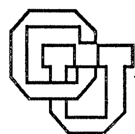
# VISUALIZING THE PERFORMANCE OF PARALLEL PROGRAMS: INTERFACE DESIGN USING TASK-CENTERED WALKTHROUGHS

Casey Boyd, Michael Jones,
Joe Thielen

CU-CS-683-93



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

# VISUALIZING THE PERFORMANCE OF PARALLEL PROGRAMS: INTERFACE DESIGN USING TASK-CENTERED WALKTHROUGHS

CU-CS-683-93 November 1993

Casey Boyd, Michael Jones,
Joe Thielen

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

# VISUALIZING THE PERFORMANCE OF PARALLEL PROGRAMS: INTERFACE DESIGN USING TASK-CENTERED WALKTHROUGHS

Casey Boyd[1], Michael Jones[1], Joe Thielen[2]

[1]Institute of Cognitive Science and
Department of Computer Science
University of Colorado
Boulder, CO 80309
cboyd@cs.colorado.edu
mjones@cs.colorado.edu

[2]U S WEST Advanced Technologies
Product Engineering and Development
Boulder, CO 80303
joe@advtech.uswest.com

## ABSTRACT
Parallel performance debugging is a task that requires large amounts of data to be evaluated by the user in an easy and efficient manner. Consequently, careful attention must be given to the design of a user interface for parallel performance visualization. The user interface should allow the user to identify and correct possible performance problems in a parallel program. We explore the application of task-centered design and the cognitive walkthrough method to this task domain and suggest some limitations of the walkthrough method.

**KEYWORDS:** Parallel performance visualization, parallel performance debugging, task-centered design, cognitive walkthrough, interface design, large-scale concurrency.

## INTRODUCTION
ProVis is designed to guide the user to identify and correct performance defects in programs. This includes letting the user obtain desired information easily, without being overwhelmed with the massive amounts of data created by the parallel program. ProVis takes the massive amount of data in the trace file and presents it in a visual format that lets a programmer recognize patterns and measure system performance.

We designed the system beginning with the user interface. The initial stage consisted of mockups of screen displays. A refinement cycle was iterated several times based on walkthroughs; first by the designers, then by domain specialists.

Our approach to visualizing parallel program execution begins with two displays: the general architecture of the parallel system, figure 1, and a control panel that also displays the source code of the program being analyzed, figure 2. The user may monitor program activity in a node or link by selecting it from the architecture diagram. A programmer running ProVis can get a sense of the overall performance and focus on specific trouble areas. ProVis was designed to enable the programmer to visually detect performance bottlenecks, especially cache thrashing, since this is a common and difficult problem to detect and determine how to resolve.

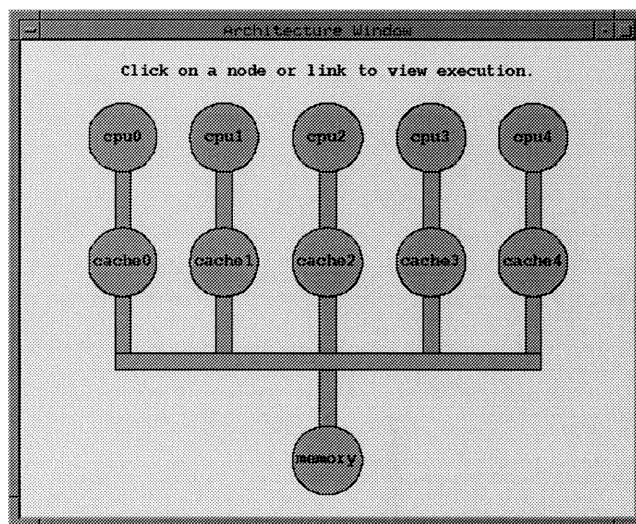



Figure 1: Architecture display.

## THE PROBLEM DOMAIN
Parallel and serial programs encounter efficiency problems when they fail to make good use of the cache. Since access to main memory is slow relative to cache memory access, programs run more efficiently when the data and instructions they need are already located in the cache a large percentage of the time they are requested. When a program's inefficiency is a result of poor cache utilization, it is difficult to determine which program segments are responsible for the inefficiencies. Our system is designed to allow a programmer to visualize a program's access patterns of cache and main memory.

