APRIL: A Processor Architecture for Multiprocessing

Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Processors in large-scale multiprocessors must be able to tolerate large communication latencies and synchronization delays. This paper describes the architecture of a rapid-context-switching processor called APRIL with support for fine-grain threads and synchronization. APRIL achieves high single-thread performance and supports virtual dynamic threads. A commercial RISC-based implementation of APRIL and a run-time software system that can switch contexts in about 10 cycles is described. Measurements taken for several parallel applications on an APRIL simulator show that the overhead for supporting parallel tasks based on *futures* is reduced by a factor of two over a corresponding implementation on the Encore Multimax. The scalability of a multiprocessor based on APRIL is explored using a performance model. We show that the SPARC-based implementation of APRIL can achieve close to 80% processor utilization with as few as three resident threads per processor in a large-scale cache-based machine with an average base network latency of 55 cycles.

1 Introduction

The requirements placed on a processor in a large-scale multiprocessing environment are different from those in a uniprocessing setting. A processor in a parallel machine must be able to tolerate high memory latencies and handle process synchronization efficiently [2]. This need increases as more processors are added to the system.

Parallel applications impose processing and communication bandwidth demands on the parallel machine. An efficient and cost-effective machine design achieves a balance between the processing power and the communication bandwidth provided. An imbalance is created when an underutilized processor cannot fully exploit the available network bandwidth. When the network has bandwidth to spare, low processor utilization can result from high network latency. An efficient processor design for multiprocessors provides a means for hiding latency. When sufficient parallelism exists, a processor that rapidly switches to an alternate thread of computation during a remote memory request can achieve high utilization.

Processor utilization also diminishes due to synchronization latency. Spin lock accesses have a low overhead of memory requests, but busy-waiting on a synchronization event wastes processor cycles. Synchronization mechanisms that avoid busy-waiting through process blocking incur a high overhead.

Full/empty bit synchronization [22] in a rapid context switching processor allows efficient fine-grain synchronization. This scheme associates synchronization information with objects at the granularity of a data word, allowing a low-overhead expression of maximum concurrency. Because the processor can rapidly switch to other threads, wasteful iterations in spin-wait loops are interleaved with useful work from other threads. This reduces the negative effects of synchronization on processor utilization.

This paper describes the architecture of APRIL, a processor designed for large-scale multiprocessing. APRIL builds on previous research on processors for parallel architectures such as HEP [22], MASA [8], P-RISC [19], [14], [15], and [18]. Most of these processors support *fine-grain interleaving* of instruction streams from multiple threads, but suffer from poor single-thread performance. In the HEP, for example, instructions from a single thread can only be executed once every 8 cycles. Single-thread performance is important for efficiently running sections of applications with low parallelism.

APRIL does not support cycle-by-cycle interleaving of threads. To optimize single-thread performance, APRIL executes instructions from a given thread until it performs a remote memory request or fails in a synchronization attempt. We show that such *coarse-grain multithreading* allows a simple processor design with context switch overheads of 4–10 cycles, without significantly hurting overall system performance (although the pipeline design is complicated by the need to handle pipeline dependencies). In APRIL, thread scheduling is done in software, and unlimited virtual dynamic threads are supported. APRIL supports full/empty bit synchronization, and provides tag support for *futures* [9]. In this paper the terms process, thread, context, and task are used equivalently.

By taking a systems-level design approach that considers not only the processor, but also the compiler and run-time system, we were able to migrate several non-critical operations into the software system, greatly simplifying processor design. APRIL's simplicity allows an implementation based on minor modifications to an existing RISC processor design. We describe such an implementation based on Sun Microsystem's SPARC processor [23]. A compiler for APRIL, a run-time system, and an APRIL simulator are operational. We present simulation results for several parallel applications on APRIL's efficiency in handling fine-grain threads and assess the scalability of multiprocessors based on a coarse-grain multithreaded processor using an analytical model. Our SPARC-based processor supports four hardware contexts and can switch contexts in about 10 cycles, which yields roughly 80% processor utilization in a system with an average base network latency of 55 cycles.

The rest of this paper is organized as follows. Section 2 is an overview of our multiprocessor system architecture and the programming model. The architecture of APRIL is discussed in Section 3, and its instruction set is described in Section 4. A SPARC-based implementation of APRIL is detailed in Section 5. Section 6 discusses the implementation and performance of the APRIL run-time system. Performance measurements of APRIL based on simulations are presented in Section 7. We evaluate the scalability of multithreaded processors in Section 8.

2 The ALEWIFE System

APRIL is the processing element of ALEWIFE, a large-scale multiprocessor being designed at MIT. ALEWIFE is a cache-coherent machine with distributed, globally-shared memory. Cache coherence is maintained using a directory-based protocol [5] over a low-dimension direct network [20]. The directory is distributed with the processing nodes.

2.1 Hardware

As shown in Figure 1, each ALEWIFE node consists of a processing element, floating-point unit, cache, main

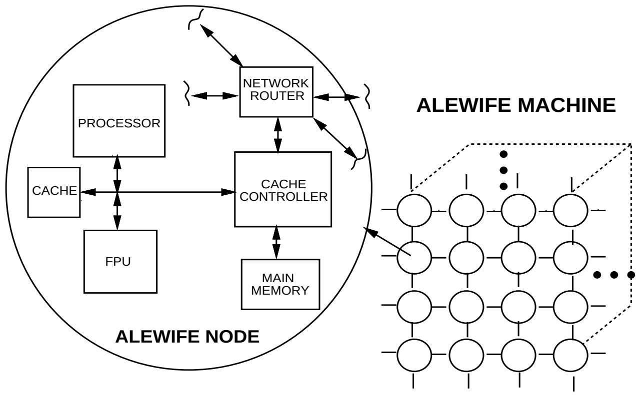

Figure 1: ALEWIFE node.

memory, cache/directory controller and a network routing switch. Multiple nodes are connected via a direct, packet-switched network.

