A Study of the Clarity of Functionally and Structurally Composed High-level Simulation Models

Manish Vachharajani
manishv@ee.princeton.edu
Department of Electrical Engineering

David I. August
august@cs.princeton.edu
Department of Computer Science

1 Introduction

Simulation is a widely used tool in evaluating and guiding high-level hardware design decisions. As a result, the accuracy of the simulation model is critical; a poor model can result in poor design choices. Since most simulation models are complex, some form of validation is needed to ensure this accuracy.

The most common validation technique is that of checking the simulation model by hand. Clearly, the reliability and efficiency of this technique is closely tied to how easy it is for someone to understand (i.e. the clarity of) the simulation model. This is especially true for validation by a third party. The clarity is also critical if third parties are to correctly modify the simulation model for their own purposes.

This technical report presents the results of an experiment that compares the clarity of models specified with a structural specification system (in particular, LSE [3]) versus a simulation model written in a sequential programming language (i.e. a sequential simulator).

The organization of the paper is as follows. Section 2 gives a brief description of how sequential simulators differ from structural models. Section 3 describes the overall experimental setup. Section 4 presents the results of the experiment.

2 Background

To manage the design of complex hardware, designers divide the system's functionality into separate communicating hardware components and design each individually. Since each component is smaller than the whole, designing the component is significantly easier than designing the entire system. If components are too complex, they too can be divided into sub-components to further ease the design process. The final system is built by assembling the individually designed components. To ensure the components will interoperate, designers, when dividing the system, agree on the communication interface of each component. This interface, which defines what input each component requires and what output each component will produce, encapsulates the functionality of each component; other parts of the system can change without affecting a particular component provided its communication interface is respected. We call this type of encapsulation and communication *structural composition*.

Leveraging encapsulation to allow this divide-and-conquer design strategy is also very common in software design. Sequential programming languages such as C or C++ use functions to encapsulate functionality. Each function has a communication interface (its arguments and return value) and this interface encapsulates the functions behavior. Software systems are built by assembling functions which communicate by calling one another and passing arguments and receiving return values. We call this type of encapsulation and composition *functional composition*.

The presence of encapsulation combined with designer familiarity and tool availability make sequential programming languages a popular tool with which to model hardware systems. However the encapsulation permitted by functional composition in sequential languages is not the same as the encapsulation provided by structural composition. This mismatch forces designers to *map* their structurally composed hardware designs to functionally composed sequential programming languages. This mapping leads to simulators that are difficult to understand [4].

Table 1: Subject responses to the questionnaire.

Subject	Years in Architecture	Wrote a C Simulator	Wrote RTL for a CPU Core	Years Experience w/ C/C++	Days Experience w/ LSE	Used Simple-Scalar
S1	3	Yes	No	5	3	Yes
S2	10	Yes	Yes	15	2	No
S3	3	No	Yes	5	2	No
S4	3	No	No	6	3	Yes
S5	3	No	No	7	3	No
S6	3	Yes	Yes	6	3	Yes
S7	3	No	Yes	8	3	Yes
S8	3	No	No	5	4	No
S9	2	No	No	14	5	No
S10	6	Yes	No	10	5	Yes
S11	7	No	No	7	2	No
S12	3	Yes	Yes	2	2	Yes
S13	7	No	Yes	10	2	No
S14	2	No	No	10	2	No
S15	1	No	No	10	2	No
S16	2	Yes	No	8	2	Yes
S17	6	No	No	6	2	No
S18	3	No	No	4	2	Yes
S20	3	No	No	5	2	Yes
S21	2	Yes	No	6	2	Yes
S22	2	Yes	No	6	2	Yes
S23	1	No	Yes	10	21	No
S24	4	No	Yes	6	7	No

The experiment presented in the next section measures how much harder it is to understand a functionally composed model versus a structurally composed model.

3 Experimental Setup

The experiment in this technical report measures the clarity of simulation models by determining how easy it is for subjects to identify properties of the hardware modeled by a simulation model. A group of subjects was asked to examine sequential simulator code modeling a microprocessor and identify properties of the machine modeled. The subjects were asked *exactly* the same questions for a model of a similar machine built in the Liberty Simulation Environment (LSE). LSE [3] is a hardware modeling framework in which models are built by connecting concurrently executing software blocks much in the same way hardware blocks are connected. Thus, LSE models closely resemble the hardware block diagrams of the machines being modeled. We call these structurally composed models *structural models* to distinguish them from functionally composed *sequential simulators*.

