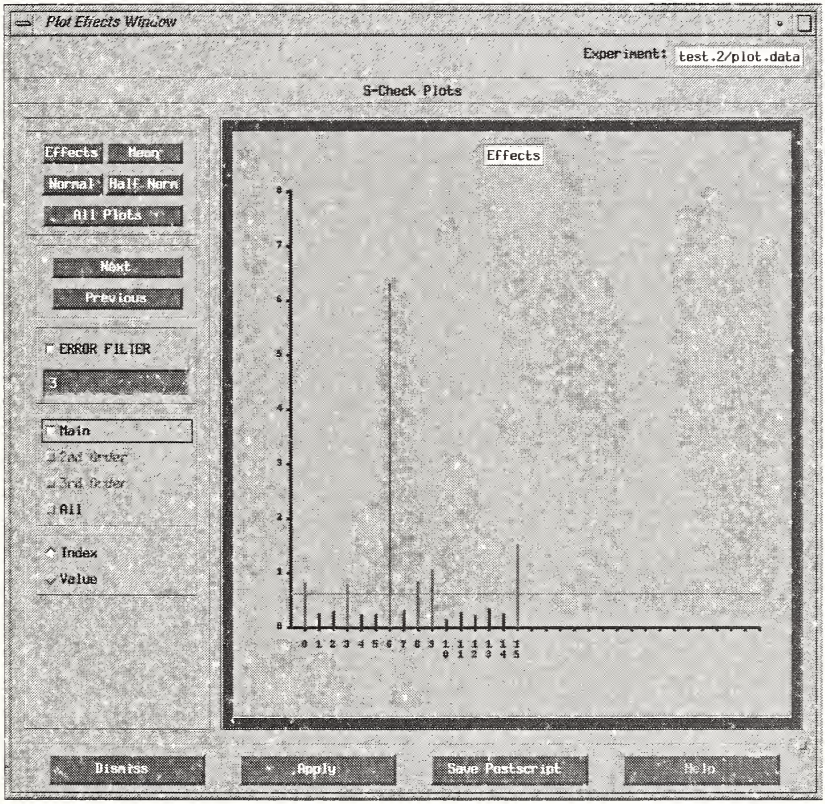
NAT'L INST. OF STAND & TECH R.I.C.
A11105 156474
NIST
PUBLICATIONS

U.S. Department of Commerce
National Institute of Standards and Technology
High Performance Systems and Services Division
Scalable Parallel Systems and Applications Group

NISTIR 6022

S-Check, by Example

Robert Snelick



QC
100
.U56
NO.6022
1997

June 1997
Supported by NIST task number 40131 and ARPA task number 7066.

NISTIR 6022

S-Check, by Example

Robert Snelick

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
High Performance Systems and Services Division
Scalable Parallel Systems and Applications Group
Gaithersburg, MD 20899-0001

June 1997

U.S. DEPARTMENT OF COMMERCE
William M. Daley, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Robert E. Hebner, Acting Director

S-Check, by Example

Robert Snelick

Abstract: *S-Check is a software tool for identifying performance bottlenecks in parallel and networked programs. The system uses and incorporates sophisticated statistical techniques and synthetic perturbation for extracting code sensitivities. The methodology demands extensive setup and execution procedures. S-Check automates much of this process. Even still, using S-Check at first glance can be formidable because of its unfamiliar scheme for performance analysis. To alleviate the initial learning curve we present a simple walk through example of how to set up and conduct an S-Check performance analysis.*

Introduction

We describe how to use the sensitivity analysis tool S-Check [3] by working through a simple example. This step-by-step procedure can be used to familiarize yourself with S-Check's basic window layout and flow (Figure 1). You can follow along on-line by going to the example code directory given with the S-Check distribution (release 2.0 and later) [2]. The sample code (a simple quicksort program) is under the *example* directory in the top level of the distribution. Start S-Check while in this directory. As we go through the example, user actions will be outlined in highlighted boxes. It is assumed that the reader is familiar with basic S-Check concepts [3,4,5,6].

Below is a listing of the steps and procedures required by S-Check to complete its analysis of your code. Some steps are trivial, requiring just a simple user response or are completely automated by S-Check. Others involve moderate input and interaction from the user.

