Presented at *NP2: Workshop on Network Processors*, held in conjunction with *The 9th International Symposium on High-Performance Computer Architecture*, Anaheim, California, February 8-9, 2003.

A Massively Multithreaded Packet Processor

Steve Melvin[†], Mario Nemirovsky[‡], Enric Musoll[‡], Jeff Huynh[‡],
Rodolfo Milito[‡], Hector Urdaneta, and Koroush Saraf[‡]

[†]*O'Melveny & Myers LLP*
275 Battery Street
San Francisco, CA 94111
smelvin@omm.com

[‡]*Kayamba, Inc.*
1299 Parkmoor Avenue
San Jose, CA 95126
{mario,enric,jeff,rodolfo,koroush}@kayamba.com

Abstract

This paper introduces a new packet processor designed for stateful networking applications: those applications where there is a requirement to support a large amount of state with little locality of access. Stateful applications require a high rate of external memory accesses, and this in turn implies a high degree of parallelism is needed. Our packet processor utilizes multiple multithreaded processing engines to support this parallelism in a design that supports 256 simultaneous threads in eight processing engines. Each thread has its own independent register file and executes instructions formatted to a general purpose ISA, while sharing execution resources and memory ports with other threads. The processor is optimized to sacrifice single threaded performance, so that a design is achieved that is realizable in terms of silicon area and clock frequency. The use of a general purpose ISA and other features achieves a design in which software porting issues are minimized.

1. Introduction

There are increasing numbers of emerging applications for packet processing devices in IP based networks that are *stateful applications*. In a stateful application, memory is updated with each packet that is processed, and a large amount of memory must be maintained. Thus, unlike applications in which packets can be processed relying on tables that change infrequently (such as layer 2 packet forwarding), stateful applications require the network device to maintain state related to which packets have been processed and in what order, and to update that state with each packet. This state is typically flow based, and in cases that there are a large number of simultaneous flows (for example 1 million or more), there can potentially be a large amount of memory that must be maintained (100's of megabytes or more). Stateful applications include quality of service enforcement, intrusion detection, sophisticated billing and monitoring among others. Many of these applications reside not in the core or at the edge, but at an enterprise gateway or data center connection point. Data rates between 1Gbps and 10Gbps are currently the biggest challenge. Rarely are these kinds of applications needed at higher data rates, and at lower data rates conventional solutions are practical.

One characteristic of many stateful applications, especially at data rates where many flows are aggregated on a single link, is the lack of memory locality. There is little temporal locality since the data being accessed by one packet is not needed again until a related packet is processed, which could be after a large number of intervening packets. Also there is also little spatial locality since often data structures are sparse with reads and writes only to small pieces of memory at a time. It is also the case that often a large memory footprint is needed, more than will fit on-chip even using advanced embedded DRAM techniques.

A consequence of these application requirements is that data caches improve performance little, if at all. In fact, data caches can often lower performance by forcing large blocks of memory (i.e. cache lines) to be read from and written to external memory, tying up this valuable resource longer than necessary. The only significant value of a data cache in these scenarios is to capture the working set locality of a single packet workload (i.e. the stack operations used to perform local computation). Our analysis has shown that the working sets for stateful applications of interest are in the range of 200 to 400 bytes. Thus, this amount of memory can easily be buffered on chip eliminating the need for a data cache and freeing up the external data memory to perform a larger number of randomly addressed memory operations.

The random address rate, or the rate at which non-sequential pieces of memory can be accessed, is a critical underlying performance metric that often governs overall performance. To achieve high random address rates in processors with low spatial locality, narrow buses using short bursts are desirable. Memory devices and access mechanisms optimized for large block transfers (such as is

needed for graphics applications) are often very inefficient in these applications, except where packet DMA is involved.

Thus, the workload for each packet must make a significant number of small external memory accesses. Our analysis of applications of interest has shown that overall workloads range from 100 to 10,000 instructions with 10-20% of all instructions making external memory accesses. Thus, there may be up to several thousand external memory accesses required to process a single packet. Given best case DRAM latencies and packet transmission times for short packets, it is clear that a high degree of packet overlap (on the order of 100s of packets simultaneously being processed) is necessary to cover this latency. Multithreading at some level is the only practical way to implement this high degree of parallelism, since to fit in a reasonable silicon area, there must be a high level of sharing of resources. Most threads are waiting for memory most of the time, and these waiting threads should consume as few resources as possible.

To satisfy these application requirements, we have designed a packet processor code-named *Porthos*, a block diagram for which is shown in Figure 1. We refer to Porthos as a "massively multithreaded" packet processor. Porthos has multiple memory ports, multiple multithreaded processing engines and other special features that emphasize efficiency over single threaded performance to achieve its design goal.

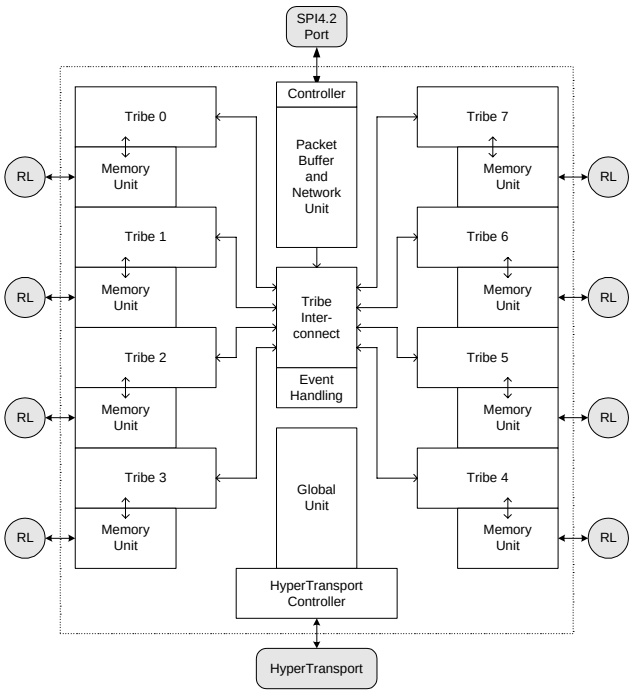

Figure 1 - Porthos Block Diagram

In this paper we will first explore the concept of random external accesses to memory and compare Porthos to other processors. Then we will discuss different memory / processor architecture. In section 4 we describe the microarchitecture of each multithreaded processing unit of Porthos in detail. Section 5 describes the special purpose network block and section 6 provides details on the interconnect block. Finally we summarize with conclusions in section 8.

