Parallel Computation for Microwave Circuit Simulation

David L. Rhodes, *Member, IEEE,* and Barry S. Perlman, *Fellow, IEEE*

***Abstract*—In this paper, the results of implementing a harmonic balance simulator, AGILE, on a variety of massively parallel computers (MPC's) is given. Descriptions of the computer hardware, which includes both shared-memory and message-passing implementations, and algorithms used to parallelize the computations are presented. The computers used include the CM-5, KSR-1, and a networked set of nonheterogeneous workstations using UNIX RPC. A key aspect is the description of the variety of algorithms used for each computer and the results obtained.**

***Index Terms*—Harmonic balance, parallel simulation.**

I. Introduction

A MASSIVELY parallel computer (MPC) is one which contains a great many individual communicating processors [typically called *nodes* or *processing elements* (PE's)]. Typical MPC's allow for tens to several hundred or even thousands of nodes, and thus in principle offer the ability to apply a large amount of computational power and memory to a problem. A key barrier, however, is the development of software and algorithms which efficiently exploits such hardware. Note that the focus here is on *parallel* computation and not vector computation—in vector computation specialized hardware, along with associated programming language interfaces or software libraries, is provided which can apply an operation(s) to an array or matrix of data (usually floating point) faster and more efficiently than an equivalent looping construct in an application language. While vector hardware can provide speed-ups, there is an inherent limitation in the speed-up possible. This is due to several factors.

First, the speed of the microprocessor's internal floating-point unit versus that of a specialized vector hardware board might be relatively fixed (say between 2–10 times) given equivalent digital fabrication processes/technologies. Dramatic gains of vector implementations made in the past could mostly be attributed to a situation where a much larger ratio was present (perhaps up to 100 times), but this was at a time when floating-point units could not be implemented directly on central processing unit (CPU) chips. With current very large scale integration (VLSI), microprocessor floating-point units are not really hardware constrained, and thus their floating-point computation speed is comparable to that of a specialized unit (perhaps it is even better since data does not need to be driven on and off chip). Secondly, the potential advantage of vector processing in *knowing* that a series of data is to be operated on has been largely eroded with pipelined CPU implementations and compiler advances in loop optimizations. Thirdly, vectorization is only applied to floating-point array or matrix operations and does not impact any other aspect (e.g., model evaluation) of the program.

Previous work on parallel circuit simulation implementation differs in a variety of ways. For example, a common portable library which operates over a variety of MPC's was used in [1]. This approach is appealing from a portability perspective but does not allow computer-targeted algorithmic optimizations. Several authors have investigated the difficult problem of solving linear matrices on MPC's (e.g., [2]) which arise in simulation, but they generally do not obtain speed-up scaling in concert with the number of computational nodes available. Attempts to create special-purpose parallel hardware for circuit simulation (e.g., [3]) ultimately cannot keep pace with general-purpose computing technology. Thus, the effort in this paper focuses on custom-developed algorithms tailored for each system under consideration. The aim is to assess the upper bounds of performance, minimization of communication overhead, etc., that are possible on MPC's. Other microwave-circuit or electromagnetic simulation studies using parallel computers (e.g., [4], [5]) did not explicitly focus on investigating various algorithmic approaches on a variety of MPC's.

AGILE's[1] [6], [7] harmonic-balance simulation entails separating the circuit into linear and nonlinear portions, assignment of initial harmonic voltages, and *balance* iterations which adjust harmonic voltages until an acceptably small error, defined as the difference between computed harmonic currents from the linear circuit and nonlinear devices, is reached. Although the nonlinear balance iterations sometimes consume a significant portion of execution time, as circuit sizes grow the tendency is for the analysis of the linear portion to dominate overall execution time. Specifically, the execution time for the nonlinear balance portion of the solution tends to grow linearly in the number of nonlinear devices for two prime reasons. First, each nonlinear element has a fixed, small number of connection points (e.g., a three-terminal transistor model) and secondly, nonlinear devices may not be strongly interacting or coupled to one another (there are obvious exceptions to this, of course). The linear solution portion, however, is generally of cubic order as this entails matrix inversion. Thus, the aim is to parallelize the linear portion of the solution.

Manuscript received March 1, 1996; revised January 24, 1997. This work was supported in part by a grant of HPC time from the following DoD Shared Resource Centers: the Army HP Computing Research Center and the Army Research Laboratory Super Computer Facility.

