Microcomputer Applications, Vol. 15, No. 1, 1996

ARCHITECTURAL SUPPORT FOR FINE-GRAIN MULTITHREADING ON STOCK PROCESSORS

S.V. Kotikalapoodi,* B. Lee,* S.-L. Lu,* and A.R. Hurson**

Abstract

This paper discusses some issues involved in providing efficient support for fine-grain multithreading and proposes the modifications required on a conventional processor. The design is aimed at reducing the costs associated with context-switching. The hardware modifications are kept to a minimum in order to maintain the functionality of a conventional RISC processor.

Key Words

Fine-grain parallelism, multithreading, RISC, context switching

1. Introduction

There seems to be a consensus that future massively parallel architectures will consist of a number of nodes, or processors, interconnected by a high-speed network. These nodes should support multiple threads and be able to switch between these threads very rapidly to tolerate latencies due to remote memory requests and synchronization, i.e., *multithreading* [1]. There is a broad spectrum of execution models for how multithreading should be accomplished [2]. Multithreading based on the conventional control-flow model offers high-performance on sequential execution that exhibits good locality, but switching and synchronization among threads have substantial execution overhead. On the other hand, dataflow architectures support rapid context switching on a per-instruction basis by associating a context with each datum. However, these architectures invariably require extensive hardware and do not use high-speed registers.

There have been a number of proposed architectures which combine the instruction-level context-switching capability with sequential scheduling [2]. In the context of multithreading, a *thread* is a sequence of statically ordered instructions such that once the first instruction in the thread is executed, the remaining instructions execute without interruption [3]. As a result, a thread defines the basic unit of work from the dataflow model point-of-view that requires synchronization only at the beginning of a thread. Observations of current dataflow projects show that there is a trend towards adopting multithreading as a viable method for building hybrid architectures that combine both features of dataflow and control-flow execution models [2].

One such project at UC Berkeley, called Thread Abstract Machine (TAM), supports the interleaving of multiple threads by an appropriate compilation strategy and program representation rather than through elaborate hardware [1]. Experiments on TAM have already shown that it is possible to implement the dataflow execution model on conventional architectures and obtain reasonable performance. This has been demonstrated by compiling Id90 [4] programs to TL0, the TAM assembly language, and finally to the native machine code for a variety of platforms, mainly CM-5 [5]. These studies also show a basic mismatch between the requirements for fine-grain parallelism and the underlying architecture, and thus considerable improvement is possible through hardware support.

Based on the aforementioned discussions, this paper presents a design modification required to efficiently support fine-grain parallelism on a conventional RISC architecture. Modifications to the instruction set architecture (ISA) are proposed to reduce the costs involved in switching among threads. The hardware modifications are kept to a minimum so as not to disturb the functionality of a conventional RISC processor. Although the discussion is based on the SPARC processor, the design issues apply to other RISC processors as well.

2. Threaded Abstract Machine

This section describes the TAM execution model. Although it grew out of work on dataflow, the TAM model of execution exposes synchronization, thread scheduling, storage management to the compiler, and is explicit in the machine language. Unlike other dataflow proposals, TAM design aims to minimize the hardware overhead of multithreading and exploit locality even under asynchronous execution.

Figure 1 illustrates a TAM activation tree. A TAM program consists of a collection of *code-blocks* where each *code-block* typically represents a loop-body or a function. A code-block is comprised of *threads* and *inlets*. Invoking a code-block involves allocating an *activation frame* — which is analogous to the stack frame for conventional subroutine calls — depositing argument values into the frame, and enabling threads for execution within the context of the frame. Initialization also consists of setting the values for synchronization counters, or *entry counters*, stored within the frame. Unsynchronizing threads requires

* Department of Electrical and Computer Engineering, Oregon State University, Corvallis, OR 97331, USA; E-mail: kotikas, benl, sllu@ece.orst.edu

** Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA; E-mail: a2h@ecl.psu.edu

(paper no. 307-647)

no counters. A resident frame is executed until it has no enabled threads. A *quantum* is the set of threads executed during a single residency of the frame.

