Structured Evaluation of Computer Systems

Günter Böckle
Hermann Hellwagner
Roland Lepold
Gerd Sandweg
Burghardt Schallenberger
Raimar Thudt
Stefan Wallstab
Siemens AG

Evaluating how well a system will perform is difficult because it is seldom done systematically. An approach developed at Siemens makes decisions explicit and the process reproducible.

System development involves deciding which configuration and characteristics will determine a successful design. Suppose you must evaluate cache designs for a new computer system. The project manager might say something like "Determine the best cache for our new computer." You know that you have to evaluate cache systems, but what does "best" really mean? High performance? Low heat? Low cost? Minimum space? Even if you learn that performance is the goal, what technical properties must your evaluation consider? Is it just cache size or cache organization? The replacement and refill strategy? The consistency strategy? Which architectures must you consider? What kind of workload will this computer system have to handle? A cache that offers high performance for computer-aided engineering tasks might perform poorly for text processing.

Even if you perform the evaluation satisfactorily, six months later a customer could ask what a different cache parametrization might yield. Will your documentation reveal why you chose a particular range of architectures and workloads? Why you chose a set of criteria for evaluation? Will you be able to repeat the evaluation and produce the same results? Can you extend the evaluation to different sets of parameters to back up your results?

Problems with evaluation

Evaluating computers and other systems is difficult for a couple of reasons. First, the goal of evaluation is typically ill-defined: Customers, sometimes even designers, either don't know or can't specify exactly what result they expect. Often, they don't specify the architectural variants to consider, and often the metrics and workload they expect you to use are ill-defined. Second, they rarely clarify which kind of model and evaluation method best suit the evaluation problem.

These problems have consequences. For one thing, the decision-maker may not trust the evaluation. For another, poor planning means the evaluation cannot be reproduced if any of the parameters are changed slightly. Finally, the evaluation documentation is usually inadequate, and so some time after the evaluation you might ask yourself, "How did I come to that conclusion?"

Structuring the evaluation process

Our experiences with evaluation projects motivated us to develop a systematic approach to evaluating computer systems. We wanted to develop a methodology that structures the evaluation process, not a general evaluation-problem solver. Our methodology aims to

- Make the entire evaluation process explicit, systematic, and reproducible.
- Decompose the evaluation problem into subproblems.
- Determine the main aspects for each evaluation subproblem: the system to evaluate at that level, the evaluation criteria (metrics), the workload (typical application), and the evaluation method (such as simulation).

- Determine how the subproblems relate to each other and how to combine their results to solve the total evaluation problem.
- Document for each evaluation subproblem what is to be evaluated and why and its characteristics and constraints.

DEFINING AN EVALUATION PROBLEM

A computer-system evaluation seeks to answer particular questions about the system, such as its performance or availability. The customer, designer, or manager must make architectural decisions or assertions about the system. These assertions are usually framed in indistinct, nontechnical terms.

Therefore, the very first step is to analyze the questions, understand the problem, and state the goals and constraints as precisely as possible.

Four evaluation aspects

An evaluation problem has four basic aspects:

- the system or subsystem (*architecture*) under consideration;
- the metrics (*criteria*) to evaluate the system; and
- the requests (*workloads*) the system has to cope with.
- the solution, reached by applying an evaluation method.

ARCHITECTURE. To specify the system's or subsystem's architecture, identify its components (hardware, software, firmware, network) and delineate its boundaries. The system might be a hardware system (like a CPU or a cache), a software system (like a database-management system or file system), or a network (comprising hardware, software, and networking components). However, we've applied the concept to other technical systems as well, for telecommunications, manufacturing, traffic control, and power plants.

The delta chart is the building block for structuring and visualizing the entire evaluation process.

To evaluate a system, define its components, structure, and parameters that affect the evaluation. Parameters can be variable or fixed; either way, you must determine and document their values. For example, when evaluating various cache designs, cache size and organization could be variable parameters, whereas the replacement and consistency strategies could be fixed.

CRITERIA. Evaluation criteria are system properties that help the customer make a decision. The criteria must appropriately and comprehensively capture the performance, quality, or availability of system services. Define the criteria so they correspond with the architecture and workload. Consider the available evaluation methods, because each method can answer only particular questions.