The authors are with the Army Research Laboratory, Sensors and Electronics Directorate, Fort Monmouth, NJ 07703 USA.

Publisher Item Identifier S 0018-9480(97)02911-6.

[1]The serial version of AGILE is available on the Web at `http://horse.arl.mil/AGILE/`.

In general, a goal for parallelizing software is to maximize the *granularity*, which refers to the *amount* of computation being performed within each node between communications to other nodes or the host. Here, one would like large granularity, which minimizes communication among nodes/host, and *load balance* which means that each node's computational load is approximately equal. As stated, the study in this paper parallelizes the linear portion of the harmonic-balance routine, where the parallelization was done such that multiple frequencies of analysis were divided equally for separate (parallel) computation. This is a relatively large granularity in that model evaluation, matrix fill, and matrix reductions are all parallelized. Note that this provides for load balancing as each node is doing almost the same amount of work (although there can be slight variations, e.g., due to frequency-dependent branches in models).

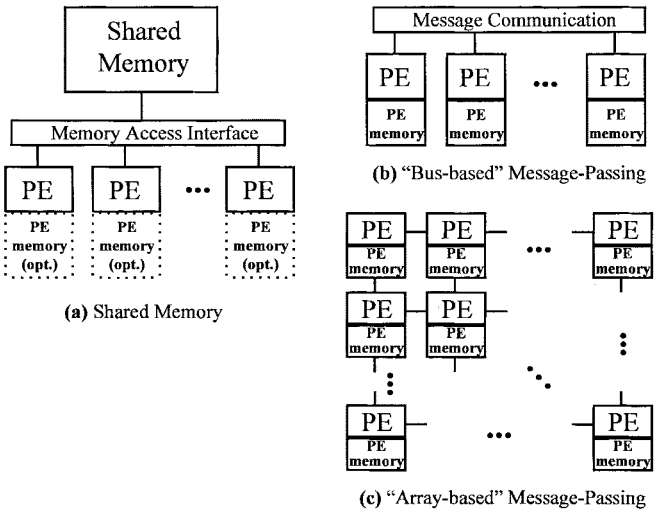

Fig. 1. Various configurations of parallel computers distinguished by how PE's or nodes communicate. (a) Shared memory. (b) "Bus-based" message-passing. (c) "Array-based" message-passing.

II. Generalized Hardware Descriptions and Definitions

The terms single instruction multiple data (SIMD) and multiple instruction, multiple data (MIMD) are encountered when referring to parallel computers. In a SIMD approach, all of the parallel computing nodes are executing the same instruction sequence (or nearly so) on different data, while in a MIMD approach each node executes its own instruction stream on its own data. The attractiveness of the SIMD approach lies in the hardware simplicity and potentially greater ease of programming since all nodes are restricted to running the same code, but this simplicity is only possible assuming that the problem at hand *strongly* fits this computational model. Although SIMD MPC's are still produced, especially for specialized embedded applications like image processing, the MIMD approach is currently favored for general-purpose parallel computing. The implementations discussed here are all using MIMD-type MPC's.

Two major parallel computer-implementation schemes serve to classify MPC's as either *shared-memory* or *message-passing*. In a shared-memory MPC, each computational node or PE has access to data memory locations which are accessible as memory locations to other nodes. A shared-memory computer might be implemented with a single memory as in Fig. 1(a) or in other ways. A shared-memory MPC would usually have local, privately available memory shown as dotted lines in Fig. 1(a) in addition to shared memory as is the case for the KSR-1 MPC discussed below. Communication among the nodes occurs via each accessing the shared memory (e.g., a node writes data, sets a flag indicating its presence, and another node reads the data and unsets the flag). In message-passing MPC's, each node has only a local memory available and each node explicitly communicates with others by sending and receiving messages through a communication mechanism. Fig. 1(b) shows a *bus*-style communication implementation, where every node can directly communicate with any other, while Fig. 1(c) shows an array style where nodes can only directly communicate with its neighbors (nonneighbor communication might be handled by allowing messages to pass through each other). As is to be expected, the underlying communication mechanism affects the efficacy of the MPC for a particular application problem. The block diagrams shown in Fig. 1 are necessarily simplified but are sufficient for purposes here (see [8] for further details). As a further distinction, in *symmetric* MPC's all nodes are completely equivalent, and, for purposes here, an *asymmetric* MPC is one where a particular node (called the *host*) is the only one capable of some particular operation, for example performing input/output (I/O) with the user or disks.

