# Combining Self-reported and Automatic Data to Improve Programming Effort Measurement

Lorin Hochstein[1], Victor R. Basili[1,2], Marvin V. Zelkowitz[1,2], Jeffrey K. Hollingsworth[1], Jeff Carver[3]
[1]Department of Computer Science, University of Maryland, College Park, MD 20742
[2]Fraunhofer Center, College Park, MD 20740
[3]Department of Computer Science and Engineering, Mississippi State University, Mississippi State, MS 39762
{lorin,basili,mvz,hollings}@cs.umd.edu, carver@cse.msstate.edu

## ABSTRACT
Measuring effort accurately and consistently across subjects in a programming experiment can be a surprisingly difficult task. In particular, measures based on self-reported data may differ significantly from measures based on data which is recorded automatically from a subject's computing environment. Since self-reports can be unreliable, and not all activities can be captured automatically, a complete measure of programming effort should incorporate both classes of data. In this paper, we show how self-reported and automatic effort can be combined to perform validation and to measure total programming effort.

## Categories and Subject Descriptors
D.2.8 [**Software engineering**]: Process Metrics

## General Terms
Measurement, Experimentation, Human Factors, Verification.

## Keywords
Effort. Manual approaches.

## 1. INTRODUCTION
One of the primary goals of software engineering research is to reduce the amount of effort required to develop software. Consequently, many empirical studies in software engineering focus on the effect of a given technology on effort. There are several easy-to-measure proxies for effort (e.g., size, complexity, defect counts), but the most direct and accurate method is to record how much time the subjects spend when performing a task.

While measuring development time in a controlled experiment sounds simple, in practice it can be a notoriously difficult task. Measuring development time directly becomes particularly challenging when the task is too large to be completed in a single work session, the subjects are not being observed directly by the experimenter, or the experimenter does not have complete control over the working environment. Problems of loss of researcher control or insight are compounded in the time just before delivery of the product. Unfortunately, this "crunch time" could provide the most interesting information about the way people work when they are under time pressure.

How time is measured can have a pronounced impact on the interpretation of results. If the measure does not capture effort consistently across activities or subjects, then it can introduce bias and may lead to drawing incorrect conclusions. Even if the measure is unbiased, an imprecise measure will reduce the power of a study, and is especially inconvenient if the goal is to develop quantitative models of the effect of certain variables on effort.

We have been conducting empirical studies to characterize how different variables affect effort in the domain of high performance computing. Our initial studies have focused on graduate students solving small parallel programming problems [5]. These studies were done in the context of the Defense Advanced Research Projects Agency (DARPA) High Productivity Computing Systems (HPCS) program. One of the goals of the Development Time working group, led by the University of Maryland, of the HPCS project is to develop methods to evaluate the productivity of high-end computing systems, in particular the next generation of systems that are currently being developed [14].

In the context of these studies, we have sought a measure of programming effort that is both accurate and complete (i.e., captures all programmer activities well). In this paper, we present our methods for collecting effort data and how we validated them through empirical studies. We show how a combination of self-reported and automatic measures of effort data can be used for assessing confidence in results and estimating total effort.

This paper is structured as follows: Section 2 is an overview of different methods for measuring effort. In Section 3, we describe how we performed an initial evaluation of our effort measures through pilot studies. In section 4, we describe how we performed a more detailed evaluation of these measures using observational studies. In section 5, we present our final algorithms for measuring and validating effort, and we conclude in section 6.

## 2. BACKGROUND
Many software engineering studies have been conducted that involve measuring effort. We can classify them broadly into four categories: self-reported, automatic, hybrid, and indirect.

### 2.1 Self-reported
The simplest instrument for measuring effort, from the experimenter's point of view, is to have the subjects keep track of


*ESEC-FSE'05*, September 5–9, 2005, Lisbon, Portugal.

their own effort using an effort log. The only instrumentation required is a paper form, although web-based entry forms are also possible. Along with recording effort, a log can also capture the type of activities that are being performed. A log can capture all of the activities related to software development, even activities that do not involve a direct interaction with a computer, such as "thinking." One example of the use of such forms is in Humphrey's Personal Software Process [8], which has programmers fill out time recording logs. This self-reported effort data is then used for tasks such as schedule estimation.

Self-reported data can be collected in two different formats: *free-form* or *pre-specified*. In a free-form log, the subject has no constraints on the description of activities or the resolution of the log. In a pre-specified log, the subject chooses from a pre-defined set of activities when filling out the log, and the resolution of the individual entries are constrained. Each approach has advantages and disadvantages. For example, we have used free-form effort logs to collect data on programmers working in industry. We have found very large differences in the granularity of the reported activities, with one programmer recording log entries in minutes, and another recording log entries in days.

Self-reported measures can vary over time, due to history or maturation effects [4]. This is a particular problem when the subjects have more interest in completing the task than complying with the protocols of the study. For example, students who are working on an assignment for a class might become less diligent with their log as they approach their deadline. Moreover, researchers using self-reported data must also worry about accuracy. For example, the student subjects described above might, consciously or not, over-estimate their effort in order to impress the instructor. Student accuracy may also vary based on other variables that are difficult to measure such as motivation to capture accurate data.

Basili et al. [3] evaluated Software Science metrics against self-reported pre-specified effort data collected from satellite ground support software projects. There was very little correlation between self-reported effort and metrics known to predict effort, and there was concern that poor self-reported data was distorting the results. They were able to validate the programmers' self-reported effort data by cross-checking against resource forms filled out by the programmers' supervisors. The authors checked for agreement between the programmers' reports and the supervisors' forms to identify which reports were more reliable. Being able to triangulate the data allowed for an evaluation of the quality/accuracy of the data. The reported effort of the more reliable reports exhibited better correlation with the metrics under investigation.

Perry et al. [11] 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. They had developers maintain time diaries, filled in at the end of each day, to recount how their time was spent during that day. The experimenters observed the developers for five days over a 12-week period to evaluate the accuracy of the logs. They found that, on average, subjects overestimated their time by about 2.8% per day. The agreement between subject and observer varied considerably across subjects, ranging from 0.58 to 0.95. Each subject tended to consistently overestimate or consistently underestimate his or her effort. They also found that a major source of error in the logs was the failure to report unexpected events such as interruptions that occurred during the course of a day. The developers often failed to account for these unplanned events when retrospectively filling in their diaries.

