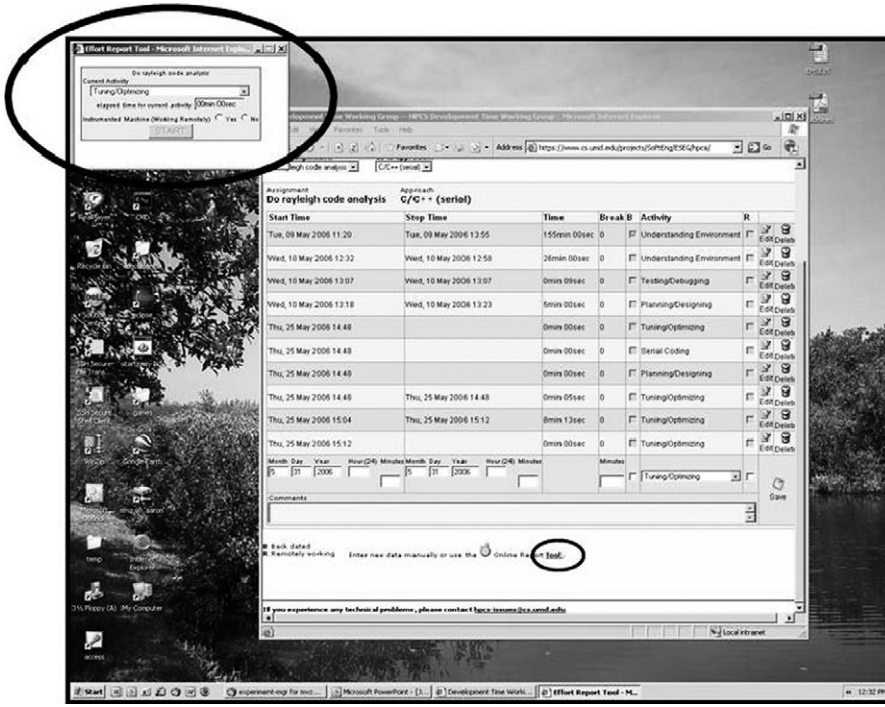

FIG. 3. Effort collection screen.

The tool automatically computes elapsed time between events and saves the data in the database. If the student stops for a period of time (e.g., goes to lunch and surfs the web), there is a *stop* button on the tool. Upon returning, the user simply clicks on *start* to resume timing.

For most HPC development, the student simply has to:

1. Log into Experiment Manager to go to effort page (Fig. 4), then click on effort tool (Fig. 3).
2. Develop program as usual.
3. Each time a new activity starts, click on the new activity in the effort tool (Fig. 2).
4. If any errors are found, the student records that defect by invoking the defect tool (Fig. 4) to explain the defect on a separate page (Fig. 5).

Only steps 3 and 4 involve any separate activity for participating in these experiments, and such activity is minimal.