2. Random External Memory Accesses

As mentioned in the previous section, the rate of random external accesses is often a more important constraint than the total amount of memory bandwidth. We introduce a term REAPS, for "random external accesses per second." The number of REAPS necessary for a given application can be compared against the maximum number a given processor is capable of.

We have extensively studied a number of applications in terms of their requirements for MIPS and REAPS. A summary of our analysis is shown in Table 1. The applications shown are Cisco Express Forwarding (CEF), packet classification, packet re-assembly, a monitoring application (Argus [1]) and an intrusion detection application (Bro [2]). The second column in Table 1 shows the average number of instructions per packet we measured for these applications.

The number of MIPS required is shown for OC48 (2.5Gbps full-duplex) with 500 byte packets. The last column illustrates the number of REAPS needed in billions per second. This data is based on our observed numbers of load and store instructions per workload, which ranges from 25% to 46%, and the simulated fraction of those that would miss in an internal cache, which ranges from 35% to 40%.

Table 1: MIPS/REAPS Requirements and Capabilities

Application Requirements at OC48, 500b packets			
App.	*Inst/packet*	*GIPS*	*GREAPS*
CEF	70	0.09	0.01
Classification	400	0.50	0.05
Re-assembly	400	0.50	0.08
Monitoring	2500	3.13	0.38
IDS	3500	4.38	0.51
Processor Capabilities			
Intel, 2Ghz		3.0	0.08
SB1250		2.4	0.16
Porthos		9.6	2.40

The last section of Table 1 illustrates MIPS and REAPS capabilities for three processors: a Pentium running at 2Ghz, a Sibyte 1250 and Porthos using realistic assumptions on IPC. This table illustrates that while the Pentium and Sibyte processors are much closer to handling these applications in terms of MIPS than in terms of REAPS. This is mainly due to the different design goals between processors running large single threaded applications (that have high temporal and spatial memory locality) and those of packet processors as discussed above. As Table 1 illustrates, the GREAPS/GIPS ratio in the applications we have studied is on the order of 1/10. While the Sibyte has GIPS/GREAPS ratio of 1/15 and the Pentium has a ratio of barely 1/40, Porthos is designed to accommodate a peak GREAPS rate of 1/4 of its sustained GIPS rate. Processors such as the Pentium are optimized

for transferring large blocks of memory externally rather than a higher number of small transfers.

3. Processor/Memory Architectures

Given the REAPS requirements of applications of interest at the rates of interest, it quickly becomes clear that multiple memory ports are required. Even the fastest external DRAM with the shortest burst length cannot deliver enough independent accesses. Our analysis has shown that eight RL-DRAM ports running at 300Mhz, each 32-bits in width are sufficient to meet the overall design requirements. A 32-bit bus width with a burst length of 2 yields 64-bits every DRAM clock cycle. With enough buffering to keep all banks busy, an independent access can be started every DRAM cycle to keep the number of random external accesses high.

Given these multiple memory ports, a significant issue is whether each memory port should be coupled to a particular processing engine, or whether there should be a decoupled arrangement where memory ports and processing engines are connected via an internal network. We have performed extensive modeling of these architecture choices and shown that higher memory utilizations are possible with a coupled architecture. This result is intuitively clear when one considers that a coupled architecture suffers less contention. A coupled architecture exposes a memory arrangement to the software/application level, which allows best case utilization to increase.

The disadvantage of a coupled architecture is that the programmer and/or compiler now must partition the application such that workloads and/or flows are split across the multiple processing elements. Exposing this detail to software had advantages in performance, but can only be taken advantage of if the applications can be fit into this model. In Porthos, we have chosen a hybrid approach, which could be called "loosely coupled." While there is a coupling of a processor to a memory port, there is also an internal switch in which each memory unit is connected to every other memory unit. This permits software running on any tribe to access any memory in any tribe, with a preferential access to the local memory port. This is possible since each ports represents a distinct region of the physical address space.

The partitioning of the memory space into distinct regions for each of the tribes is a key design detail of the Porthos chip. Partitioning is based on the assumption that a large number (at least 90%) of memory accesses for threads within a given tribe are destined for the local memory associated with that tribe. Partitioning reduces the interconnect cost, since the global interconnect is needed for only a small fraction of the total memory accesses. Partitioning also reduces contention, thereby reducing queuing delay and/or allowing a higher efficiency of external memory accesses. As discussed above, the key to this reduction in contention is that a preferential memory region is exposed to software so partitioning can be addressed statically.

Memory partitioning does represent software challenges. It will be necessary to have compiler and linker support to insulate this detail as much as possible from the programmer. It is also necessary to have hardware support to move threads between processing engines. Thus, it is more efficient to move a thread to a run on a particular processing engine that is closer to the data it is operating on than to move the data to a particular tribe. More details on the manner in which threads are moved between processing engines is discussed below. It is important to note that because of this loosely coupled architecture, memory partitioning has been moved from a correctness issue, to one of performance.

4. The Tribe Microarchitecture

Figure 1 illustrates the major blocks of Porthos. A full-duplex network interface of 10 Gbps is provided that may be channelized into 4 OC48 ports (2.5 Gbps each), or 10 1 Gbps Ethernet ports. The Packet Buffer is a FIFO that stores the packet data until it is determined if the packet should be dropped, forwarded (with modifications if necessary), or transferred off chip for later processing. Packets may be transmitted from data stored externally and may be created by software and transmitted.

The processing on packets that are resident in the chip occurs in multiple processing engines, where each engine is associated with an independent port to memory as discussed above. Each processing engine, called a *tribe*, can execute up to 32 threads simultaneously. Each tribe is in fact a multithreaded processor with two memory ports, one to external memory and one to the packet buffer. A block diagram of the microarchitecture of each tribe is shown in Figure 2, which illustrates the modules that implement the tribe and the major data path connections between those modules. A software thread would typically execute on a single packet in one tribe at a time, and may jump from one tribe to another. A HyperTransport interface is used to communicate with host processors, co-processors or other Porthos chips.

4.1. ISA Specification

Each tribe executes instructions to accomplish the necessary workload for each packet. The ISA implemented by Porthos is similar to the 64-bit MIPS-IV ISA with a few omissions and a few additions. The main differences between the Porthos ISA and MIPS-IV are summarized as follows. The Porthos ISA contains 64-bit registers, and utilizes 32-bit addresses with no TLB. There is no 32-bit mode, thus all instructions that operate on registers operate on all 64-bits. This functionality is the same as a MIPS R4000 in 64-bit mode. All memory is treated as big-endian and there is no mode bit that controls endianness. Since there is no TLB, there is no address translation and there are no protection modes implemented. This means that all code has access to all regions of memory. The physical address space of Porthos is 32-bits, so the upper 32 bits of a generated 64-bit virtual address are ignored and no translation takes place. There are no TLB related CP0 registers and no TLB instructions.