For example, an important criterion in evaluating the performance of different cache designs is the *cache-miss rate*. The miss rate depends on the reference pattern that a specific workload submits to the cache, which differs significantly among applications.

Important criteria for computer systems are performance, dependability, and functionality. Refine and quantify these criteria into metrics so you can use them for comparisons. For example, typical performance metrics are throughput, bandwidth, utilization, response time, and miss rate.[1]

Besides these technical criteria, economic and strategic criteria are also important—in fact, they often determine a product's success. Economic criteria include cost, time to market, sales, and profit. Strategic criteria include standards conformance, compatibility, and new market experience. A thorough, well-structured evaluation process can also assess a system's nontechnical criteria.

WORKLOAD. Workload denotes representative tasks or service requests, which stimulate the components relevant to the study and characterize the system's applications concisely. Obtaining the cache-miss rate, for example, requires an appropriate description of an application's reference pattern (the number of different addresses referenced, the length of the reference string, and the distribution of references across the address range, among other things).

There are four major classes of load models:[2]

- *Paper-and-pencil benchmarks*, which denote comprehensive, high-level, implementation-independent specifications of tasks a system should perform. The TPC benchmark specifications[3] for transaction processing are an example.
- Benchmark *programs*,[4,5] which consist of standardized source code like the SPEC benchmark suites.
- *Traces*, which denote collections of system requests, such as sequences of memory references.
- *Distributions and parameters*, which characterize workloads and are used mainly in conjunction with analytical models.

Our experience shows that people often approach this workload modeling unsystematically, which can yield inadequate workload models. Systematic load modeling is an important research area that demands more study.[1,6,7]

METHOD. The evaluation method you choose can employ analytical modeling, simulation, system measurement, or a combination of these techniques.[1] To determine which technique to use for a first, coarse specification, try partitioning the problem recursively into subproblems until the subproblems are small or detailed enough to solve with a single technique. In general, solving a subproblem means determining criteria values by solving or running the architecture/workload model with specific architecture/ workload parameters.

Delta charts

Our structured approach is similar to the systematic approach to performance evaluation that Jain proposed.[1] However, we base our evaluation method on a graphical visualization scheme called the delta chart, shown in Figure 1. The delta chart is the building block for structuring and visualizing the entire evaluation process.

A delta chart depicts the three aspects of system evaluation as the three edges of a triangle. On the architecture edge, dots mark specific system components and parameters (or parameter values, depending on the level of evaluation detail). On the criteria edge, dots denote the criteria or metrics to be determined (again, their level of detail corresponds to the other problem aspects). On the workload edge, dots list service requests, request parameters, or specific system requirements. These range from application or benchmark programs to parameters of synthetic workloads. The evaluation method is written inside the triangle. To simplify the notation, the type of model—Petri-net, for example—can replace the modeling and evaluation technique.

You must specify each edge of the delta chart precisely and comprehensively, in correspondence with the other edges. Visualizing an evaluation problem like this supports understanding and partitioning of the problem, illustrates alternatives, documents the work done, and facilitates communication. We recommend that both the customer and the evaluating engineer approve the initial delta chart that defines the problem. This ensures that you're solving the right problem and improves the likelihood that the decision-maker will accept the results.

Most likely, one delta chart will not suffice to formulate and solve the evaluation problem for a complex system like a multiprocessor. You must decompose the problem recursively into ever smaller and more tractable subproblems, represented by more detailed delta charts.

EVALUATION PROCESS

In our method, the evaluation process consists of two basic sets of activities: Create the evaluation graph, and then use it to perform the evaluation itself. To make this process easier, we created a Motif-based graphical editor. We hope the easy-to-use interface will encourage people to adopt systematic evaluation.

Creating the evaluation graph

Creating an evaluation graph has four steps:

- create an initial delta chart,
- refine that chart,
- determine system parameters, and
- combine, or *generalize*, delta charts to reduce evaluation time.

These steps help document the evaluation so that we can repeat it using different conditions, as well as defend the results.