III. Implementation Details and Algorithms

The algorithmic methods developed for each MPC were tailored and optimized in accordance with its features in order to extract maximum benefit from the machine, keeping in mind that the level of parallelization is limited by the number of frequencies analyzed. However, typical cases (as those shown below) usually entail tens to hundreds of frequencies and thus the approach provides for a high degree of parallelization. Three implementations, two on large, commercial MPC's and the third on a local network of workstations, are presented. The commercial MPC's used, namely the Kendall Square Research KSR-1 and the Thinking Machines CM-5, both allow for *partitions*. Partitions divide the overall machine into a set of independent smaller machines. Since none of the test cases used more than 24 frequencies, and since all KSR-1 runs were performed in a partition of 32 nodes and all CM-5 runs were in partitions with at least 24 nodes, each node computed a single frequency point although the software is generalized to allow assignments of multiple frequencies per node.

Fig. 2 shows the flow of control for the KSR-1 using *pthreads* [9]. Note that time on the horizontal axis is not to scale in Figs. 2 and 3, and that each computational node occupies a position on the vertical axis. One of the nodes acts as a host program (which can equivalently reside on any

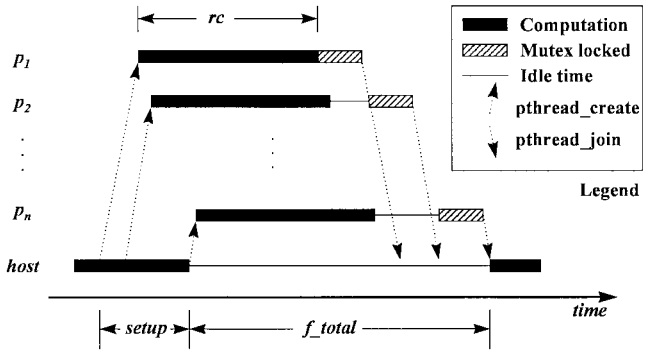

Fig. 2. Detailed execution sequence for the KSR-1. Times shown are not to scale.

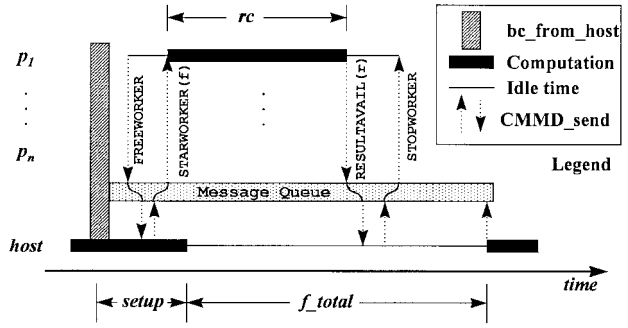

Fig. 3. Abstract execution sequence for the CM-5. Times shown are not to scale. The final *up* arrow indicates a `CMMD_send()` to halt.

node in the partition since the KSR-1 is a symmetric MPC) which uses calls to `pthread_create()` with appropriate arguments to start additional (parallel) processes. The KSR-1 is a shared-memory MPC and each pthread has access to both globally defined data and to private or local data as well. In the case of the KSR-1, the compiler automatically determines which variables are global and which are local based on the block structure of the program near the `pthread_create()` code. While this made porting of the software quite simple, there are ramifications which will be discussed later in the paper.

The *upright* directional arrows in Fig. 2 indicate the creation of a parallel process using `pthread_create()` in the figure. Each frequency is assigned its own pthread and each computes independently (in parallel). Since memory is shared, all of these threads have access to the netlist description and other state information which was established prior to their creation. At the end of the computation, a structured array is updated with the reduced admittance matrix. Recall that this computation entails model evaluation, matrix-fill, and subsequent reduction for the frequency(s) assigned to the node. As these results become available, they must be stored into a structured array, exactly like the serial version of the code. In order to avoid simultaneous memory writes from each pthread (even though each is writing to a different array element, the array in whole is considered a single data object in terms of the shared memory), a *mutex* is used. A mutex provides sequential (sole) access to shared data, the hashed sections of the figure show each parallel computational node locking the mutex, writing the data, and then unlocking. This operation, of course, is mutually exclusive (hence the name, mutex) across the nodes and the figure captures this fact (e.g., the idle time of p_2 waiting for p_1)—the need for mutexes is typical of shared-memory parallel code implementations. Immediately after spawning each thread, `pthread_join()` is called. This creates a *barrier* point in the program—the main program does not continue until all the joined threads (e.g., each parallel computation) are complete. In Fig. 2, all joins are complete at the point of the last downright arrow.