There is no floating-point unit and therefore no floating-point instructions. However the floating-point

registers are implemented as well as co-processor 2 registers. This allows additional registers (up to 95) to be used by integer codes in a way that fits within the standard MIPS ISA. Instructions that load store and move data between the regular registers and the floating-point registers (CP1 registers) are implemented. The SC, SCD, LL and LLD instructions are implemented. Additionally, there is an ADDM instruction that atomically adds a value to a memory location and returns the result. In addition there is a GATE instruction that stalls a thread to preserve packet dependencies (this is described in more detail below).

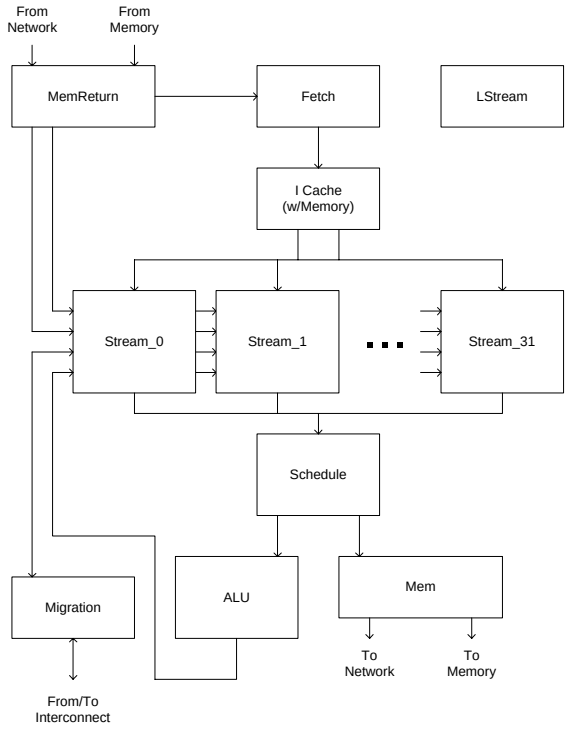

Figure 2 - Tribe Block Diagram

External events and timer interrupts are treated such that new threads are launched. Thus, a thread has no way to enable or disable these events itself since they are configured globally. This is explained in more detail below. There are four new CP0 registers: SequenceNumber, Tribe/Stream Number, FlowID and GateVector. There are also three new thread control instructions: DONE, FORK and NEXT.

Support for string search, including multiple parallel byte comparison, has been provided for in new instructions. In addition there are bit field extract and insert instructions. Finally, an optimized ones-complement add is provided for TCP checksum acceleration.

Events can be categorized into three groups: triggers from external events, timer interrupts, and thread-to-thread communication. In the first two groups, the events are not specific to any specific physical thread. In the third group, software can signal between two specific physical threads.

4.2. Thread Migration

The process by this a thread executing on a stream in one tribe is transferred to a stream in another tribe is called *migration*. We use the term "thread" to refer to a software construct, and "stream" to refer to the particular physical resources that are used by a thread when active. When migration occurs, a variable amount of context follows the thread. The CPU registers that are not transferred are lost and initialized to zero in the new tribe. Migration may occur out of order, but it is guaranteed to preserve thread priority as defined by the SequenceNumber register. Note, however, that a lower priority thread may migrate ahead of a high priority thread if it has a different destination tribe. Threads may migrate to the tribe that they are already in. A thread may change its priority by writing to the SequenceNumber register. The thread migration instruction: NEXT specifies a register that contains the destination address and an immediate that contains the amount of thread context to preserve. More details on thread migration, including deadlock avoidance are discussed below in the section on the Interconnect block.

Thread migration is a key feature of Porthos and it is important to understand how thread migration interacts with the loosely coupled processor / memory architecture. Since the memory map is not overlapped, every thread running in each tribe has access to all memory. Thus, from the standpoint of software correctness, migration is not strictly required. However, the ability to move the context and the processing to an engine that is closer to the state that the processing is operating on allows a flexible topology to be implemented. A given packet may do all of its processing in a specific tribe, or may follow a sequence of tribes. Furthermore, this decision can be made on a packet by packet basis.

The thread migration module is responsible for migrating threads into the Tribe block and out of the Tribe block. A thread can only be participating in migration if it is not actively executing instructions. During migration, a single register read or write per cycle is processed by the Stream module and sent to the Interconnect block. A migration may contain any number of registers. When an inactive stream is migrated in, all registers that are not explicitly initialized are set invalid. An invalid register will always return 0 if read. A single valid bit per register allows the register file to behave as if all registers are initialized to zero when a thread is initialized.

4.3. Flow Gating

The efficient handling of packet dependencies is an important design element to any packet processor. Many workloads we have studied have the property that most packets can be processed completely independently, but a non-trivial fraction, and sometimes even back-to-back packets require serialization. A hardware mechanism to enforce packet dependencies with no changes to software is discussed in [3]. Porthos uses a different technique, known as "flow gating," that requires software changes, but fairly modest ones.

Flow gating is a mechanism in which packet seniority is enforced by hardware through the use of a GATE instruction inserted into the packet processing workload. When a GATE instruction is encountered, the instruction execution for that packet is stalled until all older packets of the same flow have made progress past the same point. Software manually advances a packet through a gate by updating the GateVector register. Multiple gates may be specified for a given packet workload and serialization occurs for each gate individually.

Packets are given a sequence number by the packet buffer controller when they are received and this sequence number is maintained during the processing of the packet. A configurable hardware pre-classifier is used to combine specified bytes from the packet and generate a FlowID number from the packet itself. The FlowID is initialized by hardware based on the hardware hash function, but may be modified by software. The configurable hash function is also be used to select which tribe a packet is sent to. Afterward, tribe to tribe migration is under software control.

A new instruction is utilized that operates in conjunction with three internal registers. In addition to the FlowID register and the PacketSequence register discussed above, each thread contains a GateVector register. Software may set and clear this register arbitrarily, but it is initialized to 0 when a new thread is created for a new packet. A new instruction, named GATE, is implemented. The GATE instruction causes execution to stall until there is no thread with the same FlowID, a PacketSequence number that is lower, and with a GateVector in which any of the same bits are zero. This logic serializes all packets within the same flow at that point such that seniority is enforced.