The controller synthesizes a global shared memory space via messages to other nodes, and satisfies requests from other nodes directed to its local memory. It maintains strong cache coherence [7] for memory accesses. On exception conditions, such as cache misses and failed synchronization attempts, the controller can choose to trap the processor or to make the processor wait. A multithreaded processor reduces the ill effects of the long-latency acknowledgment messages resulting from a strong cache coherence protocol. To allow experimentation with other programming models, the controller provides special mechanisms for bypassing the coherence protocol and facilities for preemptive interprocessor interrupts and block transfers.

The ALEWIFE system uses a low-dimension direct network. Such networks scale easily and maintain high nearest-neighbor bandwidth. However, the longer expected latencies of low-dimension direct networks compared to indirect multistage networks increase the need for processors that can tolerate long latencies. Furthermore, the lower bandwidth of direct networks over indirect networks with the same channel width introduces interesting design tradeoffs.

In the ALEWIFE system, a context switch occurs whenever the network must be used to satisfy a request, or on a failed synchronization attempt. Since caches reduce the network request rate, we can employ coarse-grain multithreading (context switch every 50–100 cycles) instead of fine-grain multithreading (context switch every cycle). This simplifies processor design considerably because context switches can be more expensive (4 to 10 cycles), and functionality such as scheduling can be migrated into run-time software. Single-thread performance is optimized, and techniques used in RISC processors for enhancing pipeline performance can be applied [10]. Custom design of a processing element is not required in the ALEWIFE system; indeed, we are using a modified version of a commercial RISC processor for our first-round implementation.

2.2 Programming Model

Our experimental programming language for ALEWIFE is Mul-T [16], an extended version of Scheme. Mul-T's basic mechanism for generating concurrent tasks is the `future` construct. The expression `(future` X`)`, where X is an arbitrary expression, creates a task to evaluate X and also creates an object known as a *future* to eventually hold the value of X. When created, the future is in an *unresolved*, or *undetermined*, state. When the value of X becomes known, the future *resolves* to that value, effectively mutating into the value of X. Concurrency arises because the expression `(future` X`)` returns the future as its value without waiting for the future to resolve. Thus, the computation containing `(future` X`)` can proceed concurrently with the evaluation of X. All tasks execute in a shared address-space.

The result of supplying a future as an operand of some operation depends on the nature of the operation. *Non-strict* operations, such as passing a parameter to a procedure, returning a result from a procedure, assigning a value to a variable, and storing a value into a field of a data structure, can treat a future just like any other kind of value. *Strict* operations such as addition and comparison, if applied to an unresolved future, are suspended until the future resolves and then proceed, using the value to which the future resolved as though that had been the original operand.

The act of suspending if an object is an unresolved future and then proceeding when the future resolves is known as *touching* the object. The touches that automatically occur when strict operations are attempted are referred to as *implicit* touches. Mul-T also includes an *explicit* touching or "strict" primitive `(touch` X`)` that touches the value of the expression X and then returns that value.

Futures express control-level parallelism. In a large class of algorithms, data parallelism is more appropriate. Barriers are a useful means of synchronization for such applications on MIMD machines, but force unnecessary serialization. The same serialization occurs in SIMD machines. Implementing data-level parallelism in a MIMD machine that allows the expression of maximum concurrency requires cheap fine-grain synchronization associated with each data object. We provide this support in hardware with *full/empty bits*.

We are augmenting Mul-T with constructs for data-level parallelism and primitives for placement of data and tasks. As an example, the programmer can use `future-on` which works just like a normal `future` but allows the specification of the node on which to schedule the future. Extending Mul-T in this way allows us to experiment with techniques for enhancing locality and to research language-level issues for programming parallel machines.

3 Processor Architecture

APRIL is a pipelined RISC processor extended with special mechanisms for multiprocessing. This section gives an overview of the APRIL architecture and focuses on its features that support multithreading, fine-grain synchronization, cheap futures, and other models of computation.

The left half of Figure 2 depicts the user-visible processor state comprising four sets of general purpose registers, and four sets of Program Counter (PC) chains and Processor State Registers (PSR). The PC chain represents the instruction addresses corresponding to a thread, and the PSR holds various pieces of process-specific state. Each register set, together with a single PC-chain and PSR, is conceptually grouped into a single entity called a *task frame* (using terminology from [8]). Only one task frame is active at a given time and is designated by a current frame pointer (FP). All register accesses are made to the active register set and instructions are fetched using the active PC-chain. Additionally, a set of 8 global registers that are always accessible (regardless of the FP) is provided.

Registers are 32 bits wide. The PSR is also a 32-bit register and can be read into and written from the general registers. Special instructions can read and write the FP register. The PC-chain includes the Program Counter (PC) and next Program Counter (nPC) which are not directly accessible. This assumes a single-cycle branch delay slot. Condition codes are set as a side effect of compute instructions. A longer branch delay might be necessary if the branch instruction itself does a compare so that condition codes need not be saved [13]; in this case the PC chain is correspondingly longer. Words in memory have a 32 bit data field, and have an additional synchronization bit called the *full/empty* bit.

Use of multiple register sets on the processor, as in the HEP, allows rapid context switching. A context switch is achieved by changing the frame pointer and emptying the pipeline. The cache controller forces a context switch on the processor, typically on remote network requests, and on certain unsuccessful full/empty bit synchronizations.

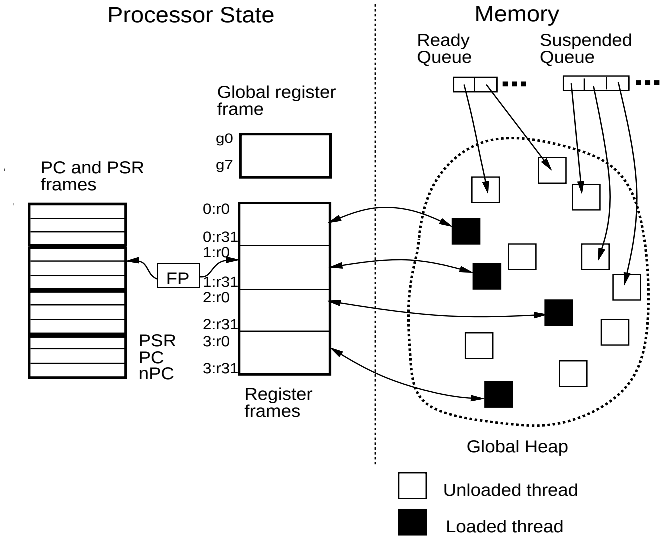

Figure 2: Processor State and Virtual Threads.