We have seen confirmation of the findings of Perry et al. in a study performed for a high-school science project. The study measured the accuracy of self-reporting of task-completion times for short tasks (<10 minutes), and found that under-estimation or over-estimation occurred consistently within subjects.

## 2.2 Automatic

If the experimenter has some control over the subject's computing environment, then the experimenter can use software to collect data from the environment. which can be used to measure effort. Data can be collected from interactions with the shell, editor, compiler, etc. By collecting events and their corresponding timestamps, the experimenter can estimate how much time the subject spent interacting with different programs. The experimenter can also try to infer the subjects' activities based on the nature of the events being captured (e.g., an active debugger may be classified as "debugging"). Furthermore, measures based on automatic data collection should be consistent across subjects and time, provided the experimenter has equal control over the computing environments of all subjects. Hackystat [8] and GRUMPS [13] are examples of such data collection systems.

The disadvantages of an automatic effort measure are the need for specialized instrumentation and for integrating the collected data to form an estimate of the effort. An automatic method generally leads the experimenter to focus on the data that is easiest to collect, rather than the data that will be most informative. Furthermore, the collected data will be a stream of timestamped events, which must be transformed into a single measure of effort. For example, effort could be estimated by adding up time intervals between events, or by slicing up time into equally-sized chunks and counting the number of chunks that contain events. Each of these methods has its own shortcomings.

Figure 1 is a graphical depiction of an interval-based method, where the circles are recorded events. Using this method, we estimate effort by adding up time intervals between events. However, this method must avoid counting time intervals that represent non-working gaps between work sessions (e.g., should $t_2$ be counted as effort or not?). Szafron and Schaeffer used an interval-based method to measure effort, ignoring time intervals larger than 15 minutes [12].

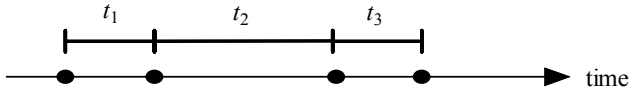


**Figure 1 Interval-based method**

Figure 2 is a graphical depiction of a chunk-based method, where time is broken up into chunks of size $t_c$, and we estimate effort by adding up the chunks that contain events. For this method, we need to determine the appropriate chunk size to avoid overestimating or underestimating.

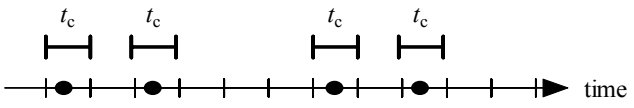


**Figure 2 Chunk-based method**

Hackystat supports a chunk-based method to estimate effort [9]. Hackystat sensors can be plugged in to different software tools (e.g., editors, shell, version-control systems). These sensors collect data on the use of these tools without user intervention. While Hackystat is designed for project monitoring rather than effort estimation, it does have some support for estimating the time spent editing source files, which it calls "active time". Active time uses the chunking method to estimate the time spent editing files. Kou and Xu analyzed the effect of chunk size on active time and concluded that the measure is not very sensitive to chunk size in the range of 3-10 minutes [10].

## 2.3 Hybrid

A hybrid measure is a combination of manual and automatic data collection. These methods rely on automatic instrumentation software that also prompts the user for some additional input (e.g., each time the user invoke the compiler, the system asks the user how long they have been working). This approach has advantages over the self-reported and fully automatic methods. The subjects are prompted for input on a regular basis and so we would expect more consistent reporting than a fully self-reported method. This approach can be used to improve the accuracy of an automatic time-interval based effort measure by distinguishing work times from break times through the help of user input.

A disadvantage is that subjects are forced to record information at the rate dictated by the software, rather than at their own pace, and they may become frustrated if they are prompted for data too often. Like the fully self-reported data, the subjects may not report their data consistently.

We have developed our own instrumentation that uses a hybrid approach to collect effort data. We have developed "wrapper" programs that instrument compilers. When users invoke these instrumented programs, they are asked questions about what type of activity they are doing, and how long they have been working. We use an interval-based approach to measure effort, applying the user input to help us identify gaps between work sessions.

Another example of combining automatic and self-reported data to measure effort is work by Atkins et al [1]. They combine monthly time sheets with change management data to estimate effort spent on particular changes.

## 2.4 Indirect

An indirect measure is one that does not measure subject activity directly, but measures something else that is believed to correlate well with effort. These measures are typically code-based: for example, size (e.g., lines of code, tokens), complexity, number of defects that occurred during development. Indirect measures can be attractive because they are typically much easier to obtain than direct measures (this is why they are used). In some cases, they may be appropriate. For example, if a researcher is concerned primarily with debugging time, then counting the number of defects may be a suitable proxy for measuring the time. However, the relationship between such measures and effort is not yet well-understood.

Basili and Reiter evaluated a number of indirect measures in an experiment where subjects developed software either as individuals with no particular methodology, in teams with no particular methodology, or in teams with a specific disciplined methodology [1]. They focused on both process metrics (which measured activity during development) and product metrics (which measured the delivered program). For process metrics, they looked at job steps and program changes, where job steps are counts of compiles and executions, and program changes are a measure of textual changes in the source code during development. For product metrics, they looked at various size metrics (line counts, routine counts, decision counts) and complexity metrics based on cyclomatic complexity. They found that the various process metrics agreed on differences between groups (disciplined teams required few job steps and fewer program changes than both ad-hoc individuals and ad-hoc teams), while the various product measures showed different trends.

# 3. PILOTING EFFORT MEASURES

We conducted studies with graduate students at various universities across the United States. One of the goals of the research is to evaluate parallel programming technologies for their effect on development time and execution time. The studies described in this section were pilots which we used to evaluate our data collection methodology, as well as to familiarize the professors with empirical studies involving human subjects. We were concerned with measuring overall development time, as well as identifying the different activities of development (e.g., parallelizing, debugging).