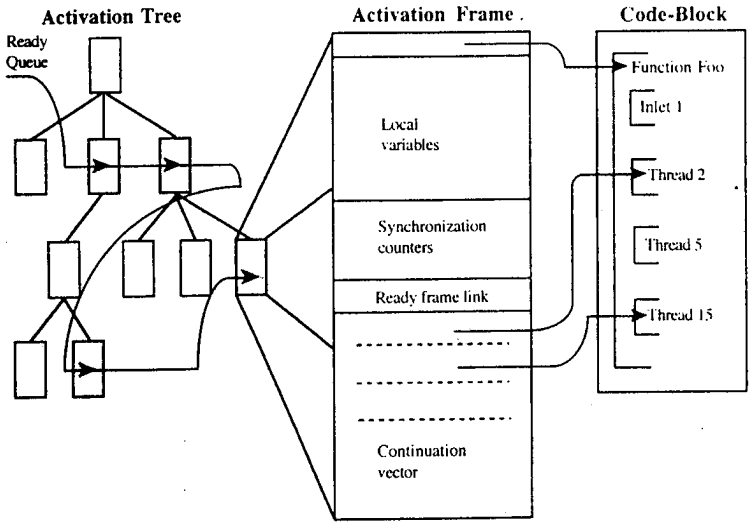


Figure 1. TAM activation tree.

TAM supports FORK instructions that enable a thread within the current activation. If the thread is an unsynchronizing one, the pointer to the thread is pushed onto the *local continuation vector* (LCV), which contains pointers to the all enabled threads within the current quantum.[1] If the thread requires synchronization, the FORK instruction decrements the entry count for the thread. If the decremented count is zero, the thread is enabled and pushed onto the LCV; otherwise, the count is stored back. A SWITCH instruction forks one of two threads depending on a condition. A STOP instruction terminates the current thread and causes some other enabled thread to begin execution. This is done by popping a thread from the LCV. When the LCV is empty or the execution of the current code-block has completed, the processor executes a SWAP instruction, which transfers the control to a frame pointed to by the *ready frame link* within the current frame. The following is a simple TL0 code for a TAM thread that adds two frame memory slots, stores the result in a register, and forks another thread.

```
THREAD 1
ADD ireg0.i = islot0.i islot1.i[2]   % Add frame slots islot0 and islot1
                                     and store the result in the TAM
                                     temporary register ireg0
FORK 4.t                             % FORK the thread "4"
STOP                                 % Pop a thread from LCV
```

TAM also supports inter-frame messages, which arise in passing arguments to an activation, returning results, and responses to the global heap accesses. This is done by associating a set of inlets with each code-block. *Inlets* are compiler-generated message handlers that copy arguments into the frame and enable threads depending on the message. For example, a SEND operation delivers a data value to an inlet relative to the target frame. A sample inlet code is show below:

```
INLET 1
    RECEIVE islot4.i   % Receive the value and store it in
                       the frame slot islot4
    POST 5.t           % Pushes the thread "5" onto LCV or RCV
    STOP               % Pop a thread from LCV
```

Global data structures in TAM provide synchronization on per-element basis to support I-structure and M-structure semantics [5]. If the I-structure element is empty, a read is deferred until the corresponding write takes place. A remote I-structure operation generates a request for a particular heap location and the corresponding response is received by an inlet. Meanwhile, the processor continues with other enabled threads. In TAM, these split-phase transactions are supported by instructions, such as IFETCH and ISTORE, which are used to read and write to the data structures, respectively.

2.1 Mapping TAM onto a Stock Processor

TAM is codified in a pseudo-machine language TL0. TL0 instructions are primarily three-address, where the operands are constants, registers, or frame locations. No fixed limit is placed on the number of TAM registers; however, the compiler tries to use them as efficiently as possible. The TAM translator is responsible for mapping TAM registers to physical registers or spill areas.