APRIL implements *futures* using the trap mechanism. For our proposed experimental implementation based on SPARC, which does not have four separate PC and PSR frames, context switches are also caused through traps. Therefore, a fast trap mechanism is essential. When a trap is signalled in APRIL, the trap mechanism lets the pipeline empty and passes control to the trap handler. The trap handler executes in the same task frame as the thread that trapped so that it can access all of the thread's registers.

3.1 Coarse-Grain Multithreading

In most processor designs to date (e.g. [8, 22, 19, 15]), multithreading has involved cycle-by-cycle interleaving of threads. Such fine-grain multithreading has been used to hide memory latency and also to achieve high pipeline utilization. Pipeline dependencies are avoided by maintaining instructions from different threads in the pipeline, at the price of poor single-thread performance.

In the ALEWIFE machine, we are primarily concerned with the large latencies associated with cache misses that require a network access. Good single thread performance is also important. Therefore APRIL continues executing a single thread until a memory operation involving a remote request (or an unsuccessful synchronization attempt) is encountered. The controller forces the processor to switch to another thread, while it services the request. This approach is called *coarse-grain multithreading*. Processors in message passing multicomputers [21, 27, 6, 4] have traditionally taken this approach to allow overlapping of communication with computation.

Context switching in APRIL is achieved by changing the frame pointer. Since APRIL has four task frames, it can have up to four threads loaded. The thread that is being executed resides in the task frame pointed to by the FP. A context switch simply involves letting the processor pipeline empty while saving the PC-chain and then changing the FP to point to another task frame.

Threads in ALEWIFE are virtual. Only a small subset of all threads can be physically resident on the processors; these threads are called *loaded threads*. The remaining threads are referred to as *unloaded threads* and live on various queues in memory, waiting their turn to be loaded. In a sense, the set of task frames acts like a cache on the virtual threads. This organization is illustrated in Figure 2. The scheduler tries to choose threads from the set of loaded threads for execution to minimize the overhead of saving and restoring threads to and from memory. When control eventually passes back to the thread that suffered a remote request, the controller should have completed servicing the request, provided the other threads ran for enough cycles. By maximizing local cache and memory accesses, the need for context switching reduces to once every 50 or 100 cycles, which allows us to tolerate latencies in the range of 150 to 300 cycles with 4 task frames (see Section 8).

Rapid context switching is used to hide the latency encountered in several other trap events, such as synchronization faults (or attempts to load from "empty" locations). These events can either cause the processor to suspend execution (wait) or to take a trap. In the former case, the controller holds the processor until the request is satisfied. This typically happens on local memory cache misses, and on certain full/empty bit tests. If a trap is taken, the trap handling routine can respond by:

1. *spinning* – immediately return from the trap and retry the trapping instruction.

2. *switch spinning* – context switch without unloading the trapped thread.

3. *blocking* – unload the thread.

The above alternatives must be considered with care because incorrect choices can create or exacerbate starvation and thrashing problems. An extreme example of starvation is this: all loaded threads are spinning or switch spinning on an exception condition that an unloaded thread is responsible for fulfilling. We are investigating several possible mechanisms to handle such problems, including a special controller initiated trap on certain failed synchronization tests, whose handler unloads the thread.

An important aspect of the ALEWIFE system is its combination of caches and multithreading. While this combination is advantageous, it also creates a unique class of thrashing and starvation problems. For example, forward progress can be halted if a context executing on one processor is writing to a location while a context on another processor is reading from it. These two contexts can easily play "cache tag", since writes to a location force a context switch and invalidation of other cached copies, while reads force a context switch and transform read-write copies into read-only copies. Another problem involves thrashing between an instruction and its data; a context will be blocked if it has a load instruction mapped to the same cache line as the target of the load. These and related problems have been addressed with appropriate hardware interlock mechanisms.

3.2 Support for Futures

Executing a Mul-T program with futures incurs two types of overhead not present in sequential programs. First, strict operations must check their operands for availability before using them. Second, there is a cost associated with creating new threads.

Detection of Futures Operand checks for futures done in software imply wasted cycles on every strict operation. Our measurements with Mul-T running on an Encore Multimax show that this is expensive. Even with clever compiler optimizations, there is close to a factor of two loss in performance over a purely sequential implementation (see Table 3). Our solution employs a tagging scheme with hardware-generated traps if an operand to a strict operator is a future. We believe that this hardware support is necessary to make futures a viable construct for expressing parallelism. From an architectural perspective, this mechanism is similar to dynamic type checking in Lisp. However, this mechanism is necessary even in a statically typed language in the presence of dynamic futures.

APRIL uses a simple data type encoding scheme for automatically generating a trap when operands to strict operators are futures. This implementation (discussed in Section 5) obviates the need to explicitly inspect in software the operands to every compute instruction. This is important because we do not want to hurt the efficiency of all compute instructions because of the possibility an operand is a future.

Lazy Task Creation Little can be done to reduce the cost of task creation if `future` is taken as a command to create a new task. In many programs the possibility of creating an excessive number of fine-grain tasks exists. Our solution to this problem is called *lazy task creation* [17]. With lazy task creation a `future` expression does not create a new task, but computes the expression as a local procedure call, leaving behind a marker indicating that a new task could have been created. The new task is created only when some processor becomes idle and looks for work, *stealing* the continuation of that procedure call. Thus, the user can specify the maximum possible parallelism without the overhead of creating a large number of tasks. The race conditions are resolved using the fine-grain locking provided by the full/empty bits.

3.3 Fine-grain synchronization

Besides support for lazy task creation, efficient fine-grain synchronization is essential for large-scale parallel computing. Both the dataflow and data-parallel models of computation rely heavily on the availability of cheap fine-grain synchronization. The unnecessary serialization imposed by barriers in MIMD implementations of data-parallellism can be avoided by allowing fine-grain word-level synchronization in data structures. The traditional `test&set` based synchronization requires extra memory operations and separate data storage for the lock and for the associated data. Busy-waiting or blocking in conventional processors waste additional processor cycles.