### Other Approaches to Parallel Visualization
Several pioneer parallel visualization systems embedded the visualization code into the parallel program itself. Balsa[1] and SDL[9] use this approach. One of the advantages of this approach is run-time visualization, which allows the developer to monitor a program's performance in real-time. However, several drawbacks

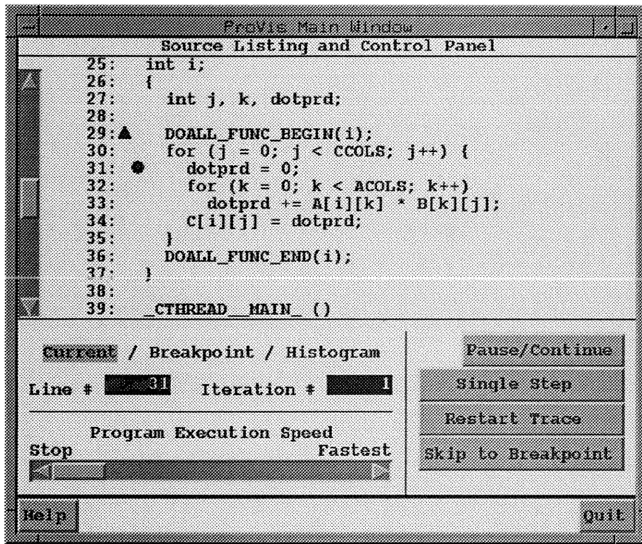



Figure 2: Control panel and source code.

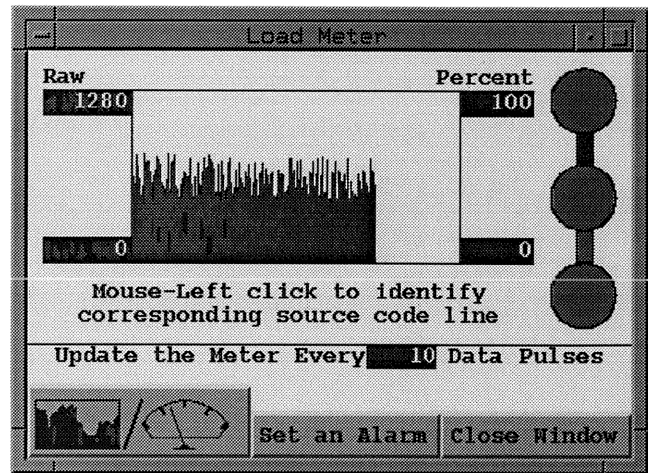



Figure 3: Histogram meter.

to this approach include the need to modify the parallel code, difficulty changing the information that is displayed and its displayed format. [9]

ProVis doesn't have these problems since it uses a trace file that was created when the program last ran. This approach has been used successfully in other parallel visualization systems, such as ParaGraph[3] and TraceView[8]. The trace file contains a description of the system architecture and program run time information. The trace file can be thought of as a script to recreate the program run-time states without running the program again.

### Design Features

If a programmer is going to be productive at debugging performance bottlenecks, certain tools must be available that can help the programmer detect thrashing, stop the trace file analysis when utilization reaches a predetermined level, and tie that to the source code. ProVis was designed to address these specific tasks for the programmer.

A histogram meter, figure 3, can be used to display the traffic or utilization level across some link. It is useful to be able to stop playback of a trace file when bus utilization reaches a predetermined point, such as 95%. ProVis offers an alarm that allows the programmer to halt playback or simply to notify the programmer when a condition is met. This feature can be used to spot trouble areas without having to look over the trace file in great detail.

Thrashing is a common performance bottleneck that programmers deal with often. Thrashing occurs when the CPU is processing a loop of code that repeatedly requests data that is not in the cache. Usually the problem results when the data being processed is too large to fit into the cache. A part of it is loaded and then the CPU requests another part that isn't loaded. The system clears the first part of the cache to accommodate the new data. When repetitive swapping of data into the cache continues, thrashing occurs, hindering system performance seriously. ProVis offers tools to help the programmer see thrashing via the touch trace meters. The meters show the memory being loaded into the cache and displaced from it, when each cache element is accessed. The programmer watches this meter carefully whenever the potential for thrashing exists to see whether parts of the cache are being repetitively cleared.