- select experiment name (pick a name to identify the experiment)

- identify run environment (*e.g.,* IBM SP2 using LoadLeveler)
- indicate your test code (your application)
- indicate compile and runtime instructions (*e.g.,* include parallel routine library)
- select type of S-Check analysis (*e.g.,* a basic screening of computational segments)
- pick suspect code locations to test (where do you suspect performance bottlenecks)
- define response interval (*e.g.,* overall run time of the program)
- select DEX plan (trade-off between cost and information)
- select number of experiment replication (increase for a *noisy* system)
- select delay value (duration of delay)
- build experiment (compile code with instrumentation)
- run experiment (run all program variants, record data)
- calculate effects (solve set of linear equations)
- view results (list or plot results from previous step)
- tune code accordingly and/or change experiment parameters and return to a prior step

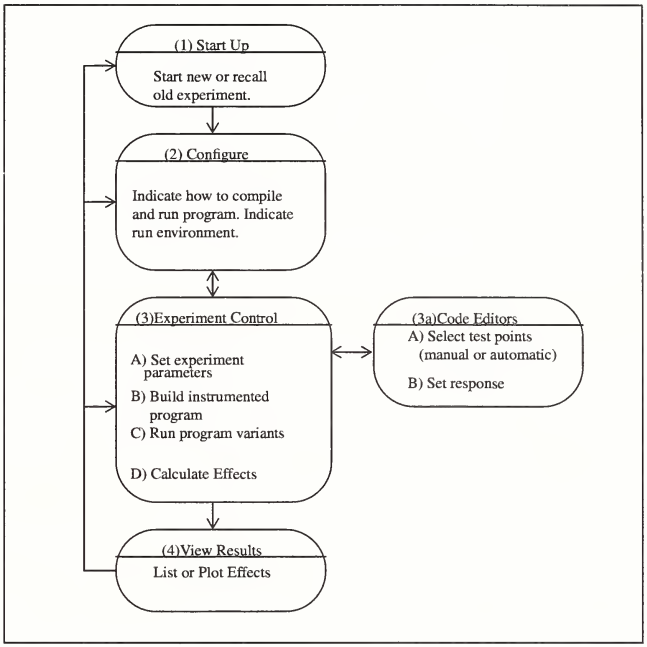


Figure 1. S-Check Window Layout and Flow.

Step-by-Step Example

Start S-Check

Before starting S-Check make sure that it is installed correctly. This includes having S-Check's executables in your path and making sure that S-Check is reading its resource file. See "How to install S-Check" in *Using S-Check* [1]. To invoke the tool just type scheck while in the *example* directory or in the directory in which your code resides.

```
% cd <your_path>/scheck2.0/example
% scheck &
```

Create Experiment

The first window you see is the Experiment List Window (Figure 2). This window allows you to create new experiments and/or recall previously saved experiments. We refer to an experiment as the entire process (static and dynamic aspects) that defines a particular S-Check analysis. To create a new experiment, give it a name and click on Open.

```
> type in test.1 in the New Experiment Name text field
> click on the Open button
```

Configure Experiment

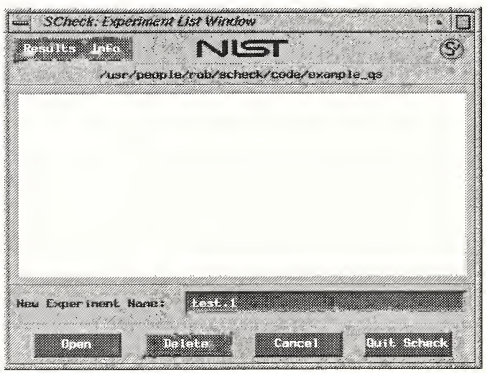


Figure 2. Experiment List Window

Since this is a new experiment, S-Check launches the Configuration Window (Figure 3) where the *platform type*, *test code*, and *experiment type* can be initialized. The Platform Type indicates to S-Check what environment your source code will run on. The Platform Type option menu allows you to select your platform. For simplicity, we demonstrate the tool on a uniprocessor Unix workstation. Therefore, we use the default Platform Type (Unix). This requires no action

by the user. Next, we tell S-Check which code we are going to test. S-Check lists all .c files in the current directory. Here we need to select all the files that will make up the executable program. Drag and hold your pointer over the files listed under the Directory Contents selection window to highlight the files. Click on the Add button to include them in the Work Set. The Work Set contains the list of files that will be edited for the purpose of picking test points. Any flags (or other attributes) that you need to set to compile or run the code are identified next. In our simple example the only parameter that needs to be filled in is the command line Arguments text field. Here we enter 1000000. This instructs the example program to sort 1000000 numbers. We enter command line arguments in the same manner as if we were running the code from the shell.