The sequential machine models were a collection of modified and unmodified versions of `sim-outorder.c` from version 3.0 of the popular SimpleScalar tool [2]. `sim-outorder.c` models a superscalar machine that executes the Alpha instruction set. The LSE models were variations of a superscalar processor model that executed the DLX instruction set. A refined version of this model was released along with the Liberty Simulation Environment in the package `tomasulodlx` [3].

To ensure that the experiment measured the quality of the models, not the knowledge of the subjects, all the subjects were either PhD students studying computer architecture or Ph.D. holders whose primary work involved computer architecture. To ascertain the background of subjects, each was given a questionnaire to determine their familiarity with computer architecture, programming languages, and existing simulation environments, in particular, SimpleScalar. A summary of the answers for this questionnaire is in Table 1.

The questionnaire itself appears in Appendix A.

Note that finding qualified subjects unaffiliated with the Liberty Research Group for this experiment was challenging given the requirements. Subjects had to be very familiar with computer architecture and also had to be familiar with LSE. Though growing in popularity, LSE is still a young system, making the second constraint the most difficult to satisfy. Only 24 of all LSE users could be recruited as subjects for this experiment. As will be seen, however, even with 24 subjects the results are quite dramatic.

Each subject was asked 2 multi-part questions for the sequential SimpleScalar model and the same questions for the structural LSE model. The multi-part question for the sequential model was asked immediately after the same question for the structural model. The questions are shown in Appendix B. The appendix shows each question along with a short description of the intent of the question. The text describing the question intent was not presented to subjects.

For the different machine models, the questions may have had different answers. There were 4 sequential simulator models:

- Stock `sim-outorder.c` which has a unified issue window and flags mispredicted branches in the decode stage, begins recovery in the write-back stage. Note that flagging mispredicted branches in the decode stage involves emulating instructions at decode which provides results that would not be available in hardware. This information is used to prevent recovery for mispredicted branches that will be squashed.
- A modified `sim-outorder.c` which had a unified issue window, but identified and resolved mispredicted branches in the commit stage, in-order.
- A modified `sim-outorder.c` which had a unified issue window, but identified and resolved mispredicted branches in the write-back stage, supporting recovery of mispredicted branches that would eventually be squashed.
- A modified `sim-outorder.c` which had a distributed set of reservation stations and the stock `sim-outorder.c` branch semantics.

There were 2 structural models built in LSE:

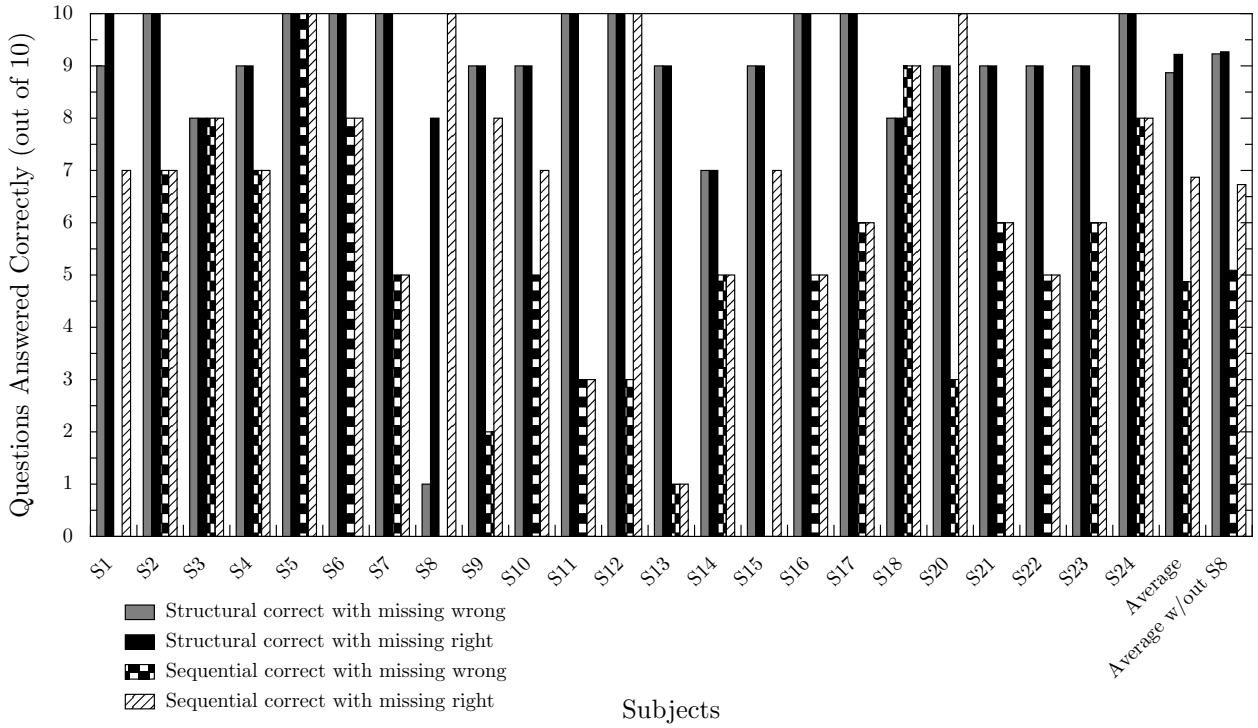
- A machine with a unified set of reservation stations with mispredicted branches identified and resolved at commit.
- A machine with a distributed set of reservation stations with the above branch resolution policy.