Software is responsible for setting a bit in the GateVector when it leaves the critical section. This will allow other packets to enter the critical section. The GateVector register represents progress through the workload of a packet. Software is responsible for setting bits in this register manually if a certain packet skips a certain gate, to prevent younger packets from unnecessarily stalling. If the GateVector is set to all 1's, this will disable flow gating for that packet since no younger packets will wait for that packet. Note that forward progress is guaranteed since the oldest packet in the processing system will never be stalled and when it completes, another packet will be the oldest packet.

A seniority scheduling policy is implemented such that older packets are always given priority for execution resources within a processing element. One characteristic of this strictly implemented seniority scheduling policy is that if two packets are executing the exact same sequence of instructions, a younger packet will never be able to overtake an older packet. In certain cases, the characteristic of no-overtaking may simplify handling of packet dependencies in software. This is because a no-overtaking processing element enforces a pipelined implementation of packet workloads, so the oldest packet is always guaranteed to be ahead of all younger packets. However, a seniority based instruction scheduler and seniority based cache replacement can only behave with no-overtaking if packets are executing the exact same sequence of instructions. If conditional branches cause packets to take different paths, a flow gate would be necessary. Flow gating in conjunction with no-overtaking processing elements allow a clean programming model to be presented that is efficient to implement in hardware.

4.4. Tribe Pipeline

The Tribe block contains an instruction cache and register files for 32 threads. The tribe block interfaces with the Network block (for handing packet buffer reads and writes), the Interconnect block (for handling thread migration) and the Memory block (for handling local memory reads and writes).

The Tribe block consists of three decoupled pipelines. The fetch logic and instruction cache form a fetch unit that will fetch from two threads per cycle according to thread priority among the threads that have a fetch available to be performed. The Stream block within the Tribe contains its own state machine that sequences reads and writes to the register file and executes certain instructions. Finally the scheduler, global ALU and memory modules form an execute unit that schedules operations based on global priority among the set of threads that have an instruction available for scheduling. At most one instruction per cycle is scheduled from any given thread.

Globally, up to three instructions can be scheduled in a single cycle (one ALU, one packet buffer memory and one external memory). Many instructions can be fully executed within the Stream block, not requiring global scheduling. Thus, the peak instantaneous instruction execution rate is 32. Practically, the maximum rate of instruction execution is limited by the fetch unit, which can fetch up to eight instructions each cycle. A sustained execution rate of four instructions per cycle has been simulated under typical workload conditions.

4.4.1. Instruction Fetch. The instruction fetch mechanism fetches four instructions from two threads for a total fetch bandwidth of eight instructions. The fetch unit includes decoders so that eight decoded instructions are delivered to two different stream modules in each cycle. There is a 16K byte instruction cache shared by all threads that is organized as 1024 lines of 16 bytes each, separated into four ways of 256 lines. The fetch mechanism is pipelined, with the tags accessed in the same cycle as the data. The fetch pipeline is illustrated in Figure 3.

Fetch priority is based on the seniority scheduling mechanism described below. This gives priority to the oldest two packets that have a fetch request pending. The instruction cache is a two-read one-write design so writes can take place without interfering with instruction fetches. Porthos does not implement any speculative fetch, so a thread will not be a candidate for fetch until all pending branches have been resolved. In the case of straight-line code, a stream block can detect there are no branches in cycle F4 (corresponding to cycle S1 discussed below) and in the following cycle become a candidate for fetching another four instructions. This means that a single thread could theoretically request four instructions every four

cycles, but in most cases a stream block could not execute instructions at this rate.

4.4.2. Stream Modules. The Stream modules (one per stream for a total of 32 within the Tribe block) are responsible for sequencing reads and writes to the register files, executing branch instructions, and handling certain other arithmetic and logic operations. A Stream module receives four decoded instructions at a time from the Fetch mechanism and saves them for later processing. One instruction is processed at a time, with some instructions taking multiple cycles to process. Since there is only a single port to the register file, all reads and writes must be sequenced by the Stream block. The basic pipeline of the Stream module is shown in Figure 4.