Each study was done in a graduate level class about parallel programming. We collected data on students as they worked on parallel programming assignments that were required coursework. These assignments were generally due two weeks after they had been assigned, and required on the order of ten hours of work to complete.

The student programs had to be compiled and run on a parallel machine, which the students accessed through remote login. This gave us some control over the students' environment, as we could collect data on the remote machine. However, we could not collect data if the students worked on a local machine (e.g., serial program development, editing source code, etc.). While students were encouraged to develop on the remote machines, we could not force them to do so.

## 3.1 Data collection

We chose to use a self-reported measure of effort as well as a hybrid measure of effort. For the self-reported measure, we asked the students to keep an effort log, to report how much time they spent each day in different activities, which we call *self-reported effort*. Figure 3 shows the log format and one entry. We also asked the subject to specify, for each entry, whether they were working on an instrumented machine. We used a web-based form to collect the data.

| Effort (hours) | Thinking (Understanding the problem) | Thinking (Designing a solution) | Experimenting with environment | Adding functionality | Parallelizing | Tuning | Debugging | Testing | Other ( specify activity) |
|---|---|---|---|---|---|---|---|---|---|
| 0.25 | X | X | | | | | | | |

**Figure 3 Effort log**

To collect data for automatically measuring effort, we instrumented the compiler. On each compile, the timestamp was recorded and a copy of the submitted source code was captured. Each time the user invoked the instrumented compiler, we asked them to specify their *work time*: how long they had been working since their last compile. This question is optional, and if they do not respond it means that they have been working continuously (see Figure 4).

```
$ mpicc life.c
How long (in minutes) have you been working
before this compile?
(Hit enter if you have been working
continuously since last compile)
>
```

**Figure 4 Question asked by compiler**

## 3.2 Instrumented effort measure

We used an interval-based hybrid measure, defined as follows:

$$E = \sum_i f(t_i, w_i) \quad (1)$$

where $E$ is total effort, $t_i$ is the i'th time interval (between compiles *i-1* and *i*), $w_i$ is the work time specified by the user (0 by default), and $f$ is the following function:

$$f(t_i, w_i) = \begin{cases} w_i & w_i > 0 \\ t_i & w_i = 0 \wedge t_i \le T_1 \\ \overline{w} & w_i = 0 \wedge t_i > T_1 \wedge \overline{w} > 0 \\ T_2 & w_i = 0 \wedge t_i > T_1 \wedge \overline{w} = 0 \end{cases} \quad (2)$$

In this equation, $\overline{w}$ is the average work time specified by the subject across all compiles. If the subject specifies a work time, we use that as the time interval. If the subject does not specify a work time, we use the actual time interval, provided it falls below a threshold $T_1$. If the user does not specify a time interval and the actual time interval exceeds $T_1$, we use the mean work time specified by the subject, as an estimate of the time interval. If the subject never specified a work time throughout the development process, we use the value $T_2$. We determine $T_1$ and $T_2$ from analysis of the collected data of the pilot studies.

## 3.3 Experimental setting

The pilot studies took place in courses at the following universities: University of Maryland, MIT, University of California Santa Barbara, and University of Southern California. Each class had 1-4 assignments, and the focus of each assignment was the implementation of a program to be run on a parallel machine, typically in MPI [6], OpenMP [5], or both.

## 3.4 Analysis

### *3.4.1 Estimating $T_1$*

We use $T_1$ as a threshold to determine if an interval should be counted as continuous work. This is necessary because subjects do not always specify a work time at the start of a work session. An ideal value for $T_1$ would be longer than the longest interval of true continuous work, and shorter than the shortest break, so that it would perfectly classify intervals as being work or breaks. We assume the longest interval of work is shorter than the shortest break. In practice, this won't be true, since many work sessions involve brief interruptions, but for the purposes of our algorithm we would still count them as work. Recall that Perry et al. [9] found that people tend to forget about unscheduled interruptions when keeping retrospective logs, so we hoped this would not be a source of disagreement.

The method we use to estimate $T_1$ is to compare the time intervals ($t_i$) where users specified a work time with the time intervals where users did not specify a work time. We expect the distribution of time intervals where users did specify a work time to be similar to the distribution of time intervals that represent breaks, and we expect the distribution of time intervals where users did not specify a work time to be similar to the distribution of time intervals that represent continuous work. This method does not depend on the accuracy of reported effort and we do not expect the results to vary by size of the assignment. Note that roughly 20% of the subjects never specified a work time throughout an entire assignment.

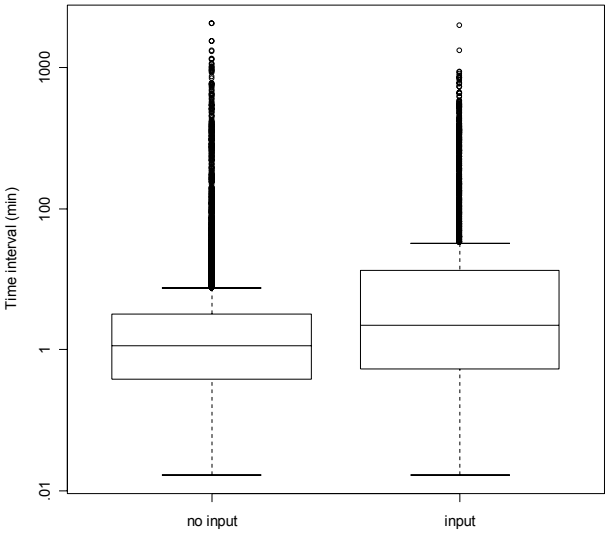


**Figure 5 Distribution of time intervals**

In Figure 5, we show box plots of these two groups of time intervals: when users specify work time ("input") and when users do not specify work time ("no input"). This data is across 4 classes, 8-20 participating students in each class, with 1-4 assignments in each class, and represents roughly 18,000

compiles. We had expected most of the time intervals labeled "input" to be larger than those labeled "no input", but there is considerable overlap among the two data sets. The very high values for "no input" suggest that students do not always fill out the work time after coming back from a break, illustrating the need for a good $T_1$ estimate. More surprising is the many low values for intervals when the user specifies a work time. The median time interval is under 3 minutes: more than half of these intervals are from students specifying a work time when it has been less than 3 minutes since their last compile. We can only conclude that these students either did not understand when they were supposed to answer this question, or were intentionally answering incorrectly (e.g., just hitting "1" each time).