The base LSE model was built for instructional and demonstration purposes and thus the machine was modeled at a fairly low-level and closely resembles the hardware. This made the model an excellent reference point for clarity. There are fewer LSE models than sequential simulators because the low-level of the base LSE model (the machine actually computed results in the pipeline instead of pre-computing the results in the decode stage) made the construction of models that used data that would be unavailable in hardware difficult.

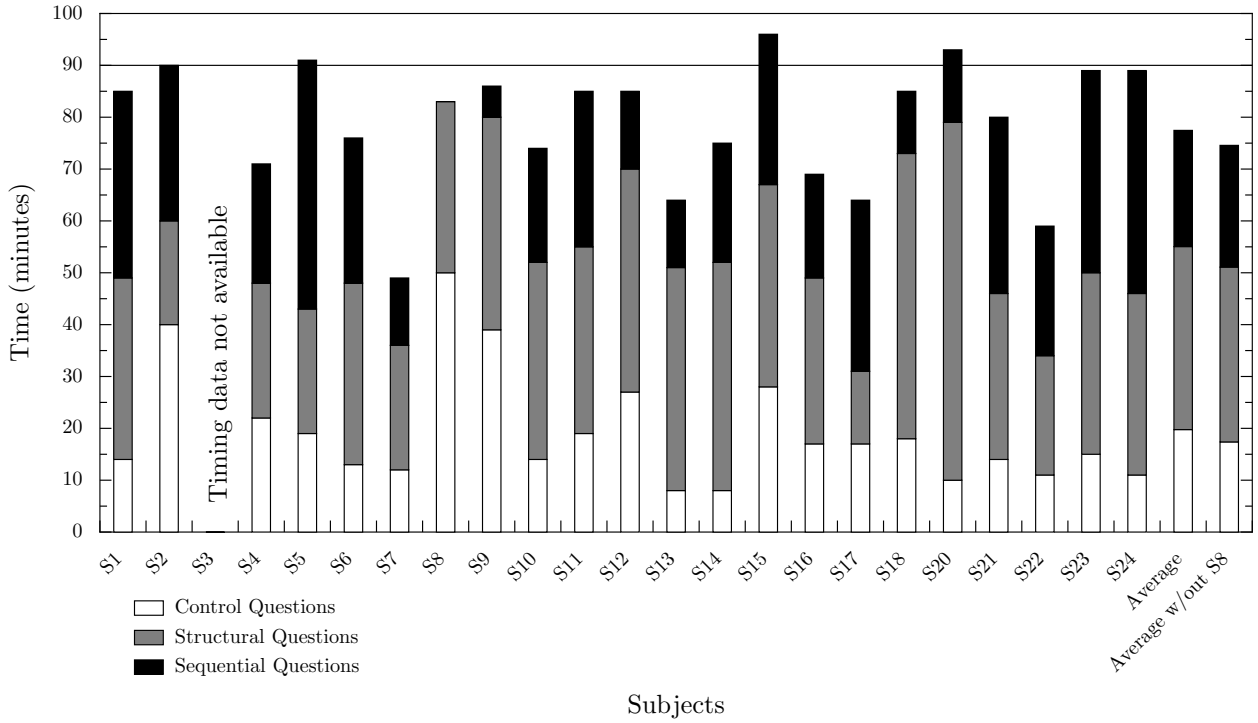
Subjects were given 90 minutes to answer 2 control questions (shown in Appendix B as B.1 and B.2) and 2 questions for the structural model and the sequential simulator (Shown as B.3 and B.4). The control questions were used to determine if the subjects had basic familiarity with LSE since each had less than one week of experience with the tool. The answers to the control questions indicated that all subjects, except subject S19, understood LSE enough for the purposes of this experiment. Accordingly, the data presented here excludes subject S19. It is clear from the questionnaire results in Table 1 that all remaining subjects were familiar with the C programming language, the language with which the sequential model was constructed.