INITIAL DELTA CHART. Begin the evaluation by specifying the goals—the result you want to derive and the constraints you must consider. Then formulate the evaluation problem in the initial delta chart according to the three main aspects: architecture, criteria, and workload. Figure 2 shows an example in which the evaluation problem is the performance of a multiprocessor system that will run database and office applications.

REFINING THE CHART. To perform the evaluation, we must refine each edge of the chart in Figure 2. The architecture specification, "multiprocessor," is rather general. Analyzing all features of a multiprocessor that can influence performance is a costly, unmanageable task. We start the chart refinement by specifying which architectural properties we actually need to measure or model.

In this example, we can separate the multiprocessor architecture into three parts: processors, caches, and buses. The other parts of the architecture, such as I/O, memory, and software, are fixed. The processor type is also fixed, but the number of processors is an architectural property that the evaluation should consider. Thus, refining the first triangle yields the chart in Figure 3. Next we refine "cache architecture." The example represents cache architecture by two main properties, "cache size" and "associativity." This produces a new delta chart, shown in Figure 4a. We refine cache size further, as Figure 4b shows, by listing cache sizes of 1, 2, and 4 megabytes. We refine associa-

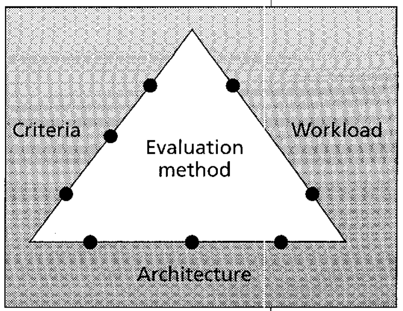


Figure 1. Delta chart template. A delta chart is a graphical visualization scheme for structuring and visualizing the entire evaluation process.

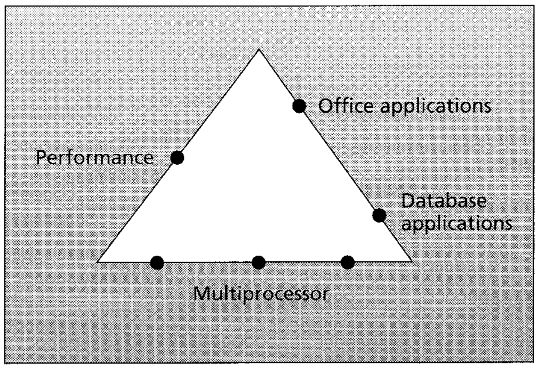


Figure 2. Initial delta chart for the multiprocessor example.

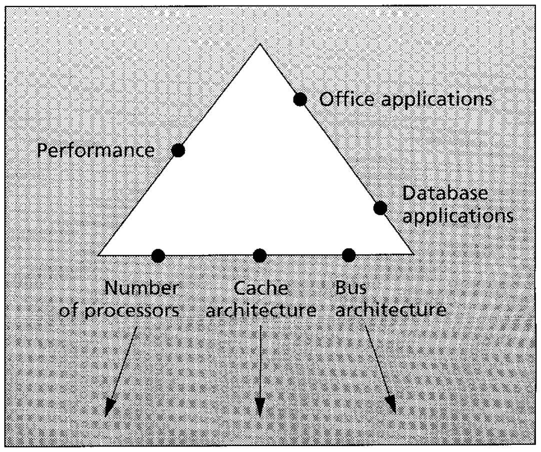


Figure 3. Refined multiprocessor delta chart.

tivity by listing the number of sets (two, four, or eight) (Figure 4c).

We can refine more than the architecture. The "performance" attribute on the criteria edge of our triangle is quite general. Here we show "system performance" and "bus throughput," although our example will consider only system performance. The example represents the workload for office and database applications by benchmarks for each of these kinds of processing. Figure 4d depicts all of this.

The evaluation process corresponds to a bottom-up walk through the evaluation graph.

These examples demonstrate *concretization,* a particular kind of refinement. Concretization can separate an architecture (or criterion, or load) into one or more submodules. Alternatively, it can realize an evaluation aspect more specifically. We continue the concretization refinement until we get delta charts that we can solve by applying a single evaluation method.