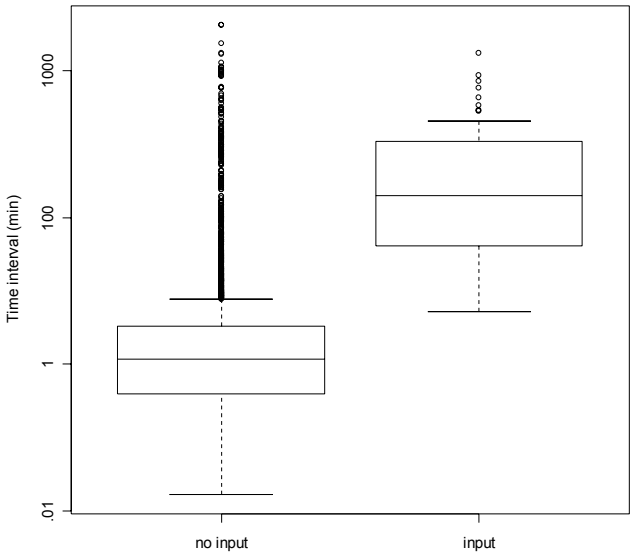


**Figure 6 Filtered distribution of time intervals**

To filter these out, we eliminated the data from all subjects who ever specified a work time when the actual time between compiles was 5 minutes or less, since it is reasonable that if a subject had compiles that were 5 minutes apart, they were working continuously, and entering a work time would be incorrect. This reduces our total sample from 18,000 to about 5,000 compiles. This updated plot is in Figure 6. Now there is considerably less overlap between the two distributions. For the "input" intervals, the first quartile is 43 minutes, which implies that 75% of the time intervals that were effectively classified by the subjects as breaks were over 43 minutes. It seems reasonable to round this up to $T_1$=45 minutes.

### *3.4.2 Estimating $T_2$*

To choose an appropriate value for $T_2$ we need to know how long students typically spend working before their first compile when they begin a work session. This should be equal to their "work time", which is how long they say they have been working before a compile, since they are only supposed to specify a work time when they have returned from a break.

Figure 7 shows a box plot of the work times specified by graduate students in the Applied Parallel Computing course at the University of California, Santa Barbara, over four parallel programming assignments. The plot suggests that the work time distributions are quite different across subjects (note that the y-axis is logarithmic). Even within subjects, there can be a great deal of variation (e.g., subject 18 exhibits variation from 1 minute to 100 minutes).

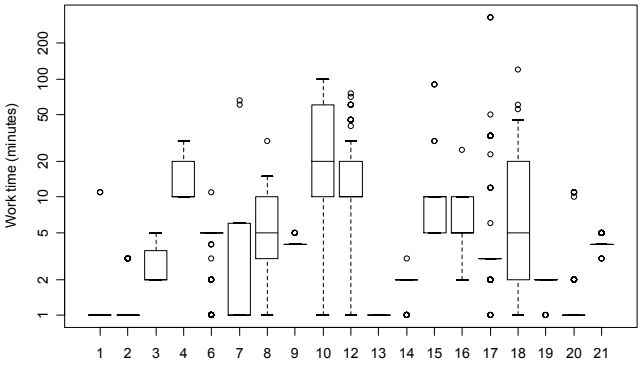


**Figure 7 Work time distributions across subjects**

Figure 8 shows the distribution of work times for the entire class. The median work time is 4 minutes and the mean is 4.8 minutes. Therefore, we felt that a reasonable value for the time interval estimate in the absence of work time to be $T_2$ =5 minutes.

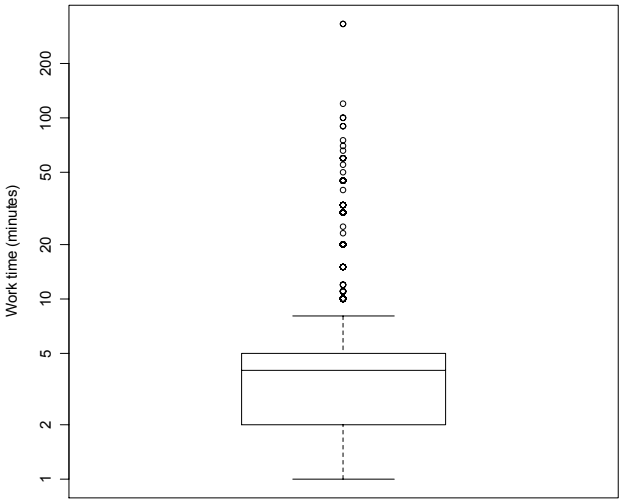


**Figure 8 Work time distribution for entire class**

Figure 9 shows the results when we apply equation 2 to the data collected from the three courses ($T_1$=45 minutes, $T_2$=5 minutes). To compute reported effort, we only count entries where the subjects specify that they are working on the instrumented machine. Note that each data point is one student solving one assignment. Since some classes had up to 4 assignments, a single subject may appear up to 4 times on this plot. Table 1 shows the summary information for the deviations (instrumented effort – self-reported effort) across subjects, where *n* is the number of subjects, *mean* is the average of the deviations across all subjects, *stdev* is the standard deviation of the deviations, and *%* is the mean of the deviations divided by the mean effort (To compute mean effort, we take the mean of the self-reported and instrumented effort for each subject, and average this across all subjects). The negative mean deviation indicates that the instrumented effort tends to underestimate the effort reported by the students.

**Table 1 Summary info for deviations**

| n | mean | stdev | % |
|---|---|---|---|
| 56 | -3.7 hours | 11.7 hours | 26% |

### *3.4.3 Examining the reported data*

Subjects claim that they spend about 80% of their development time working on the instrumented machine, averaged across all subjects who submitted effort logs. More than half claim that they spent 100% of their time on the instrumented machine.

