

High-Throughput, Low-Memory Applications on the Pica Architecture

D. Scott Wills, *Member, IEEE*, Huy H. Cat, José Cruz-Rivera, *Member, IEEE*, W. Stephen Lacy, *Member, IEEE*, James M. Baker Jr., John C. Eble, *Student Member, IEEE*, Abelardo López-Lagunas, *Member, IEEE*, and Michael Hopper, *Member, IEEE*

Abstract—This paper describes Pica, a fine-grain, message-passing architecture designed to efficiently support high-throughput, low-memory parallel applications, such as image processing, object recognition, and data compression. By specializing the processor and reducing local memory (4,096 36-bit words), multiple nodes can be implemented on a single chip. This allows high-performance systems for high-throughput applications to be realized at lower cost. The architecture minimizes overhead for basic parallel operations. An operand-addressed context cache and round-robin task manager support fast task swapping. Fixed-sized activation contexts simplify storage management. Word-tag synchronization bits provide low-cost synchronization. Several applications have been developed for this architecture, including thermal relaxation, matrix multiplication, JPEG image compression, and Positron Emission Tomography image reconstruction. These applications have been executed using an instrumented instruction-level simulator. The results of these experiments and an evaluation of Pica's architectural features are presented.

Index Terms—Fine-grain parallelism, image processing architectures, through-wafer interconnects, MIMD architectures.

1 INTRODUCTION

TODAY'S commercial microprocessors are general purpose and low cost, providing reasonable computing power in PCs and workstations. For higher levels of performance, parallel computers offer greater computing power at higher cost, leveraging from high-volume microprocessor systems by using the same processors, memory, and software technologies.

Not all high-performance applications require the generality inherited from microprocessor-based parallel systems. High-throughput, low-memory applications (e.g., image processing, object recognition, and data compression) demand a small but specialized set of operations. Floating point units, large local memories, and associated caches are not required since a large fraction of operands come from integer data streams. Although the benefits of leveraging from commercial microprocessors are substantial, there is an opportunity for a more efficient dedicated architecture to address these applications. By better utilizing the silicon resource, this architecture can provide high-performance at low cost.

This paper introduces Pica, an architecture designed specifically for high-throughput, low-memory applications. A Pica node provides traditional RISC operations plus low overhead support for communication, synchronization, naming, and task and storage management. A small local memory (4,096 36-bit words) provides low-latency storage for instructions and intermediate data. This node complexity can be implemented using a fraction of the transistors

available on a chip in current technology, allowing *multi-node chips*. As integration densities increase, the extra area per chip can be used for additional processing nodes. Recent research has explored single-chip processing nodes [3], [7], [14]. Since these architectures target general applications, they employ external dense memory chips. Our approach does not use external dense memory arrays, reducing system cost for low-memory applications.

The implementation and performance of applications are critical in evaluating this type of architecture. A set of applications including thermal relaxation, matrix multiplication, data compression, and image reconstruction have been implemented on an instruction-level simulator for Pica. These experiments, presented in this paper, provide an evaluation of the architecture with respect to the following issues:

- Can algorithms be adapted for a high-throughput, low-memory architecture?
- What performance is possible for a collection of very simple nodes?
- Are parallel mechanisms effectively utilized in the architecture?

This paper describes the Pica high-throughput, low-memory architecture, and presents an evaluation of its application and performance on several applications. The next section describes related research. Section 3 presents the novel features of the Pica architecture. Then, Section 4 describes several applications developed for Pica and their simulated performance. An analysis of the effectiveness of Pica's architectural features for these applications is presented in Section 5. The status of the project and future plans are described in the final section.

• The authors are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0250.
E-mail: scotty@ee.gatech.edu.

Manuscript received 18 May 1994.

For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number 100865.0.

2 RELATED RESEARCH

A significant number of recent architectures have utilized multithreading to tolerate parallel overhead. In *coarse-grained* multithreading, a processing node switches rapidly between a set of resident tasks (or threads) to mask communication and synchronization delays with useful computation. In the time between context switches, the processor executes a single, active thread. *Fine-grained* multithreading (not to be confused with the term *fine-grained parallelism*, used elsewhere in this paper) relaxes the single-thread restriction, allowing instructions from multiple threads to execute concurrently in the processor's pipelines. Coarse-grained designs include Sparcle [1], the Message Driven Processor (MDP) [7], *T (pronounced "START") [16], and Pica. Fine-grained multithreaded architectures include HEP [21], Horizon [22], Tera [2], XIMD [26], and the MIT M-Machine [11], [12]. Fine-grained multithreading is most appropriate for processors with a large number of heavily pipelined functional units, since such designs benefit greatly from the increased instruction parallelism of multiple execution threads. Because each Pica node uses a single integer ALU with limited pipelining, coarse-grained multithreading is a more natural execution domain.

Since coarsely multithreaded processors cannot overlap context switching overhead with useful ALU operations from other threads, the context switch penalty limits the granularity (or run-length) of parallel tasks. The processor's strategy for managing the register state of threads greatly impacts the switching time. Sparcle (and, to a lesser extent, the MDP) both maintain multiple register contexts to avoid saving and restoring registers on context switches. While the switching operation is fairly efficient for a thread in a register context (14 cycles for Sparcle), transferring other threads to or from this active set generally requires saving and restoring entire register sets. Register load/store operations are reduced in *T since the compiler knows precisely at what points thread execution will suspend or resume and can place appropriate spill/restore code at these locations using live variable analysis. Pica's use of the context cache (inspired by the Named State Register File [17], [18]) differs from the above approaches in that it decouples register management from the context switch operation, performing load/store operations on a demand-driven basis. This ensures that a minimal number of memory operations are executed. To further improve context switch time, Pica uses dedicated hardware for task management (as opposed to the use of interrupts in Sparcle) and a simplified round-robin scheduling mechanism, achieving a context switching time of only two instruction cycles.

3 PICA ARCHITECTURE

The Pica execution architecture is designed for handling high message traffic consisting of small, ephemeral tasks. In order to achieve acceptable performance in this fine-grain domain, parallel overhead must be minimized, while node area is kept small. This section provides an overview of the microarchitecture, highlighting key aspects of the design.

3.1 Microarchitecture

The basic functional blocks of the Pica microarchitecture are shown in Fig. 1. The *network router* routes messages through the node, forming that node's contribution to the communication network. The *network interface* buffers incoming messages and signals the context manager that a memory context is required. When it obtains access to local memory, the network interface writes the message contents directly into the allocated, fixed-length context without interrupting the processor. The *datapath* consists of a 32-bit integer ALU and shifter, and several special-purpose registers. In order to keep design complexity and task swapping overhead low, a simple pipeline is employed. Instruction operands are accessed from a 32-word context cache, which supports two read and one write accesses on each cycle. The *instruction unit* fetches and decodes instructions for execution. It includes a 16-word instruction cache. In the current implementation, the instruction cache is organized as a fully associative cache with four-word lines. The replacement policy is FIFO. The *context manager* serves three functions:

- 1) It maintains a queue of suspended and ready tasks for execution,
- 2) It allocates task storage for incoming messages and deallocates storage as the tasks complete, and
- 3) It arbitrates requests from the network interface, instruction cache, and context cache controller for access to local memory.

Several functions provided by the modules are described in the following sections.

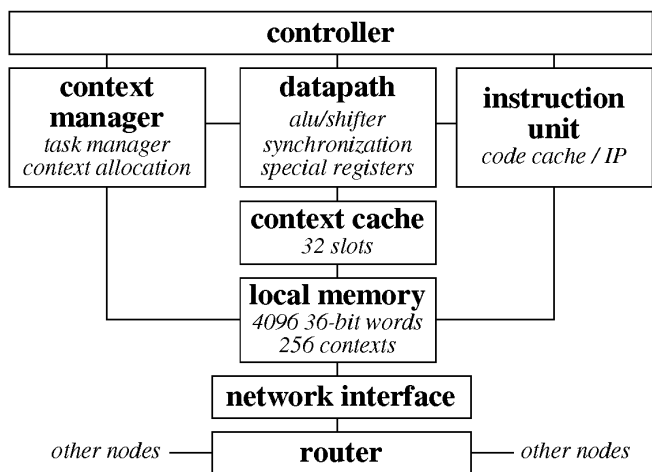


Fig. 1. Pica microarchitecture.