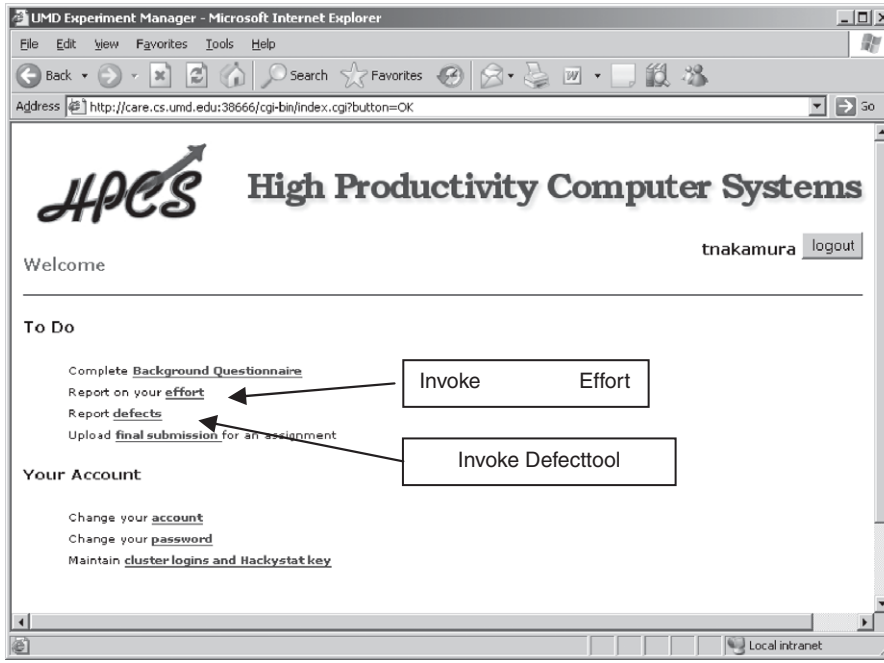

FIG. 4. Student view of Experiment Manager.

3.3.1 Data Sanitization

While personal data collected by the experiments must be kept private, we would like to provide as much data as possible to the community as part of our analysis. The sanitization process exports 'safe' data into a database that can be made accessible to other researchers, running on a separate machine.

The sanitization process is briefly described in Fig. 6. Each data object we obtained in an experiment is classified as one of:

1. *Prohibited*: Data contains personal data we cannot reveal (e.g., name or other personal identifiers).
2. *Clean*: Data we can reveal (e.g., effort data for development of another clean object).
3. *Modified*: Data we can modify to make it clean (e.g., removing all personal identification in the source program).

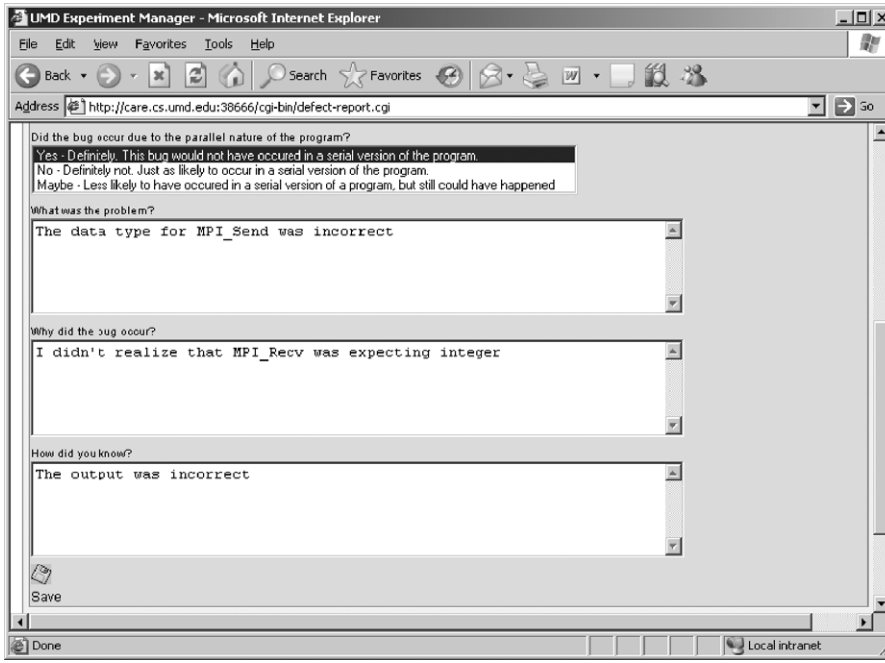

FIG. 5. Defect reporting tool.

Clean data can be moved to the analysis server and modified data can also be moved. Only prohibited data cannot be exported to others desiring to look at our collected database. Our sanitization process on data consists of the following four functions:

1. *Normalization* – Normalize the time stamps for each class on a common basis. By making each time stamp relative to 0 from the beginning of that experiment, information about in which semester it was collected (and hence from which school the data was collected) is hidden.
2. *Discretization* – Since grades are considered private data, we define a mapping table that maps grades on a small set, such as {good,bad}. Converting other interval or ratio data into less specific ordinal sets, while it loses granularity, it helps to present anonymity.
3. *Summarization* – With some of the universities, we can give out source code if we remove all personal identifiers of the students who wrote the code. But in