We observe both overreporting and underreporting in this data. Two of the largest instances of overreporting are from the same subject on two different assignments. This subject reported 44 hours on one assignment and 63 hours on another. When we examined the collected data in detail, it appeared that the subject was either significantly overestimating their effort, or the subject was not invoking the instrumented compiler while working, despite claims in the effort log to be working on the instrumented machine. This subject tended to use relatively large entry sizes: specifying several 10-hour entries and one 14-hour entry. By comparison, the median entry across all subjects and assignments was 3.8 hours.

There are three subjects with instrumented effort of over 10 hours whose effort logs consisted of a single one-hour entry, clear instances of underreporting. It is notable that these subjects all came from the same class, so perhaps there was some unknown factor which caused students in this class to be less diligent with their logs.

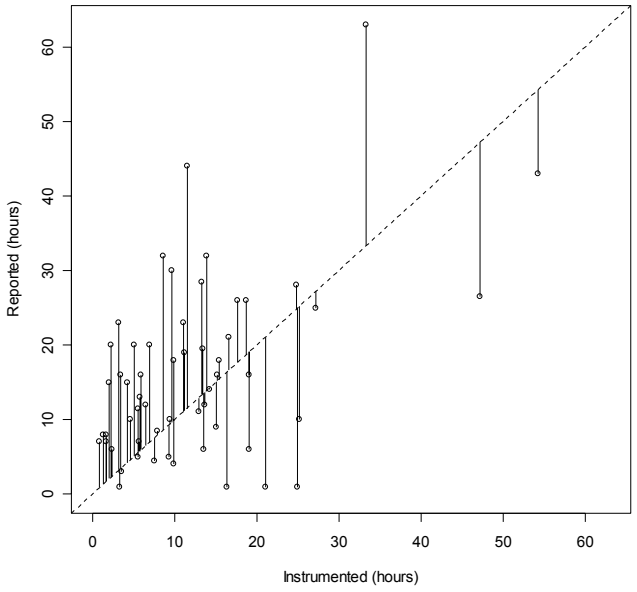


**Figure 9 Reported vs. instrumented effort**

## 3.5 Lessons learned

We learned several lessons from these pilot studies:

- The *work time* (time spent working before first compile, after having returned from a break) varies considerably across subjects.
- Subjects will not always specify when they have returned from a break, even if prompted to at each compile.
- Subjects will sometimes specify work time even when the gap in question is quite small. Either they do not understand the question or are specifying breaks at too fine a level of granularity for our purposes.
- Students claim that they spend most of their time (>80%) working on the instrumented machine, so instrumented effort can potentially be within 20% of total effort.
- Underreporting and overreporting are significant issues. They appear to be the source of the largest deviations.

# 4. VALIDATING EFFORT MEAURES

## 4.1 Motivation

In the pilot studies, we tried to increase the accuracy of our instrumented effort by asking the subjects for additional information. However, we found that the data provided by the subjects was often inconsistent with data collected automatically. We did not have sufficient confidence in either instrumented effort or self-reported effort for a proper evaluation.

We decided to run a series of observational studies to compare the various effort measures to a more accurate effort measure, obtained through direct observation. We also wanted to collect more detailed information about what types of activities our effort measures do not capture well.

An observational study involves a single subject who solves one of the programming problems from the classroom studies. We apply all of the same data collection measures which were used in the classroom studies (effort logs, instrumented compilers). The additional factor in these studies is a passive observer, who sits with the programmer and keeps a separate log of the programmer's activities. The goal of the observer is to provide an accurate measure of effort against which the other measures can be compared.

## 4.2 Modification to data collection

For these studies, we also collected two new types of automatic data. We used Johnson's Hackystat [7] system for collecting the time spent in the editor and commands sent to the shell. We also captured information on when the user submitted a job. We were able to do so because the subjects were using a Beowulf cluster on which we could instrument the batch scheduler used for job submission. Unlike the compiler instrumentation, we did not ask the subject questions when a job was submitted.

Our initial compiler instrumentation software asked the subject questions on each compile. Some subjects in previous studies found these questions irritating, especially when they had to recompile due to a syntax error. To reduce the frequency of questions, we modified the compiler instrumentation to only ask the subject questions if there were no syntax errors in the source code).

We made several modifications to our effort logs. We switched from a web-based interface to paper effort logs as we suspected that the web interface may have been related to underreporting in the previous studies. Between our first and second observational study we modified the effort log so that the subject had to specify

| Date | Start time | Stop time | Breaks (minutes) | Total time (minutes) | Activity | Comments |
|---|---|---|---|---|---|---|
| 10/4 | 3:15 PM | 3:42 PM | | 27 | Serial coding | |

**Figure 11 Updated effort log**

start and stop times for each activity (see Figure 11). This increases the precision of the log, and allows us to compare the logs more directly with automatically collected data.

## 4.3 Instrumented effort measure

The modification of the compiler instrumentation introduces a problem into our instrumented effort measure.

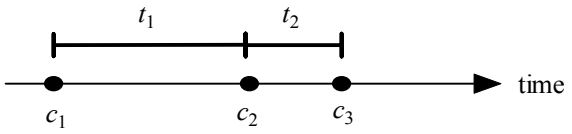


**Figure 10 Dealing with compile-time errors**

Consider the scenario depicted in Figure 10. Let $c_1$,$c_2$,$c_3$ be captured compile events, and assume that the subject took a break between $c_1$ and $c_2$. Consider the case where $c_2$ is a failed compile (syntax error) and $c_3$ is a successful compile. The instrumentation will not ask a question at $c_2$, but it will ask at $c_3$. If the subject specifies a work time, the algorithm will incorrectly use that work time in place of $t_2$, when it should be used in place of $t_1$. To avoid this problem, we do not use failed compiles in computing instrumented effort, though we still record when they occur. Ignoring compiles that are not successful should not impact instrumented effort unless the time to fix syntax errors exceeds $T_1$ (45 minutes).