3.2 Storage Management

To support fast storage allocation and deallocation, node memory is divided into fixed length blocks, or *contexts*, of 16 words each. Contexts are used to store task data, persistent data objects, and program code. The size of a context is based on results from [24]. For a set of fine-grain benchmark programs, 95.8 percent of the tasks require less than eight words of storage. 99.0 percent require less than 16 words. Objects larger than 16 words are constructed using collections of 16-word contexts with some increase in access cost. Fixed-length contexts reduce storage management

costs, but result in unused space at the end of a context. This penalty is attenuated by the elimination of memory fragmentation and the fact that small objects tend to be ephemeral. In [24], the median lifetime of task storage is approximately 100 operations.

Each task can directly access two contexts through instruction operand addressing. The *Active Context* is created when the message that invokes the task is received, and is initialized with the message parameters. The *Data Context* can be any context on the node. These 32 words are accessed in the same manner as registers in a traditional RISC architecture. Load and store instructions are used to access data not available in the active and data contexts.

In the initial Pica implementation, local memory contains 4,096 36-bit words. For a proposed 50 MIPS node, this provides 328 bytes per MIPS, which is significantly lower than commercial microprocessor systems. Local memory is organized as 256 contexts. It is implemented with dense cell memories, and is organized in 256 rows of 16 words (576 bits) in length. As a row is accessed, it is stored in a row buffer to exploit spatial locality. The local memories of the nodes form a global address space, shown in Fig. 2. The maximum address space is four billion words in a one-million-node system. Support for a consistent shared address space is not provided.

node ID	context ID	context offset
(20 bits)	(8 bits)	(4 bits)

Fig. 2. Global address format.

3.3 Task Management

Because this architecture targets fine-grained tasks, low-overhead task management is a critical design goal. Task management includes the following operations:

- 1) *task creation*,
- 2) *task selection/activation*,
- 3) *task suspension*, and
- 4) *task termination*.

Pica supports these operations in hardware via the *context management table* (CMT). The CMT contains an entry for each local memory context. An entry indicates whether a context is allocated and if it is associated with a task. Additional task data (e.g., instruction pointer and data context) are also stored in a CMT entry.

Tasks are created via message-passing. When a message arrives at a node, a context is allocated to store the message contents, and a new task is created and queued for execution. The starting location of the task handler is defined in the message header. These operations are performed in the CMT without interrupting the currently executing task. Tasks requiring additional automatic or persistent storage can allocate contexts via the `ALLOCATE` instruction during their execution. Messages can also deliver and install handler code on nodes during system configuration. Handler lengths are not fixed. Several handlers can be stored in a single context, or one handler can spread across multiple contexts. Limited storage prohibits long, sequential procedures.

The multithreaded execution of tasks on a Pica node is nonpreemptive. Once a task begins execution, it continues until

- 1) it completes,
- 2) it encounters an unready synchronization, or
- 3) it voluntarily suspends using the `SUSPEND` instruction.

Tasks are scheduled for execution by visiting valid CMT entries in a round-robin fashion. When a task is activated, its active context, data context, and instruction pointers are copied from the CMT entry into dedicated CPU registers. When a task suspends, its current IP is saved in the CMT entry. When a task completes, it invalidates its CMT entry via the `END` instruction.

Because round-robin scheduling is used, a task blocked by a synchronization fault will poll the blocking memory location when its turn arrives. While more processor instructions are performed compared to explicit signaling schemes, this simpler approach reduces the overhead of task swapping and synchronization. The use of fixed-size contexts, dedicated CMT hardware, and a programming model with no explicitly loaded CPU registers allows task swapping to occur quickly. On the cycle a task suspends, the instruction pointer and data context are written into the CMT using the active context ID. On the following cycle, the preselected next task (round-robin) is loaded from the CMT into instruction pointer, active context, and data context registers.

3.4 Context Cache

Operands can be values in the active and data contexts, values in other contexts, and immediate constants. There are no explicitly loaded registers in the programming model. To avoid the latency associated with accessing dense cell memories, a context cache is included between local memory and the datapath.

The context cache assumes the role of registers in a traditional load-store architecture, providing fast, multiported access of operands. Unlike traditional registers, context cache entries are managed automatically at runtime based on temporal access locality. This is similar to the named-state storage described in [18]. In the current implementation, the context cache is organized as a direct mapped cache with 32 sets and one-word lines. The cache employs a write-back update policy. The context cache provides low-overhead task swapping, since entries are not swapped back to local memory unless the slot is required by another task (i.e., lazy context swapping).

Fig. 3 shows the fields of a context cache entry. The context ID is the cache tag which is compared to either the active task or data context register during a context slot access. Since Pica has a three-operand instruction set architecture, three simultaneous accesses of the cache occur during each processor cycle. The use of a direct mapped context cache allows these tag comparisons to occur in parallel with the datapath operation, but prior to the destination write. In the case of a context cache miss, the instruction is stalled while the missing lines are loaded from memory. When a task ends, cache entries associated with the active context are invalidated.

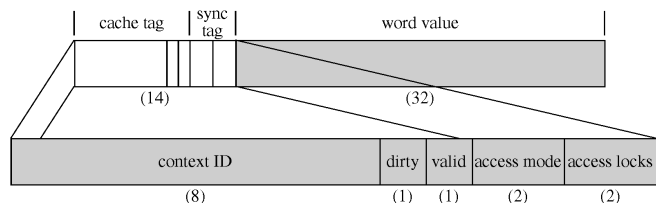


Fig. 3. Context cache fields.

Simulations show that context cache miss frequencies are comparable to register load/store frequencies in a traditional register-based architecture (67-95 percent hit rate). However register spilling and restoring due to task switches are substantially reduced.

3.5 Synchronization

Synchronization in a parallel system is composed of two components: a piece of named synchronization state, and an internode communication mechanism to allow access of that state. Communication is provided via message sends. The named state, which indicates the status of synchronization events (both data and control), is maintained as annotated storage locations. In this way, synchronization event naming is supported via the existing storage naming scheme already described.

Each word in local memory includes a 32-bit data value and a four-bit synchronization tag. Two bits of the tag are *read lock* and *write lock*. If a lock bit is set, the corresponding access to the word is blocked, resulting in the suspension of the active task. The two remaining bits define the *access mode* for the word. The access mode defines how the lock bits are to be modified during an access. For example, a data synchronization (e.g., full/empty bits in HEP [20], Tera [2]) is accomplished by initially setting the read-lock bit of a word. The mode is defined as **D-SYNC**, indicating that a write operation should clear the read-lock bit. Fig. 4 illustrates three of the supported access modes. A fourth access mode, **READ-WRITE** does not change lock bits during accesses. In this mode, the lock bits are directly manipulated using the **READ-TAG** and **WRITE-TAG** instructions to efficiently create other synchronization types (e.g., barrier synchronization).

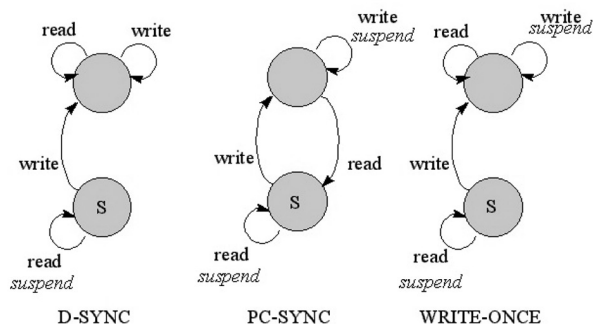


Fig. 4. Synchronization state diagrams.

To support low-cost synchronization, the testing and manipulation of the synchronization tag are performed by hardware in the context cache. When a three-operand in-

struction executes, the synchronization tags of the three operands are accessed and modified in accordance with the access modes. Three state machines (two read access and one write access) determine whether the access of an operand

- 1) results in a suspension, and
- 2) requires modification of the access bits.