4 Experimental Results

Figure 1(a) summarizes the results of the experiment presented in Section 3. Since the subjects had limited time, not all subjects could answer all questions. The first pair of bars corresponds to responses regarding the structural model. The first bar in the pair assumes that all questions left unanswered were answered



(a) Number of correctly answered questions, by subject.



(b) Total time taken for each group of questions, by subject.

Figure 1: Results of the simulator clarity experiment.

incorrectly. The second bar in the pair of bars shows the same data assuming that all unanswered questions were answered correctly. The second pair of bars shows the same information for questions regarding the sequential simulator. From the graph, we can see in each case subjects were able identify the machine properties at least as accurately with the structural model as they were with the sequential simulator, usually much more accurately. On average, subjects answered 8.87 questions correctly with the structural model versus 4.87 for the sequential model. Even if unanswered questions are assumed correct for *only* the sequential simulator, almost all subjects answered more questions correctly with the structural model. The only subjects who do better with the sequential simulator, under these circumstances, are subjects S8 and S20. Subject S20 simply failed to respond to 7 out of 10 of the questions for the sequential simulator. S8 spent an unusually long time on the control questions and thus did not have time to answer any questions regarding the sequential simulator. In fact, S8 only completed the first two parts of the first non-control question for the structural model (questions B.3(a) and B.3(b)) as numbered in Appendix B) and no questions for the sequential simulator. Clearly subject S8 is an outlier.

If subject S8 is excluded from the data, we see that, on average, 9.23 questions were answered correctly for the structural model versus 5.09 for the sequential simulator. Even when unanswered questions are assumed correct for the sequential simulator and incorrect for the structural model, the structural model still has 9.23 correctly answered questions versus 6.73 for the sequential simulator. The only subject who actually answers more questions correctly for the sequential simulator is Subject S18. The best explanation for this is that S18 was extremely familiar with `sim-outorder.c`. The model that the subject examined was the stock `sim-outorder.c` model and, the subject takes very little time to answer the sequential questions. An interesting experiment would have been to test S18 on a modified version of the `sim-outorder.c` models, but a retest was not possible.

Figure 1(b) summarizes the time taken to answer each class of questions: control questions, questions for the structural model, and questions for the sequential model. With S8 excluded, the average time taken per question for the structural model is greater by about 10 minutes when compared to the time for questions about the sequential simulator. Some of this gap is due to the fact that some subjects did not complete all the sequential simulator questions, and the time taken for unanswered questions does not count toward the average. Despite this, the increased number of correct responses for the structural model is not likely due to the extra time spent on those questions; many of the total times are well below the 90 minute time limit. One conclusion to be drawn from the timing data is that the sequential simulator is extremely misleading. Subjects voluntarily spend less time answering the sequential simulator questions despite plenty of extra time. Presumably, this is due to their confidence in their rapid analysis. Yet, the subjects frequently mischaracterize properties of the machine modeled. Timing data for subject S3 was unavailable because the subject failed to record the data during the examination.

