Lightweight Implementation of the POSIX Threads API for an On-Chip MIPS Multiprocessor with VCI Interconnect

Frédéric Pétrot and Pascal Gomez
ASIM/LIP6
Université Pierre et Marie Curie
France

Abstract

This paper relates our experience in designing from scratch a multi-threaded kernel for a MIPS R3000 on-chip multiprocessor. We briefly present the target architecture build around a VCI compliant interconnect, and the CPU characteristics. Then we focus on the implementation of part of the POSIX 1003.1b and 1003.1c standards. We conclude this case study by simulation results obtained by cycle true simulation of an MJPEG video decoder application on the multiprocessor, using several scheduler organizations and architectural parameters.

1. Introduction

Applications targeted to *SoC* implementations are often specified as a set of concurrent tasks exchanging data. Actual co-design implementations of such specifications require a multi-threaded kernel to execute the parts of the application that has been mapped to software. As the complexity of applications grows, more computational power but also more programmable platforms are useful. In that situation, on-chip multiprocessors with several general purpose processors are emerging in the industry, either for low-end applications such as audio codec, or for high end applications such as video decoders or network processors. Compared to multiprocessor computers, such integrated architectures feature a shared memory access with low latency and potentially very high throughput, since the number of wires on chip can be much greater than on a printed card board.

This paper relates our experience in implementing from scratch the POSIX thread API for an on-chip MIPS R3000 multiprocessor architecture. We choose to implement the POSIX thread API for several reasons:

- It is standardized, and is *de facto* available on many existing computer systems,
- It is well known, taught in universities and many applications make use of it,
- The 1003.1c defines no more than 5 objects, allowing to have a compact implementation.

All these facts make the development of a bare kernel easier, because its relies on a hopefully well behaved standard and API, and allows direct functional comparison of the same application code on a commercial host and on our multiprocessor platform.

The main contribution of this paper is to relate the difficult points in developing a multiprocessor kernel general enough to support the POSIX API on top of a generic interconnect. The problems that we have encountered, such as memory consistency, compiler optimization avoidance, interrupt dispatching, are outlined. We also want this kernel to support several types of organization: Symmetric using a single scheduler, Distributed using one scheduler per processor, Distributed with centralized synchronization, with or without task migration, etc, in order to compare them experimentally.

2. Target architecture and basic architectural needs

The general architecture that we target is presented on Figure 1. It makes use of one or more MIPS R3000 as CPU, and a Virtual Chip Interconnect [1] compliant interconnect on the which are plugged memories and dedicated hardware when required. The MIPS CPU has been chosen because it is small and efficient, and also widely used in embedded applications [2]. It has the following properties:

- two separated caches for instruction and data,
- direct mapped caches, as for level one caches on usual computers,
- write buffer with write update and write through policy. Write back policy allows to minimize the memory traffic and allows to build burst when updating memory, particularly useful for SD-RAM, but it is very complicated to ensure proper memory consistency[3],
- no memory management unit (MMU), logical addresses are physical addresses. Virtual memory isn't particularly useful on resource constrained hardware because the total memory is fixed at design time. Page protection is also not an issue in current SoC implementations. Finally, the multi-threaded nature of the software makes it natural to share the physical address space.

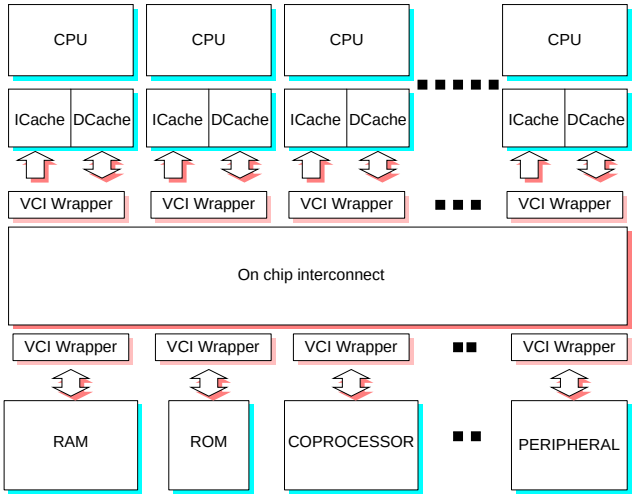

Figure 1. General VCI based SoC architecture

The interconnect is VCI compliant. This ensures that our kernel can be used with any interconnect that has VCI wrappers. This means:

- it is basically a shared memory architecture, since all addresses that go through a VCI interface are seen alike by the targets,
- the actual interconnect is not known: only the services it provides are. VCI doesn't say anything about cache consistency or interrupt handling, cached or uncached memory space and so on.