The features described so far are useful only if the programmer can tie the information to the source code, so that corrections can be made. Performance anomalies show up in the histogram display. When the user clicks on the histogram at the point where the anomaly appears, ProVis indexes into the trace file and highlights the corresponding source code line in the control panel, figure 2.

ProVis also allows the programmer to adjust the speed at which the trace file is played. This gives the ability to look at certain areas of code slowly, so that performance problems that might have otherwise gone unnoticed can be detected and fixed.

## THE DESIGN APPROACH

The design and development of our program performance visualizer, ProVis, has followed certain principles for user interface design. Some of these principles have been suggested by others. Our findings help substantiate their work.

### Principles

**1. Initial designs should be produced and modified quickly.**[4] This principle implies that the design prototype should not require the designer to program. Programming forces the designer to be concerned with details that aren't directly related to the interface design itself. A short design and modification cycle is important so that changes can be quickly tested.

**2. The design process should be task-centered.**[7] One can not produce a useful system without knowing what the system is intended to do. A specific task or set of tasks must be chosen for the system and then all design considerations must be weighed by their contribution (or lack thereof) to the solution of the task. A task-centered approach keeps the designer focused and gives a measuring stick against which all design decisions can be judged for relevance and importance.

**3. All information provided by the system should be relevant to the task.** Any information or control that does not apply to the functionality or purpose of a window, dialog box, the system, etc. should be eliminated. The superfluous information will only serve to confuse and distract the user. The following design principles argue in favor of this:[6, p. 236]

1. Make the repertory of available actions salient.
2. Offer few alternatives.
3. Require as few choices as possible.

We interpret these to mean: 1) all control options appearing in any window should be directly relevant to the information displayed in that window, 2) no window should have more control options than are absolutely necessary to use the contents of the window, and 3) each control option should offer only those selections needed to work on the class of tasks targeted by the system.

**4. The system should guide the user in performing the task or solving the problem.** The system should provide enough information as analytical data and system help to support the user's purpose. This information should guide the user to the solution rather than make it necessary to hunt for or infer it. The system should provide information that would be expected given the task, and the information should be presented in a comprehensible format.

**5. The system should enable new users to learn quickly what features are available and how to use them.**[6] A usable system should have an intuitive feel about it. Users are rarely willing to spend much time studying a user's manual. If the system cannot be understood and used with a little exploration, users are likely to become frustrated and stop using the system. The system should provide ready access to all the information it can provide. There should not be any hidden ways to get desired information. If the system will provide a piece of information, it should be obvious how to get that information.

### Why Task-Centered?

Domain-specificity may not limit the scope of application of a computer system enough when you are designing for expressiveness and flexibility within the constraint of limited complexity. Fulfilling those goals may require narrowing the scope further than the domain level down to a class of tasks. An interface designer can choose several representative tasks and use them to guide the design process. We used specific tasks to design an interface adequate for the chosen task class while deliberately leaving out all system features needed for tasks not in the target class.[7]

By limiting its scope, our interface can present the right information in full measure without overwhelming the user. Also we limited the methods for controlling the program to a reasonable number by offering no more options than were necessary.

Once one makes the decision to follow this approach to interface development and determines the base class of tasks, it is still necessary to refine the interface with user walkthroughs. The walkthrough phase ensures that the interface design will support the relevant solution methods and processes. One might find that the task class is still too broad. Several advantages of a walkthrough method using mock-ups of the interface are that one can design without programming, use a short refinement iteration cycle, and the cost of correcting fundamental design errors is low.

### Preparing for the Walkthrough

To guide our system-building effort we used the cognitive walkthrough method when we designed our system's interface. The cognitive walkthrough method is a procedure for systematically evaluating the features of an interface.[6, 5]

We chose a specific task to focus the walkthrough. It represented the class of tasks for which the new system was targeted. The class of tasks was the debugging of efficiency pathologies in parallel programs, with special attention to the use of memory structures. The specific task was a matrix multiply program whose execution behavior presented slow performance.