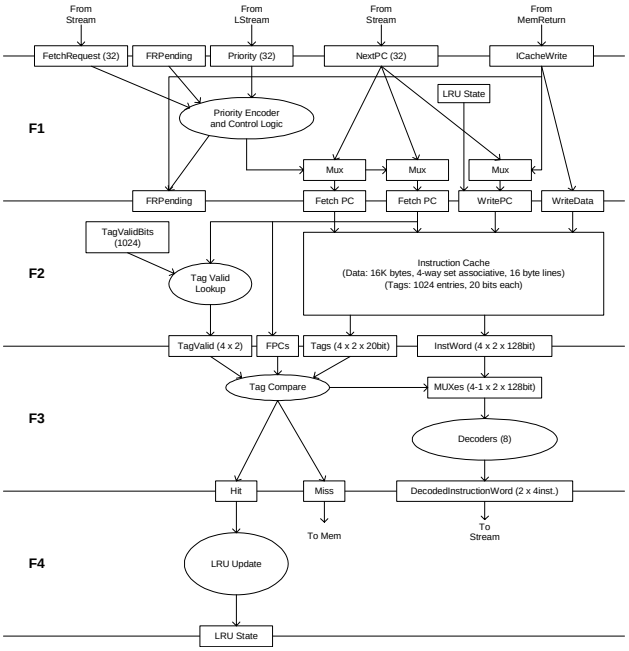

Figure 3 - Fetch Pipeline Timing

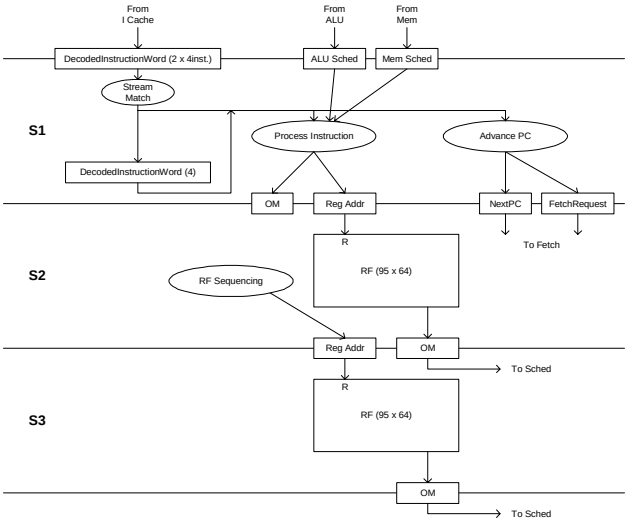

Figure 4 - Stream Timing - Register Read

Note that in cases where only a single operand needs to be read from the register file, the instruction would be available for global scheduling with only a single RF read stage. Each register contains a ready bit that is used to determine if the most recent version of a register is in the register file, or it will be written by an outstanding memory load or ALU instruction. In spite of the fact that MIPS is a three operand architecture, requiring up to three cycles to process each instruction, in practice the number of register cycles is only slightly more than one. This is due to the fact that many instructions use only one or two operands, and many operands are in flight (requiring a write, but not a read). Since single threaded performance can be sacrificed, it is sufficient to supply only a single register port for each thread and suffer the additional latency when multiple cycles are required to sequence the register file. Single ported register files are far smaller and lower in power consumption than multi-ported register files. Extending this concept, it would be viable to combine two stream blocks and utilize only a single register file between them. This would have effectively 1/2 of a register port per thread.

Writes returning from the Network block and the Memory block must also be sequenced to the register file. The register write sequencing pipeline of the Stream block is shown in Figure 5. When a memory instruction, or an instruction for the global ALU is encountered, the operation matrix, or OM, register is updated to reflect a request to the global scheduling and execute mechanism.

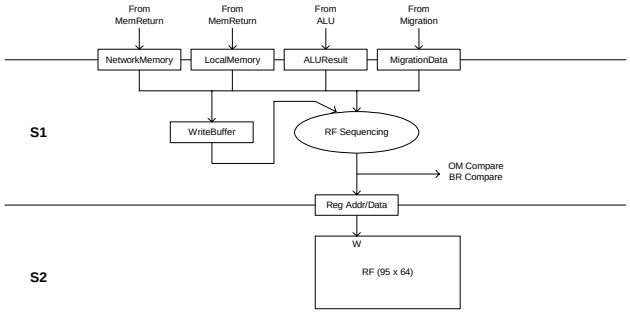

Figure 5 - Stream Timing - Register Write

Branch instructions are executed within the Stream module as illustrated in Figure 6. Branch operands can come from the register file, or can come from outstanding memory or ALU instructions.

The branch operand registers are updated in the same cycle in which the write to the register file is scheduled. This allows the execution of the branch to take place in the following cycle. Since branches are delayed in the MIPS ISA, the instruction after the branch instruction must be processed before the target of the branch can be fetched. The earliest that a branch delay slot instruction can be processed is the same cycle that a branch is executed. Thus, a fetch request can be made at the end of this cycle at the earliest. The processing of the delay slot instruction would occur later than this if it was not yet available from the Fetch pipeline.

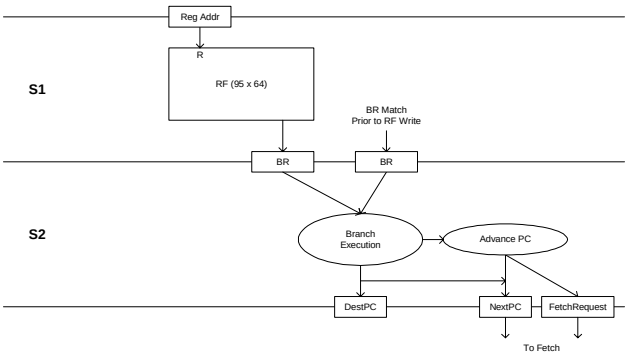

Figure 6 - Stream Timing - Branch Resolution

4.4.3. Scheduling and Execute. The scheduling and execute modules schedule up to three instructions per cycle from three separate streams and handle register writes back to the stream modules. The execute pipeline is shown in Figure 7. Streams are selected based on what instruction is available for execution (only one instruction per stream is considered a candidate), and on the overall stream priority. Once selected, a stream will not be able to selected in the following cycle since there is a minimum two cycle feedback to the Stream block for preparing another instruction for execution. This means that the maximum instruction rate per thread is 0.5 IPC. In practice a higher IPC could be achieved since some instructions are executed within the Stream block. However, given expected memory latencies and realistic workloads, an IPC per thread of 0.1 to 0.2 are expected.

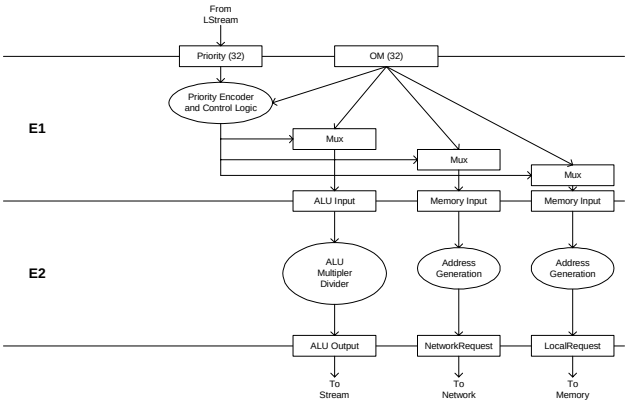

Figure 7 - Execute Timing

4.4.4. Thread Priority. Thread priority (used for fetch scheduling and execute scheduling) is maintained by the "LStream" module. This module also maintains a gateability vector used to implement FlowGating. The LStream module is responsible for determining for each thread whether or not it should stall upon the execution of a "GATE" instruction, or can continue. This single bit per thread is exported to each Stream block. Any time a change is made to any CP0 register that can potentially affect gateability, the LStream module will export all 0's on its gateability vector (indicating no thread can proceed past a GATE), until a new gateability vector is computed.

Changes that affect gateability are rare. They are as follows:

1. A new thread is created, it will be migrated in with its own sequence number, gate vector and flow ID register;

2. An existing thread is deactivated, either due to a DONE instruction or a NEXT instruction (migration out to another tribe);

3. A thread explicitly updates one of its gateability CP0 registers (sequence number, gate vector, flow ID)using the MTC0 instruction.

Thread priority is based on a seniority concept, where older packets have priority over younger packets. Packet age is determined by the sequence number that represents order of arrival and is generated by the Network block. This means that when a packet first migrates to a tribe, it may get very little execution resources until older packets finish or are migrated out. In most cases this is the desirable behavior since the packet buffer represents a FIFO where packets must be either processed or moved off-chip to avoid head of line blocking. In some cases, other thread priority schemes are needed, so thread priority can be overridden by software when necessary. One example of special case thread would be a background thread not associated with a packet. It may need a guaranteed minimum level of service, with the minimal impact to other threads. Another example is a thread operating on a packet that is no longer in the packet buffer.