DETERMINING PARAMETERS. To determine system performance, the criterion for our evaluation goal, we use a Petri-net cache model. However, for Petri nets, the load cannot consist of benchmarks themselves. Instead we must represent the load by parameters that characterize the benchmarks. Such parameters are *average memory request length, average interrequest time* (the average time between two memory accesses), and *cache-miss rate*. This further concretization leads to the delta chart in Figure 4e, annotated with the method to be applied, "Petri-net model."

To derive the parameters for Petri-net analysis for these applications, the single process and the single processor on which each application runs determine request length and interrequest time. Therefore, we can find the parameters by measuring them on a single-processor system running the actual benchmarks. This measurement, an evaluation problem itself, appears as the delta chart in Figure 4g.

The kind of refinement we applied to create Figure 4g differs from the previous ones. So far, the left edge has become the left edge of the succeeding triangle after a refinement. This time, the right edge, with the cache-load parameters, switched to the left side. The new right edge shows measures of new criteria. This kind of refinement is called *parameter determination*. The delta chart in Figure 4g needs no further refinement. We can solve it as a straightforward evaluation by applying the method "measurement." However, the measurement task might not be so simple. We need experts to perform these measurements, a tool to get the desired data, and benchmarks that characterize the applications.

A third load parameter, the cache miss rate, remains from the delta chart in Figure 4e. An expert can derive this parameter by modeling the cache and evaluating its performance. This example uses the Petri-net modeling method. Again, we observe a change in the triangle's edge from 4e to 4f: The load in 4e becomes the measure of the criterion determined by the next step. This chart employs the analytical cache-modeling method. We must then adapt the load representation to the kind of model. Figure 4f specifies the load with the following parameters: number of addresses, reference string length, and static and dynamic sequence length.

Now we face the same problem as before: The load consists of parameters we cannot yet measure. However, for the applications in the example, these parameters depend mostly on the *kinds* of processors used, not the number. Thus, we can determine these parameters by measuring them on a single-processor system running the benchmarks that represent the applications.

GENERALIZING CHARTS. That last step gives us a delta chart, Figure 4h, that needs no further refinement. We can perform the measurement as a straightforward evaluation problem, as above. Delta charts 4g and 4h differ only in one edge, the criteria. They both represent measurements on the same processor using the same kind of load. Therefore, we can combine both delta charts into a new one, reducing the work we need to do. For the resulting delta chart 4i, we need only one set of measurements for both sets of criteria on the left edge.

Joining two delta charts that differ only in one edge by combining the items on the differing edges is called *generalization*. This combination reduces the cost of our evaluation by avoiding multiple applications of evaluation methods.

EVALUATION PROCESS RULES. As the preceding example illustrates, there are the following rules for developing an evaluation process.

- The basic evaluation graph starts with an initial delta chart that uses architecture, evaluation criteria, and workload as its edges.
- We then refine each delta chart in the graph by creating new triangles until we get charts that we can process as simple evaluation problems. Each delta chart represents an evaluation subproblem. An evaluation method written into the triangle characterizes these subproblems.

Our approach uses three kinds of refinement to produce an evaluation graph:

- *Concretization* copies a delta chart to one or more new ones by changing one edge. The technique replaces a part by its major subparts or subfunctions. For example, we can concretize "cache" by "cache size" and "cache associativity." Bifurcating the evaluation graph denotes cases where we evaluate these parts separately.
- *Parameter determination,* as in our example, spawns successors for a delta chart by copying its right edge (load parameters) to their left edges (the criteria).
- *Generalization* combines two delta charts that differ only in one edge into a new chart. The union of the items attributed to the differing edges becomes the new delta chart's corresponding edge.

Of course, the evaluation graph is not fixed indefinitely. We can change it whenever a particular solution turns out to be infeasible. Documenting each decision by noting the reasons for choosing each refinement or generalization supports an important goal of our evaluation methodology: The graph makes the whole evaluation process repeatable and plausible.