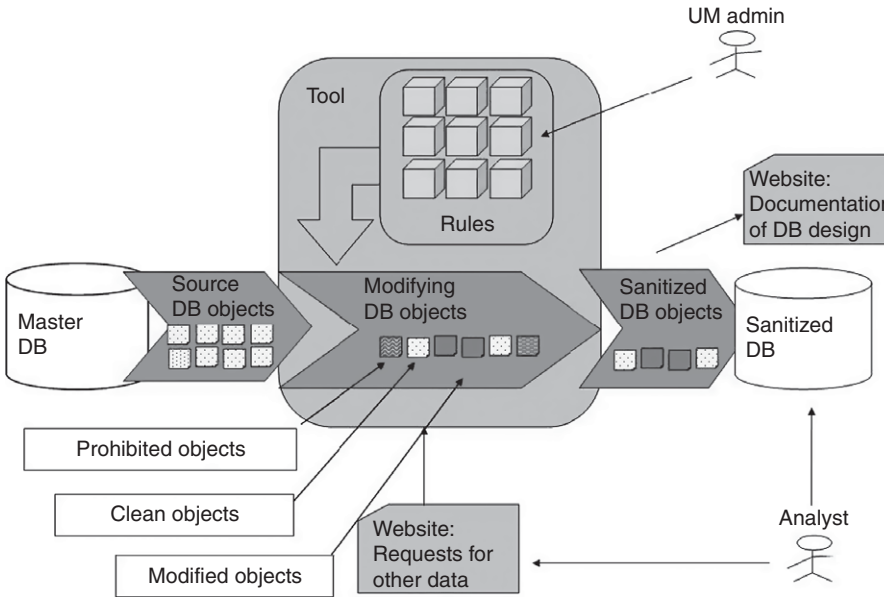

FIG. 6. Basic sanitization process.

some cases, we are prohibited from doing even that. If we cannot give out source code, we can collect patterns and counts of their occurrence in the source code. For example, we can count lines of code, or provide analyses of 'diffs' of successive versions of code.

4. *Anonymization* – We can create hash values for dates, school names, and other personal identifiers.

4. Current Status

Our Experiment Manager framework currently contains data from 25 classroom experiments conducted at various universities in the United States (Fig. 7). While some of the experiments preceded the Experiment Manager (and motivated its development) and their data was imported into the system; perhaps half the experiments used parts or all of the system.

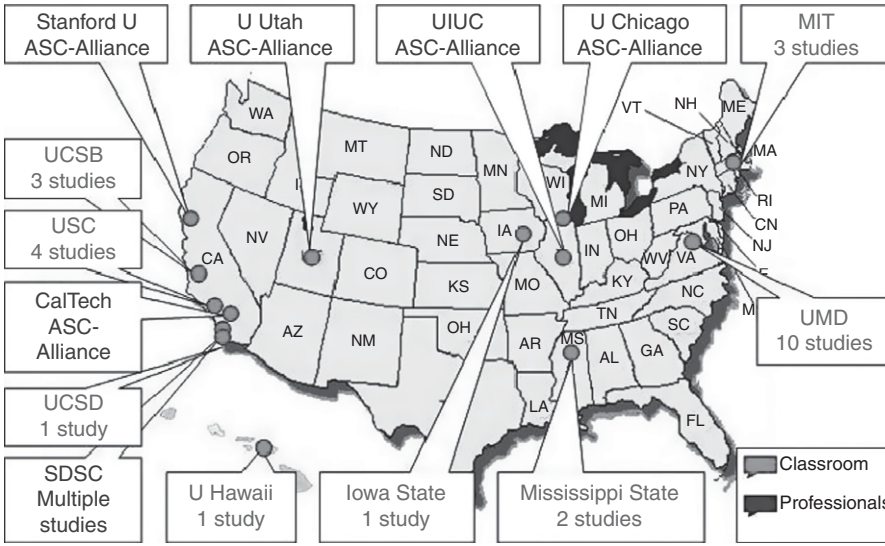

FIG. 7. Completed studies.