> leave Platform Type set at Unix and Experiment Type set to Screening
> highlight all files in Directory Contents by dragging the pointer on them
> add the selected files to the Work Set by clicking on the Add button
> enter 1000000 in the Arguments text field
> click on the OK button to exit the Configuration Window

Next we select the Experiment Type. There are 4 options: Screening, Barrier, Communication, and Scaling. In our example, we just want to evaluate the impact of certain suspect computational code segments. Therefore, we keep the default selection (Screening). This will run S-Check basic sensitivity analysis. Clicking on OK

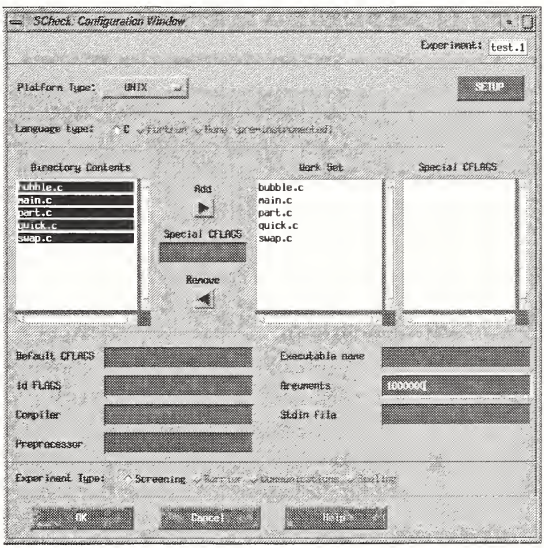


Figure 3. Configuration Window

exits this window and leads us to the Experiment Control Window.

> **> open bubble.c by double-clicking on it in the Experiment Control Window**
> **> move your pointer to line 21 in bubble.c and click on "swap(&list[i], ...);"**
> **> move your pointer to line 10 and click on the code segment "{"**
> **> click on the OK button to exit the editor**

Select Test Locations

Before we can access many of the functions of the Experiment Control Window (Figure 5), we must first select the code segments that we wish to investigate and define the response interval. So we will do that first. One way to select points to screen is to use S-Check's Factor Editor Window (Figure 4). In the upper left hand corner of the Experiment Control Window is the list of Work Set files we previously selected in the Configuration Window. To open a Factor Editor, double-click on the desired file.

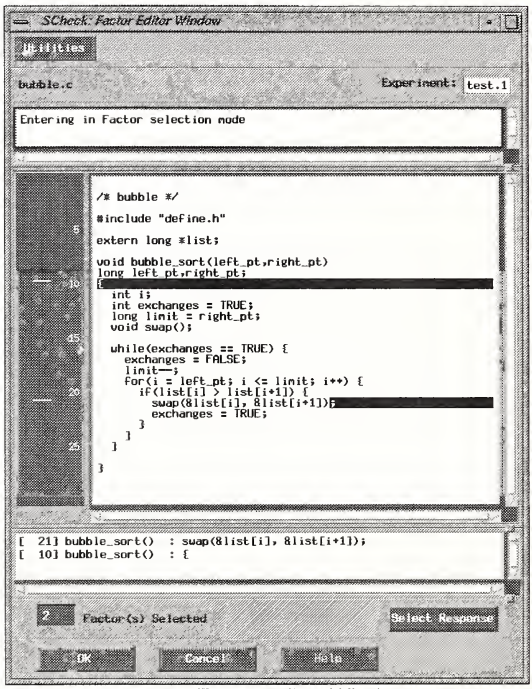


Figure 4. Factor Editor Window

We select test locations (called *factors*) by simply clicking on the desired code segment. This action highlights a section of text where you clicked. The delay instrumentation will be inserted between the left most character and the right most character of the reverse video. Some exceptions to this rule for certain compound statements exist, see Chapter 4 in the user's guide [1].

> **> open part.c by double-clicking on it**
> **> move to line 23 and click on the "more_to_do" variable, the " {" section of the line is highlighted (the body of the while loop will be tested)**
> **> click on the OK button to exit the editor**

Once a factor has been selected, the factor count is updated for the current file that is being edited. A global factor count for the experiment is maintained on the Experiment Control Window (Figure 5). A list of the currently selected factors are shown in a panel under the source code viewer. In our example Factor Editor screen (for bubble.c) we selected two test points. For the entire program we will select three locations (two in bubble.c and one in part.c).

A Factor can be removed by clicking on the location again. The text area will no longer be highlighted in reverse video. Factors can also be selected in groups with the use of S-Check's automatic factor selection utility. See the Profile button under the Utility Menu on the Experiment Control Window. We don't use this feature in our simple case study, but it can be very useful in large programs.