We also saw in the previous studies that subjects sometimes specify work times that exceed the time interval between compiles. This led to a further refinement of algorithm that checks for this occurrence:

$$f(t_i, w_i) = \begin{cases} w_i & w_i > 0 \wedge t_i \geq w_i \\ t_i & (w_i = 0 \wedge t_i \leq T_1) \vee (t_i < w_i) \\ \overline{w} & w_i = 0 \wedge t_i > T_1 \wedge \overline{w} > 0 \\ T_2 & w_i = 0 \wedge t_i > T_1 \wedge \overline{w} = 0 \end{cases} \quad (3)$$

In addition to our previous instrumented effort measure, we applied a new instrumented effort measure that incorporates all captured events (i.e., compiles, run, edits, shell commands). We no longer incorporate the work time in this measure because we expect too much overlap between work time and other captured events. We use a simple interval-based algorithm where we only include the interval if it exceeds a certain threshold, otherwise it does not contribute to the estimate. We used a threshold of 45 minutes, as before.

## 4.4 Experimental setting

For the first observational study we chose a small, relatively simple problem, used in two of the classes from our second round of studies. The problem was the "Buffon-Laplace needle problem", a Monte Carlo simulation to estimate the value of $\pi$ by dropping needles on a grid and counting how many needles intersect with the gridlines. The task was to solve the problem in serial in C, and in parallel using the MPI library on a Beowulf cluster. The subject was a member of our research group who had taken the graduate-level high performance computing course at the University of Maryland. The study took place in a single session and ran for a little over two hours.

For our second observational study, we chose a more difficult problem from the set of assignments: Conway's game of life. We kept the same objectives as before (serial implementation in C, parallel implementation using MPI on a Beowulf cluster). The subject was a professional programmer working for a research lab involved with parallel applications. Though the focus of his day-to-day work was not MPI-related, he had experience with MPI. The programming task took about nine hours to complete, which was done in six sessions over the course of two weeks.

## 4.5 Analysis of first observational study

Figure 12 graphically depicts the data we collected from the first observational study using the automatic instrumentation, as well as effort measures using our method and Hackystat's "active time" measure. The x-axis represents the actual clock time of the observation. On the y-axis, *compiles* refers to successful compiles, *runs* refers to submissions to the batch scheduler for parallel runs, *edits* refers to edit events captured by Hackystat, *cmds* refers to commands typed in the shell, *inst. effort* is our effort measure described in the previous section (using only data captured from compiles), and *active time* refers to Hackystat's "active time" estimate, which is a measure of how much time is spent editing a file. The vertical lines indicate the beginning and end times of the session as recorded by the observer.

We can see that the instrumented effort has some (small) gaps at the beginning and end of the session. The gap near the beginning of the session is because the subject slightly underestimated when specifying the work time. The gap at the end of the session shows that instrumented effort fails to take into account effort after the last compile. We were able to capture the time before the first compile because of work time specified by the subject. From the figure, it appears that our instrumented effort may have been more accurate if we used all captured events, rather than rely solely on compile events.

Active time captures most of the effort, with two gaps. The first gap (10:20-10:40) is due to the initial "thinking" time, where the subject is thinking about the problem and is working out a solution on paper. The second gap (12:05-12:10) is due to the subject doing testing, measuring the effect of compiler optimization on execution time.

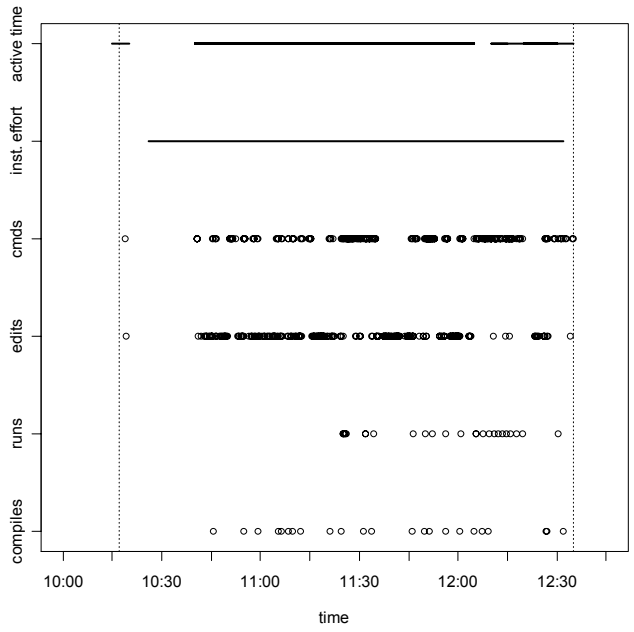


**Figure 12 First observational study**

Table 2 gives a summary of the different effort measures, where *Direct obs.* Refers to direct observation, *Rep. effort* refers to reported effort, *Inst. effort* refers to instrumented effort, and *Active time* refers to Hackystat active time. We see that instrumented effort underestimated by a very small amount (<5 minutes), with active time underestimating somewhat more (15 minutes) The reported effort overestimated by about 17 minutes.

We believe that the subject kept the effort log as accurately as can be reasonably expected. The effort log errors were most likely due to the resolution of the log: the subject was asked to specify time in hours for each activity (see Figure 3). Most entries were either 0.25 h or 0.5 h, which is probably the finest level of granularity at which a person can estimate their time spent in a particular activity. If ideal conditions yielded errors of about 13%, we don't expect to do better than about 20% in a classroom study.

**Table 2 First observational study results**

| | Direct obs. | Rep. effort | Inst. effort | Active time |
|---|---|---|---|---|
| **Time** | 2.17 h | 2.45 h | 2.10 h | 1.92 h |
| **Error** | 0 h | +0.28 h | -0.07 h | -0.25 h |
| **Error %** | 0 % | +13% | -3% | -12% |

The results of this first observational study, though encouraging, were most likely not representative. Most problems are more difficult and require work to be done across multiple sessions, which is what we speculated to be the most likely source of errors in our effort measure. This study does give us some lower bounds on how accurate we should expect our effort measures to be.

## 4.6 Agreement measure: fidelity

As we mentioned in section 4.2, between the first and second observational study, we modified the effort log so we could directly compare the self-reported effort with the instrumented effort. We adopted the *fidelity* measure described in Perry et al. [9] to give us a sense of how much we should trust the self-reported effort. They define fidelity as the overlap between the two measures, divided by the measure which is considered more accurate.