4.1 Experiment Manager Effectiveness

Before using the experiment manager, we discovered many discrepancies between successive classroom projects which prevented merging the results. Some of these were:

1. *It was not often obvious for how many processors the final programs were written.* Since a major goal of HPC programming is to divide an algorithm to run on multiple processors, this speedup (i.e., relative decrease in execution time by using multiple processors) is a critical measure of performance. Without knowing the initial goals for each student assignment, it was unclear how to measure performance goals for each class.
2. Related to the previous problem, *the projects all had differing requirements for final program complexity* (e.g., the number of replicated cells needing to be computed). How big a grid (e.g., number of replicated cells) were required in which to compute an answer and measure performance? This affected student programming goals.
3. *Grading requirements differed.* Was performance on an HPC machine important? Sometimes just getting a valid solution mattered. Maximum speedup, or the decrease in execution time of an HPC machine over a serial implementation, was sometimes the major goal.

By using our collected database of potential assignments, as well as our checklist of project attributes, this problem has lessened across multiple classes recently, allowing for the combination of results across different universities.

The IRB process seems like a formidable roadblock the first time any new professor encounters it. Often, in contacting faculty at a new university we would lose a semester's activity simply because the IRB approval process was too onerous the first time it was attempted. With our experience of IRB issues, and our collection of IRB forms required by various universities, this no longer is a major problem.

A related problem was the installation of software on the host computer for the collection of data. Again, this often meant the delay by a semester since the installation process was too complex. This was a major driving force to host much of this software as a web server at the University of Maryland, with a relatively simple UMDinst package that needed to be installed at each university's site.

The effort tool (pictured earlier as Figs. 2 and 3) also solved some of our data collection problems. We can collect effort data by three ways (Hackystat at the level of editor and shell event time stamps, manual data via programmer filled-in forms, and compiler time stamps via UMDinst). All give different results [13]. The use of the effort tool greatly eases the data collection problem, which we believe increases the reliability of such data.

Most of our results, so far, are anecdotal. But we have been able to address new universities and additional classes in a more methodical manner at present and believe the Experiment Manager software is a major part of this improvement.

4.2 Experiment Manager Evolution

The system is continuing to evolve. Current efforts focus on the following tasks:

- We are evolving the user interface to the Experiment Manager web-based tool. The goal is to minimize the workload of various stakeholders (i.e., roles) for setting up the experiment environment, registering with the system, entering the data, and conducting an analysis. We would like to develop small native applications that provide a more integrated interface to the operating system, making it less disruptive to users.
- We want to continue our experimentation with organizations that are often behind firewalls. Although we are currently studying professionals in an open environment, we want to use the Experiment Manager in this environment. Although the UMDinst instrumentation package can be set up in secure environments, the collected data cannot be directly uploaded to the University of Maryland servers. We have planned extensions to the Experiment Manager architecture to better

support the experimentations with these organizations. Working in these environments is necessary to see how professionals compare to the students.

- We will continue to evolve our analysis tools. For example, our prototype experience bases for evolving hypotheses and high end computing defects (e.g., www.hpcbugbase.org) will continue to evolve both in content and usability.
- We will evolve problem-specific harnesses that automatically capture information about correctness and performance of intermediate versions of the code during development to ensure that the quality of the solutions (specifically, correctness and performance) is measured consistently across all subjects.

This also requires us to evolve our experience bases to generate performance measures for each program submitted in order to have a consistent performance and speedup measure for use in our workflow and time to solution studies.

Our long-range goal is to allow the community access to our collected data. This requires additional work on a sanitized database that removes personal indicators and fulfills legal privacy requirements for use of such data.

4.3 Supported Analyses

The Experiment Manager was designed to ease data analysis, in order to support the investigation of a range of different research questions. Some of these analyses are focused in detail on a single developer being studied, while others aggregate data over several classes, allowing us to look across experimental data sets to discover influencing factors on effective HPCS development.

4.3.1 Views of a Single Subject

One view of a subject's work patterns is provided directly by the instrumentation. We refer to this view as the *physical level view* since it objectively reports incontrovertible occurrences at the operating system level, such as the time stamp of each compilation.