Preparation for the walkthrough consisted of sketching out a tentative system design, including an example of every input method and every output display. It was necessary to consider precisely each logical path through the system. How would the display look when the system startup command was given? What buttons, scroll bars, and other mouse-sensitive areas would be needed in the initial display to control the displayed material and to access the rest of the system? What would be the contents of windows called up by such interactions?

Preparing the tentative system design constituted a considerable part of the design effort. However at this point "implementing" was limited to producing a diagrammatic representation on paper. So the consequences of design mis-decisions and their redesign were light. It was possible to change the design quickly in response to iterative refinements.

### Making the Walkthrough Method Work

The mock-up of the interface went through four iteration cycles. The first was based on sketches made on a

whiteboard during an early meeting between the design team and our collaborators. The second was internal to the project team and the last two were performed by our collaborators/users under the direction of the project team. We considered it imperative that the system as mocked up should represent full functionality before presenting it to the users for a formal walkthrough. It should have no gaps to block a user from accessing every portion of the system.

The project designers acted out the user's role in the first and second walkthroughs. Our lack of expertise with the task kept us from seeing needed enhancements and our familiarity with the design made us overlook confusing or troublesome points, limiting the overall value of those walkthroughs. Yet this step ensured a successful first walkthrough with the users because we checked the design for completeness. This was particularly valuable since one dimension we wanted to test with the users was ease of learning the system. Each iteration by the users trained them in the use of the interface, so we were careful not to expose them to it before it was ready.

Our two collaborators are professors of computer science with research interests in visualization of parallel program execution. Owing to this, they brought a high level of commitment to the project. They have substantial experience with exploring and learning how to use new computer systems. These factors and their commitment probably contributed to the success of our walkthroughs. Anyone using this method of evaluating interfaces ought to be aware of this as a potential factor.

Our experience demonstrated that the mock-up is a useful component in the system design cycle. We found that each refinement cycle using the mock-up produced numerous valuable changes to the design. The walkthroughs yielded information that clarified decisions about the organization of the interface and techniques used to seek and display information. We detected where the system was hard to use and why. We learned how the system could increase the value of the information it provides to the users as they work on the task.

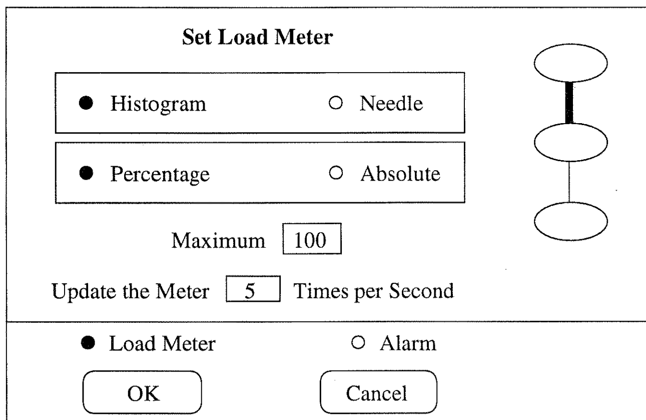



Figure 4: Initial dialog box design.

## DESIGN EVOLUTION

> The most important lesson is the unpredictability of good design: The large number of features of the final design that were not and could not have been anticipated in the initial design. These features were only discovered and incorporated because of the focus on users and user testing in the design process.[2, p. 538]

### Eliminating the Dialog Boxes

An example of the unpredictability of good design is illustrated by our replacing the original dialog boxes, figure 4, with a more direct interaction, figure 3. In the initial design the user would use the mouse to select a component of the architecture to be monitored. A dialog box with all the customization options available was then presented to the user. After setting the desired options, a meter conforming to the selected options replaced the dialog box.

The dialog boxes reduced screen clutter by removing some parameter setting functions from displays that users keep on the screen. During the walkthroughs, the subjects found that the dialog boxes interfered with using the system. They did not want to cancel and then regenerate a meter to change its properties.

The designers responded in two ways. First by eliminating some unnecessary information and interactions contained in the dialog boxes and second by incorporating what remained into the displays that the dialog boxes customized. This turned out to be a complete win as we succeeded in making the interaction simpler, more direct and more customizable without giving up any necessary system features.