Modifications of access bits are delayed until possible task suspensions and cache slot replacements are complete. Since these comparisons are completed concurrently with instruction execution, the cycle time is not significantly increased.

3.6 Communication

Low overhead communication is supported with the **SEND** family of instructions. These instructions can initiate a new message, inject one or two words from the active context into the network, and append an end-of-message marker. **SEND** instructions were first used in the Message Driven Processor [7]. When a message is received, a context is allocated in local memory to store it. This occurs without interrupting the active task. The maximum message length is 16 words.

The message format is shown below:

destination	type	message ID	arg-1	arg-2	arg-3	...	arg-n
-------------	------	------------	-------	-------	-------	-----	-------

The *destination* field specifies the destination of the message and is removed in route. The *message type* field defines the handling of the message when it arrives at the destination. The *message ID* field defines the handler to be invoked at the destination. Finally, a list of arguments is passed to the invoked handler. The maximum arguments allowed is 16, since they will be stored in the active context on the destination.

3.7 Sequential Instructions

Efficient support for sequential code sequences remains an important requirement, even for fine-grain tasks. The Pica instruction set retains a basic suite of sequential instructions: conditional branches, logical operations, shifts, and arithmetic operations. Instruction word operands specify slots in the active and data context, and immediate values. The **LOAD** and **STORE** instructions support accesses to other contexts in local memory by specifying both the context ID and offset. Synchronization locks are tested for local memory accesses.

3.8 Network Requirements

High-throughput applications and a low-memory processing node make I/O a critical aspect of this architecture. Experiments indicate that each executed instruction requires between 0.1-0.5 words of I/O. If a Pica processor executes instructions at 50 MIPS, the I/O rate can be as high as 25 Mwords/sec. This requires 800 Mbits/sec of I/O. Conventional interconnect technology is inadequate for supporting dense, three-dimensional arrays of multinode chips. To satisfy the high I/O requirement, an integrated three-dimensional network incorporating through-wafer optoelectronic interconnect is being developed at Georgia Tech [25]. A detailed study of this network, including a definition of the topology, routing algorithms, and performance comparisons to other networks, is presented in [13]. Message traces collected from the applications described in this paper have

been executed on a flit-level network simulator. For these problem sizes, additional communication latency due to blocking is not significant in overall execution time, suggesting that throughput of the proposed network is adequate.

4 APPLICATIONS

Most parallel algorithms are implemented assuming the availability of a large local memory. Adapting these algorithms to an architecture with 3,000 times less memory per MIPS than the typical Mbyte/MIPS microprocessor node can be difficult. The first step in evaluation is the implementation of some important applications in this low-memory architecture.

Four applications have been implemented on Pica. The first two applications, thermal relaxation and matrix multiplication, are widely used, well-understood parallel benchmarks. In addition, problems in image compression (JPEG) and medical imaging (Positron Emission Tomography) were chosen to illustrate the feasibility of a low-memory, high-throughput approach to practical problems.

The applications were executed using an instruction-level simulator (XP2). XP2 includes context cache modeling, constant instruction latency, and message delivery latency based on network distance and message length. XP2 is instrumented to collect statistics on tasks, messages, instructions, and contexts. System configuration time (i.e., installing code and predefined data) is not included in the execution time. Problem sizes and machine configurations were selected based on limitations of the XP2 execution platform (a SPARCstation 2 with 64 MB physical memory). However, all applications can be scaled to larger data sets and machine configurations. All applications were programmed in assembly language. An extended C compiler for Pica is currently being developed.

A description of the applications, and an evaluation of their execution on Pica is described in the following sections.

4.1 Thermal Relaxation

The first application solves Laplace's differential equation ($\nabla^2 u = 0$) using the Gauss-Jacobi method. In this application, thermal equilibrium is computed for a thin, rectangular plate in contact with heat sources. Thermal relaxation is well suited for data parallel execution. The algorithm divides the plate into an array of equal sized cells. On each iteration, each cell assumes the average temperature of its four neighbors. This process is repeated until all temperatures converge.

4.1.1 Thermal Relaxation Implementation on Pica

For the implementation on Pica, the problem size is a 64×64 cell plate with each node computing the value of four cells. The 4,096-cell plate is simulated for 50 iterations on 1,024 nodes. On one iteration, each cell broadcasts its current temperature to its four neighbors. It then computes its new temperature as the average of its neighbor's current values.

Each cell of the plate is represented by a single task. This algorithm uses a special context on each node to hold data from neighboring nodes. For each cell, two of its neighbors

reside locally and two reside on another node. Neighbors on different nodes must communicate with messages. For local neighbors, a shared data context is used to communicate.

This parallel algorithm must synchronize to ensure that all nodes are on the same iteration. However, a barrier synchronization is overly restrictive. Each cell can provide its new temperature only after all old values have been consumed. Each value should be accessed exactly once. This fuzzy synchronization [10] is supported using the producer-consumer primitive. Simulations on Pica have shown that a producer-consumer synchronization provides an order-of-magnitude lower synchronization overhead than a barrier synchronization. The concurrency for this application is shown in Fig. 5. Other results of the execution are discussed in Section 5.

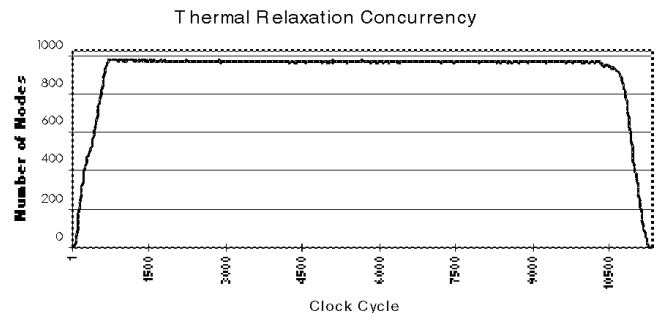


Fig. 5. Thermal relaxation concurrency.

4.2 Parallel Matrix Multiplication

Matrix multiplication is a popular benchmark for parallel machines since its well-understood operation is highly parallel. The square subblock decomposition algorithm for parallel matrix multiplication [8] was chosen to evaluate the Pica architecture since it is well suited for message-passing execution. The implemented algorithm multiplies two $M \times M$ input matrices, A and B . The algorithm assumes that a set of virtual processors, P_{ij} ($1 \leq i, j \leq N$), are organized into a logical $N \times N$ array. At the start of the algorithm, each virtual processor, P_{ij} receives a partition of each input matrix, A_{ij} and B_{ij} , where each submatrix has dimensions $(M/N) \times (M/N)$. P_{ij} also maintains an $(M/N) \times (M/N)$ submatrix, C_{ij} , which contains a partition of the product matrix at completion. Finally, each processor maintains temporary storage for two $(M/N) \times (M/N)$ submatrices used in intermediate calculations of the final partition, C_{ij} . These are denoted by $P_{ij}.A$ and $P_{ij}.B$. The algorithm uses three parallel procedure calls, *MultiplyAccumulate* (*Macc*), *RowBroadcast*, and *ColumnRoll*. These procedures and the algorithm itself are described by the pseudocode

Fig. 6 (where the “ \rightarrow ” operator indicates the invocation of a procedure by a specific virtual processor).

4.2.1 Matrix Multiplication Implementation on Pica

In the Pica implementation of the algorithm, the three parallel procedures, *Macc*, *RowBroadcast*, and *ColumnRoll*, are defined in

Fig. 6 are realized using three task handlers: *Macc*, *RowBroadcast*, and *ColumnRoll*. At the start of the program, four

```

Parallel Procedures (A, B, C, Anew, Bnew are matrices):
Pi,j → MultiplyAccumulate (A, B, C)      ;; returns C + A * B
Pi,j → RowBroadcast (Anew)             ;; Pi,j·A = Anew, 1 ≤ j ≤ N
Pi,j → ColumnRoll (Bnew)              ;; Pi,(j+1) mod N·B = Bnew

Algorithm:
Pi,j·B = Bi,j, 1 ≤ i, j ≤ N
for k = 0 to N-1 {
  Pi,(i+k) mod N → RowBroadcast (Ai,(i+k)), 1 ≤ i ≤ N
  Ci,j = Pi,j → MultiplyAccumulate (Pi,j·A, Pi,j·B, Ci,j), 1 ≤ i, j ≤ N
  Pi,j → ColumnRoll (Pi,j·B)
}

```

Fig. 6. Matrix multiply pseudocode.