The method used on the asymmetric CM-5 is quite different and uses asynchronous message-passing as implemented in the CMMD library [10]. In Fig. 3, tagged messages (which include data) are shown passing between nodes and host. The process starts by having the host *broadcast* the netlist (a broadcast is a message which is sent simultaneously to all nodes); each node then parses it, and performs necessary memory allocations. This establishes the common state needed by all of the computational nodes such that they can each participate in a common computation—this step is not necessary in the KSR-1 implementation since all parallel nodes have direct access to the netlist, instance hierarchy, etc., due to shared memory. Each CM-5 node then sends a message `FREEWORKER` which is (eventually) read by a dispatch loop on the host informing it that it is available for assignment. The host dispatches the `STARTWORK` message which includes the frequency assignment (102 bytes total, indicated as "f" in the figure). At the end of computation (once again including model evaluation, matrix-fill, and reduction), the node sends a message with tag `RESULTAVAIL` which also contains the reduced matrix result (1612 bytes indicated as "r"). Finally the host sends the `STOPWORK` message to halt the node. For simplicity, Fig. 3 shows this process for only a single node, but of course this process is happening in parallel for all nodes. As mentioned earlier, in the cases shown below, each node has only one frequency to compute.

Also note that the messages are passed *asynchronously*; a message queue is depicted in the figure to which messages are posted and picked up to capture this fact. The alternative is *synchronous* message-passing which requires that both sender and receiver wait for each other to reach tag-matched communication calls. Synchronous message-passing is usually simpler from a programming point of view since at the point of communication the sender and receiver of the particular message are known to each be at their respective communication routine calls at that time point (often referred to as a *rendezvous*). However, unless code execution time between communication calls is extremely balanced, one of the senders or receivers will arrive at its respective communication call before the other and will have to wait (also known as *block*) for the other to reach its communication call. Assuming that the data being sent is not immediately needed, this can cause unnecessary delays in overall execution. In the case here of independent frequency analyses, it does not matter which result is available first, the host continues on only after all frequency results have been returned. Even though

the computation is, in principle, highly load balanced, it is certainly possible to encounter far from ideal circumstances. For example, suppose that all of the model evaluations for the first frequency are, say, ten times longer than at other frequency points and that the matrix fill and reduction times are much less than the model evaluation time. In such a case, if synchronous communication were used to receive the reduced matrix data in frequency-order (which, along with the assumption of load balance, would be the correct assumption since they are issued in frequency order) the host would (unnecessarily) block the receipt of the other-than-first frequency results since it would be waiting to receive the first result while all of the others have already been finished and, hence, available for receipt. Synchronous communication in the code here would also prevent the assignment of another computation (e.g., another frequency point) to the nodes which complete earlier. Thus, asynchronous message-passing is used throughout the implementation. Fig. 3 captures possible delays between the asynchronous send/receive by using curved segments in the *message queue*.

In addition to the MPC parallel versions, the use of a UNIX RPC on a local area network of heterogeneous workstations was used to form an asymmetric parallel computing capability [11]. An UNIX daemon called `agilerd` was registered as an RPC program and ran on several workstations. This daemon performed exactly the same job as the node programs of the CM-5 version, including the sizes of data being passed. Namely, it received and parsed the netlist, evaluated component models, and filled and reduced the matrix. The only logical difference was that broadcasting of the netlist was not used, but each node was sent this information in individual messages. Thus, Fig. 3 can be used to visualize the data communications for the RPC version with the exception that, instead of the broadcast, each node is sent the netlist in separate transmissions—this is due to the fact that only point–to–point message-passing is available. The RPC version of the program is run on a host workstation which also has a list of other workstations (computational nodes) on which it expects the `agilerd` daemon to be available. This eliminates the need for `FREEWORKER` communication and furthermore, as the remote daemons do not need to be stopped, the need for `STOPWORK` messages is also eliminated. This daemon was ported to both Sun SparcStations and HP 700 workstations, forming a nonheterogeneous computational capability. For consistency of data analysis, however, RPC case results are presented using equivalent Sun SparcStations only.