TL0 registers are implemented on the SPARC processor using a single register window [1]. The single register window is divided into three categories: special-function registers, thread registers, and inlet registers. The special-function registers hold important variables and constants such as pointer to the top of the LCV (lcv), node ID, frame pointer (fp), pointer to the base of current code-block (cbbase), and a pointer to frame scheduling queue. The TL0 instruction pointer and the inlet instruction pointer are both mapped onto the SPARC program counter register. There are sixteen thread registers that are under the control of the register allocator. The eight inlet registers are generally reserved for inlets but may be used by the register between successive polls to hold thread temporaries.

2.2 Measurements

In this section, measurements obtained from running benchmark programs on CM-5 with 64 SPARC processors are presented and analyzed. These measurements were obtained from the instruction mixes provided by the TAM group at UC Berkeley and will be the basis for analyzing and identifying how the costs involved in thread scheduling on stock RISC processors can be reduced [6]. These programs were written in *Id* and then compiled to TAM [4]. The TAM code is then translated to SPARC. Six bench-

[1] If the frame is not active, threads are pushed onto its *remote continuation vector* (RCV). Thus, LCV can be viewed as fast, short-lived extensions of RCV.

[2] All registers and frame slots are statically typed. 'i' indicates that it is of integer type.

marks ranging from 50 to 1,000 lines are used. For a more detailed explanation of the benchmarks, see [1].

The instructions are grouped into six categories as shown in Table 1. *Arithmetic* includes integer and floating-point operations; *messages* include instructions executed for SENDs and inlets; *heap* includes global I-structure and M-structure accesses; *control* represents all control-flow instructions such as FORK, SWAP, SWITCH, and moves to initialize entry counters; and *others* include instructions such as traps.

Table 1
Dynamic Instruction Mix Statistics for the Benchmark Programs

	Gamteb	QS	MMT	Simple	Speech	Paraffins
Arithmetic	11.67%	6.41%	41.63%	20.79%	30.76%	6.86%
Messages	43.96%	35.73%	21.82%	44.75%	37.14%	22.01%
Heap	7.19%	7.79%	7.93%	11.31%	12.13%	15.68%
Control	35.96%	49.98%	28.61%	23.01%	19.96%	55.43%
Others	1.21%	0.09%	0%	0.14%	0.01%	0.01%

3. Design for Efficient Context Switching

As can be seen from Table 1, control instructions constitute a significant part of the overhead for supporting fine-grain multithreading. It shows that the control overhead varies in a range from 20% to 55% of the total instructions executed, depending on the nature of the program. The control overhead is mainly due to three instructions: FORK, SWITCH, and STOP. The TAM translator optimizes FORKs or SWITCHes by pushing these instructions to the end of the thread and combining them with STOP to form simple branches. If FORK is to an immediately following thread, the branch becomes a fall-through. A FORK or a SWITCH that cannot be optimized into a branch will attempt to push a thread onto the LCV before continuing with the execution of the current thread. This is shown in Figure 2. Table 2 shows the mapping of these instructions to the SPARC processor and indicates the relative cost of supporting TAM thread scheduling instructions [6].

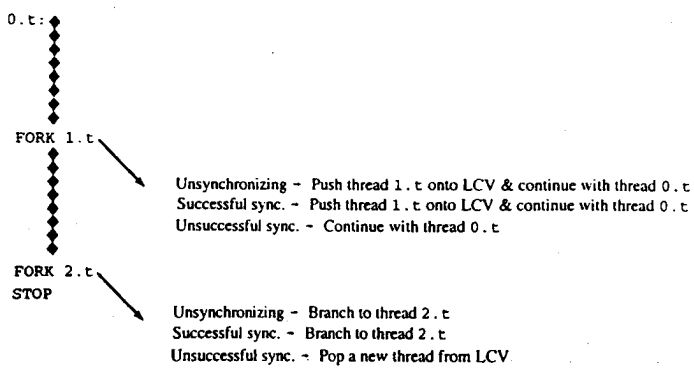


Figure 2. Control transfer of FORK.