5. Network Block

The network block contains a packet buffer as well as a variety of other control logic and data structures. A block diagram of the Network block is shown in Figure 8. The packet buffer is an on-chip 256K-byte memory that holds packets while they are being worked on. The packet buffer is a flexible FIFO that keeps all packet data in the order it was received. Thus, unless a packet is discarded, or consumed by the chip (by transfer into a local memory), the packet will be transmitted in the order that it was received. The packet buffer architecture allows for a efficient combination of "pass-through" and "reassembly" scenarios.

In a pass-through scenario, packets are not substantially changed; they are only marked, or modified only slightly before being transmitted. The payload of the packets remains substantially the same. Pass-through scenarios occur in TCP-splicing, monitoring and traffic management applications. In a re-assembly scenario, packets must be consumed by the chip and buffered into memory where they are re-assembled. After re-assembly, processing occurs on the reliable data stream and then re-transmission may occur. Re-assembly scenarios occur in firewalls and load balancing. Many applications call for a combination of pass-through and reassembly scenarios.

The Packet Buffer module interacts with software in the following ways:

- Providing the initial values of some GPRs and CP0 registers at the time a thread is scheduled to start executing its workload.
- Satisfying the requests to the packet buffer memory

- Satisfying the requests to the configuration registers, for instance
- Hash function configuration
- Packet table read requests
- Packet status changes (packet to be dropped, packet to be transmitted out)
- Allocating space in the packet buffer for software to construct packets.

Frames of packets arrive to the Packet Buffer through a configurable number of ingress ports and leave the Packet Buffer through the same number of egress ports. The maximum ingress/egress interleave degree depends on the number and type of ports, but it does not exceed 4.

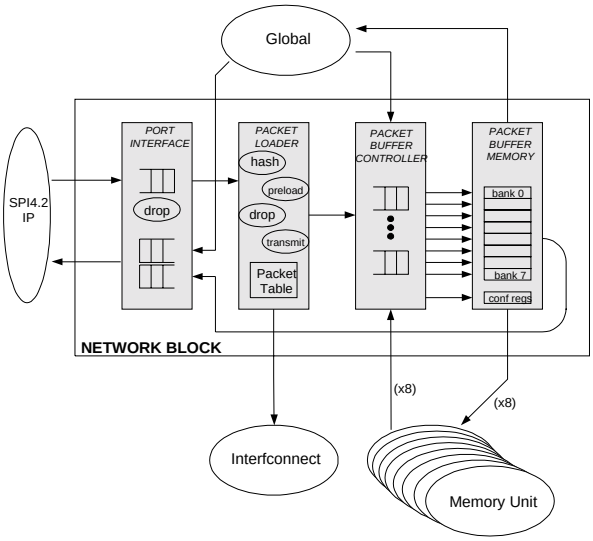

Figure 8 - Network Block Overview

Software is responsible to complete the processing of the oldest packets that the Packet Buffer module keeps track in a timely manner, namely before:

1. The subsequent newest packets fill up the packet buffer so that no more packets can be fit into it. At 300MHz core frequency, the peak rate of ingress data of 10Gbps and a packet buffer size of 256KB, this will occur in approximately 200 microseconds;
2. There are 512 total packets in the system, from the oldest to the newest, no matter whether packets in between the oldest and the newest have been dropped (or DMA out to external memory) by software.

Otherwise the Packet Buffer module will drop the incoming frames.

5.1. Packet Buffer Address Space

Two regions of the Porthos chip 32-bit physical address space are controlled directly by the Packet Buffer module:

- the packet buffer memory: 256KB of memory where the packets are stored as they arrive. Software is responsible to take them out of this memory if needed (for example, in applications that need re-assembly of the frames)
- the configuration register space: 16KB (not all used) that contains the following sections:
- the configuration registers themselves: are used to configure some functionality of the Packet Buffer module.
- the packet table: contains status information for each of the packets being kept track of.
- the get room space: used for software to request consecutive chunks of space within the packet buffer.

Software can perform any byte read/write, half-word (2-byte) read/write, word (4-byte) read/write or double word (8-byte) read/write to the packet buffer.

Even though the size of the packet buffer memory is 256KB, it actually occupies 512KB in the logical address space of the streams. This has been done in order to help minimizing the memory fragmentation that occurs incoming packets are stored into the packet buffer. This mapping is performed by hardware and in consequence packets are always stored consecutively into the 512KB of space from the point of view of software.

Software should only use the packet buffer to read the packets that have been stored by the Packet Buffer module, and to modify these packets. Allocating portions of the 512KB logical address space (for scratch pad purposes, for example) other than by means of the get room command (explained later on) can lead to undefined results. The requests from the 8 tribes are treated fairly; all the tribes have the same priority in accessing the packet buffer.

5.2. Completing And Dropping Packets

Software eventually has to decide what to do with a packet that sits in the packet buffer, and it has two options:

- Complete the packet: the packet will be transmitted out whenever the packet becomes the oldest packet in the packet buffer.
- Drop the packet: the packet will be eventually removed from the packet buffer.

In both cases, the memory that the packet occupies in the packet buffer and the entry in the packet table will be made available to other incoming packets as soon as the packet is fully transmitted out or dropped. Also, in both cases, the Packet Buffer module does not guarantee that the packet will be either transmitted or dropped right away.

Software completes and drops packets by writing into the ‘done’ and ‘drop’ configuration registers, respectively. The information provided in both cases is the sequence number of the packet. For the completing of packets, the following information is also provided:

- Header growth offset: an 10-bit value that specifies how many bytes the start of the packet has either grown (positive value) or shrunk (negative value) with respect the original packet. The value is encoded in 2’s complement. If software does not move the start of the packet, this value should be 0.
- Encoded egress channel.
- Encoded egress port.

The head of the packet is allowed to grow up to 511 bytes and shrink up to the minimum of the original packet size and 512).

5.3. Get Room Command

Software can transmit a packet that it has generated through the GetRoom mechanism. This mechanism works as follows:

- Software requests some space to be set aside in the packet buffer. This is done through a regular read to the GetRoom space of the configuration space. The address of the load is computed by adding the requested size in bytes to the base of the GetRoom configuration space.
- The Packet Buffer module will reply to the load
  - Unsuccessfully: it will return a '1' in the MSB bit and '0' in the rest of the bits
  - Successfully: it will return in the 32 LSB bits the physical address of the start of the space that has been allocated, and in bits [47..32] the corresponding sequence number associated to that space.
- Software, upon the successful GetRoom command, will construct the packet into the requested space.
- When the packet is fully constructed, software will complete it though the packet complete mechanism explained before.