IV. Results

Three microwave circuits of varying sizes were used as the basis for testing the implementations. The first, called `test1`, is composed of a main circuit utilizing three instances of one parametric subcircuit, FET_SRL2, and a single instance of another, FET_EE. The main circuit has 45 local components and 35 circuit nodes (not to be confused with computational nodes) while FET_SRL2 has a single component and one internal circuit node and FET_EE has nine components and two internal circuit nodes. Although AGILE takes advantage of hierarchy in its analysis, if this description were flattened, it would comprise 75 components and 44 circuit nodes. The second circuit is called `test2`. It has a similar descriptive organization as `test1`, but is larger. If flattened, it would have 106 components and 86 circuit nodes. The final description is called `test3`. It is much larger than `test2`, utilizing 63 parametric circuit descriptions which, if flattened, represent 1076 components and 798 circuit nodes.

TABLE I
Averaged Raw Data (seconds)

		KSR-1	CM-5	RPC
	set_up	0.780	0.500	0.614
test1	worst_rc	0.247	0.230	2.731
	f_total	0.859	0.265	5.592
	set_up	1.563	1.550	1.828
test2	worst_rc	2.092	2.310	34.180
	f_total	2.718	2.389	38.620
	set_up	11.67	4.36	n/a
test3	worst_rc	7.44	7.06	n/a
	f_total	11.72	7.30	n/a

These circuits represent a spread in complexity/size from moderate to fairly large. Note that `test3` is a complete description of a moderately sized two-stage monolithic microwave integrated circuit (MMIC) amplifier, while `test1` and `test2` are different implementations of a low-noise amplifier (LNA). Each circuit was analyzed at 24 frequencies. Detailed execution times for the parallel versions are shown in Table I. Note that `test3` data for the RPC version is not available. In each case, the times referred to as `set_up` represent the time needed to prepare for the parallel computation with respect to each of the implementations developed. The time *rc* represents the amount of computation performed by a node without inclusion of communication time; that is, it is the amount of productive computation performed directly on a node. In each case, all node `rc` times were monitored, and the average over multiple runs of the longest of these is presented as `worst_rc` in the table. The time, `f_total`, is the amount of time for the frequency loop portion of parallel computation from the host's perspective. This does not include `set_up` time and thus the total time (in this portion of the program) is `set_up` + `f_total`. These timings can be used to assess the efficacy of the parallelization and algorithms used.

It is important to note that for the RPC version, only two computational nodes (and separate host) are employed and, hence, each is analyzing 12 frequency points, whereas for the MPC's, each node is analyzing just one. From the `worst_rc` data, it can be seen that the computational power of all MPC/RPC nodes is approximately equal; for example, the `worst_rc`'s for the `test1` circuit are 0.247, 0.230, and 0.228 (2.731/12) s for the KSR-1, CM-5, and RPC node (Sun SPARC), respectively. Each data point presented in Table I was averaged from five runs; for the KSR-1 and CM-5, commands were used to ensure that no other users were using the partition helping to ensure fair timing results. Also recall that the `test3` problem was not run on the RPC version

TABLE II
PARALLEL EFFICIENCY

	KSR-1	CM-5	RPC
test1	0.29	0.87	0.49
test2	0.77	0.97	0.89
test3	0.64	0.97	n/a

TABLE III
ELAPSED TIMES (SECONDS)

	KSR-1	CM-5	RPC	Serial
test1	1.64	0.77	6.21	6.1
test2	4.28	3.94	40.45	50.4
test3	23.39	11.66	n/a	107.8

and, hence, related results are not available. The `agilerd` daemon was "killed" before each run—since dynamic memory allocation using malloc is used extensively, this prevented data "free's" from a previous run from counting in the timing analysis.