Macc tasks corresponding to four virtual processors are created on each physical processor. Each *Macc* task performs multiply-accumulate operations by maintaining pointers to four data contexts containing the submatrix operands manipulated on each iteration of the algorithm. To evaluate the efficiency of Pica's communication and synchronization mechanisms, a relatively small grain size of nine elements was chosen for each submatrix, allowing it to fit into a single 16-word context.

Communication of submatrices between *Macc* tasks is accomplished via the family of SEND operations. When a message arrives at a processing node, its contents are copied into a fixed-length context allocated by the context manager and a *RowBroadcast* or *ColumnRoll* handler is invoked to process the message. Because the network interface copies a received message into local memory directly without interrupting the executing task, the overhead of buffering incoming messages is minimal. Since storage for each message is allocated and initialized using dedicated hardware, the *RowBroadcast* and *ColumnRoll* tasks need only pass the address of the new context to the appropriate *Macc* task. Each submatrix context is explicitly deallocated by the *Macc* task once it is consumed in a multiply-accumulate operation; the use of fixed-size contexts allows a submatrix to be deallocated via a single FREE instruction.

Pica's use of tagged memory locations allows an efficient producer-consumer synchronization to be implemented. A *Macc* task is prevented from prematurely performing its multiply-accumulate operation by initializing its A and B

submatrix pointers as empty data-synchronizations at the start of each iteration. This overhead, a single WRITE-TAG instruction for each pointer, is small. An attempt to access a pointer before it is updated results in the automatic suspension of the *Macc* task. Because global barriers are not used to separate each iteration of the algorithm, it is possible for messages intended for different iterations to arrive at a node while a *Macc* task is processing submatrices from a previous iteration. To prevent message handlers from prematurely updating submatrix pointers or from executing out of order, each *Macc* task maintains an iteration count in its context. Each arriving message also specifies the iteration for which the data is intended. The task invoked to process the message compares the iteration count in the message against that of the target *Macc* task. If the counts match, the message-handler task updates the pointer location; otherwise, it voluntarily suspends and attempts the test again when it reaches the head of the task queue later. The number of polling attempts required is small since the natural data dependencies of the algorithm limit the overlap of messages from different iterations. This was confirmed by simulation.

4.2.2 Matrix Multiplication Simulation Results

The Pica implementation of the square subblock decomposition algorithm was simulated for 150×150 input matrices using 625 physical processors. Results of the simulation show that the average task consumes 186 processor cycles during its lifetime and has an average run length of 53 cycles. These statistics include 2,500 computation-intensive

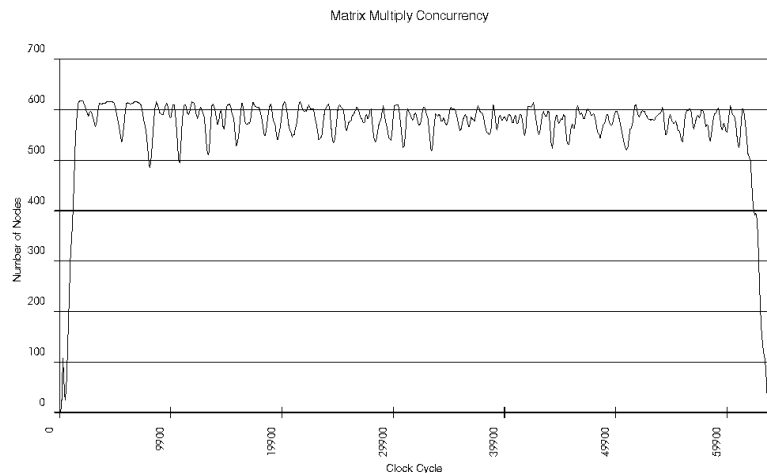


Fig. 7. Matrix multiply concurrency.

Macc tasks which exist for the duration of the program and 185,000 message-handler tasks whose running times are considerably shorter.

The average cycles consumed by message-handler tasks are of primary interest since these tasks are the predominant sources of communication and synchronization overhead for this application. Examination of other statistics reveals that a message-handler task consumes an average of 24 processor cycles and experiences less than two suspensions due to unready synchronizations. The latter statistic indicates that *RowBroadcast* and *ColumnRoll* tasks require an average of only two unsuccessful polling attempts before they are able to deliver their submatrices to *Macc* tasks. Hence, the use of busy-waiting for suspended tasks does not seriously impact processor utilization (which is approximately 90 percent). High processor utilization indicates that maintaining multiple threads allows communication and synchronization delays to be masked with useful computation. This is confirmed by the concurrency plot in Fig. 7. If global barriers had been used to guarantee lock-step execution, the plot would exhibit large, periodic drops in concurrency. The sustained level of concurrency in Fig. 7 results from overlapped computation and communication.

4.3 JPEG Image Compression

One important application in the field of image processing is image compression. The JPEG image compression standard was developed by the Joint Photographic Experts Group, which reports to both the ISO and the CCITT [23]. The JPEG standard includes several different algorithms for compression. An algorithm based on the Discrete Cosine Transform (DCT) was implemented on Pica. The steps involved in the JPEG encoder are as follows:

- 1) The image is first divided into 8×8 pixel blocks.
- 2) The two-dimensional DCT is calculated for each block. This results in an 8×8 matrix of output coefficients, which represent the spatial frequency components of the block.
- 3) The output coefficients are quantized, which emphasizes the low-frequency components and de-emphasizes the high-frequency components.
- 4) The quantized values are then encoded using Huffman coding. The DC component is treated differently than the AC components in this step. Instead of encoding the actual amplitude, the difference between the amplitude of the DC component and that of the previous 8×8 block is encoded.
- 5) The AC components are encoded using a combination of run-length encoding and Huffman coding. The coefficients are processed in a zigzag manner in order to encode the low-frequency components before the high-frequency components.

After the image has been divided into the 8×8 blocks, each block can be transformed and quantized independently of other blocks. The AC components of each block can be encoded in parallel. However, the DC components must be encoded sequentially because differential coding is used. Finally, the encoded data must be output in a specific order. This step, therefore, must also be executed sequentially.

In addition, the 8×8 DCT can be calculated using row-column decomposition, where a one-dimensional DCT is applied to each row of the 8×8 block, and then applied again to each column of the result. Therefore, for each 8×8 block, the rows can be transformed in parallel; the results are redistributed; then the columns are transformed in parallel.

4.3.1 JPEG Implementation on Pica

To implement the JPEG algorithm, the Pica nodes were divided into four different groups, depending upon the part of the algorithm that each performs:

- 1) Reader nodes—These nodes read in an 8×8 block and send the data out to a group of row nodes.
- 2) Row nodes—These nodes each receive eight data points from a reader node (representing one row of the 8×8 block), calculate the DCT of the data, and send the results out to a group of column nodes.
- 3) Column nodes—These nodes each receive data points from eight different row nodes (representing the data for a single column), calculate the DCT of the data, quantize the results, and then send the data to an encoder node.
- 4) Encoder nodes—These nodes receive 64 quantized coefficients from the column nodes, encode the data, and then output the results.

A single 8×8 block of the image is read in by a single reader node, divided into rows, and sent to a group of eight row nodes. The data then moves to a group of eight column nodes, and is finally sent to a single encoder node. This group of 18 nodes can operate on blocks in a pipelined fashion, processing up to four blocks simultaneously. For efficient execution, each group of nodes must process a number of blocks. For this implementation, 16 groups of 18 nodes (a total of 288 nodes) were used to compress a 256×256 image. The image is divided into 1,024 8×8 blocks, and each group of nodes processes 64 different blocks.

To pipelined execution, it is necessary to synchronize nodes so that each processes data from one block at a time. For example, a row node cannot begin processing a new block of data until it has sent the results of its previous calculations to the column nodes.