Based on the costs shown in Table 2, the main areas for improvement are synchronizing threads and STOPs, which represent a large percentage of the control overhead. This is because unsynchronizing FORKs are optimized into fall-through and branches, thus avoiding the cost of popping a thread from the frame. On the other hand, an unsuccessful synchronizing branch requires that a STOP instruction be executed to pop a new thread from the LCV, which requires a total of 13 cycles (8 plus 5). Unsuccessful synchronizing branches can constitute as much as 11% of the total TL0 instructions executed and, therefore, are appropriate for optimization. Another source of overhead is the SPARC processor's lack of post-decrement capabilities to efficiently implement the pushing of threads.

Table 2
Mapping of TAM Instructions on to SPARC

Operation	SPARC instructions		Cycle
FORK a thread			
Fall-through	No additional overhead		0
Branch to thread			
Unsynchronizing	`ba thr_addr`	; Branch to thread address thr_addr (PC relative)	1
Successful sync.	`ldb sync[fp],tmp1`	; Load the count into reg. tmp1	2
	`subcc tmp1,1,tmp1`	; Decrement the count	1
	`be,a thr_addr`	; Branch if count is zero. Make annulling† and put first thread inst. in the delay slot.	1
Unsuccessful. sync.*	`ldb sync[fp],tmp1`	; Load the count into reg. tmp1	2
	`subcc tmp1,1,tmp1`	; Decrement the count	1
	`be thr_addr`	; Branch on zero.	2
	`stb tmp1,sync[fp]`	; Otherwise, store the count	3
Push thread			
Unsynchronizing	`set Lthr-cbbase‡,tmp2`	; tmp2 gets thread offset addr.	1
	`sth tmp2,[lcv]`	; Store the 16-bit offset	3
	`sub lcv,2,lcv`	; Decrement the pointer	1
Successful sync.	`ldb sync[fp],tmp1`	; Load the count	2
	`subcc tmp1,1,tmp1`	; Decrement the count	1
	`bnz continue`	; Test the entry count (untaken)	2
	`set Lthr-cbbase,tmp2`	; tmp2 gets thread offset addr.	1
	`sth tmp2,[lcv]`	; Store the 16-bit offset	3
	`sub lcv,2,lcv`	; Decrement the pointer	1
Unsuccessful. sync.	`ldb sync[fp],tmp1`	; Load the count	2
	`subcc tmp1,1,tmp1`	; Decrement the count	1
	`bnz continue`	; Test the entry count (taken)	1
	`stb tmp1,sync[fp]`	; Use annulling to store the count	3
SWITCH	FORK+2 cycles (worst case) for branching depending on the condition		
STOP	`lduh [2+lcv],tmp`	; Load offset	2
	`add lcv,2,lcv`	; Increment the pointer	1
	`jmp [tmp+cbbase]`	; Add the offset to the code-block base and jump to the thread	2

* The cost of a branch to an unsuccessful synchronizing thread does not include the STOP instruction that must be executed in order to schedule the next thread.

† SPARC provides a special bit called *annul* bit for branch instructions. For conditional branch instructions, if this bit is 1, indicated by appending ",a", then the delay instruction is executed before transfer of control.

‡ cbbase points to the base of the code-block and Lthr is an absolute thread address. Therefore, Lthr-cbbase gives the offset of the thread from the base of the code-block.

3.1 The Proposed Method

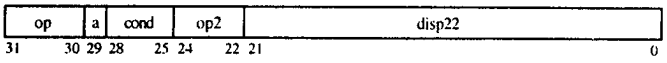
In order to reduce the overhead of implementing unsuccessful synchronizing branches, a method is proposed to eliminate the need to access the frame for the LCV, thereby decreasing the thread-switching time. This is done by allocating a special register r_lcv to hold the content of the top of the LCV. To make use of this register, a new instruc-

tion called *conditional double branch and pop*, cdbp, has been added. All the synchronizing branches now contain the instruction

```
cdbp thr_addr
```

After the entry count has been decremented, the cdbp instruction jumps to the thread at the location thr_addr if the count is zero; otherwise, it jumps to the address given by r_lcv. If the control transfers to the location given by r_lcv, the r_lcv register is updated by popping the next enabled thread pointer into it, and then the delay slot is used to store back the entry count.