APRIL adopts the full/empty bit approach used in the HEP to reduce both the storage requirements and the number of memory accesses. A bit associated with each memory word indicates the state of the word: full or empty. The load of an empty location or the store into a full location can trap the processor causing a context switch, which helps hide synchronization delay. Traps also obviate the additional software tests of the lock in `test&set` operations. A similar mechanism is used to implement I-structures in dataflow machines [3], however APRIL is different in that it implements such synchronizations through software trap handlers.

3.4 Multimodel Support Mechanisms

APRIL is designed primarily for a shared-memory multiprocessor with strongly coherent caches. However, we are considering several additional mechanisms which will permit explicit management of caches and efficient use of network bandwidth. These mechanisms present different computational models to the programmer.

To allow software-enforced cache coherence, we have loads and stores that bypass the hardware coherence mechanism, and a *flush* operation that permits software writeback and invalidation of cache lines. A loaded

Type	Format	Data transfer	Control flow
Compute	`op s1 s2 d`	`d` ← `s1 op s2`	`PC+1`
Memory	`ld` *type* `a d`	`d` ← `mem[a]`	`PC+1`
	`st` *type* `d s`	`mem[a]` ← `s`	`PC+1`
Branch	`j`*cond* `offset`		`if` *cond* `PC+offset` `else PC+1`
	`jmpl offset d`	`d` ← `PC`	`PC+offset`

Table 1: Basic instruction set summary.

context has a *fence counter* that is incremented for each dirty cache line that is flushed and decremented for each acknowledgement from memory. This fence counter may be examined to determine if all writebacks have completed. We are proposing a block-transfer mechanism for efficient transfer of large blocks of data. Finally, we are considering an interprocessor-interrupt mechanism (IPI) which permits preemptive messages to be sent to specific processors. IPIs offer reasonable alternatives to polling and, in conjunction with block-transfers, form a primitive for the message-passing computational model.

Although each of these mechanisms adds complexity to our cache controller, they are easily implemented in the processor through "out-of-band" instructions as discussed in Section 5.

4 Instruction Set

APRIL has a basic RISC instruction set augmented with special memory instructions for full/empty bit operations, multithreading, and cache support. The attraction of an implementation based on simple SPARC processor modifications has resulted in a basic SPARC-like design. All registers are addressed relative to a current frame pointer. Compute instructions are 3-address register-to-register arithmetic/logic operations. Conditional branch instructions take an immediate operand and may increment the PC by the value of the immediate operand depending on the condition codes set by the arithmetic/logic operations. Memory instructions move data between memory and the registers, and also interact with the cache and the full/empty bits. The basic instruction categories are summarized in Table 1. The remainder of this section describes features of APRIL instructions used for supporting multiprocessing.

Data Type Formats APRIL supports tagged pointers for Mul-T, as in the Berkeley SPUR processor [12], by encoding the pointer type in the low order bits of a data word. Associating the type with the pointer has the advantage of saving an additional memory reference when accessing type information. Figure 3 lists the different type encodings. An important purpose of this type encoding scheme is to support hardware detection of futures.

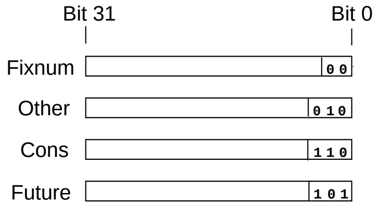

Figure 3: Data Type Encodings.

Future Detection and Compute Instructions Since a compute instruction is a *strict* operation, special action has to be taken if either of its operands is a future. APRIL generates a trap if a future is encountered by a compute instruction. Future pointers are easily detected by their non-zero least significant bit.

Memory Instructions Memory instructions are complex because they interact with the full/empty bits and the cache controller. On a memory access, two data exceptions can occur: the accessed location may not be in the cache (a cache miss), and the accessed location may be empty on a load or full on a store (a full/empty exception). On a cache miss, the cache/directory controller can trap the processor or make the processor wait until the data is available. On full/empty exceptions, the controller can trap the processor, or allow the processor to continue execution. Load instructions also have the option of setting the full/empty bit of the accessed location to empty while store instructions have the option of setting the bit to full. These options give rise to 8 kinds of loads and 8 kinds of stores. The load instructions are listed in Table 2. Store instructions are similar except that they trap on full locations instead of empty locations.

A memory instruction also shares responsibility for detecting futures in either of its address operands. Like compute instructions, memory instructions also trap if the least significant bit of either of their address operands are non-zero. This introduces the restriction that objects in memory cannot be allocated at byte boundaries. This, however, is not a problem because object allocation at word boundaries is favored for other reasons [11]. This trap provides support for implicit future touches in operators that dereference pointers, *e.g.*, `car` in LISP.

Name	Type	Reset f/e bit	EL[1] trap	CM[2] response
`ldtt`	1	No	Yes	Trap
`ldett`	2	Yes	Yes	Trap
`ldnt`	3	No	No	Trap
`ldent`	4	Yes	No	Trap
`ldnw`	5	No	No	Wait
`ldenw`	6	Yes	No	Wait
`ldtw`	7	No	Yes	Wait
`ldetw`	8	Yes	Yes	Wait

[1]Empty location. [2]Cache miss.

Table 2: Load Instructions.

Full/Empty Bit Conditional Branch Instructions Non-trapping memory instructions allow testing of the full/empty bit by setting a condition bit indicating the state of the memory word's full/empty bit. APRIL provides conditional branch instructions, `Jfull` and `Jempty`, that dispatch on this condition bit. This provides a mechanism to explicitly control the action taken following a memory instruction that would normally trap on a full/empty exception.

Frame Pointer Instructions Instructions are provided for manipulating the register frame pointer (FP). FP points to the register frame on which the currently executing thread resides. An `INCFP` instruction increments the FP to point to the next task frame while a `DECFP` instruction decrements it. The incrementing and decrementing is done modulo the number of task frames. `RDFP` reads the value of the FP into a register and `STFP` writes the contents of a register into the FP.

Instructions for Other Mechanisms The special mechanisms discussed in Section 3.4, such as `FLUSH` are made available through "out-of-band" instructions. Interprocessor-interrupts, block-transfers, and `FENCE` operations are initiated via memory-mapped I/O instructions (`LDIO`, `STIO`).