Each sending node creates a producer-consumer synchronization. Before sending data, the node accesses this synchronization and suspends if the data cannot be consumed. When the receiver is ready for new data, it sends a message to the sending node enabling it to send. In the cases where a node broadcasts data to multiple nodes (e.g., each row node sends data to eight column nodes), a counter is added to create a barrier synchronization. This mechanism is used to synchronize data transfer between the reader and row nodes, between the row and column nodes, and between column and encoder nodes.

Encoder nodes must also be synchronized so that encoded data is output in the correct order. This is accomplished by passing a token around a ring of nodes. Each encoder node maintains a producer consumer synchronization. The encoder node accesses this synchronization before outputting its data. The system is initialized with one encoder node enabled to send its results. When this node

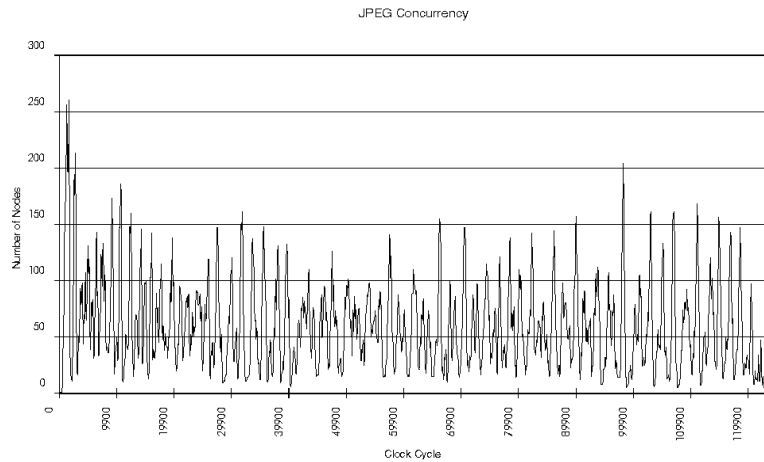


Fig. 8. JPEG concurrency plot.

completes, it sends a message to the next encoder, enabling its transmission. The transmission of the final node reinitializes the ring for the next pipelined iteration.

4.3.2 JPEG Simulation Results

The compression of a 256×256 image takes 124,287 cycles, which is 2.5 mS 50 MIPS nodes. The concurrency plot in Fig. 8 shows that the number of active processors initially reaches a sharp peak as the 16 “pipelines” (groups of 18 nodes) fill up with blocks of data. Following this, sequential nature of the output phase begins to reduce the number of active nodes. This is inherent in the algorithm, as encoded data values must be output in sequential order.

4.4 Positron Emission Tomography (PET)

This application, used in cardiology diagnosis, computes a three-dimensional image from sensor data. The physical configuration of PET image reconstruction is shown in Fig. 9. The objective is to reconstruct an $N \times N$ clinical image, λ , from the sinogram (projection) data, $n_{t\theta}$, acquired in the form of photon counts across a number of bins (tubes), t , located at different view angles, θ . The image reconstruction process can be performed via the Maximum Likelihood-Expectation Maximization (ML-EM) algorithm described by the iterative update equation [19],

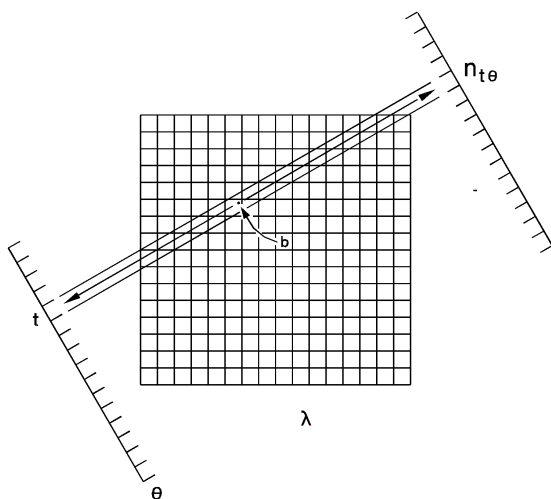


Fig. 9. Schematic of PET application.

$$\lambda_{k+1}(b) = \frac{\lambda_k(b)}{\xi(b)} \sum_{t=1}^T \frac{n(t)p(b,t)}{\sum_{b'=1}^B \lambda_k(b')p(b',t)} = \lambda_k(b) \cdot \beta(b) \quad (1)$$

where $\lambda_k(b)$ is the k th estimate of the radionuclide density function being imaged at box b , $n(t)$ is the number of photon counts at tube t , $p(b,t)$ is the probability that a photon emitted at box b is detected by tube t (where the indexing of the two-dimensional array λ is stacked with respect to b and the angular indexing of the detector-count, $n_{t\theta}$, is absorbed by stacking with respect to t). The $\xi(b)$ array is a scaling factor array that remains constant over the iterations and the $\beta(b)$ array is the update factors array, which captures the summations and scaling factors of (1). The summations in the denominator and numerator of (1) correspond to a projection (ray-sum) operation across the λ array and a back-projection (replication) operation across the β array, respectively.

Even though the ML-EM algorithm has been demonstrated to provide higher quality image reconstructions than can be achieved via the more commonly employed Filtered Backprojection (FBP) algorithm, the use of the ML-EM algorithm within the clinical environment has been thwarted by the large computational time and memory requirements involved in its implementation. This situation has led to numerous research efforts toward the development of efficient ML-EM algorithmic formulations targeting a variety of computational platforms, with particular effort being geared toward parallel implementations. However, these attempts have failed to provide the necessary speed-ups required for the ML-EM to be used clinically for large PET datasets. The main limiting factor in these formulations has been the execution models employed, as these have limited the performance levels and problem sizes that can be tackled. Fortunately, as evidenced by the simulation results to be presented below, the high-throughput, low-memory execution model supported by Pica offers an excellent platform for a novel operator-based formulation of the ML-EM algorithm that is expected to provide image reconstructions within clinical timeframes.

4.4.1 PET Implementation on Pica

The memory intensive nature of the ML-EM algorithm

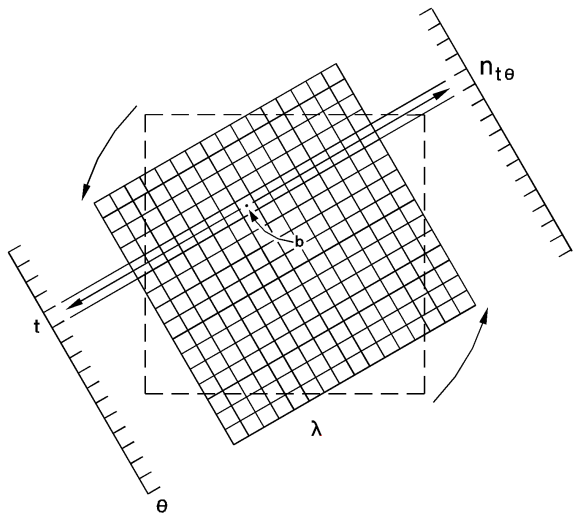


Fig. 10. Rotated PET approach.

arises particularly because of the large number of $p(b, t)$ values that must be stored and the large datasets targeted. A typical PET dataset consists of 21 slices (images) of 256×256 pixels each with data obtained across 120 angles. In developing an algorithmic formulation that is well suited to the Pica execution model, an attempt must be made to optimally partition the problem so that the low-memory per node specification does not pose a major limitation. In this respect, we have developed an operator-based formulation that is based on the observation that if only geometric factors are considered, the $p(b, t)$ factors in (1) can be absorbed by a sufficiently accurate rotation of the λ and β arrays (Fig. 10) [6]; allowing the memory-intensive ML-EM algorithm to be cast into a communications-intensive formulation. This observation allows (1) to be recast into the operator-based equation,

$$\lambda_{k+1} = \frac{\lambda_k}{\xi} \sum_{\theta} R_{-\theta} \left\{ B_t \left(\frac{n_{t\theta}}{P_{t\{R_{\theta}(\lambda_k)\}}} \right) \right\} = \lambda_k \cdot \beta \quad (2)$$

where three vector operators are introduced: R_{θ} , B_t , and P_t . R_{θ} is a rotation operator that rotates an $N \times N$ array by the specified angle θ , B_t is a back-projection operator which replicates each element of a column vector of length N across the corresponding row of an $N \times N$ array, and P_t is a projection operator that calculates the ray-sum of an $N \times N$ array on a row-by-row basis.