Performing the evaluation

Once we have created the graph, the actual evaluation begins. The evaluation process corresponds to a bottom-up walk through the evaluation graph, from the leaves to the root. The graph's leaves contain evaluation methods we apply by building and evaluating models, whether analytically or by simulation. First, we process all leaves by applying their methods. Next, we process the leaves' predecessor nodes. When a leaf is a generalization of several predecessors, we simply divide the results into subsets. In the example (Figure 4i), we divided the measurement results into two subsets: "number of addresses, reference string length, static sequence length, dynamic sequence length" and "average memory request length, average interrequest time." This solves the evaluation task represented by such a delta chart.

To determine a predecessor's parameters, we can insert values derived by evaluating a leaf into the corresponding edge of the preceding delta chart. Then we can solve the problem the preceding delta chart represents by applying the corresponding evaluation method (see Figure 4f).

When a delta chart is a concretization of a predecessor, we must combine the results of all this predecessor's successors. This can mean just noting the results, such as when we concretize one application into several workload representations. This combination can also mean a complicated analytical composition, for example, combining performability measures of different parts. Some cases, particularly in their uppermost triangles, produce a set of alternative results, such as performance curves for architectural alternatives.

Supporting evaluation graphically

To promote dissemination and acceptance of systematic evaluation in industry, we have developed a graphical editor for evaluation graphs. The Motif-based tool, called Graphedit, lets users conveniently construct, annotate, modify, display, print, and store evaluation graphs.

In addition to the usual graphical editor functions, Graphedit has the following notable features:

- You can annotate individual delta charts and arcs, as well as entire subgraphs, with keywords, short texts, and