> **> open main.c by double-clicking on it**
> **> click on the Select Response button**
> **> scroll down to line 40 and select it (a B is placed in the annotation column)**
> **> scroll down to line 44 and select it (a E is placed in the annotation column)**
> **> exit the Factor Editor window**

Set Response Interval

Next we need to define the *response interval* which is the section of the code we want to improve. This response measure is usually set to record the run time for the whole program. To define the response interval, we open up a Factor Editor for the file where we want to set it, usually the main driver. When started, a Factor Editor by default is automatically set to factor selection mode (as we have seen previously). To change this mode we use the Select Response button (lower right on the Factor Editor Window). This button is used for setting the Begin (B) and End (E) of the timing interval. Upon clicking on the Select Response button, the cursor changes to a red downward pointing arrow. This symbol indicates that the

Begin can be set (click to mark it). After the Begin location is set the cursor points upward and is ready for defining the End point. Another click sets the End location and completes the response interval.

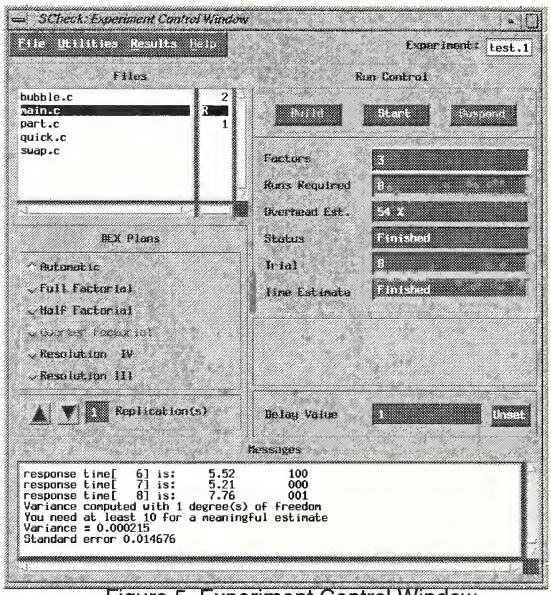


Figure 5. Experiment Control Window

Now that the test points and response interval are defined, we can set up the rest of the experiment. We do this using the Experiment Control Window. Here we can open Factor Editors (as discussed), select a *designed of experiment plan*, set the number of *replications*, and determine the *delay value*. In S-Check, all of these have default settings or can be generated. In our example, we use the default settings.

> click on the Build button of the Experiment Control Window

Build Experiment

In addition, we use the Experiment Control Window to build and start experiments. To build an experiment we click on the Build button. This compiles the program with our instrumentation instructions. Once the program is built, S-Check runs sample tests to determine a suitable delay value. The progress of this phase (as well as other phases) is chronicled in the Messages area (bottom of window).

Start Experiment

Upon completion of this phase, the Start button becomes functional. We begin the experiment by selecting this button. S-Check proceeds to execute

each program variant and displays various status information on the Experiment Control Window. A running trace of each trial run is shown in the Messages area. The response time and the corresponding delay pattern is displayed.

> **> click on the Start button to begin the experiment**
> **> upon completion, select the List Effect button from the Results menu**

View Results

Upon successful completion of the experiment, the List button under the Results menu is enabled. Select this choice to bring up the List Effects Window (Figure 6) to view the sensitivity analysis. S-Check displays the effects for each factor defined in the factor selection phase of experiment initialization. It can also display the interaction effects for these factors (if available). An interaction effect indicates the affect of code efficiency changes on two or more factors together.

An effect reflects the impact the code efficiency changes (the delay) had on the run time of the program. The higher the effect, the more likely the corresponding code segment is a bottleneck. The tuning effort begins at the highest ranked code segment. The significance of an effect is determined by its relative magnitude to other effects and a standard error

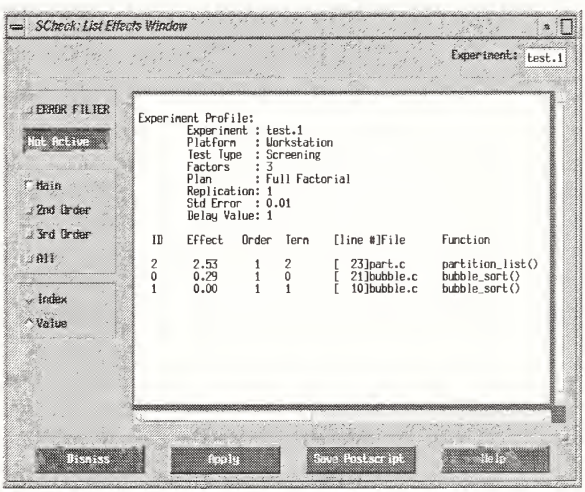


Figure 6. List Effects Window

measure. Effects should be at least 3 standard errors from zero to be considered significant. As shown in the List Effects Window, each effect measure is related back to a corresponding section of code. Double-clicking the right mouse button on a code segment opens a Factor Editor for viewing the code segment in question. Investigation (improvement) of code sections proceeds down the ordered list until a desired performance level is obtained. Alternatively, further exploration of the program can be performed by discarding some or all factors and adding more factors and then retesting. The process can be iterative.