On the R3000, we do not distinguish between *user* and *kernel* modes, and the whole application runs in *kernel* mode. This allows to spare cycles when *a)* accessing the uncached memory space (in *kernel* space in the MIPS R3000) that contains the semaphore engine and hardware module, *b)* using privileged instructions, to set the registers of coprocessor 0, necessary mainly to mask/unmask the interrupts and use the processor identifier. The number of cycles spared is at least 300, to save and restore the context and analyze the cause of the call. This is to be compared with the execution times of the kernel functions given in Table 2.

The architecture needs are the following:

Protected access to shared data. This is done using a spin lock, whose implementation depends on the architecture. A spin lock is acquired using the `pthread_spin_lock` function, and released using the `pthread_spin_unlock`. Our implementation assumes that there is a very basic binary semaphore engine in uncached space such that reading a slot in it returns the value of the slots and sets it to 1. Writing in a slot sets it to 0. Other strategies can make use of the read modify write opcode of the VCI standard, but this is less efficient and requires that each target is capable of locking its own access to a given initiator, thus requiring more resources per target.

Cache coherency. If the interconnect is a shared bus, the use of a snoopy cache is sufficient to ensure cache coherence. This has the great advantage of avoiding any processor to memory traffic. If the interconnect is VCI compliant (either bus or network), an access to a shared variable requires either to flush the cache line that contains this variable to obtain a fresh copy of the memory or to have such variables placed in uncached space. This is due to the fact that VCI doesn't allow the building of snoopy caches, because the VCI wrapper would have to know both the cache directory entries and be aware of all the traffic, and that is simply no possible for a generic interconnect. Both solutions are not very satisfactory as the generated traffic eats-up bandwidth and leads to higher power consumption, particularly costly for on-chip applications. However, this is the price to pay for interconnect genericity. In any case, synchronization for the access to shared data is mandatory. Using caches is meaningful even for shared data only used once because we benefit from the read burst transfers on the interconnect.

Processor identification. The CPUs must have an internal register allowing their identification within the system. Each CPU is assigned a number at boot time, as some startup actions should be done only once, such as clearing the `bss` area and creating the scheduler. Also, the kernel sometimes needs to know which processor it runs on to access processor specific data.

Compared to other initiatives, [4] for example, our kernel is designed for multiprocessor hardware. We target a lightweight distributed scheduler with shared data objects, each having its own lock, and task migration.

3. Overview of the pthread specifications

The main kernel objects are the threads and the scheduler. A thread is created by a call to:

```
int pthread_create(
    pthread_t *thread, pthread_attr_t *attr,
    void *(*start)(void *), void *arg)
```

This executes the thread whose behavior is the `start` function called with `arg` as argument. The `attr` structure contains thread attributes, such as stack size, stack address and scheduling policies. Such attributes are particularly useful when dealing with embedded systems or SoCs, in the which the memory map is not standardized. The value returned in the `thread` pointer is a unique identifier for the thread.

A thread can be in one of 5 states, as illustrated by the Figure 2. Changing state is usually done using some *pthread* function on a shared object. Exceptions to this rule is going from RUNNABLE to RUN, which is done by the scheduler using a given policy, and backward from RUN to RUNNABLE using `sched_yield`. This function does not belong to the POSIX thread specifications in the which there is no way to voluntarily release the processor for a thread. This function is usually called in a timer interrupt handler for time sharing. In our implementation, the thread identifier is a pointer to the thread structure. Note that POSIX, unlike more specialized API such as OSEK/VDX[5], doesn't