5 An Implementation of APRIL

An ALEWIFE node consists of several interacting subsystems: processor, floating-point unit, cache, memory, cache and directory controller, and network controller. For the first round implementation of the ALEWIFE system, we plan to use a modified SPARC processor and an unmodified SPARC floating-point unit.[1] There are several reasons for this choice. First, we have chosen to devote our limited resources to the design of a custom ALEWIFE cache and directory controller, rather than to processor design. Second, the register windows in the SPARC processor permit a simple implementation of coarse-grain multithreading. Third, most of the instructions envisioned for the original APRIL processor map directly to single or double instruction sequences on the SPARC. Software compatibility with a commercial processor allows easy access to a large body of software. Furthermore, use of a standard processor permits us to ride the technology curve; we can take advantage of new technology as it is developed.

[1]The SPARC-based implementation effort is in collaboration with LSI Logic Corporation.

Rapid Context Switching on SPARC SPARC processors contain an implementation-dependent number of overlapping register windows for speeding up procedure calls. The current register window is altered via SPARC instructions (`SAVE` and `RESTORE`) that modify the Current Window Pointer (CWP). Traps increment the CWP, while the trap return instruction (`RETT`) decrements it. SPARC's register windows are suited for rapid context switching and rapid trap handling because most of the state of a process (*i.e.*, its 24 local registers) can be switched with a single-cycle instruction. Although we are not using multiple register windows for procedure calls within a single thread, this should not significantly hurt performance [25, 24].

To implement coarse-grain multithreading, we use two register windows per task frame – a user window and a trap window. The SPARC processor chosen for our implementation has eight register windows, allowing a maximum of four hardware task frames. Since the SPARC does not have multiple program counter (PC) chains and processor status registers (PSR), our trap code must explicitly save and restore the PSRs during context switches (the PC chain is saved by the trap itself). These values are saved in the trap window. Because the SPARC has a minimum trap overhead of five cycles (for squashing the pipeline and computing the trap vector), context switches will take at least this long. See Section 6.1 for further information.

The SPARC floating-point unit does not support register windows, but has a single, 32-word register file. To retain rapid context switching ability for applications that require efficient floating point performance, we have divided the floating point register file into four sets of eight registers. This is achieved by modifying floating-point instructions in a context dependent fashion as they are loaded into the FPU and by maintaining four different sets of condition bits. A modification of the SPARC processor will make the CWP available externally to allow insertion into the FPU instruction.

Support for Futures We detect futures on the SPARC via two separate mechanisms. Future pointers are tagged with their lowest bit set. Thus, direct use of a future pointer is flagged with a *word-alignment trap*. Furthermore, a strict operation, such as subtraction, applied to one or more future pointers is flagged with a modified *non-fixnum trap*, that is triggered if an operand has its lowest bit set (as opposed to either one of the lowest *two* bits, in the SPARC specification).

Implementation of Loads and Stores The SPARC definition includes the Alternate Space Indicator (ASI) feature that permits a simple implementation of APRIL's many load and store instructions (described in Section 4). The ASI is available externally as an eight-bit field. Normal memory accesses use four of the 256 ASI values to indicate user/supervisor and instruction/data accesses. Special SPARC load and store instructions (`LDASI` and `STASI`) permit use of the other 252 ASI values. Our first-round implementation uses different ASI values to distinguish between flavors of load and store instructions, special mechanisms, and I/O instructions.

Interaction with the Cache Controller The cache controller in the ALEWIFE system maintains strong cache coherence, performs full/empty bit synchronization, and implements special mechanisms. By examining the processor's ASI bits during memory accesses, it can select between different load/store and synchronization behavior, and can determine if special mechanisms should be employed. Through use of the Memory Exception (MEXC) line on SPARC, it can invoke synchronous traps corresponding to cache misses and synchronization (full/empty) mismatches. The controller can suspend processor execution using the MHOLD line. It passes condition information to the processor through the Coprocessor Condition bits (CCCs), permitting the full/empty conditional branch instructions (`Jfull` and `Jempty`) to be implemented as coprocessor branch instructions. Asynchronous traps (IPI's) are delivered via the SPARC's asynchronous trap lines.

6 Compiler and Run-Time System

The compiler and run-time system are integral parts of the processor design effort. A Mul-T compiler for APRIL and a run-time system written partly in APRIL assembly code and partly in T have been implemented. Constructs for user-directed placement of data and processes have also been implemented. The run-time system includes the trap and system routines, Mul-T run-time support, a scheduler, and a system boot routine.

Since a large portion of the support for multithreading, synchronization and futures is provided in software through traps and run-time routines, trap handling must be fast. Below, we describe the implementation and performance of the routines used for trap handling and context switching.

6.1 Cache Miss and Full/Empty Traps

Cache miss traps occur on cache misses that require a network request and cause the processor to context switch. Full/empty synchronization exceptions can occur on certain memory instructions described in Section 4. The processor can respond to these exceptions by *spinning*, *switch spinning*, or *blocking* the thread. In our current implementation, traps handle these exceptions by switch spinning, which involves a context switch to the next task frame.

In our SPARC-based design of APRIL, we implement context switching through the trap mechanism using instructions that change the CWP. The following is a trap routine that context switches to the thread in the next task frame.

```
rdpsr psrreg ; save PSR into a reserved reg.
save         ; increment the window pointer
save         ; by 2
wrpsr psrreg ; restore PSR for the new context
jmpl  r17    ; return from trap and
rett  r18    ; reexecute trapping instruction
```

We count 5 cycles for the trap mechanism to allow the pipeline to empty and save relevant processor state before passing control to the trap handler. The above trap handler takes an additional 6 cycles for a total of 11 cycles to effect the context switch. In a custom APRIL implementation, the cycles lost due to PC saves in the hardware trap sequence, and those in calling the trap handler for the PSR saves/restores and double incrementing the frame pointer could be avoided, allowing a four-cycle context switch.