5.4. Initial migration

When the packets have been fully received by the Packet Buffer module and they have been fully stored into the packet buffer memory, the first migration of those packets into one of the tribes will be initiated. The migration process consists of a request of a migration to a tribe, waiting for the tribe to accept the migration, and providing some control information of the packet to the tribe. This information will be stored by the tribe in some software visible registers of one of the available streams.

The Packet Buffer module assigns to each packet a flow identification number and a tribe number using configurable hashing hardware. The packets will be migrated to the corresponding initial tribes in exactly the same order of arrival. This order of arrival is across all ingress ports and, if applicable, all channels. If a packet has to be migrated to a tribe that has all its streams active, the tribe will not accept the migration. The Packet Buffer module will keep requesting the migration until the tribe accepts it.

After the migration has taken place, the following registers are initialized in one of the streams of the tribe: the PC will be initialized with a configurable vector; CP0 registers will be initialized with an initial FlowID number obtained by the hashing hardware and a SequenceNumber that contains the order of arrival of the packet; and two GPRs will be initialized with the ingress port/channel of the packet and the 32-bit logical address where the packet resides.

6. Interconnect

The Interconnect block consists of 3 modules: Event, Arbiter and Crossbar. A block diagram of the Interconnect block is shown in Figure 9. The Event module collects event information and activates a new stream to process the event. The Arbiter module performs arbitration between sources and destinations. The Crossbar module directs data from sources to destinations

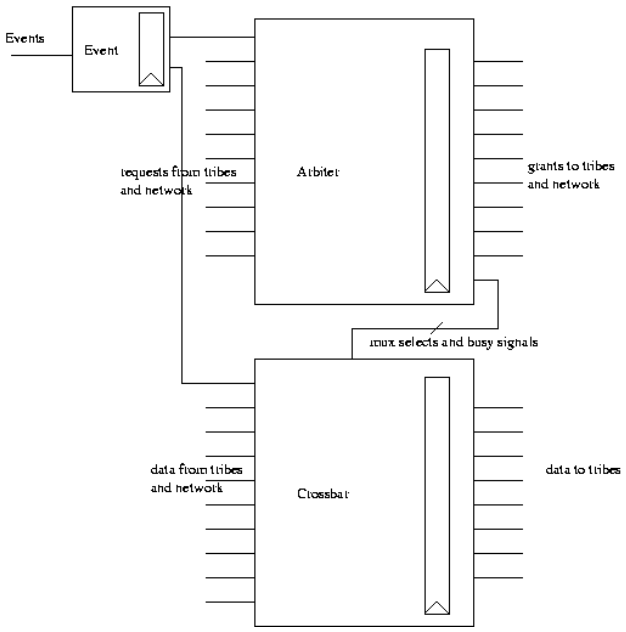

Figure 9 - Interconnect Overview

6.1. Arbiter

There are 11 sources of requests, the 8 tribes, the network block, the event handling module and transient buffers. Each source tribe can make up to 7 requests, one for each destination tribe. The network block, event handling module, and transient buffers each can make one request to one of the 8 tribes. If there's a request from transient buffers to a tribe, that request has the highest priority and no arbitration is necessary for that tribe. If transient buffers are not making request, then arbitration is necessary.

The arbiter needs to match the requester to the destination in such a way as to maximize utilization of the interconnect, while also preventing starvation. We use round-robin prioritizing scheme. There are two stages. The first stage selects one non-busy source for a given non-busy destination. The second stage resolves cases where the same source was selected for multiple destinations. Each source and each destination has a round-robin pointer. This points to the source or destination with the highest priority. The round-robin prioritization logic begin searching for the first available source or destination beginning at the pointer and moving in one direction.

The arbitration scheme described above is "greedy," meaning it attempts to pick the requests that can proceed, skipping over sources and destinations that are busy. In other words, when a connection is setup between a source and a destination, the source and destination are locked out from later arbitration. With this scheme, there are cases when the arbiter starves certain context. It could happen that two repeated requests, with overlapping transaction

times, can prevent other requests from being processed.

To prevent this, the arbitration operates in two modes. The first mode is "greedy" mode as described above. For each request that cannot proceed, there is a counter that keeps track of the number of times that request has been skipped. When the counter reaches a threshold, the arbitration will not skip over this request, but rather wait at the request until the source and destination become available. If multiple requests reach this priority for the same source or destination, then one-by-one will be allowed to proceed in a strict round-robin fashion. The threshold can be set via the Greedy Threshold configuration register.

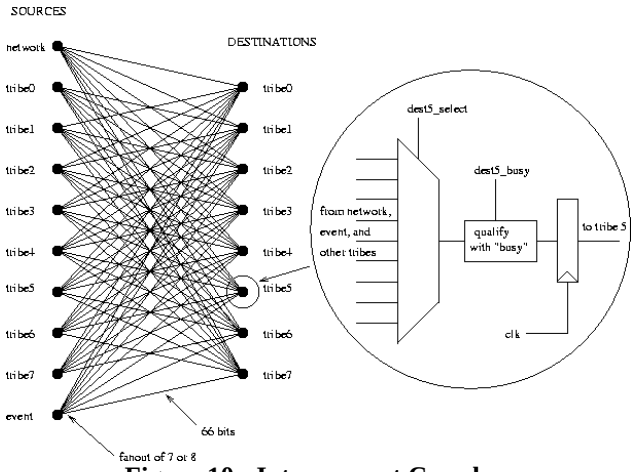

Figure 10 - Interconnect Crossbar

6.2. Deadlock Resolution

Deadlock occurs when the tribes in migration loops are all full, i.e. tribe 1 request migration to tribe 2 and vice versa and both tribes are full. The loops can have up to 8 tribes. To break a deadlock, Porthos uses two transient buffers in the interconnect, with each buffer capable of storing an entire migration (66 bits times maximum migration cycles). The migration request with both source and destination full (with destination wanting to migrate out) can be sent to a transient buffer. The transient stream becomes highest priority and initiate a migration to the destination, while at the same time the destination redirect a migration to the second transient buffer. Both of these transfers need to be atomic, meaning no other transfer is allowed to the destination tribe and the tribe is not allowed to spawn new stream within itself. The migrations into and out of a transient buffers use the same protocol as tribe-tribe migrations.