In this paper, one is mostly interested in determining communication overhead and preferred approaches. To estimate communications overhead, a definition for parallel-efficiency (P), is given as

$$\mathrm{P} = \frac{\texttt{worst\_rc}}{\texttt{f\_total}}. \tag{1}$$

The amount of time from the host's (or user's) perspective is `f_total`, while the longest node computation is `worst_rc`, so that the P factor captures the ratio of useful node computation versus host elapsed time in the frequency loop. Table II shows the evaluation of P for the times shown in Table I. Except for the drop for `test3` on the KSR-1 (the 0.64 figure), the general trend to increased efficiency for larger granularity is witnessed—that is as the size of the problem increases (from `test1` to `test3`), the parallel efficiency improves. On the CM-5, the larger-sized problems enjoy a 97% (0.97) efficiency, which implies that, from the host's perspective, only 3% of the overall frequency loop time is spent in communications. For the RPC version, a separate timing test was done which determined that RPC communication (on the local area network used) takes about 300 ms. This rather large factor reflects itself in the poor efficiency on the smaller problem (`test1`). As expected, the situation improves for the larger problem (`test2`) just due to the larger granularity of the computations.

KSR-1 results are more difficult to explain. As should be expected, P increases from 0.29 to 0.77 for `test1` and `test2` problems, but then inexplicably decreases somewhat to 0.64 for the larger `test3` problem. Notice that while the `set_up` times for `test1` and `test2` are comparable between the CM-5 and KSR-1, this is not the case for the `test3` problem (from Table I). On the other hand, the `worst_rc` for `test3`, as well as for the other problems, are comparable between the KSR-1 and the CM-5. The conclusion then, is that the KSR-1 requires a relatively larger time to do the `pthread_create()`'s which is basically the only code executed as part of the `set_up` time, for the `test3` problem—but the question is "why?" One of the only differences is that the actual executable image for AGILE when running larger circuits is progressively larger, as dynamic memory is allocated based on need. A large executable image size at the point of pthread creation indeed appears to be the difficulty that the KSR-1 has with `test3`. Since the compiler fully automates allocation of data to shared or (pthread) local memory, there is not much that can be done to alleviate this situation. Note that the goal here was to contrast shared-memory and message-passing implementations; it might be expected that the KSR-1 would perform as well as the CM-5 if its message-passing capabilities were used.

V. CONCLUSION

From a user's perspective, elapsed run time is what is important. Table III shows the elapsed times (for this portion of the program) for each of the parallel implementations and a serial version running on a Sun SparcStation-II (with 16 megabytes); note that the elapsed time shown for each parallel version is `set_up` + `f_total`. With the exception of `test1` on the RPC version, each of these is clearly superior to the serial version.

For simplicity in programming, the shared-memory model is superior in terms of ease of development and the level of assistance provided by the compiler, and this model achieves reasonable parallelization efficiencies (at least for this problem). The message-passing versions on the other hand required quite a bit of code changes, development of nonhost node or daemon programs, etc. However, custom-programmed message-passing appears superior from a performance perspective, even at the level of a local area network of workstations (RPC version) when the problem is large enough to overcome a high communication time delay.

REFERENCES

[1] E. Pajarre, T. Ritoniemi, and H. Tenhunen, "PAR-APLAC: Parallel circuit analysis and optimization," in *Euro-DAC'92 Conf. Proc.*, Hamburg, Germany, Sept. 7–10, pp. 584–589.

[2] K. Mayaram, P. Yang, J. Chern, R. Burch, L. Arledge, and P. Cox, "A parallel block-diagonal preconditioned conjugate-gradient solution algorithm for circuit and device simulations," in *IEEE Int. Conf. Comput.-Aided Design* , Santa Clara, CA, Nov. 11–15, 1990, pp. 446–449.

[3] J. O. Hamblen and C. O. Alford, "A parallel computer architecture for continuous simulation," *IEEE Trans. Aerosp. Electron Syst.*, vol. 24, pp. 719–725, Nov. 1988.

[4] V. Rizzoli, F. Mastri, F. Sgallari, and V. Frontini, "The exploitation of sparse-matrix techniques in conjunction with the piecewise harmonic-balance method for nonlinear microwave circuit analysis," in *IEEE Microwave Theory Tech.-S Dig.*, vol. 3. Dallas, TX, May 1990, pp. 1295–1298.

[5] C. Eswarappa, P. P. M. So, and W. J. R. Hoefer, "Efficient field-based cad of microwave circuits on massively parallel computer using TLM and Prony's method," in *IEEE Microwave Theory Tech.-S Dig.*, vol. 3. San Diego, CA, May 1994, pp. 1531–1534.