6.2 Future Touch Trap

When a future touch trap is signalled, the future that caused the trap will be in a register. The trap handler has to decode the trapping instruction to find that register. The future is resolved if the full/empty bit of the future's value slot is set to full. If it is resolved, the future in the register is replaced with the resolved value; otherwise the trap routine can decide to *switch spin* or *block* the thread that trapped. Our future touch trap handler takes 23 cycles to execute if the future is resolved.

If the trap handler decides to block the thread on an unresolved future, the thread must be unloaded from the hardware task frame, and an alternate thread may be loaded. Loading a thread involves writing the state of the thread, including its general registers, its PC chain, and its PSR, into a hardware task frame on the processor, and unloading a thread involves saving the state of a thread out to memory. Loading and unloading threads are expensive operations unless there is special hardware support for block movement of data between registers and memory. Since the scheduling mechanism favors processor-resident threads, loading and unloading of threads should be infrequent. However, this is an issue that is under investigation.

7 Performance Measurements

This section presents some results on APRIL's performance in handling fine-grain tasks. We have implemented a simulator for the ALEWIFE system written in C and T. Figure 4 illustrates the organization of the simulator. The Mul-T compiler produces APRIL code, which gets linked with the run-time system to yield an executable program. The instruction-level APRIL processor simulator interprets APRIL instructions. It is written in T and simulates 40,000 APRIL instructions per second when run on a SPARCServer 330. The processor simulator interacts with the cache and directory simulator (written in C) on memory instructions. The cache simulator in turn interacts with the network simulator (also written in C) when making remote memory operations. The simulator has proved to be a useful tool in evaluating system-wide architectural tradeoffs as it provides more accurate results than a trace driven simulation. The speed of the simulator has allowed us to execute lengthy parallel programs. As an example, in a run of **speech** (described below), the simulated program ran for 100 million simulated cycles before completing.

Evaluation of the ALEWIFE architecture through simulations is in progress. A sampling of our results on the performance of APRIL running parallel programs is presented here. Table 3 lists the execution times of four programs written in Mul-T: **fib**, **factor**, **queens** and **speech**. **fib** is the ubiquitous doubly recursive Fibonacci program with '`future`'s around each of its recursive calls, **factor** finds the largest prime factor of each number in a range of numbers and sums them up, **queens** finds all solutions to the n-queens chess problem for $n = 8$ and **speech** is a modified Viterbi graph search algorithm used in a connected speech recognition

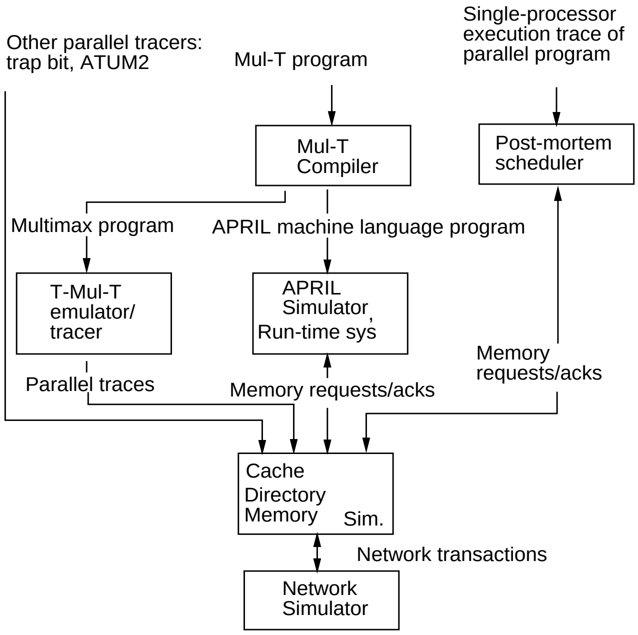

Figure 4: Simulator Organization.

system called SUMMIT, developed by the Spoken Language Systems Group at MIT. We ran each program on the Encore Multimax, on APRIL using normal task creation, and on APRIL using lazy task creation. For purposes of comparison, execution time has been normalized to the time taken to execute a sequential version of each program, *i.e.*, with no futures and compiled with an optimizing T-compiler.

The difference between running the same sequential code on T and on Mul-T on the Encore Multimax (columns "T seq" and "Mul-T seq") is due to the overhead of future detection. Since the Encore does not support hardware detection of futures, an overhead of a factor of 2 is introduced, even though no futures are actually created. There is no overhead on APRIL, which demonstrates the advantage of tag support for futures.

The difference between running sequential code on Mul-T and running parallel code on Mul-T with one processor ("Mul-T seq" and 1) is due to the overhead of thread creation and synchronization in a parallel program. This overhead is very large for the **fib** benchmark on both the Encore and APRIL using normal task creation because of very fine-grain thread creation. This overhead accounts for approximately a factor of 28 in execution time. For APRIL with normal futures, this overhead accounts for a factor of 14. Lazy task creation on APRIL creates threads only when the machine has the resources to execute them, and performs much better because it has the effect of dynamically partitioning the program into coarser-grain threads and creating

		T	Mul-T					
Program	System	seq	seq	1	2	4	8	16
fib	Encore	1.0	1.8	28.9	16.3	9.2	5.1	
	APRIL	1.0	1.0	14.2	7.1	3.6	1.8	0.97
	Apr-lazy	1.0	1.0	1.5	0.78	0.44	0.29	0.19
factor	Encore	1.0	1.4	1.9	0.96	0.50	0.26	
	APRIL	1.0	1.0	1.8	0.90	0.45	0.23	0.12
	Apr-lazy	1.0	1.0	1.0	0.52	0.26	0.14	0.09
queens	Encore	1.0	1.8	2.1	1.0	0.54	0.31	
	APRIL	1.0	1.0	1.4	0.67	0.33	0.18	0.10
	Apr-lazy	1.0	1.0	1.0	0.51	0.26	0.13	0.07
speech	Encore	1.0	2.0	2.3	1.2	0.62	0.36	
	APRIL	1.0	1.0	1.2	0.60	0.31	0.17	0.10
	Apr-lazy	1.0	1.0	1.0	0.52	0.27	0.15	0.09

Table 3: Execution time for Mul-T benchmarks. "T seq" is T running sequential code, "Mul-T seq" is Mul-T running sequential code, 1 to 16 denote number of processors running parallel code.

fewer futures. The overhead introduced is only a factor of 1.5. In all of the programs, APRIL consistently demonstrates lower overhead due to support for thread creation and synchronization over the Encore.