The proposed instruction can be easily incorporated into the existing SPARC instruction set. The general format for branch and jump instructions of SPARC is shown below [7].



The *op* field is zero for branch instructions, and *op2* field decodes several different branches, such as conditional, unconditional, and floating-point conditional, etc. For a conditional branch instruction, if the *a* bit is 1, the delay slot is executed only if the branch is taken; otherwise, the delay slot is always executed. The condition *op2*=3 is not yet implemented, which can be used for the proposed instruction.

The cdbp instruction can be incorporated in the SPARC processor with only minor changes to the existing four-stage pipeline datapath. The microoperations for the different stages is given as follows:

- Fetch Stage: Fetch the cdbp instruction.
- Decode Stage: Decode instruction and compute PC+thr_addr. Operands r_lcv and cbbase (which are implied) are read from the register file.
- Execute Stage (1): By this time, the condition code from the previous instruction is available. Thus, if the condition is not true (entry count≠0), the address r_lcv+cbbase is generated and loaded into the PC. If the condition is true (entry count=0), nothing has to be done and the following stages are skipped.
- Execute Stage (2): Increment the stack pointer, lcv ← lcv +2.
- Execute Stage (3): Put the address of the top of the LCV (lcv) on the bus and load the value into the temporary buffer.
- Write Back Stage: Write back the content of the top of the LCV from the temporary buffer into the register r_lcv.

Based on this, the cdbp instruction requires two cycles if the condition is true (because of the delay slot); otherwise, it requires three cycles.

In addition to cdbp instruction, one more instruction is proposed to allow pushing of threads faster.

```
std rd, [rs] ;  store rd into [rs] and decrement rs
```

This instruction is similar to the store instruction except that the additional work of decrementing rs is done in the Execute Stage.

With these modifications, branch-to-synchronizing-thread instructions and push-thread instructions are mapped as follows:

Branch to synchronizing thread:

```
ldb   sync[fp],tmp1      ; Load the entry count into reg. temp1 (2 cycles)
subcc tmp1,1,tmp1        ; Decrement the count (1 cycle)
cdbp  thr_addr           ; Branch to thread thr_addr if the count is zero, else branch to
                           thread pointed to by r_lcv and pop a thread into r_lcv (2
                           cycles if tmp1 is zero; else 3 cycles)
stb   tmp1,sync[fp]      ; Use the delay slot to store back entry count (3 cycles)
```

Push unsynchronizing thread onto LCV:

```
std   r_lcv,[lcv]        ; First push the existing thread pointer onto LCV (3 cycles)
set   Lthr-cbbase,r_lcv  ; Store the thread pointer in r_lcv (1 cycle)
```

Push synchronizing thread onto LCV:

```
ldb   sync[fp],tmp1      ; Load the count (2 cycles)
subcc tmp1,1,tmp1        ; Decrement the count (1 cycle)
bnz,a continue           ; Test the count (1 cycle for taken/2 cycles for not taken)
stb   tmp1,sync[fp]      ; Use delay slot to store the count if synchronization fails (3 cycles)
std   r_lcv,[lcv]        ; Otherwise, push the existing thread pointer onto LCV (3 cycles)
set   Lthr-cbbase,r_lcv  ; Store the thread pointer in r_lcv (1 cycle)
```

Since the cdbp instruction uses r_lcv to hold the top of the LCV, STOP is changed to:

```
orcc  g0,g0,g0           ; Clear the zero flag (1 cycle)
cdbp  nowhere            ; Unconditionally branch to thread pointed to by r_lcv and pop a
                           thread into r_lcv (3 cycles)
```

With these changes, branch to a thread on successful synchronization now requires five cycles; however, unsuccessful synchronization requires only nine cycles compared to 13 cycles (including STOP) without the cdbp instruction. The std instruction also speeds up the pushing unsynchronizing threads onto the LCV from five cycles to four cycles. In addition, adding this instruction reduces the time to implement pushing thread on successful synchronization from ten to nine cycles, while that of unsuccessful synchronization remains unchanged. The STOP instruction also only requires four cycles. Table 3 summarizes the overall impact of these changes on the cycle cost for Gamteb and Paraffins. The cycle cost for each TAM instruction is obtained by adding the total clock cycles for the set of SPARC instructions to which it is mapped. The *average cycle costs* are obtained by taking the summation of the products of the percentage of each instruction type with its cycle cost.