### Increasing Direct Manipulation in the Histogram Meters

Several other design enhancements to the histogram meter were suggested during the walkthroughs. One suggestion was the ability to zoom in on a section of the histogram to get a magnified view of the data. Another was to present in the histogram the correspondence between percent utilization and raw traffic count on the bus being monitored. These two ideas resulted in the addition of editable y-axis labels that reflect both the percentage of use and the raw use of the bus. As an example of their use, consider changing the upper percentage label from 100 to 50. The view of the histogram data would be magnified to display the lower half of the histogram in the same visual space. A corresponding change would automatically be made by ProVis in the upper raw data label to reflect 50% of the maximum raw bus traffic.

As the subjects studied the histogram data from a task-centered perspective, they realized that they would be watching for patterns of excessive activity (spikes in the histogram display). Consequently they requested the ability to correlate patterns in the display with lines in

the source code. The new design highlights the source code line corresponding to a position in the histogram selected with the mouse.

### Adding Task-Specific Information to the Cache Touch Trace Meter

A primary goal of the system is to help a programmer detect thrashing during program execution. In the walkthroughs the subjects reported that a high utilization of the bus between the cache and main memory does not necessarily mean that thrashing is occurring. To make this determination the programmer also needs to know the activity in the cache. While the histogram can show a bus utilization rate it can not show cache use patterns. The cache touch trace meter is designed to show access patterns within the cache. In the initial design only accesses by the CPU were shown. However, the subjects realized that besides knowing which data in the cache were being used, they also needed to know which data were being replaced. The touch trace meter was extended to show both kinds of information side-by-side so that the two could be compared.

A comment similar to the following was made during the walkthrough. "Now that I have seen a high cache miss rate, I need to know which variables are responsible." To satisfy this need, the designers added a color-coded filter to the touch trace meter. The user is able to select program variables and display their activity in the cache using different colors. The user is then able to see which variables are consuming large portions of the cache and which are being constantly swapped in and out.

### A Brief Description of Some of the Other Design Changes Suggested by the Walkthroughs

We moved some controls to associate them more closely with the relevant displays. In the first walkthrough we realized that users would simultaneously display several meters of the same type. Multiple histograms might confuse users if there were no way to remember which architectural component was being monitored by which meter. To remind users we added a reduced-size image of the architecture model to each meter with the monitored component highlighted.

Our collaborators suggested ideas for improving the usefulness of breakpoints. One suggestion allowed restarting the trace file and skipping ahead to the first or next breakpoint without updating the meters. Eliminating the display computation and screen updates improves the execution speed of ProVis through the parts of the program that the user does not want to see.

Another suggestion increased control flexibility when setting breakpoints. Thrashing problems often occur inside the body of a loop. Users can set breakpoints by both line number and the number of times the line has been executed in the trace file. This allows large segments of the program trace to be skipped without having to stop on each loop iteration.

## CONCLUSIONS

### Is There an Inherent Limitation in Using a Mock-Up for Design Refinement Iterations - a Point of Diminishing Returns?

> While some aspects of new technology may be difficult to simulate we have never encountered a design problem in which at least some important aspects could not be usefully simulated.[2, p. 534]

A mock-up may not be useful for settling all interface design issues. Some important aspects will not be possible to simulate usefully.

After their second iteration our walkthrough participants perceived an inability to imagine any further how to use the system to solve problems without having a running program to give live results, rather than made-up displays. They felt they would require active and precisely responsive behavior from the system to engage in further evaluation of the interface's usability.

We could have mocked up more thoroughly realistic results showing high fidelity to displays of the sort that would be produced by running the system on a problem. But one obstacle is that we would need an implementation of the system to get such displays. Another obstacle is that we could not have anticipated the problem-solving paths needed by our participants to cover all the possibilities they would want to try in pursuing a solution for the target task.

For ProVis an additional difficulty in the mock-up phase came from the program's feature of displaying temporally visible patterns in aid of the problem-solving process. Paper mock-ups or any other static display could not in principle present the dynamic visual images of time-based patterns that are needed in this problem class to understand how to proceed toward a problem solution.