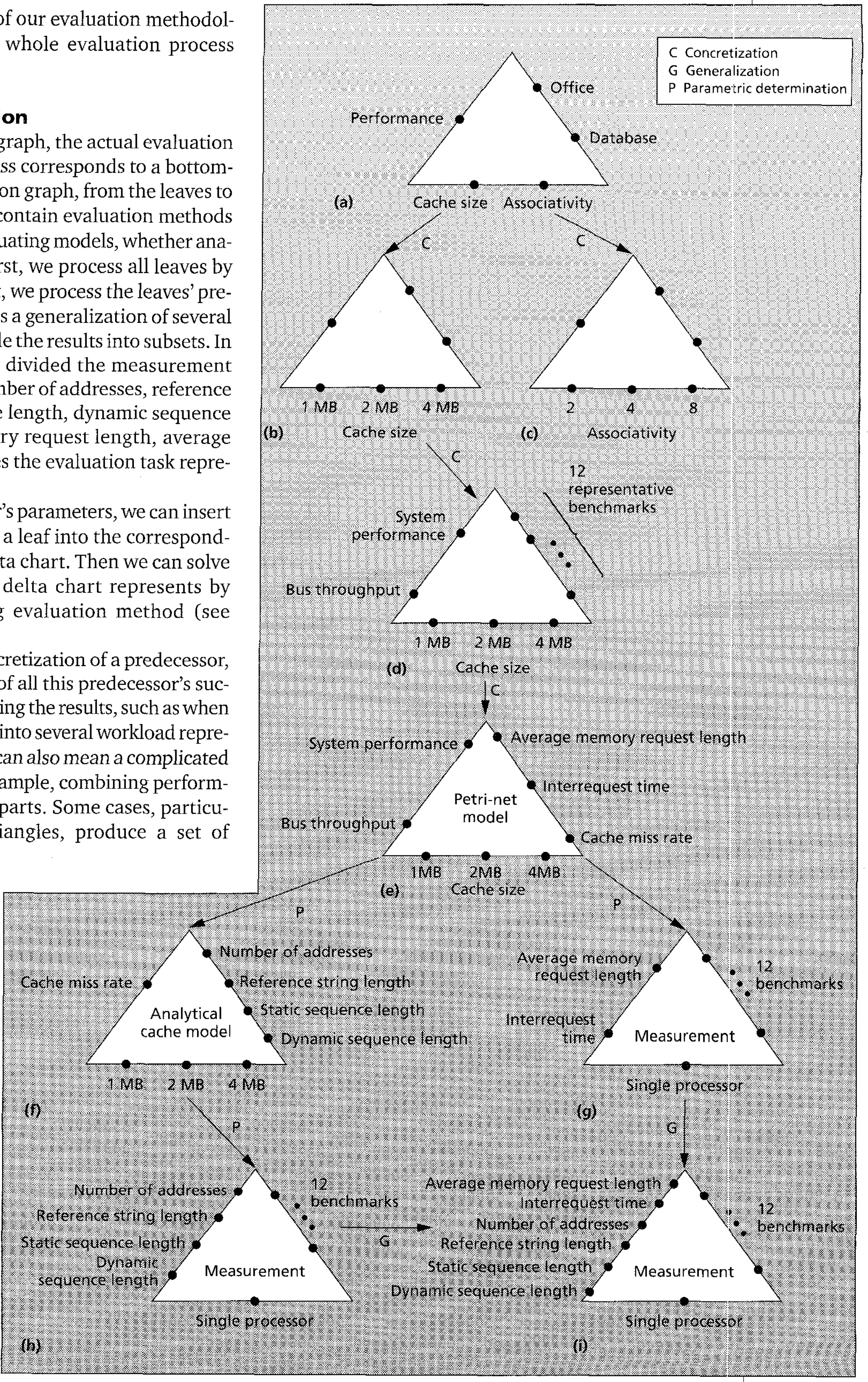
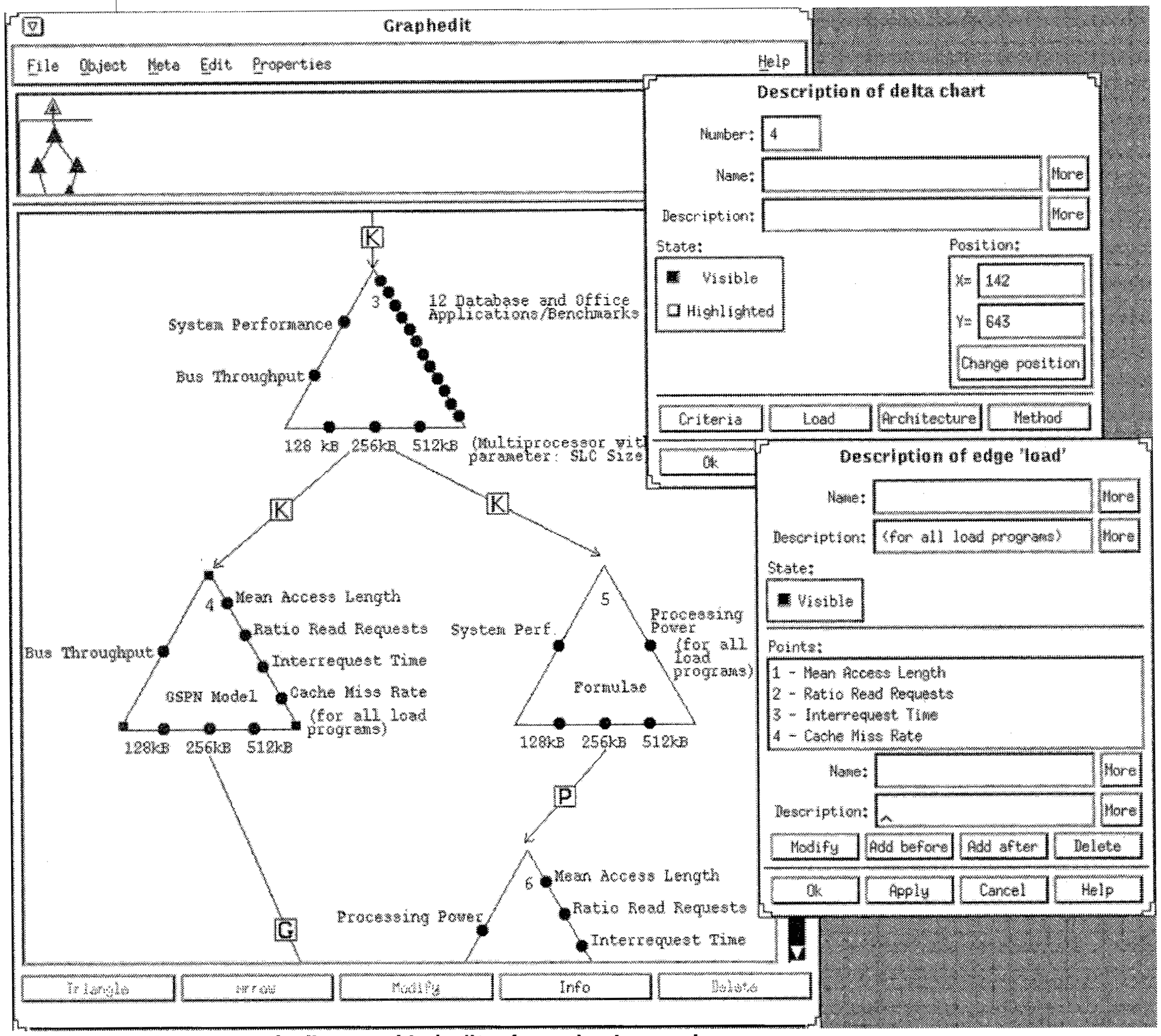