## 4.7 Analysis of second observational study

Table 3 gives a summary of the accuracy of the different methods. *Inst. effort* refers to the instrumented effort measure from equation 3 (using only compile data), and *All* refers to instrumented effort which takes into account all of the different types of data.

Note that the increase in the accuracy of the effort log, decrease in accuracy of instrumented effort, and increase in accuracy of the Hackystat measure compared to the first observational study. While it is unwise to draw conclusions from two data points, these results support our hypotheses that a more precise effort log improves accuracy, and that instrumented effort decreases in accuracy when the work is done over multiple sessions.

**Table 3 Second observational study results**

| | Direct obs. | Rep. Effort | Inst. effort | Active time | All |
|---|---|---|---|---|---|
| **Time** | 9.05 h | 8.98 h | 8.28 h | 8.60 h | 9.08h |
| **Error** | 0.00 h | -0.07 h | -0.77 h | -0.45 h | +.03 h |
| **Error %** | 0% | -1% | -8% | -5% | +0.3% |

If we compute the fidelity to check the agreement of the self-reported effort and the instrumented effort (*All*), assuming the instrumented effort to be more accurate, we get an agreement of 0.97, a very high value which is consistent with Table 3.

Figure 13 shows activity for the second session of the study, and illustrates some of the problems with our effort measure (it does not depict the instrumented effort measure which takes all types of events into account). On the first compile, the subject did not specify the time spent working, so the instrumentation used the average work time specified (18 minutes). The observed time spent working before first compile was only 3 minutes. The subject spent this entire session debugging the serial version of the program, so no parallel runs were recorded. On this plot, we can also see the gap between the time of the last compile for the day and the time when the subject actually stopped working: our compiler-based effort measure has no mechanism for estimating this time. These two errors illustrate the weakness of an interval-based method for estimating effort.

The subject spent much of this session looking at source code and output of previous runs. While he was interacting with the editor (e.g., scrolling, switching from one file to another), no editing was taking place, resulting in gaps in the Hackystat "active time" measure. Note that for the second large gap in active time (10:35-10:55), there are no captured events at all: no compiles, edits, shell commands, or runs. This illustrate the weakness of a chunking-based method for estimating effort.

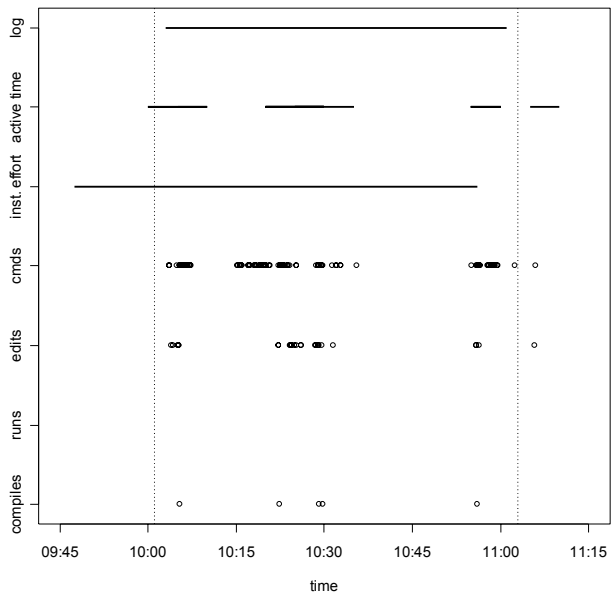


**Figure 13 Activity for second session**

The instrumented effort measure which takes into account all types of events clearly performs the best, achieving within 1% of the observed effort. For this measure, we used $T_1$=45 minutes, $T_2$=0. We initially tried larger values of $T_2$, which always resulted in overestimates. In our study, the subject always began a session by logging in to the remote machine, so the first recorded event coincided very closely with the beginning of a work session, and the last recorded event (the logout command) coincided with the end of the work session. Note that if only compiles are used, the same algorithm yields an effort of 6.2 hours, which is an error of about -30%. This suggests that compiles do not occur frequently enough to be relied upon exclusively for estimating effort without additional information.

## 4.8 Lessons learned

We learned several lessons from these observational studies:

- Instrumented effort underestimates due to time spent working before the first compile, time spent working after the last compile and time intervals where work was being done outside of the instrumented environment.
- Active time, which is a chunk-based automated measure, underestimates because of time intervals where work is done that doesn't trigger edit events.
- If we incorporate events recorded by Hackystat sensors into our instrumented effort, we capture more of the effort.
- An effort log that asks for start/stop times simplifies the task of evaluating self-reported effort against instrumented effort, and can potentially increase the accuracy of the effort measure.
- For an effort log that does not use start/stop times, the limit to the granularity of effort reporting is about 15 minutes.
- For an effort log that does not use start/stop times, we do not expect to get better than about 20% agreement on reported and instrumented effort.
- Instrumented effort accuracy may decrease when subjects program over multiple sessions because effort errors occur at the beginning and end of sessions.
- When a subject is debugging, sometimes no events are captured because the subject is simply looking at source code and program output. This may be particularly problematic for chunk-based effort measures.

# 5. EFFORT ALGORITHMS

## 5.1 Instrumented effort

We found that if there is sufficient automatically collected data (e.g., edit events, shell commands), then a simple, interval-based measure works well:

$$E = \sum_i f(t_i) \quad (4)$$

$$f(t) = \begin{cases} t & t \le T_1 \\ 0 & t > T_1 \end{cases} \quad (5)$$

where $E$ is the effort, $t_i$ is the time interval between captured events, and $T_1$ is a constant. Based on our analysis, we concluded that $T_1$=45 minutes was reasonable. In our observational studies, the largest interval where the subject was working but no activity was recorded was 20 minutes.