Measurements for multiple processor executions on APRIL (2 – 16) used the processor simulator without the cache and network simulators, in effect simulating a shared-memory machine with no memory latency. The numbers demonstrate that APRIL and its run-time system allow parallel program performance to scale when synchronization and task creation overheads are taken into account, but when memory latency is ignored. The effect of communication in large-scale machines depends on several factors such as scheduling, which are active areas of investigation.

8 Scalability of Multithreaded Processor Systems

Multithreading enhances processor efficiency by allowing execution to proceed on alternate threads while the memory requests of other threads are being satisfied. However, any new mechanism is useful only if it enhances *overall system performance*. This section analyzes the system performance of multithreaded processors.

A multithreaded processor design must address the tradeoff between reduced processor idle time and increased cache miss rates, network contention, and context management overhead. The private working sets of multiple contexts interfere in the cache. The added interference misses coupled with the higher average traffic generated by a higher utilized processor impose greater bandwidth demands on the interconnection network. Context management instructions required to switch the processor between threads also add to the overhead. Furthermore, the application must display sufficient parallelism to allow multiple thread assignment to each processor.

What is a good performance metric to evaluate multithreading? A good measure of system performance is system power, which is the product of the number of processors and the average processor utilization. Provided the computation of processor utilization takes into account the deleterious effects of cache, network, and context-switching overhead, the processor utilization is itself a good measure.

We have developed a model for multithreaded processor utilization that includes the cache, network, and switching overhead effects. A detailed analysis is presented in [1]. This section will summarize the model and our chief results. Processor utilization U as a function of the number of threads resident on a processor p is derived as a function of the cache miss rate $m(p)$, the network latency $T(p)$, and the context switching overhead C:

$$U(p) = \begin{cases} \frac{p}{1+T(p)m(p)} & \text{for } p < \frac{1+T(p)m(p)}{1+Cm(p)} \\ \frac{1}{1+Cm(p)} & \text{for } p \geq \frac{1+T(p)m(p)}{1+Cm(p)} \end{cases} \qquad (1)$$

When the number of threads is small, complete overlapping of network latency is not possible. Processor utilization with one thread is $1/(1+m(1)T(1))$. Ideally, with p threads available to overlap network delays, the utilization would increase p-fold. In practice, because the miss rate and network latency increase to $m(p)$ and $T(p)$, the utilization becomes $p/(1+m(p)T(p))$.

When it is possible to completely overlap network latency, processor utilization is limited only by the context switching overhead paid on every miss (assuming a context switch happens on a cache miss), and is given by $1/(1+m(p)C)$.

The models for the cache and network terms have been validated through simulations. Both these terms are shown to be the sum of two components: one component independent of the number of threads p and the other linearly related to p (to first order). Multithreading is shown to be useful when p is small enough that the fixed components dominate.

Let us look at some results for the default set of system parameters given in Table 4. The analysis assumes 8000 processors arranged in a three dimensional array. In such a system, the average number of hops between a random pair of nodes is $nk/3 = 20$, where n denotes network dimension and k its radix. This yields an average round trip network latency of 55 cycles for an unloaded network, when memory latency and average packet size are taken into account. The fixed miss rate comprises first-time fetches of blocks into the cache, and the interference due to multiprocessor coherence invalidations.

Parameter	Value
Memory latency	10 cycles
Network dimension n	3
Network radix k	20
Fixed miss rate	2%
Average packet size	4
Cache block size	16 bytes
Thread working set size	250 blocks
Cache size	64 Kbytes

Table 4: Default system parameters.

Figure 5 displays processor utilization as a function of the number of threads resident on the processor when context switching overhead is 10 cycles. The degree to which the cache, network, and overhead components impact overall processor utilization is also shown. The ideal curve shows the increase in processor utilization when both the cache miss rate and network contention correspond to that of a single process, and do not increase with the degree of multithreading p.

We see that as few as three processes yield close to 80% utilization for a ten-cycle context-switch overhead

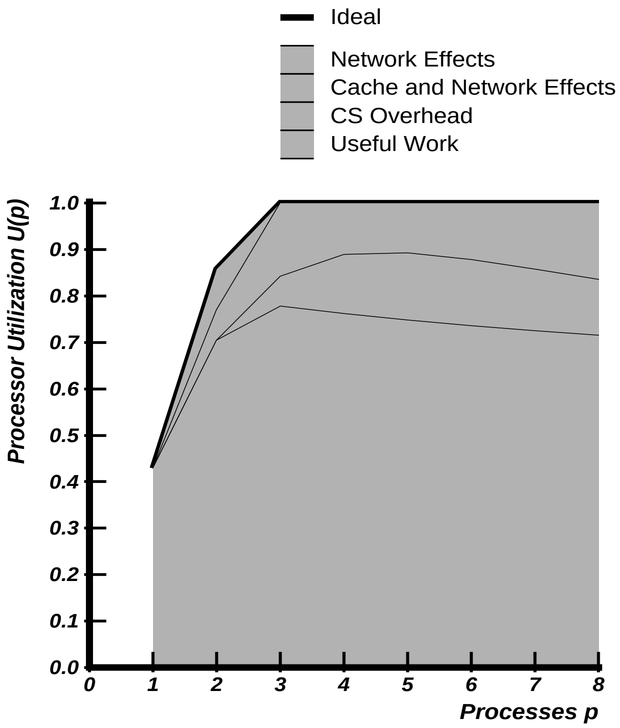

Figure 5: Relative sizes of the cache, network and overhead components that affect processor utilization.

which corresponds to our initial SPARC-based implementation of APRIL. This result is similar to that reported by Weber and Gupta [26] for coarse-grain multithreaded processors. The main reason a low degree of multithreading is sufficient is that context switches are forced only on cache misses, which are expected to happen infrequently. The marginal benefits of additional processes is seen to decrease due to network and cache interference.