This method begins by detecting only possibility of deadlock and not the actual deadlock condition. It allows forward progress while looking for the actual deadlock, although there maybe cases where no deadlock is found. It also substantially reduces the hardware complexity with minimal impact on performance.

A migration that uses the transient buffers will incur an average of 2 migration delays (a migration delay is the number of cycles needed to complete a migrate). The delays don't impact performance significantly since the migration is already waiting for the destination to free up.

Using transient buffers will suffice in all deadlock situations involving migration, including simple deadlock loops involving 2 to 8 tribes, multiple deadlock loops with 1 or more shared tribes, multiple deadlock loops with no shared tribe and multiple deadlock loops that are connected. In the case of multiple loops, the transient buffers will break one loop at a time. The loop is broken when the transient buffers are emptied.

Hardware deadlock resolution cannot solve the deadlock situation that involve software dependency. For example, a tribe in one deadlock loop waits for some result from a tribe in the another deadlock loop that has no tribe in common with the first loop. Transient buffers will service the first deadlock loop and can never break that loop.

6.3. Event module

Upon hardware reset, event module spawns a new stream in tribe 0 to process reset event. This reset event comes from global block. The Event module spawns a new stream via the interconnect logic based on external and timer interrupts.

Each interrupt is maskable by writing to Interrupt Mask configuration registers in configuration space. There are two methods an interrupt can be directed. In the first method, the interrupt is directed to any tribe that is not empty. This is accomplished by the event module making requests to all 8 destination tribes. When there is a grant to one tribe, the event module stops making requests to the other tribes and start a migration for the interrupt handling stream. In the second method, the interrupt is being directed to a particular tribe. The tribe number for the second method as well as which method are specified using Interrupt Method configuration registers for each interrupt.

The event module has a 32-bit timer which increments every cycle. When this timer matches the Count configuration register, it activates a new stream via the migration interconnect.

6.4. Crossbar module

This is a 10 inputs, 8 outputs crossbar. Each input is comprised of a "valid" bit, 64-bit data, and a "last" bit. Each output is comprised of the same. For each output, there's a corresponding 4-bit select input which selects one of 10 inputs for that particular output. Also, for each output, there's a 1-bit input which indicates whether the output port is being selected or busy. This "busy" bit is ANDed with the selected "valid" and "last" so that those signals are valid only when the port is busy. The output is registered before being sent to the destination tribe.

7. Project Status

Porthos was designed and developed by the authors while at the company FlowStorm, Inc. FlowStorm was founded in July 2001 as a venture funded startup. Development continued for more than a year and included

the RTL coding and verification of all major functionality, partial synthesis and timing evaluation. Performance modeling, software development and system engineering were also extensive. Some of the authors are continuing development of a massively multithreaded processor in a new company, Kayamba, Inc.

8. Conclusions

In this paper we have presented *Porthos*, a new packet processor designed for stateful applications. The design of Porthos is derived from fundamental observations about stateful networking applications: that a significant number of off-chip accesses with little locality need to be supported for each packet. In order to efficiently implement the requisite large number of parallel packets, Porthos implements a massively multithreaded processor with a strong bias toward efficiency over single threaded performance.

Multithreaded processors have a long history as both commercial projects including the HEP-1 [4], Horizon [5], and Tera [6] as well as academic research [7] [8]. More recent commercial designs have included the Compaq Alpha EV-8 [9], the Clearwater Networks CNP810SP [10], Intel [11], and announced development by Sun Microsystems among others. Porthos differs from this previous work in several important ways, and it is for this reason that we refer to it as "massive multithreading", or MMT. We define a MMT processor as one with most (but perhaps not all) of the following characteristics:

Hardware support for large numbers of threads (on the order of 100s, rather than 2-8);
Single threaded performance has been sacrificed:
no branch prediction
no speculative fetch or execution
single port (or less) per register file per thread
no ALU bypass (back to back dependent operations are not possible)
a maximum of one operation per thread per cycle.

This results in a design that can practically implement 256 threads in a reasonable silicon area (projected at 120 square millimeters), at a reasonable power (projected at 12 watts). It is also the case that since the applications tend to be limited by external memory device access time, internal clock frequency is not a strong performance factor. At a projected clock frequency of 300Mhz using 0.15 micron technology, the design well exceeds other processor designs operating at 1Ghz and above.

Besides providing an MMT network processor, another key innovation in Porthos is the implementation of a loosely coupled processor / memory architecture. This allows performance to be optimized by software and provides topological flexibility (through the use thread migration) and a uniform software model (so that code with still work if memory partitioning issues are ignored). Finally, the use of flow gating allows efficient memory synchronization with minimal software porting issues.

9. References

[1] Argus - The All Seeing System and Network Monitoring Software, www.tcp4me.com.

[2] Paxson, V. "Bro: A System for Detecting Network Intruders in Real-Time," *7th Annual USENIX Security Symposium*, January 1998.

[3] Melvin, S., and Patt, Y. "Handling of Packet Dependencies: A Critical Issue for Highly Parallel Network Processors," *International Conference on Compiler, Architecture, and Synthesis for Embedded Systems (CASES)*, October 2002.

[4] Smith, B., "Architecture and Applications of the HEP Multiprocessor Computer System," *Proceedings, SPIE Real-Time Signal Processing IV*, August 1981.

[5] Kuehn, J., and Smith, B., "The Horizon Supercomputing System: Architecture and Software," Proceedings, Supercomputing 1988, November 1988.

[6] Alverson, R., Callahan, D. Cummings, D., Koblenz, B. Porterfield, A., and Smith, B. "The Tera Computer System," *Proceedings, 1990 International Conference on Supercomputing*, June 1990.

[7] Yamamoto, W., and Nemirovsky, M., "Performance Estimation of Multistreamed, Superscalar Processors," *27th Hawaii International Conference on System Sciences*, January 1994.

[8] Tullsen, D., Eggers, S., and Levy, H., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," Proceedings of the 22nd Annual International Symposium on Computer Architecture," June 1995.

[9] Emer, J. "Simultaneous Multithreading: Multiplying Alpha Performance," *Microprocessor Forum*, 1999.

[10] Nemirovsky, M. "XStream Logic's Optical Network Processor," *Microprocessor Forum*, 2000.

[11] Hinton, G., and Shen, J. "Intel's Multi-Threading Technology," *Microprocessor Forum*, 2001.