Figure 8 shows such a physical view for a 9 hour segment of work done by a given subject. The *x*-axis represents time and each dot on the graph represents a compile event. Although we have the tools to measure physical activities with a high degree of accuracy, this type of analysis does not yield much insight. For example, although we know how often the compiler was invoked, we do not know why: We cannot distinguish compilations which add new functionality from compilations which correct problems or defects that could have been avoided. This is an important distinction to make, if we want to know the cause of unnecessary rework so it can be avoided.

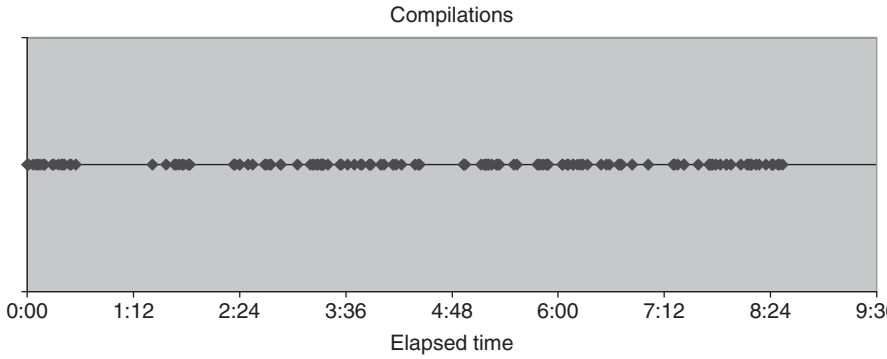

FIG. 8. Compilation events for one subject.

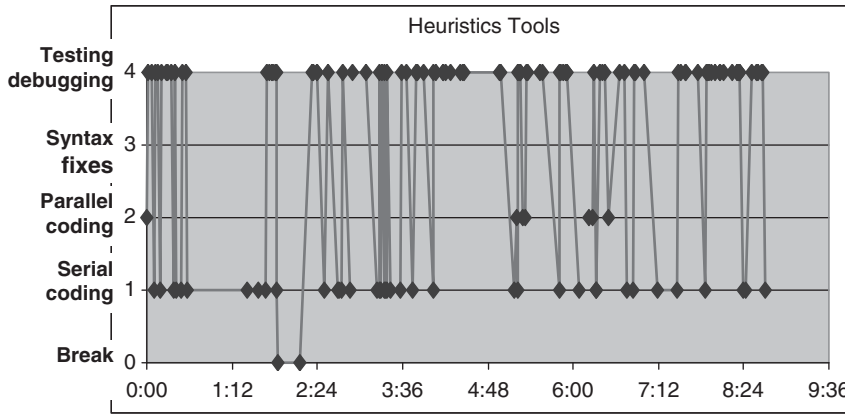

FIG. 9. Types of development activities for one subject.

A second approach is to use the logged compile times along with a snapshot of the code at each such compile and then apply a set of heuristics to guess at the semantics of the activities requiring that compilation. We have built such a tool that allows us to use various algorithms to manipulate these heuristics. The purpose of determining the semantic activities is to build baselines for predicting and evaluating the impact of new tools and languages.

Figure 9 uses the same data as Fig. 8 to illustrate this view. By evaluating the changes in the code for each compile, we can infer an activity being performed by the programmer. Again, each dot represents one compile but in this case the activity preceding each compile has been classified as one of:

- *Serial coding*: The developer is primarily focused on adding functionality through serial code. This is inferred since most of the changes since the previous compiler was in new code being added.
- *Parallel coding*: The developer is adding code to take advantage of multiple processors, not just adding function calls to the parallel library. We decided to separate out this activity since the amount of effort spent in this activity is indicative of how difficult it is to take advantage of the parallel architecture for solving the problem. This is inferred since parallel execution calls (such as to the MPI library) were added to the program.
- *Syntax fixes*: The developer is fixing errors from a previous compile. We can determine this since the previous compile failed, and the source program is changed with no intervening execution.
- *Testing and debugging*: The developer is focused on finding and fixing a problem, not adding new functionality. This activity can be identified via some typical and recognizable testing strategies, such as when a high percentage of the code added before a compile were output statements (so that variable values can be checked at runtime); creating/modifying test data files instead of the main code block; or removing test code back out of the system at the end of a debugging session. Our hypothesis is that effort spent on these activities can come from misunderstanding of the problem or the proposed solution and so could be addressed with more sophisticated aides given to developers.