Figure 4. Multiprocessor evaluation graph. (a) Delta chart refining cache architecture; decomposed by (b) cache size and (c) associativity; (d) workload measured by application benchmarks; (e) parametrized for Petri-net analysis; (f) *determining the cache-miss rate; (g) determining the other load parameters for* the Petri-net model; (h) determining cache workload parameters; (i) generalizing 4g and 4h for measurement.


Graphedit
File Object Meta Edit Properties Help
K
3
12 Database and Office Applications/Benchmarks
System Performance
Bus Throughput
128 kB 256kB 512kB (Multiprocessor with parameter: SLC Size)
K
K
4
Mean Access Length
Ratio Read Requests
Interrequest Time
Cache Miss Rate
(for all load programs)
Bus Throughput
GSPN Model
128kB 256kB 512kB
5
System Perf.
Processing Power (for all load programs)
Formulae
128kB 256kB 512kB
P
G
6
Mean Access Length
Ratio Read Requests
Interrequest Time
Processing Power
Triangle Arrow Modify Info Delete

Description of delta chart

Number: 4

Name:

Description:

State: ■ Visible □ Highlighted

Position: X= 142 Y= 643

More More Change position Criteria Load Architecture Method Ok

Description of edge 'load'

Name:

Description: (for all load programs)

State: ■ Visible

Points:
1 - Mean Access Length
2 - Ratio Read Requests
3 - Interrequest Time
4 - Cache Miss Rate

Name:

Description:

More More More More Modify Add before Add after Delete Ok Apply Cancel Help

Figure 5. Graphedit, a graphical editor for evaluation graphs.

comprehensive texts. For instance, you can use this feature to explain certain design decisions.

- You can display or hide texts, either for each item (a delta chart, chart edge, or arc between charts) individually, or for all charts (or arcs). This lets you document the evaluation process in close detail, while keeping the corresponding graph easy to survey.
- Again for readability's sake, you can collapse subgraphs into, and expand them from, single *meta charts* represented by triangular symbols.
- You can survey and rapidly navigate a large, complex evaluation graph using a sum-up view. You can still perform simple editing functions on this representation.
- You can print the sum-up view, selected objects or pages, or the whole graph, sized up or down as desired. A simple page-preview facility is available as well.

Figure 5 shows how Graphedit appears to the user. The tool runs on Unix-based workstations and personal computers. Contact us for more detailed information.

WE HAVE USED THIS METHOD to evaluate mainly computer systems and communication systems design, including multiprocessor systems, a parallel file system, and an ATM network. Now we are disseminating the method within Siemens.