The parallelization of the operator-based ML-EM equation was performed as follows. Since the projection and backprojection operations work over individual rows of appropriately aligned (rotated) arrays, the most natural way to distribute the data arrays among PEs is in a row-wise manner (an alternative block-wise partitioning is discussed in [6].) In order to pipeline the projection and backprojection operations and to subscribe to the low-memory per node constraint, data allocation is such that the P-PEs are allocated one row each of the 256×256 λ_k and $R_{\theta}\{\lambda_k\}$ arrays and the sine and cosine values for the 120 view angles to be processed (for a total of 752 words), while the B-PEs are allocated one row each of the 256×256 β and

ξ arrays, the sine and cosine values for the 120 view angles, and 120 $n_{t\theta}$ values (for a total of 872 words). The collaboration between P- and B-PEs is then as follows. Each P-PE is responsible for reconstructing one row of the image array, λ , while each B-PE is responsible for computing one row of the update array, β . The two sets of PEs collaborate in a pipelined fashion by having the P-PEs rotate the λ_k array to a particular view angle, θ , compute the projections of the rotated array, $R_{\theta}\{\lambda_k\}$, and then send these values to the B-PEs in charge of the same row of the β array as the P-PE's row of the λ array. The B-PEs, which are provided with the necessary $n_{t\theta}$ values, take the received projection values as arguments in computing this angle's contribution to the update factors array, β , backproject the $n_{t\theta}/P_t\{R_{\theta}(\lambda_k)\}$ value across a temporary array, β' , and then rotate this array to the negative of the view angle being processed before adding it to the β array. Once all views have been processed, the B-PEs divide the β values by the corresponding ξ values and send the scaled β values to the appropriate P-PEs so that the latter may update their assigned λ_k values, producing λ_{k+1} .

The implementation of the *projection* and *backprojection* operations are straightforward since each task is in charge of one row of either the λ or β array. The *projection* operations consist of adding all the elements within one row of the rotated λ array, while the *backprojection* operations consist of replicating the scaled detector-count value across all columns of a given row of the β array. The *rotation* operator, however, is more complicated. All tasks within a given group collaborate on a rotation by exchanging pixels according to the mapping established by the axis transformation equations $x_r = x \cdot \cos \theta + y \cdot \sin \theta$ and $y_r = -x \cdot \sin \theta + y \cdot \cos \theta$. Each task obtains those pixels of the nonrotated matrix that map into the pixel locations in its domain (row) through *remote access* requests to the task holding the desired pixels. Upon receiving a *remote access* request a task will send a *reply* message which includes the requested pixel value to the originating task. Since the quality of the reconstructed images is heavily dependent on the accuracy of the rotations, the algorithm is modified so that each task calculates the coordinates of and requests the values for four "subpixels" within each pixel. This process is equivalent to replacing each pixel in the nonrotated matrix by a block of four "subpixels," each with value equal to $1/4$ the original.

Two forms of communication are identified in the Pica MLEM; intragroup communications within the boundaries of the B-PE and P-PE sets (*rotation*), and intergroup communications, where a single P-PE communicates with a single corresponding B-PE, and vice versa (*backprojection* and *update*). In all cases, barrier synchronization mechanisms are used. To avoid running out of contexts during the rotation stage, the tasks must voluntarily suspend after every pixel request to clear up their message queue. During the *backprojection* stage, where the P-PEs send their corresponding B-PEs partner the computed projection values, barrier synchronizations assure that the B- and P-PEs are aligned to the same view-angle. In the *update* stage, the P-PEs are refrained from starting a new iteration until each receives all the update factors from their corresponding B-PE partners. Once this condition is satisfied, the P-PEs will proceed to update their copy of the λ array.

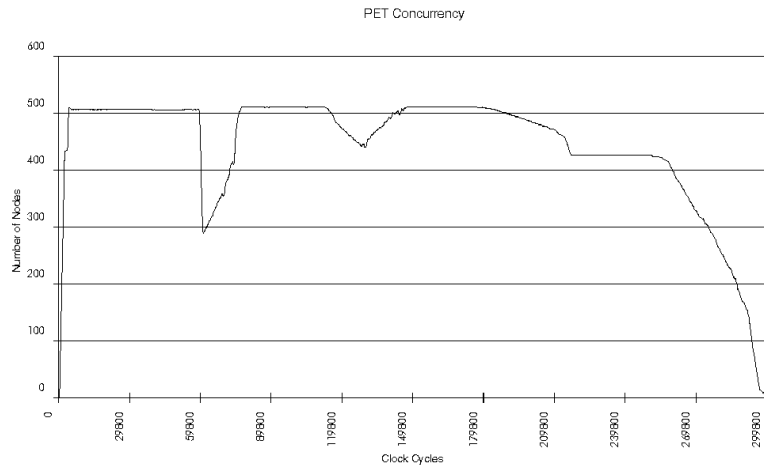


Fig. 11. PET concurrency.

TABLE 1
SUMMARY OF SIMULATION RESULTS

	Thermal Relaxation (64 × 64)	Matrix Multiply (150 × 150)	JPEG	PET
Nodes Used	1,024	625	288	513
Network Dimensions	10 × 11 × 10	9 × 9 × 8	16 × 8 × 3	4 × 9 × 16
Total Execution Cycles	11,323	64,206	124,287	302,277
Total # of Dynamic Contexts	404,037	197,525	194,623	2,810,304
Avg # of Dynamic Contexts	8,313	10,565	1,306	26,627
Avg Contexts Allocated (% of system)	3.8 %	6.6 %	1.8 %	20.3 %
Peak Dynamic Contexts (% of system)	3.4 %	7.9 %	2.1 %	21.0 %
Code Size (% of system)	17.2 %	7.3 %	9.3 %	10.4 %
Total Tasks Created	401,990	189,401	177,440	2,784,449
Avg Active Cycles per Task	23.3	186.0	30.9	46.3
Avg Run Length per Task	12.5	53.0	26.9	11.4
Avg Suspended Cycles per Task	150.7	1,072.5	179.3	54.8
% Execution Time Spent Polling	2.0 %	3.6 %	70.0 %	0.0 %
Context Cache Hit Rate	67 %	92 %	82 %	95 %
Avg Cycles per Message Creation	2.0	5.6	2.3	2.5
Total # of Messages	401,989	189,400	177,439	2,784,448
Avg Message Length	2.0	8.2	2.1	2.5
Avg Time in Flight per Message	8.9	13.6	8.5	9.3

4.4.2 PET Simulation Results

Discrete-event simulation results indicate that the average time Pica takes to process one view of a 256×256 image is approximately 90,000 cycles. Thus, the total execution time for the reconstruction of the typical dataset associated with PET cardiac studies (21 slices, 32 iterations, 120 views) would be 44 seconds (for a 4,096 node system with the targeted 50 MIPS processors and the simultaneous reconstruction of eight image slices at a time). This execution time translates to well over two orders of magnitude compared to current sequential implementations on high-end workstations, and compares favorably to other parallelized formulations [4], [15] (particularly with respect to the dataset sizes that can be tackled).

The operator-based ML-EM formulation allows efficient use of Pica's processing power, as the concurrency plot of Fig. 11 shows. While a full-scale simulation of the operator-based ML-EM algorithm has been carried out for the reconstruction of a 256×256 image from projection data acquired across 120 view angles [6], Fig. 11 corresponds to a

scaled-down simulation of the reconstruction of a 256×256 image from four view angles. The scaled-down simulation allows the details of the reconstruction process to be better assessed and can be readily extrapolated to larger datasets. Each plateau in the plot corresponds to the various views being processed. Note that the shape of each region will depend on the actual view being processed and the fact that the P- and B-PEs operate in a pipelined manner, processing contiguous views simultaneously.

5 PERFORMANCE ANALYSIS OF PICA

In this section, simulation results obtained from the four applications are used to evaluate Pica's mechanisms for

- 1) *storage management*,
- 2) *synchronization*,
- 3) *task management*, and
- 4) *communication*.