It can be seen from this comparison that by introducing cdbp and std instructions, the largest reduction in cost comes from the elimination of STOPs that are required after branch to unsuccessful synchronizing threads. This results in 18.1%/11.3% reduction in the average cycle cost for 38.6%/30.76% of the total TL0 instructions in Paraffins and Gamteb. This improvement can be achieved by simply changing the control unit, and the datapath itself requires no major modifications.

4. Conclusion

Previous experiments on TAM have shown that fine-grain parallelism can be supported by conventional RISC processors [1, 8]. Our results show that considerable improve-

Table 3
A Comparison of Cost and Frequencies of TL0 Thread Synchronization and Scheduling Instructions

Type	Cycle cost		Paraffins	Gamteb
	Original	Modified	(% of control insts.)	(% of control insts.)
FORK a thread				
Fall through	0	0	4.09%	3.6%
Branch to thread				
Unsynchronizing	1	1	10%	2.95%
Successful sync.	4	5	7.63%	7.62%
Unsuccessful sync. and STOPs required after branch to unsuccessful synchronizing threads	8 5	9 0	27.94%	19.98%
Push thread onto LCV				
Unsynchronizing	5	4	0%	0.13%
Successful sync.	10	9	7.37%	15.27%
Unsuccessful sync.	7	7	13.43%	12.12%
SWITCH a thread†				
Branch to thread				
Unsynchronizing	3	3	7.43%	4.8%
Successful sync.	6	7	0.33%	4.16%
Unsuccessful sync.	10	11	0.29%	7.34%
Push thread onto LCV				
Unsynchronizing	7	6	14.09%	5%
Successful sync.	12	11	0%	1.76%
Unsuccessful sync.	9	9	0%	7.15%
STOPs	5	4	7.38%	8.1%
Percentage of total TL0 instructions			38.6%	30.76%
Average cycle cost (original/modified)			7.34/6.01	8.05/7.14

† SWITCH is basically a FORK preceded by a conditional branch instruction. In SPARC, conditional branch is implemented by be thr instruction that requires 1 cycle if taken and 2 cycles if not taken. Therefore, the cycle cost for SWITCH is based on the worst case of 2 cycles.

ment in the execution time can be obtained by optimizing the execution of TAM control instructions with minimal hardware changes. This is achieved by adding the instructions cdbp and std, which can be incorporated into ISA of the SPARC processor with only a slight change in the control unit.

The introduction of cdbp and std instructions is only an initial step towards reducing the gap between the requirements of fine-grain multithreading and the underlying architecture. Another area for improving the performance of fine-grain multithreading through better hardware support is in handling messages. As shown in Table 1, the message overhead for Simple can be as much as 45% of the total instructions executed, of which 95% of the instructions are for executing inlets. Based on this, our future work will be to relegate the responsibility of handling inlet messages to a separate coprocessor [2]. However, the difficult challenge will be to ensure atomicity between the operations of the main processor and the inlet-processor without compromising per-processor performance and system integrity.

Acknowledgements

We would like to thank the TAM group at UC Berkeley for providing the TL0 to SPARC translator. Special thanks goes to Klaus Schauser and Computation Structures Group at MIT, including Andy Shaw and Boon Ang. We are especially grateful to Seth Goldstein for many fruitful interactions and providing the TAM instruction mixes and the mapping of TL0 to SPARC assembly instructions.

References

[1] D.E. Culler et al., TAM – A compiler-controlled threaded abstract machine, *J. of Parallel and Distributed Computing*, June 1993.

[2] B. Lee & A.R. Hurson, Dataflow architectures and multithreading, *IEEE Computer*, August 1994, 27–39.