### Principles Revisited

**1. Initial designs should be produced and modified quickly.** We achieved this goal by using a paper mock-up of the system that allowed our test users to walk through the system and suggest possible changes. These changes were quickly incorporated and the cycle repeated. We have also argued that certain systems reach a point of diminishing returns with this approach. A static display cannot do justice to the temporal patterns of activity that ProVis needs to model. We do, however, advocate the mock-up approach for initial design work because much can be accomplished with this method.

**2. Design should be problem-centered.** We chose a single, specific problem for ProVis. Because we were able to focus on this task, each meter and display was carefully chosen and designed to aid a programmer to detect and correct thrashing problems in a program.

**3. All information provided by the system should be relevant to the task.** We eliminated dialog boxes because they turned out to be an unnecessary layer that our test users found cumbersome. The relevant information and options they contained were more closely linked with the meter. Selecting an architectural component immediately produces the desired meter. The meter attributes can then be modified directly.

**4. The system should guide the user in performing the task or solving the problem.** ProVis provides the ability to monitor cache and link activity by selecting parts of an architectural model. The system correlates meter display activity to the corresponding source code line with the click of a mouse button. The system also lets the user set alarms to detect automatically when activity exceeds a certain threshold. The histogram and other displays are common methods of displaying information and are easily read and understood.

**5. The system should enable new users to learn quickly what features are available and how to use them.** We have attempted to label clearly each system component. The number of choice points in ProVis was streamlined so that a few buttons in each window would provide access to all features and options. We were able to forego the use of menus and integrate all the command options of the system into the information displays. Admittedly this approach may not work for a system with many choices to be made, but it was feasible for our design. Further testing should show whether ProVis can be learned by exploration.

## Summary

ProVis may turn out to be a more useful system because it was designed with a specific task in mind. The information displays are designed to guide the user to correct thrashing problems in a parallel program. The system is designed to ease learning by new users. It supports a strategy of learning by exploration. All information provided by the system is readily accessible and relevant to the task class. We achieved these results primarily by taking a task-centered design approach and by refining the design repeatedly based on feedback from systematic walkthroughs with users.

## Acknowledgments

We would like to thank Professors Dirk Grunwald and Gary Nutt of the CU Computer Science department for their time, support, and valuable contributions to this project. We also thank Tony Sloane for providing the trace files. Professor Clayton Lewis gave us valuable guidance about applying the walkthrough method. We appreciate the helpful criticism by several reviewers of earlier drafts of this paper.

# References

[1] M. H. Brown and R. Sedgewick. A system for algorithm animation. In *Proceedings of SIGGRAPH '84*, pages 177–186. Association for Computing Machinery, July 1986.

[2] John D. Gould and Clayton Lewis. Designing for usability: key principles and what designers think. In R. M. Baecker and W. A. S. Buxton, editors, *Readings in Human-Computer Interaction*, chapter 11, pages 528–539. Morgan Kaufmann, Los Altos, CA, 1987.

[3] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[4] D. Austin Henderson, Jr. The Trillium user interface design environment. In R. M. Baecker and W. A. S. Buxton, editors, *Readings in Human-Computer Interaction*, chapter 12, pages 584–590. Morgan Kaufmann, Los Altos, CA, 1987.

[5] Robin Jeffries, James R. Miller, Cathleen Wharton, and Kathy M. Uyeda. User interface evaluation in the real world: a comparison of four techniques. In *Proceedings of CHI'91*, pages 119–124. Association for Computing Machinery, New York, 1991.

[6] Clayton Lewis, Peter Polson, Cathleen Wharton, and John Rieman. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *Proceedings of CHI'90*, pages 235–242. Association for Computing Machinery, New York, 1990.

[7] Clayton Lewis, John Rieman, and Brigham Bell. Problem-centered design for expressiveness and facility in a graphical programming system. Technical Report CU-CS-479-90, Department of Computer Science, University of Colorado, June 1990.

[8] A. D. Malony, D. H. Hammerslag, and D. J. Jablonowski. Traceview: A trace visualization tool. *IEEE Software*, 8(5):19–38, September 1991.

[9] Gruia-Catalin Roman. Language and visualization support for large-scale concurrency. In *Proceedings of the 10th International Conference on Software Engineering*, pages 296–308. IEEE Computer Society Press, April 1988.