A summary of simulation statistics is shown in Table 1.

Pica's limited local memory was not an obstacle to effi-

TABLE 2
DYNAMIC INSTRUCTION PROFILE

	Thermal Relaxation (64 × 64)	Matrix Multiply (150 × 150)	JPEG	PET
ALU	48.09 %	52.97 %	53.77 %	59.37 %
Branch	15.12 %	10.66 %	14.98 %	18.88 %
Storage Management	17.30 %	29.34 %	10.87 %	4.21 %
Synchronization	0.01 %	0.88 %	0.94 %	0.00 %
Task Management	10.79 %	3.14 %	9.35 %	12.12 %
Communication	8.62 %	3.02 %	5.78 %	5.42 %

cient implementation of high-throughput applications. For the examined applications, the storage requirements are well within system limits. For data storage, PET utilized an average of 20 percent of all available contexts, while Thermal Relaxation utilized an average of only three percent of available contexts. For code storage, Matrix Multiply, JPEG, and PET utilized between seven percent to 11 percent of the available contexts. The PET application operates on image sizes of clinical interest while consuming less than one third of the total system memory.

Another aspect of storage management is the efficiency of Pica's mechanisms for allocating, deallocating, and accessing local memory. The use of fixed-sized contexts allows rapid allocation and deallocation of contexts while avoiding memory fragmentation. A context size of 16 words is justified by the fine-grained nature of tasks and its similarity to the size of a typical register file. The context cache improves single-threaded performance by reducing the number of LOAD/STORE instructions required. The PET application illustrates the effectiveness of data context accesses. Loads and stores constitute only 4.2 percent of the executed instructions. Table 2 shows the dynamic instruction profile of the four applications.

Pica supports synchronization via tagged memory locations. This minimizes synchronization overhead by overlapped synchronization checking with ALU operations. If this can be accomplished without increasing datapath cycle time, synchronization overhead is reduced to tag initialization. This overhead is to be less than one percent of executed instructions in all applications. More complex synchronization types (e.g., barriers) are implemented by combining primitive tag types with datapath operations. The 4-bit synchronization tag represents a memory area overhead of 11 percent.

Pica's support of task management is designed to minimize context switching overhead. This is critical for efficient execution since task run lengths range from 11 to 53 cycles. The context switching penalty has two components. The initial cost of a task swap is two cycles. An additional cost is incurred when conflict and capacity misses occur in the context cache. Simulations suggest reasonable overall context cache hit rates. Both PET and Matrix Multiply exhibit average hit rates greater than 90%, while JPEG achieves an 82 percent hit rate. The lower hit rate for Thermal Relaxation (67 percent) is due to the presence of compulsory misses encountered by the large number of ephemeral tasks which execute only once. Memory accesses due to context cache miss rates are significantly lower than load/store memory accesses for register-based machines.

Suspended tasks remain in the active task queue, polling

the locking synchronization. This approach simplifies task management hardware, reducing chip area requirements. Moreover, the number of processor cycles consumed by a polling attempt is small since the locking synchronization can be checked in one or two cycles and the context switching time is two cycles. Simulations suggest the percentage of polling cycles is less than four percent for Thermal Relaxation, Matrix Multiplication, and PET. This is verified by examining the high levels of system concurrency in Fig. 7 and Fig. 11 (these plots do not include processors that are polling or context switching). The high polling percentage for JPEG (70 percent) is a consequence of the inherent sequentiality of the algorithm.

Pica supports low overhead message sending operations via the SEND instructions. A message can be injected into the network using a small number of instructions. In Matrix Multiply, the average message length is 9.2 words (including the message header), but requires only 5.6 instructions to inject. Message reception overhead is reduced by buffering received messages without interrupting the executing task. The number of words of I/O per instruction for these applications ranges from 0.04 words/instruction in Matrix Multiplication to 0.09 words/instruction in Thermal Relaxation.

6 RESEARCH STATUS

The initial Pica design being implemented targets 37 Kbits of static storage (220K transistors, 15 mm^2) and 100K transistors for the datapath and controller. This should allow a four-node chip using current technology. A 3.2 Gbits/second/chip optical network is also being designed [25]. Test components of the processing node and optical network are being fabricated. Simulations of the network performance have been performed [13].

Plans for a full-scale prototype are underway, but are still preliminary. In a target prototype system, each processing node will contain 4,096 36-bit words of local memory (256 contexts), a 32-bit integer processor, and a network interface. The target node performance is 50 MIPS. A chip contains four Pica nodes and 3.2 Gbits/sec I/O bandwidth. The full-scale prototype will employ a 2.5 supply voltage in addition to other low power techniques to keep total chip power below 500 mW.

Packaging issues related to this project are being addressed in the systems integration, optoelectronics, and thermal management thrust areas of Georgia Tech's Low Cost Packaging Research Center. In a full-scale system (4,096 nodes), a processing plane contains 64 chips (256 nodes, 12,800 MIPS) and measures approximately 10 cm by 10 cm.

Sixteen planes contain 1,024 chips (4,096 nodes, 204,800 MIPS) and fit inside a cube 10 cm on a side. 820 Gbits/sec of system I/O bandwidth is available from chips on the top and bottom surfaces of the cube. Sides of the cube are available for power and cooling mechanical connections.

Smaller systems (64 nodes to 256 nodes) can employ less aggressive interconnect and more conventional cooling techniques. Increases in VLSI technology will simplify system design by increasing node density and reducing the number of chips in a system.

A compiler for a higher-level language (Pica-CC) and a runtime system is also being developed for this architecture.

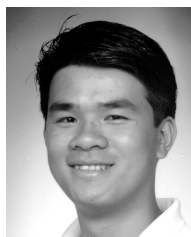
This paper has presented Pica, a fine-grain, massively parallel architecture targeted for high-throughput, low-memory applications. Pica reduces the percentage of VLSI resources traditionally devoted to dense cell memory to allow more processing nodes-per-chip and higher utilization of transistors. Reduced memory-per-node is compensated for using area arrays of optoelectronic devices combined with through-wafer transmission to allow off-chip bandwidth to scale with the number of nodes-per-chip. As the JPEG and PET examples illustrate, practical applications can benefit from a high-throughput, low-memory approach to parallel computing. Additional applications in the object recognition domain are currently being developed. Using optical interconnect and a low-memory approach to parallel computation, the Pica system intends to break the one-node-per-chip barrier in MIMD architectures.