[3] K.E. Schauser, D.E. Culler, & T. von Eicken, Compiler-controlled multithreading for lenient parallel languages, *Proc. of the 1991 Conf. on Functional Programming Languages and Computer Architecture*, Cambridge, MA, August 1991.

[4] R.S. Nikhil, ID language reference manual version 90.1, *Technical Report* CSG Memo 284-2, MIT Lab for Comp. Science, Cambridge, MA, 1991.

[5] Arvind, R.S. Nikhil, & K.K. Pingali, I-structures: Data structures for parallel computing, *Proc. of the Graph Reduction Workshop*, Sante Fe, NM, October 1986.

[6] S.C. Goldstein, The implementation of a threaded abstract machine, *Technical Report* UCS/CSD 94-818, University of California, Berkeley, EECS, May 1994.

[7] Sparc International, Inc., Menlo Park, California: The SPARC architecture manual, version 8 (Prentice Hall, 1992).

[8] E. Spertus, et al., Evaluation of mechanisms for fine-grained parallel programs in the J-machine and the CM-5, *Proc. of the 20th Int'l Symp. on Computer Architecture*, San Diego, CA, May 1993.

Biographies

Sridhar Kotikalapoodi received his B.S. degree in electronics engineering from Osmania University, India in June

1991 and M.S. degree in electrical and computer engineering from Oregon State University, Corvallis, in September 1994. Since October 1994, he has been working at Micro Linear Corporation, San Jose, CA, as a Design Engineer.

Ben Lee received his B.E. degree in electrical engineering in 1984 from the Department of Electrical Engineering at State University of New York at Stony Brook, and his Ph.D. degree in computer engineering in 1991 from the Department of Electrical and Computer Engineering at the Pennsylvania State University. While at Penn State, he was supported by a Government Scholarship from Korea. During 1985-1991, he was a Teaching Assistant, a Research Assistant, and an Instructor at the ECE Department at Penn State. Since 1991 Prof. Lee has been an Electrical and Computer Engineering Faculty Member at Oregon State University. His research interests include computer architecture, parallel processing, dataflow architectures, and multithreading.

Shih-Lien Lu received his B.S. degree in electrical engineering and computer science from the University California Berkeley, and his M.S. and Ph.D. degrees in computer science from the University California Los Angeles. He worked for USC/ISI on the MOSIS Project for six years while pursuing his Ph.D. degree. In 1991 he joined the Department of Electrical and Computer Engineering at Oregon State University. His current technical interests include self-timed circuits and systems, computer arithmetics and computer architecture. He is a member of the IEEE and IEEE Computer Society.

A.R. Hurson is a Computer Science and Engineering Faculty Member at The Pennsylvania State University. His research for the past 14 years has been directed toward the design and analysis of general as well as special purpose computer architectures. He has published over 140 technical papers in areas including computer architecture, parallel processing, dataflow architecture, database systems and database machines, multidatabases, object oriented databases, and VLSI algorithms. Dr. Hurson served as the Guest Co-Editor of special issues of the *IEEE Proceedings on Supercomputing Technology*, the *Journal of Parallel and Distributed Computing on Load Balancing and Scheduling*, and the *journal of integrated computer-aided engineering on multidatabase and interoperable systems.* He is the co-author of the *IEEE Tutorials on Parallel Architectures for Database Systems, Multidatabase Systems: An advanced solution for global information sharing, Parallel architectures for data/knowledge base systems*, and *Scheduling and Load Balancing in Parallel and Distributed Systems.* He is also the co-founder of the *IEEE Symposium on Parallel and Distributed Processing.*

Professor Hurson has been active in various IEEE/ACM Conferences and has given tutorials for various conferences on dataflow processing, database management systems, supercomputer technology, data/knowledge-based systems, scheduling and load balancing, and parallel computing. He served as a member of the IEEE Computer Society Press Editorial Board and an IEEE Distinguished speaker. Currently, he is serving in the IEEE/ACM Computer Sciences Accreditation Board.