Interpretation of Results

The 3 factors we selected to analyze are for demonstration purposes and correspond to the operations of (1) exchanging elements (using swap()) in a bubble sort routine (2) calls to the bubble sort routine, included here for a baseline measure and (3) comparing elements in a list. A typical result produced by S-Check will look like:

ID	Effect	Order	Term	[line #] File	Function	Text
2	2.43	1	2	[23]part.c	partition_list()	while(more_to_do) {
0	0.25	1	0	[21]bubble.c	bubble_sort()	swap(&list[i], &list[i+1]);
1	0.01	1	1	[10]bubble.c	bubble_sort()	{

Standard Error = +/- 0.02

Of the three elements that we decided to investigate, the main comparison loop (ID=2) of partition_list() is ranked the highest in terms of its overall performance cost. This is to be expected given that one of the central operations involved in the quicksort algorithm is comparing and subsequently exchanging elements to split lists. In comparison bubble_sort() is less costly because it is only used to sort sublists of sufficiently (economically wise) small length. In bubble_sort(), we examined two code areas: swapping elements (ID=0) and the function's entry point (ID=1). As expected (for our data set), the swapping phase's impact on performance exceeds that of the function's entry point impact. In fact, since calling the bubble sort plays a minor role in the algorithm, its (ID=1) influence on performance is barely noticed in S-Check's analysis. In contrast the key component of the sort, the partitioning phase (ID=2), is resoundingly brought to attention.

The simple example presented here describes S-Check's core usage and shows how to apply its performance analysis to tune programs. Although the example is trivial, the same procedure and analysis just as easily applies to complex programs on a variety of parallel and networked architectures. The tool is quite robust--the largest code it has analyzed had 44, 000 lines.

Save Experiment & Results

Our experiment can be revisited at a later or more convenient time by saving the experiment and results. Results are saved under the Save button in the Results menu. Experiment settings are save under the Save button in the File Menu. The use of these mechanisms can aid and log an ongoing performance evaluation of an application.

```
> select Save under the Results menu on the Experiment Control window
> name the file and click on SAVE
> select Save under the File menu on the Experiment Control window
> select Quit under the File menu on the Experiment Control window
```

Exit S-Check

To exit S-Check, select the Quit button under the File menu on the Experiment Control Window.

Conclusion

We illustrated S-Check's multistage process for assessing and identifying performance bottlenecks in parallel and distributed codes. The intention was to introduce the user to the basic S-Check concept and process. For clarity and brevity, many features and options were intentionally omitted. The users guide and related publications can further explain S-Check usage.

Disclaimer

The National Institute of Standards and Technology (NIST) contribution is not subject to copyright in the United States. Certain commercial products are identified in this paper to adequately specify experimental procedures. Such identification does not imply recommendation or endorsement by NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

References

[1] R. Snelick et al., Using S-Check, *NISTIR 5789*, February 1996. (An updated version of this document is available at the URL **http://www.scheck.nist.gov/scheck**).

[2] S-Check's URL and ftp site. URL: **http://www.scheck.nist.gov/scheck**; anonymous ftp site: **ftp:cmr.ncsl.nist.gov**

[3] R. Snelick, S-Check: A Tool for Tuning Parallel Programs. *Proceedings of the 11th International Parallel Processing Symposium (IPPS'97)*, Geneva. April 1-5, 1997, pages 107-112.

[4] G. Lyon, R. Snelick, and R. Kacker, Synthetic-perturbation tuning of MIMD programs, *Journal of Supercomputing* 8(1)(1994) 5-8.

[5] R. Snelick, J. Ja'Ja', R. Kacker, and G. Lyon, Synthetic-perturbation techniques for screening shared memory programs, *Software - Practice and Experience* 24(8)(1994) 679-701.

[6] R. Snelick, M. Indovina, M. Courson, A. Kearsley, Tuning Parallel and Networked Programs with S-Check. To appear in the *Proceedings of 1997 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas, Nevada. June 30-July 3, 1997.