Such a view helps us understand better the approach used by the subject, and how much of his/her time was spent on rework as opposed to adding new functionality. The duration data associated with each activity also helps us identify interesting events during the development: For example, when a large amount of time is spent debugging, analysts can focus on events preceding the debugging activity to understand what type of bug entered the system and how it was detected. This type of information can be used to understand how hard or easy it is for the developer to program in a given environment, and allows us to reason about what could be changed to improve the situation.

4.3.2 Validation of Workflow Heuristics

The heuristics to date have been developed by having researchers examine the collected data in detail (e.g., examining successive changes in source code versions). However, the accuracy of these heuristics is not generally known. We have developed a tool that provides a programmer with information about the inferred activities in real time. Using the tool, the developer then provides feedback about whether the

heuristic has correctly classified the activity. This allows us to evaluate how well the heuristics agree with the programmer's belief about the current development activity.

4.3.3 *Views of Multiple Subjects Across Several Classes*

From the same data, total effort can be calculated for each developer and examined across problems and across classes to understand the range of variation and whether causal factors can be related to changes in other measures of outcome, such as the performance of the code produced.

We note that for our experimental paradigm to support effective analyses, subjects in different classes who tackle the same problem using the same HPC approach should exhibit similar results regarding effort and the performance achieved. Moreover, we must be able to find measurable differences between subjects who, for example, applied different approaches to the same problem. In a previous paper [19], we presented some initial analyses of the data showing that both conditions hold.

Such analyses have been instrumental in developing an understanding of the effectiveness of different HPC approaches in different contexts. For example, Fig. 10 shows a comparison of effort data for two HPC approaches, 'OpenMP'

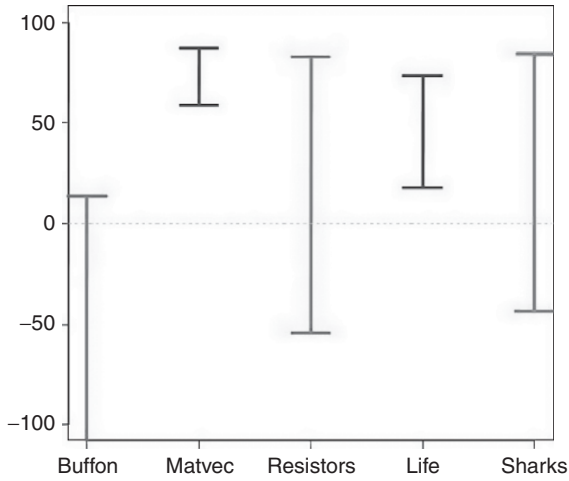

FIG. 10. Percentage effort reduction for OpenMP over MPI.

and 'MPI.' The percentage of effort saved by using OpenMP instead of MPI is shown for each of five parallel programming problems ('Buffon,' 'Matvec,' etc.) representing different classes of parallel programs. Thus, 50 on the *y*-axis represents 50% less effort for OpenMP; −50% would indicate that OpenMP required 50% more effort. The height of each bar represents the range of values from across an entire dataset of subjects. As can be seen, in two cases OpenMP yielded better results than MPI as all subjects required less effort; for two other cases although there were some who required less effort for MPI, the majority of data points indicated an effort savings associated with OpenMP. In only one case, for the Buffon problem, did MPI appear to give most subjects a savings in effort. As we gather more datasets using the Experiment Manager tool suite, we will continue this type of analysis to understand what other problems are in the set for which MPI requires less effort, and what it is about these situations that sets them apart from the ones where OpenMP was the less effort-intensive approach. (Interested readers can find a description of these approaches and programming problems in other publications [19].)