REFERENCES

- [1] A. Agarwal, J. Kubiawicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin, "Sparcle: An Evolutionary Processor Design for Large-scale Multiprocessors," *IEEE Micro*, June 1993.
- [2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," *Proc. Int'l Symp. Computer Architecture*, pp. 1-6, Sept. 1990.
- [3] W.C. Athas and C.L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *Computer*, vol. 21, no. 8, pp. 9-24, Aug. 1988.
- [4] C.S. Butler and M.I. Miller, "Maximum a Posteriori Estimation for SPECT Using Regularization Techniques on Massively Parallel Computers," *IEEE Trans. Medical Imaging*, vol. 12, pp. 84-89, Mar. 1993.
- [5] C.M. Chen, S.Y. Lee, and Z.H. Cho, "Parallelization of the EM Algorithm for 3-D PET Image Reconstruction," *IEEE Trans. Medical Imaging*, vol. 10, pp. 513-522, Dec. 1991.
- [6] J.L. Cruz-Rivera, E.V.R. DiBella, D.S. Wills, T.K. Gaylord, and E.N. Glytsis, "Parallelized Formulation of the Maximum Likelihood Expectation Maximization Algorithm for Fine-Grain Message-passing Architectures," *IEEE Trans. Medical Imaging*, pp. 758-762, Dec. 1995.
- [7] W.J. Dally, J.A.S. Fiske, J.S. Keen, R.A. Lethin, M.D. Noakes, P.R. Nuth, R.E. Davison, and G.A. Fyler, "The Message-Driven Processor: A Multicomputer Processor Node with Efficient Mechanisms," *IEEE Micro*, vol. 12, no. 2, pp. 23-39, Apr. 1992.
- [8] G.C. Fox et al., "Matrix Algorithms on a Hypercube I: Matrix Multiplication," *Parallel Computing*, vol. 4, no. 1, 1987.
- [9] T.M. Guerrero, S.R. Cherry, M. Dahlbom, A.R. Ricci, and E.J. Hoffman, "Fast Implementations of 3D PET Reconstruction Using Vector and Parallel Programming Techniques," *IEEE Trans. Nuclear Science*, vol. 40, pp. 1,082-1,086, Aug. 1993.
- [10] R. Gupta and M. Epstein, "Achieving Low Cost Synchronization in a Multiprocessor System," *PARLE '89, Future Generations Computer Systems*, vol. 6, no. 3, pp. 255-269, Eindhoven, The Netherlands, Dec. 1990.
- [11] S.W. Keckler, "A Coupled Multi-ALU Processing Node for a Highly Parallel Computer," Technical Report. 1355, Massachusetts Inst. of Technology Artificial Intelligence Laboratory, Cambridge, Mass., 1992.
- [12] S.W. Keckler and W.J. Dally, "Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pp. 202-213, 1992.
- [13] S. Lacy, J. Cruz-Rivera, and D.S. Wills, "The Offset Cube: A Three-Dimensional Multicomputer Network Topology Using Through-Wafer Optics," Submitted to *IEEE Trans. Parallel and Distributed Systems*.
- [14] D. May, R. Sheperd, and P. Thompson, "The T9000 Transputer," *Proc. 1992 Int'l Conf. Computer Design: VLSI in Computers and Processors*, pp. 209-212, Oct. 1992.
- [15] A.W. McCarthy and M.I. Miller, "Maximum Likelihood SPECT in Clinical Computation Times Using Mesh-Connected Parallel Computers," *IEEE Trans. Medical Imaging*, vol. 10, pp. 426-436, Sept. 1991.
- [16] R.S. Nikhil, G.M. Papadopoulos, and Arvind. "T: A Multithreaded Massively Parallel Architecture," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pp. 156-167, 1992.
- [17] P.R. Nuth, "The Named-State Register File," PhD dissertation, Massachusetts Inst. of Technology, Aug. 1993.
- [18] P.R. Nuth and W.J. Dally, "A Mechanism for Efficient Context Switching," *Proc. 1991 IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors*, pp. 301-304, Oct. 1991.
- [19] L.A. Shepp and Y. Vardi, "Maximum Likelihood Reconstruction for Emission Tomography," *IEEE Trans. Medical Imaging*, vol. 1, pp. 113-121, Oct. 1982.
- [20] B. Smith, "The Architecture of HEP," *Parallel MIMD Computation: HEP Supercomputer and Its Application*, J.S. Kowalik, ed., chapter 1, pp. 41-55. MIT Press, 1985.
- [21] B.J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *SPIE*, vol. 298, pp. 241-248, 1981.
- [22] M.R. Thistle and B.J. Smith, "A Processor Architecture for Horizon," *Proc. Supercomputing '88*, pp. 35-41, Nov. 1988.
- [23] G.K. Wallace, "Overview of the JPEG (ISO/CCITT) Still Image Compression Standard," *SPIE Advent Technology Series*, vol. AT 1, pp. 358-371.
- [24] D.S. Wills and W.J. Dally, "Pi: A Parallel Architecture Interface," *Proc. Fourth Symp. Frontiers of Massively Parallel Computation*, pp. 345-352, Oct. 1992.
- [25] D.S. Wills, W.S. Lacy, C. Camperi-Ginestet, B. Buchanan, H.H. Cat, S. Wilkinson, M. Lee, N.M. Jokerst, and M.A. Brooke, "A Three-Dimensional High-Throughput Architecture Using Through-Wafer Optical Interconnect," *IEEE J. Lightwave Technology*, vol. 13, no. 6, pp. 1,085-1,092, June 1995.
- [26] A. Wolfe and J.P. Shen, "A Variable Instruction Stream Extension to the VLIW Architecture," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 2-14, Apr. 1991.



D. Scott Wills received his BS in physics from Georgia Tech in 1983, and his SM, EE, and ScD in electrical engineering and computer science from MIT in 1985, 1987, and 1990, respectively. He is an associate professor of electrical and computer engineering at Georgia Institute of Technology. His research interests include VLSI architectures, high-throughput portable processing systems, optoelectronics-enabled systems, and multicomputer interconnection networks. He is a member of the IEEE and the IEEE Computer Society.



Huy H. Cat received the BScPE (valedictorian) degree in computer engineering from the University of Central Florida in 1992, the MS in electrical engineering from the Georgia Institute of Technology in 1994, and is currently pursuing the PhD degree in electrical engineering. His research interests include architectures for fine-grained parallel processing, high-performance VLSI architectures, and Smart Pixel Systems.



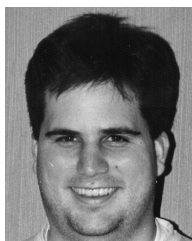
José Cruz-Rivera received the BSEE degree from the University of Puerto Rico-Mayaguez in 1991 and the MS and PhD degrees in electrical engineering from the Georgia Institute of Technology in 1992 and 1996, respectively. He is currently an assistant professor in the Electrical and Computer Engineering Department of the University of Puerto Rico-Mayaguez. His research interests include optoelectronic computing systems, parallel algorithm characterization, and image processing applications. Other professional interests include the development of teacher training programs for K-12 math and science education, as well as the use of cooperative learning methods for undergraduate engineering instruction. Dr. Cruz-Rivera is a member of the IEEE, Optical Society of America, and the Sigma Xi research society.



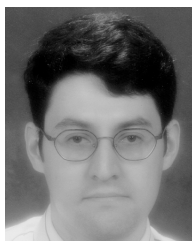
W. Stephen Lacy received the BS degree in electrical engineering from Christian Brothers University, Memphis, Tennessee, in 1991, and received the MSEE and PhD degrees in electrical engineering from the Georgia Institute of Technology in 1993 and 1997, respectively. He is currently an assistant professor in the Computer Systems Engineering Department at the University of Arkansas, Fayetteville. His research interests include interconnection networks for parallel and distributed computing systems, simulation and performance evaluation of parallel systems, and the application of high-performance VLSI/packaging techniques to computer architecture. Dr. Lacy is a member of the Tau Beta Pi, Alpha Chi, and Phi Beta Kappa academic honor societies and a member of the IEEE.



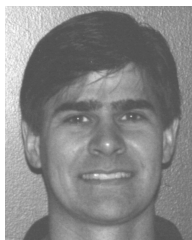
James M. Baker Jr. is a doctoral student in the VLSI Architectures Group at Georgia Tech. He received his BS and MS degrees in electrical engineering from Virginia Tech in 1988 and 1990, respectively. His research interests include operating systems, parallel architecture, and image processing. His doctoral research focuses on run-time operating systems for fine-grain parallel architectures.



John C. Eble received his BCmpE in computer engineering and MSEE in electrical engineering from Georgia Tech in 1993 and 1994, respectively. He is currently a doctoral student in the VLSI Architectures Group and the Gigascale Integration Group at Georgia Tech. His research interests include low-power, high-throughput programmable image processing architectures and generic system and architecture performance modeling. His doctoral research focuses on the development of a generic system simulator, GENESYS, and architectural techniques for energy efficient system operation. He is a student member of the IEEE and ACM.



Abelardo López-Lagunas is a doctoral student in the VLSI Architectures Group at Georgia Tech. He received his BS in electrical engineering from the Instituto Tecnológico y de Estudios Superiores de Monterrey in 1988, and his MS degree in electrical engineering from Georgia Tech in 1991. His research interests include parallel architectures, VLSI design, and parallel compilers. His doctoral research focuses on hardware support for fine-grain architectures. He is a member of the IEEE and the IEEE Computer Society.



Michael Hopper is a PhD candidate at the Georgia Institute of Technology. He received his BSEE and MSEE degrees from the University of Alabama in 1989 and 1991. His research interests include fine grain parallel systems, parallelizing compilers, multithreaded and superscaler microprocessors, and object oriented programming methodology. He is a member of the IEEE as well as the Eta Kappa Nu and Tau Beta Pi honor societies.