Note that the experiment is skewed in favor of the sequential simulator. First, subjects were asked the LSE question immediately before being asked the same question for the sequential simulator. Thus, subjects could use the hardware-like model to understand what to look for in the sequential simulator. Second, many of the subjects were familiar with the stock `sim-outorder.c` simulator. Third, all subjects had many years of experience using the C language (the language used for `sim-outorder.c`) and less than a week of experience with LSE (the tool used to build the structural model). Fourth, no subject had ever seen a full LSE processor model before the experiment. Finally, the testing environment did not permit subjects to use the LSE visualization tool [1] to graphically view block diagrams of the structural specification. Experience indicates that this tool significantly simplifies model understanding; subject responses to the questions indicated that much of their time answering questions about the structural model was spent drawing block diagrams. Despite all this, the results clearly indicate that sequential simulators are significantly more difficult to understand.

Appendix

A Questionnaire

1. How many years of experience do you have in computer architecture or related areas?
2. If you have a PhD, did you do your PhD thesis in computer architecture?

3. Have you ever written a simulator for a processor microarchitecture?
   (a) Please list some of the major features of the microprocessor (e.g. out-of-order issue, speculation, etc.)
   (b) In which language/tool was the simulator built (e.g. C, LSE, Verilog, ...)
4. Have you ever designed your own processor core?
   (a) Please list some of the major features of the microprocessor (e.g. out-of-order issue, speculation, etc.). If more than one, choose the most advanced.
   (b) Was the design fabricated or was it part of a course project where the chip was never laid out and fabricated?
5. Which publicly available/commercial processor simulation tools have you used?
6. How long have you been using the C or C++ programming language?
7. How often do you program/look at C/C++ code?
8. How long have you been using the Liberty Simulation Environment (LSE)?
9. How often do you program/look at LSE code?
10. Which of the given exercises did you complete?

B Exam Questions

B.1 Chained Delay

B.1.1 Purpose

This is a control question to see if the subjects understand some LSE basics. The specifications are simply delay chains, the first has a delay of 3, the second a delay of 5.

B.1.2 Question

(a) In the configuration below, if a signal is present on port in of instance delay in cycle 0, on what cycle does it the data to appear on the port out of instance delay?

```
module delayelement {
  using corelib;

  inport in:'a;
  outport out:'a;

  instance delay1:delay;
  instance delay2:delay;
  instance delay3:delay;

  in -> delay1.in;
  delay1.out -> delay2.in;
  delay2.out -> delay3.in;
  delay3.out -> out;
};

instance delay:delayelement;
```

(b) If a signal is present on port in of instance delay in cycle 0, on what cycle does it the data to appear on the port out of instance delay?

```
module delayn {
  using corelib;

  inport in:'a;
  outport out:'a;

  var i:int;
  parameter n:int;

  if(n <= 0) {
    punt("Error, parameter n must be greater than or equal to 1");
  }

  instance delays:instance ref[];
  for(i=0;i<n;i++) {
    delays[n] = new instance(delay,"delay${i}");
  }
  in -> delays[0];
  for(i=0;i<n-1;i++) {
    delays[i].out -> delays[i+1].in;
  }
  delays[n-1].out->out;
};

instance delay:delayn;
delay.n=5;
```

B.2 Hierarchical Modules

B.2.1 Purpose

This is another control question to see if the subjects understand some LSE basics. The mystery specification `foo` is a hierarchically constructed demultiplexor. The `in1` signal is the data input, the `in2` signal is the control that selects which output will get the input, and the `out` signal is the output of the demux. All subjects had seen this example previously, if they had completed the exercises designed to help them understand Liberty. Note that the demux used in real specifications is a leaf module in the LSE library.

B.2.2 Question

This question will center around the following module description:

```
module foo {
  using corelib;

  inport in1:bool;
  inport in2:int;

  outport out:bool;

  instance fanout1:tee;
  instance fanout2:tee;
  instance gates:instance ref[];
```

```
  in1 -> fanout1.in;
  in2 -> fanout2.in;

  var i:int;
  for(i=0;i<3;i++) {
    gates[i] = new instance(<<<gates${i}>>>, gate);
    fanout1.out -> gates[i].in;
    fanout2.out -> gates[i].control;
    gates[i].out -> out;
    gates[i].gate_control = <<<
      if(LSE_signal_extract_data(cstatus[0]) == LSE_signal_something) {
        return (*cdata[0]) == ${i};
      } else if(LSE_signal_extract_data(cstatus[0]) == LSE_signal_nothing) {
        return 0;
      } else {
        return -1;
      }
    >>>;
  }
};
```

(a) In a few words please describe the input/output behavior of the data signals of this module. (You don't need to worry about the enable and ack signals).