4.4 Evaluation

We have been performing classroom experiments in the HPC domain since early 2003. While we have not performed a careful controlled experiment of its effectiveness, we have observed anecdotally that the Experiment Manager avoids many of the problems others (including ourselves) have observed in running experiments. Many of these problems have already been reported in this paper. We have been able to run the same experiment across multiple classes in multiple universities and combine the results. Data is collected reliably and consistently across multiple development platforms. We have been able to obtain data to install into our database effortlessly without the need for students to perform any post-development activity. Faculty, who are not experimental researchers, have been able to run their own experiments with only minimal help from us. And finally, the ability to sanitize data allows us to provide copies of datasets to others wanting to perform their own analysis without running into IRB and privacy restrictions.

5. Related Work

There are various other projects that either support software engineering experiments, or support automatic data collection during development, but not both.

The SESE system [1] has many similarities: it is web-based and supports features such as managing subjects, supporting multiple roles, administering questionnaires,

capturing time spent during the experiment, collection of work products, and monitoring of subject activity. By comparison, Experiment Manager supports additional data capture (e.g., intermediate source files and defects) and data analysis (e.g., sanitization and workflow analysis).

PLUM, back in 1976, was one of the first systems to automatically collect development data [24]. It, along with Hackystat [15], Ginger2 [22], Marmoset [21], and Mylyn [17] are examples of systems which are designed to collect data during the development process, but do not have data management facilities that are specifically oriented towards running multiple experiments. Hackystat, which we are using in the Experiment Manager, can collect data from several different types of applications (e.g., vi, Emacs, Eclipse, jUnit, and Microsoft Word) via sensors. It was originally designed for project monitoring rather than running experiments. We have adopted the use of some of the Hackystat sensors into our data collection system. Ginger2 is an environment for collecting an enormous amount of low-level detail during software development, including eye-tracking and skin resistance. Marmoset is an Eclipse-specific system which captures source code snapshots at each compile, and is designed for computer science education research. Mylyn (originally called Mylar) is also an Eclipse-specific system. Mylyn provides support for task-focused code development and includes a framework for capturing and reporting on information about Eclipse usage.

6. Conclusions

The classroom provides an excellent opportunity for conducting software engineering experiments, but the complexities inherent in this environment makes such research difficult to perform across multiple classes and at multiple sites. The Experiment Manager framework supports the end-to-end process of conducting software engineering experiments in the classroom environment. This allows many others to run such experiments on their own in a way that allows for the appropriate controls of the experiment so that results across classes and organization at geographically diverse locations can be compared. The Experiment Manager significantly reduces the effort on behalf of the experimentalists who are managing the family of studies, and on the subjects themselves, by applying heuristics to infer programmer activities.

We have successfully applied the Experiment Manager framework and with each application are learning and improving the interface, simplifying the use by students, making its use of value in shrinking the overall problem solving process by students, for example, various forms of harnesses, the support for analysis, in order to get a thorough understanding of the HPC development model.

ACKNOWLEDGMENTS

This research was supported in part by Department of Energy contract DE-FG02-04ER25633 and Air Force grant FA8750-05-1-0100 to the University of Maryland. Several students worked on the Experiment Manager including Patrick R. Borek, Thiago Escudeiro Craveiro, and Martin Voelp.

REFERENCES

[1] Arisholm E., Sjoberg D. I. K., Carelius G. J., and Lindsjom Y., September 2002. A web-based support environment for software engineering experiments. *Nordic Journal of Computing*, **9**(3): 231–247.

[2] Basili V. R., Selby W. R., and Phillips T.-Y., November 1983. Metric analysis and data validation across Fortran projects. *IEEE Transactions on Software Engineering, SE-9*, **6**: 652–663.

[3] Basili V., and Green S., July 1994. Software process evolution at the SEL. *IEEE Software*, **11**(4): 58–66.

[4] Basili V., July 1997. Evolving and packaging reading technologies. *Journal of Systems and Software*, **38**(1): 3–12.

[5] Basili V., Shull F., and Lanubile F., July 1999. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, **25**(4): 456–473.