[6] B. Epstein, S. Perlow, D. Rhodes, J. S. Schepps, M. Ettenberg, and R. Barton, "Large-signal MESFET characterization using harmonic balance," in *IEEE Microwave Theory Tech.-S Dig.*, New York, May 1988, pp. 1045–1048.

[7] R. Barton and D. Rhodes, "Cost minimization via simulation-based meta-modeling," in *1994 US Conf. GaAs MANufacturing TECHnology (MANTECH)*, Las Vegas, NV, May 1–5, 1994, pp. 97–100.

[8] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. San Francisco, CA: Morgan Kaufmann Publishers, 1996.

[9] *KSR Parallel Programming Guide.* Waltham, MA, Kendall Square Research Corporation, Feb. 1992.
[10] *CMMD Reference Manual.* Cambridge, MA, Thinking Machines Corporation, v. 3.0, May 1993.
[11] *Network Programming Guide.* Mountain View, CA, Sun Microsystems, Inc., Revision B, Mar. 1993.

David L. Rhodes (S'80–M'80) received the B.S.E.E degree from Rutgers University, New Brunswick, NJ, in 1980, and the M.S.E.E. degree from Princeton University, Princeton, NJ, in 1982, while working at RCA Laboratories, David Sarnoff Research Center, Princeton, NJ. He is now pursuing a Ph.D. in electrical engineering/computer engineering at Princeton University.

He joined RCA Laboratories in June of 1980, working in the Microwave Division. He primarily worked on CAD for microwave applications including the areas of simulation, optimization, data visualization, and statistical design techniques. He is now Branch Chief of the Solid State Electronics Branch of the Sensors and Electronics Directorate (SED) of the Army Research Laboratory (ARL), Fort Monmouth, NJ. He was chair of IEEE SCC-30, *Analog HDLs*, until October 1996 and is working on or directing efforts in the areas of hardware/software co-design, parallel simulation, meta-modeling/statistical design, active device modeling, and analog HDL's. He was tri-service Leader of the Computational Electronics and Nanoelectronics area of the high-performance computing modernization program software initiative from September 1994 until September 1995. He was the Technical Program Chair for the first and second (1994 and 1995) International Conference on Electronic Hardware Description Languages (ICEHDL) (called SHDL in 1994). He also serves on the Editorial Board for the *International Journal of Microwave and Millimeter-Wave Computer-Aided Engineering*.

Barry S. Perlman (M'65–SM'71–F'86) received the B.E.E. degree in electrical engineering, College of the City University of New York, New York, in 1961, and the M.S.E.E. and Ph.D. degrees in electrophysics from the Polytechnic Institute of New York, in 1964 and 1973, respectively.

He is Chief, RF & Electronics Division, U.S. Army Research Laboratory, Sensors and Electron Devices Directorate (ARL/SEDD), Adelphi, MD, and Ft. Monmouth, NJ. He is responsible for basic and applied research and exploratory development in RF sensing and advanced electronic device technologies. He directs technical programs in ultra-wide-band radar, millimeter-wave imaging, microwave/millimeter-wave technology, phenomenology, architectures and algorithms, electronic materials, devices and monolithic circuits, electrophysics and modeling, high-speed/power devices, vacuum electronics, frequency-control devices, and technologies for signal generation, transmission, reception control, and processing. Prior to his current position, he was Director, Electronics Division for the Physical Sciences Directorate of ARL and prior to that, Head, Design Research, Microwave Laboratory, David Sarnoff Research Center (formerly RCA Laboratories) in Princeton, NJ. He holds four U.S. patents and has published more than 70 technical papers.

Dr. Perlman is a member of Sigma Xi, and a Registered Professional Engineer in the State of New York. He is a contributing member to the MTT AdCom where he serves as Chairman of the Intersocietal Liaison Committee and an active member of the Technical Program Committee. He serves as an Army representative to the Advisory Group on Electron Devices (AGED) and the DoD JDL/Reliance Sub Panel on RF Technology. He is a member of the Army's Electronic Coordinating Committee (ECOG), the Technical Advisory Committee (TAC) for the URI/High-Frequency Microelectronics Center, University of Michigan, and the NSF/CAEME policy board at the University of Utah. He is also a member of URSI Commission D. He has received several awards including four RCA Outstanding Engineering Achievement Awards for research in microwave/millimeter-wave device, circuit, and design technology.