If it is possible to collect a subset of the above data (e.g., compile timestamps), then the simple method above will under-estimate the effort. We can increase the accuracy by asking the subjects questions at compile-time and using a hybrid measure to estimate the effort:

$$E = \sum_i f(t_i, w_i) \quad (6)$$

$$f(t_i, w_i) = \begin{cases} w_i & w_i > 0 \wedge t_i \ge w_i \\ t_i & (w_i = 0 \wedge t_i \le T_1) \vee (t_i < w_i) \\ \overline{w} & w_i = 0 \wedge t_i > T_1 \wedge \overline{w} > 0 \\ T_2 & w_i = 0 \wedge t_i > T_1 \wedge \overline{w} = 0 \end{cases} \quad (7)$$

where $E$ is the effort, $t_i$ is the time interval between events, $w_i$ is the *work time* (amount of time the subject indicated that they working before the event, 0 if no response), $\overline{w}$ is the average work time for the subject, and $T_1$ and $T_2$ are constants. From our analysis, we found $T_1$=45 minutes, $T_2$=5 minutes to be reasonable values.

## 5.2 Total effort and confidence

To obtain a measure of total effort, we combine instrumented and self-reported effort measures using the following equation:

$$E_{total} = E_{inst} + (1-k)E_{rep} \quad (8)$$

where $E_{total}$ is total effort, $E_{inst}$ is instrumented effort, $E_{rep}$ is self-reported effort, and $k$ is the fraction of the self-reported effort that corresponds to work done on an instrumented machine.

We use *fidelity* as a confidence measure to help us judge whether we should use the self-reported effort or discard it, based on how well it agrees with the instrumented effort:

$$f = \frac{Ov(E_{inst}, kE_{rep})}{E_{inst}} \quad (9)$$

where $f$ is fidelity, and $Ov$ is the overlap between two effort measures.

A disadvantage of *fidelity* is that it only measures underreporting. Measuring overreporting by comparing with instrumented effort is difficult because a subject may still be working even if no activities are recorded, as we saw in the observational studies. An alternate method is to look for coarse-grained self-reported entries, which are more likely to be suspect. We have found that large effort entries are a warning sign that overreporting may be occurring. In studies that follow the ones described here, we ask subjects to specify for each entry whether it was filled in close to the actual work, or retrospectively. This information may provide experimenters with additional warnings that the data is suspect.

# 6. CONCLUSION

In this paper, we have described how we evolved self-reported and automatic methods of data collection through a series of empirical studies, and how we combined them to assess our confidence in the data and measure total effort for small programming problems. We found that effort estimates based on automatically collected data can be quite accurate if sufficient data can be collected, particularly if the data comes from multiple sources. This data can be augmented by self-reported data to capture activities that would be missed by automatic instrumentation. However, the quality of self-reported data will vary considerably across subjects, and it should be validated against automatically collected data.

It is not clear whether a similar approach could be adopted in industrial environments, where much of the effort is spent in phases other than coding and there are barriers to using instrumention. Furthermore, larger projects may not require effort measures at the same level of precision as small experiments.

# 7. ACKNOWLEDGEMENT

This research was supported in part by Department of Energy contract DE-FG02-04ER25633 to the University of Maryland. We wish to acknowledge the contributions of the various faculty members and their students who have participated in the various experiments we have run over the past 2 years. This includes Alan Edelman at MIT, John Gilbert at the University of California Santa Barbara, Mary Hall and Jacqueline Chame at the University of Southern California, Alan Snavely at the University of California San Diego, Uzi Vishkin and Alan Sussman at the University of Maryland. We would like to acknowledge Sima Asgari, Taiga Nakamura, Jaymie Strecker and Forrest Shull for their feedback. We would also like to acknowledge Adrianne Fuentes from Charles Flowers High School in Prince George's County, Maryland.

# 8. REFERENCES

[1] Atkins, D., Ball, T., Graves, T., and Mockus, A. Using version control data to evaluate the impact of software tools. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, (Los Angeles, CA, May 16-22 1999)

[2] Basili, V.R., and Reiter, R.J. Evaluating automatable measures of software development. In *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity and Cost* (Oct. 1979)

[3] Basili, V.R., Selby, W.R., and Phillips, T.-Y. Metric analysis and data validation Across Fortran projects. *IEEE Transactions on Software Engineering, SE-9*, 6 (Nov. 1983), 652-663.

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

[5] Carver, J., Asgari, S., Basili, V., Hochstein, L., Hollingsworth J. K., Shull, F., and Zelkowitz, M. Studying code development for high performance computing: the HPCS program. In *Proceedings of the International Workshop on Software Engineering for High Performance Computing Systems Applications* (*SE-HPCS '04*) (Edinburgh, Scotland, May 24 2004).

[6] Dagum, L., and Menon, R. OpenMP: An industry-standard for shared-memory programming. In *IEEE Computational Science & Engineering*, 5, 1 (Jan. 1998), 46-55.

[7] Dongara, J.J., Otto, S.W., Snir, M., and Walker, D. A Message passing standard for MPP and workstations. In *Communications of the ACM, 39, 7* (Jul. 1996), 84-90.

[8] Humphrey, W.S. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

[9] Johnson, P.M., Kou, H., Agustin, J.M., Chan, C., Moore, C. A., Miglani, J., Zhen, S., and Doane, W.E. Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the International Conference on Software Engineering (ICSE '03)* (Portland, OR, May 3-10, 2003).

[10] Kou, H., and Xu, X. *Most Active File Measurement Validation In Hackystat*. Technical Report CSDL-02-09, University of Hawaii, Honolulu, HI, 2002.

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

[12] Szafron, D., and Schaeffer, J. An experiment to measure the usability of parallel programming systems. In *Concurrency: Practice and Experience, 8,* 2 (Mar. 1996), 147-166.

[13] Thomas, R., Kennedy, G.E., Draper, S., Mancy, R., Crease, M., Evans, H., and Gray, P. Generic usage monitoring of programming students. In *Proceedings of the 20th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education (ASCILITE '03)* (Adelaide, Australia, Dec 7-10, 2003).

[14] Zelkwoitz, M., Basili, V., Asgari, S., Hochstein, L., Hollingsworth, J., and Nakamura, T. Measuring productivity on high performance computers. In *Proceedings of the International Symposium on Software Metrics (Metrics '05)* (Como, Italy, Sep. 19-22 2005).