Why is utilization limited to a maximum of about 0.80 despite an ample supply of threads? The reason is that *available network bandwidth limits the maximum rate at which computation can proceed.* When available network bandwidth is used up, adding more processes will not improve processor utilization. On the contrary, more processes will degrade performance due to increased cache interference. In such a situation, for better system performance, effort is best spent in increasing the network bandwidth, or in reducing the bandwidth requirement of each thread.

The relatively large ten-cycle context switch overhead does not significantly impact performance for the default set of parameters because utilization depends on the product of context switching frequency and switching overhead, and the switching frequency is expected to be small in a cache-based system. This observation is important because it allows a simpler processor implementation, and is exploited in the design of APRIL.

A multithreaded processor requires larger caches to sustain the working sets of multiple processes, although cache interference is mitigated if the processes share code and data. For the default parameter set, we found that caches greater than 64 Kbytes comfortably sustain the working sets of four processes. Smaller caches suffer more interference and reduce the benefits of multithreading.

9 Conclusions

We described the architecture of APRIL – a coarse-grain multithreaded processor to be used in a cache-coherent multiprocessor called ALEWIFE. By rapidly switching to an alternate task, APRIL can hide communication and synchronization delays and achieve high processor utilization. The processor makes effective use of available network bandwidth because it is rarely idle. APRIL provides support for fine-grain tasking and detection of futures. It achieves high single-thread performance by executing instructions from a given task until an exception condition like a synchronization fault or remote memory operation occurs. Coherent caches reduce the context switch rate to approximately once every 50–100 cycles. Therefore context switch overheads in the 4–10 cycle range are tolerable, significantly simplifying processor design. By providing hardware support only for performance-critical operations and migrating other functionality into the compiler and run-time system, we were able to simplify the processor design even further.

We described a SPARC-based implementation of APRIL that uses the register windows of SPARC as task frames for multiple threads. A processor simulator and an APRIL compiler and run-time system have been written. The SPARC-based implementation of APRIL switches contexts in 11 cycles. APRIL and its associated run-time system practically eliminate the overhead of fine-grain task creation and detection of futures. For Mul-T, the overhead reduces from 100% on an Encore Multimax-based implementation to under 5% on APRIL. We evaluated the scalability of multithreaded processors in large-scale parallel machines using an analytical model. For typical system parameters and a 10 cycle context-switch overhead, the processor can achieve close to 80% utilization with 3 processor resident threads.

10 Acknowledgements

We would like to acknowledge the contributions of the members the ALEWIFE research group. In particular, Dan Nussbaum was partly responsible for the processor simulator and run-time system and was the source of a gamut of ideas, David Chaiken wrote the cache simulator, Kirk Johnson supplied the benchmarks, and Gino Maa and Sue Lee wrote the network simulator. We appreciate help from Gene Hill, Mark Perry, and Jim Pena from LSI Logic Corporation for the SPARC-based implementation effort. Our design was influenced by Bert Halstead's work on multithreaded processors. Our research benefited significantly from discussions with Bert Halstead, Tom Knight, Greg Papadopoulos, Juan Loaiza, Bill Dally, Steve Ward, Rishiyur Nikhil, Arvind, and John Hennessy. Beng-Hong Lim is partly supported by an Analog Devices Fellowship. The research reported in this paper is funded by DARPA contract # N00014-87-K-0825 and by grants from the Sloan Foundation and IBM.

References

[1] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. September 1989. MIT VLSI Memo 89-566, Laboratory for Computer Science.

[2] Arvind and Robert A. Iannucci. *Two Fundamental Issues in Multiprocessing.* Technical Report TM 330, MIT, Laboratory for Computer Science, October 1987.

[3] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, (Springer-Verlag Lecture Notes in Computer Science 279)*, September/October 1986.

[4] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *Computer*, 21(8):9–24, August 1988.

[5] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. June 1990. To appear in IEEE Computer.

[6] W. J. Dally et al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 189–196, IEEE, New York, June 1987.

[7] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event or-

dering in multiprocessors. *IEEE Computer*, 9–21, February 1988.

[8] R.H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, IEEE, New York, June 1988.

[9] Robert H. Halstead. Multilisp: A Language for Parallel Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[10] J. L. Hennessy and T. R. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[11] J. L. Hennessy et al. Hardware/Software Tradeoffs for Increased Performance. In *Proc. SIGARCH/SIGPLAN Symp. Architectural Support for Programming Languages and Operating Systems*, pages 2–11, March 1982. ACM, Palo Alto, CA.

[12] M. D. Hill et al. Design Decisions in SPUR. *Computer*, 19(10):8–22, November 1986.

[13] Mark Horowitz et al. A 32-Bit Microprocessor with 2K-Byte On-Chip Instruction Cache. *IEEE Journal of Solid-State Circuits*, October 1987.

[14] R.A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Hawaii, June 1988.

[15] W. J. Kaminsky and E. S. Davidson. Developing a Multiple-Instruction-Stream Single-Chip Processor. *Computer*, 66–78, December 1979.

[16] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implemenation*, June 1989.

[17] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel tasks. In *Proceedings of Symposium on Lisp and Functional Programming*, June 1990. To appear.

[18] Nigel P. Topham, Amos Omondi and Roland N. Ibbett. Context Flow: An Alternative to Conventional Pipelined Architectures. *The Journal of Supercomputing*, 2(1):29–53, 1988.

[19] Rishiyur S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? In *Proceedings 16th Annual International Symposium on Computer Architecture*, IEEE, New York, June 1989.

[20] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12), December 1984.

[21] Charles L. Seitz. The Cosmic Cube. *CACM*, 28(1):22–33, January 1985.

[22] B.J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[23] SPARC Architecture Manual. 1988. SUN Microsystems, Mountain View, California.

[24] P. A. Steenkiste and J. L. Hennessy. A Simple Interprocedural Register Allocation Algorithm and Its Effectiveness for LISP. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, January 1989.

[25] David W. Wall. Global Register Allocation at Link Time. In *SIGPLAN '86, Conference on Compiler Construction*, June 1986.

[26] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings 16th Annual International Symposium on Computer Architecture*, IEEE, New York, June 1989.

[27] Colin Whitby-Strevens. The Transputer. In *Proceedings 12th Annual International Symposium on Computer Architecture*, IEEE, New York, June 1985.