[6] Basili V., McGarry F., Pajerski R., and Zelkowitz M., May 2002. Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Laboratory. In *IEEE Computer Society and ACM International Conference on Software Engineering*, pp. 69–79. Orlando, FL.

[7] Campbell D. T., and Stanley J. C., 1963. Experimental and Quasi-Experimental Designs for Research. Houghton-Mifflin, Chicago.

[8] Carlson W., Culler D., Yellick K., Brooks E., and Warren K., 1999. Introduction to UPC and language specification (CCS-TR-99–157). Technical report, Center for Computing Sciences.

[9] Carver J., Jaccheri L., Morasca S., and Shull F., 2003. Issues in using students in empirical studies in software engineering education. In *International Symposium on Software Metrics*, pp. 239–249. Sydney, Australia.

[10] Choy R., and Edelman A., 2003. MATLAB*P 2.0: A unified parallel MATLAB. Singapore MIT Alliance Symposium.

[11] Dagum L., and Memon R., January 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, **5**(1): 46–55.

[12] Dongarra J. J., Otta S. W., Snir M., and Walker D., July 1996. A message passing standard for MPP and workstations. *Communications of the ACM*, **39**(7): 84–90.

[13] Hochstein L., Basili V., Zelkowitz M., Hollingsworth J., and Carver J., September 2005. Combining self-reported and automatic data to improve effort measurement. In *Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 356–365. Lisbon, Portugal.

[14] Hochstein L., Carver J., Shull F., Asgari S., Basili V., Hollingsworth J. K., and Zelkowitz M., November 2005. HPC Programmer Productivity: A Case Study of Novice HPC Programmers, Supercomputing 2005. Assoc. for Computing Machinery and Institute for Electronic and Electrical Engineers, Seattle, WA.

[15] Johnson P. M., Kou H., Agustin J. M., Zhang Q., Kagawa A., and Yamashita T., August 2004. Practical automated process and product metric collection and analysis in a classroom setting: Lessons

learned from Hackystat-UH. In *International Symposium on Empirical Software Engineering,* Los Angeles, California.

[16] Miller J., September 2000. Applying meta-analytical procedures to software engineering experiments. *Journal of Systems and Software*, **54**(1): 29–39.

[17] Murphy G. C., Kersten M., and Findlater L., July/August 2006. How are Java Software Developers using the eclipse IDE? *IEEE Software*, **23**(4): 76–83.

[18] Perry D. E., Staudenmayer N. A., and Votta L. G., 1995. Understanding and improving time usage in software development. Volume 5 of Trends in Software: Software Process John Wiley & Sons, New York.

[19] Shull F., Carver J., Hochstein L., and Basili V., 2005. Empirical study design in the area of high performance computing (HPC). In *International Symposium on Empirical Software Engineering,* Noosa Heads, Australia.

[20] Sjoberg D., Hannay J., Hansen O., Kampenes V., Karahasanovic A., Liborg N., and Rekdal A. C., September 2005. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, **31**(9): 733–753.

[21] Spacco J., Strecker J., Hovemeyer D., and Pugh W., 2005. Software repository mining with Marmoset: an automated programming project snapshot and testing system. In *Proceedings of the International Workshop on Mining Software Repositories*, pp. 1–5. St. Louis, Missouri.

[22] Torii K., Mastumoto K., Nakakoji K., Takada Y., Takada S., and Shima K., July 1999. Ginger2: An environment for computer-aided empirical software engineering. *IEEE Transactions on Software Engineering*, **25**(4): 474–492.

[23] Voelp M., August 2006. Diploma Thesis, Computer Science. University of Applied Sciences, Mannheim, Germany.

[24] Zelkowitz M. V., October 1976. Automatic program analysis and evaluation. In *International Conference on Software Engineering*, pp. 158–163. San Francisco, CA.

[25] Zelkowitz M. V., and Wallace D., May 1998. Experimental models for validating computer technology. *IEEE Computer*, **31**(5): 23–31.

[26] Zelkowitz M. V., Basili V., Asgari S., Hochstein L., Hollingsworth J., and Nakamura T., September 2005. Productivity measures for high performance computers. In *International Symposium on Software Metrics,* Como, Italy.