(b) In the following configuration that uses the module defined above, what is printed out by the collector in timestep 0. What is the data status (LSE_port_something, nothing, or unknown) of each in port (in[0], in[1], ...) at the end of cycle 0. (Note that the data collector is only called if data is received on the port).

```
using corelib;

instance input1:source;
instance input2:source;
instance munger:foo;
instance hole:sink;

input1.create_data=source::create_constant("TRUE");
input2.create_data=source::create_constant("1");

input1.out ->:bool munger.in1;
input2.out ->:int munger.in2;

LSS_connect_bus(munger.out,hole.in,3);

collector in.resolved:hole {
  record = <<<
    if(LSE_signal_data_is_known(status) &&
       LSE_signal_data_is_present(status) &&
       !LSE_signal_data_is_known(prevstatus)) {
      printf(LSE_time_print_args(LSE_time_now));
      printf(": porti=%d %d\n", porti, *datap);
    }
  >>>;
};
```

B.3 Issue Windows and Reorder Buffers

B.3.1 Purpose

This question is designed to identify how easy it is to identify large structures, especially when a simulator is modified. It is *very* difficult in SimpleScalar [2], for example, to split the issue window into multiple issue windows in a clean way. A hack is pretty easy to concoct, but hard to understand. The modified versions of SimpleScalar used for this question were modified for other work unrelated to this experiment so the modified versions of of SimpleScalar are still representative. The answer to this question depends on which specification version of the specifications the subjects saw.

B.3.2 Question

(a) In the machine modeled in the configuration question3/machine1.xxx, Examine file question3/machine1.xxx lines nnn through mmm. This is the highest level code that describes the issue window/reservation stations.

Does the machine described have a unified issue window (excluding the load store queue), or a distributed set of reservation stations (still ignoring the load store queue).

(b) How many entries does each reservation station/issue window have?

(c) In order to recover from speculation, the machine in this configuration also keeps track of the issue order of instructions. Is it possible to have an instruction in the reorder buffer but not in the issue window? That is to say, can there be more decoded instructions in flight (not committed) than the cumulative size of the reservations stations/issue window. (Hint: The code for the commit stage of the machine is in pipestages.xxx, lines nnn to mmm.)

B.4 Branch Resolution Policy

B.4.1 Purpose

This question is designed to identify how easy it is to identify policy decisions normally thought of as sequential algorithms, such as what happens when a branch needs to commit. This type of sequential understanding often hides lots of complexity when you have to coordinate behavior among concurrent elements. LSE forces you to make these design decisions. We wanted to see if subjects can tell what those decisions are. For this question, the correct answer depends on the version of the configuration the user saw.

B.4.2 Question

In the machine modeled in the file question5/machine1.xxx, what is the branch resolution policy of the machine? In particular:

(a) Identify the line or lines of code which identify (detect) mis-speculated branches. (Hint: Look at lines mmm-nnn of pipestages.xxx)

(b) In what stage of the pipe are mis-speculated branches identified?

(c) Is this the same stage which initiates branch recovery (Recovery is initiated when the machine stops fetching from the wrong path)?

(d) If not, then what stage initiates branch recovery?

(e) Do instructions reach the pipeline stage where recovery is initiated in-order?

(f) If the answer to question 5 was no, is it possible to begin recovery on a later branch instruction (later in program order) before initiating recovery on an earlier branch?

(g) If the answer to question 6 was no, what hardware structure ensures that the branches are processed in order? How does that structure communicate with the pipeline stage?

References

[1] J. Blome, M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty simulation environment as a pedagogical tool. In *Proceedings of the 2003 Workshop on Computer Architecture Education (WCAE)*, June 2003.

[2] D. Burger and T. M. Austin. The SimpleScalar tool set version 2.0. Technical Report 97-1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[3] The Liberty Research Group. Web site: http://www.liberty-research.org/Software/LSE.

[4] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, pages 271–282, November 2002.