You can use the method for evaluating all kinds of technical systems, and we believe you could adapt it to the analysis of social systems and organizations as well. To confirm this, we evaluated the "performance" of our own organization, Siemens Corporate R&D. In addition to achieving the central goals of structuring and improving the evaluation process, the method turned out to facilitate communication among the people involved. Because of its requirement for explicitly formulating evaluation goals and aspects, the method supports focusing on the essentials and avoiding misunderstandings. Altogether, it helps reduce the time that complex evaluations take. ▌

References

1. R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, New York, 1991.
2. H. Hellwagner, "Workload Classification," *Proc. Workshop on Workload Modeling*, Universität der Bundeswehr, Neubiberg, Germany, 1994 (in German).
3. "TPC Benchmark Descriptions," Transaction Processing Performance Council, San Jose, Calif., 1996; http://www. tpc. org/.
4. "SPEC Newsletter," Standard Performance Evaluation Corp., Manassas, Va., 1996; http://www.specbench.org/spec/.
5. R.P. Weicker, "An Overview of Common Benchmarks," *Computer*, Dec. 1990, pp. 65-75.
6. M. Calzarossa and G. Serazzi, "Workload Characterization: A Survey," Proc. IEEE, Aug. 1993, pp. 1,136-1,150.
7. P. Heidelberger and S.S. Lavenberg, "Computer Performance Evaluation Methodology," *IEEE Trans. Computers*, Dec. 1984, pp. 1,195-1,220.

Günter Böckle *is a research manager at Siemens Corporate Research and Development. His technical experience includes modeling, simulation, assessment, and design of various systems, including communications protocols, processor and system architectures, and compilers. He developed methods and tools for automatic parallelization and is currently working in the field of systems engineering. He is the author of* Exploitation of Fine-Grain Parallelism *(Springer Verlag, Berlin and New York, 1995). Böckle received a Diplom in mathematics and a doctorate of science, both from Stuttgart University.*

Hermann Hellwagner *is an associate professor of parallel computer architecture at the Technical University of Munich. He was previously with Siemens Corporate Research and Development, where he participated in the research this article reports. His current research interests are in parallel processing, particularly distributed shared-memory systems, and the performance evaluation of computer systems. Hellwagner received an MS and a PhD in computer science, both from the University of Linz, Austria. He is a member of the IEEE Computer Society. His home page is http:// wwwbode.informatik.tu-muenchen.de/~hellwagn/.*

Roland Lepold *is a department head and product manager at the division for Automation Systems for Machine Tools, Robots, and Special Machines Division of Siemens in Erlangen, Germany. He has done research in system modeling and evaluation techniques at Siemens Corporate Research and Development in Munich. His research interests are fault-tolerant architecture design and reliability and performance modeling. Lepold received a Diplom in computer science from the University of Karlsruhe and a PhD in computer science from the Polytechnic Research Institute of the University in Mulhouse, France.*

Gerd Sandweg *is head of the development department at INCA, Industrial Computer Application. Formerly, he headed the Systems Technology Department at Siemens Corporate Research and Development. His interests are computer architecture, integrated circuit design and testing, and systems architecture. He is the coauthor of* Selbsttest digitaler Schaltungen *(Oldenbourg Verlag, Berlin, 1990) on the self-test of digital circuits. Sandweg received a PhD in electrical engineering from the Technical University of Munich.*

Burghardt Schallenberger *heads the Microcomputer Components Business Unit of the Siemens Semiconductor Group in Munich. At Siemens Corporate Research and Development, he worked on chip, computer, and system architectures. He also lectures in computer architecture at the University of Heidelberg. Schallenberger received a Diplom in physics and a PhD in natural sciences, both from Rheinisch-Westfälische Technische Hochschule (RWTH), Aachen, Germany.*

Raimar Thudt *is a research engineer at Siemens Research and Development, working on performance evaluations of computer systems and networks. His current research topics are in communication systems analysis and design, focusing on ATM networks, and in high-performance simulation techniques and tools, focusing on parallel and distributed simulation. Thudt received an MS in computer science from the Technical University of Munich.*

Stefan Wallstab *is a research engineer at Siemens Corporate Research and Development. He has worked on several VLSI chip architecture and design-automation projects. His current research interests focus on hardware architectures for information security, VHDL-based architecture and logic-level synthesis, and FPGA-based ASIC emulation. Wallstab received a Dipl.-Ing. from the Technical University of Munich. He is a member of Verein Deutscher Ingenieure, the Association of German Engineers.*

Address questions about this article to Böckle at Corporate Research and Development, Siemens AG, D81730 Munich, Germany; e-mail Guenter.Boeckle@zfe.siemens.de. For information about Siemens, visit http://www.siemens.de.