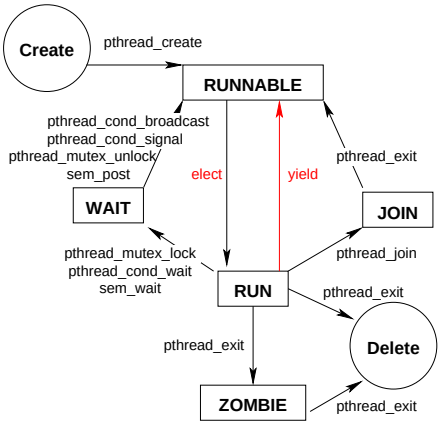

Figure 2. Pthread states

provide a mean for static thread creation. It is a shortcoming because most embedded applications do not need dynamic task creation. A thread structure basically contains the context of execution of a thread and pointers to other threads.

The scheduler manages 5 lists of threads. It may be shared by all processors (Symmetrical Multi-Processor), or exist as such on every processor (Distributed). The access to the scheduler must be performed in critical section, and under the protection of a lock. However, this lock can be taken for only very few instructions if the other shared objects have their own locks, which allows for greater parallelism.

Other implemented objects are the spin lock, mutex, conditions, and semaphores. Spin locks are the low level test and set accesses, that usually perform active polling. Mutex are used to sequentialize access to shared data by suspending a task is the access is denied. Conditions are used to voluntarily suspend a thread waiting for a condition to become true. Semaphore are mainly useful when dealing with hardware, because `sem_post` is the only function that can be called in interruption handlers.

4. Implementation

Our implementation is a complete redesign that doesn't make use of any existing code. Most of it is written in **C**. This **C** is not particularly portable because it makes use of physical addresses to access the semaphore engine, the terminal, and so on. Some assembly is necessary for the deeply processor dependent actions: access to coprocessor0 registers, access to processor registers for context saving and restoring, interruption handling and cache line flushing.

To avoid a *big lock* on the scheduler, every mutex and semaphore has its own lock. This ensures that the scheduler lock will be actually acquired only if a thread state changes, and this will minimize scheduler locking time, usually around 10 instructions, providing better parallelism.

In the following, we assume that access to all shared variables are a fresh local copy of the memory in the cache.

Booting sequence

Algorithm 1 describes the boot sequence of the multiprocessor. The identification of the processors is determined in a pseudo-random manner. For example, if the interconnect is a bus, the priorities on the bus will define this order. It shall be noted that there is no need to know how many processors are booting. This remains true for the whole system implementation.

Two implementation points are worth to be seen. A weak memory consistency model [6] is sufficient to access the shared variable *proc_id*, since it is updated after a synchronization point. This model is indeed sufficient for POSIX threads applications. The *scheduler_created* variable must be declared with the `volatile` type qualifier to ensure that the compiler will not optimize this seemingly infinite loop.

The `main` thread is executed by processor 0, and, if the application is multi-thread, it will create new threads. When available, these threads will be executed on any processor waiting for an available thread. Here the execute() function will run a new thread without saving the current context, since the program will never come back at that point. The thread to execute is chosen by the elect() function. Currently, we have only implemented a FIFO election algorithm.

Algorithm 1 Booting sequence

```
Definition and statically setting shared variables to 0
scheduler_created ⇐ 0                          No scheduler exists
proc_id ⇐ 0                                    First processor is numbered 0
mask interruptions                             Done by all processors
Self numbering of the processors
spin_lock(lock), set_proc_id(proc_id++), spin_unlock(lock)
set stack for currently running processor
if get_proc_id = 0 then
    clear .bss
    scheduler and main thread creation
    scheduler_created ⇐ 1                      indicates scheduler creation
    enable interruptions
    goto the main function
else
    Wait until scheduler creation by processor 0
    while scheduler_created = 0 end while
    Acquire the scheduler lock to execute a thread
    spin_lock(scheduler)
    execute(elect())
end if
```

Context Switch

For the R3000, a context switch saves the current value of the CPU registers into the context variable of the thread that is currently executing and sets the values of the CPU registers to the value of the context variable of the new thread to execute. The tricky part is that the return address of the function is a register of the context. Therefore, restoring a context sends the program back where the context was

saved, not to the current caller of the context switching routine.

An important question is to define the registers and/or variables that belong to the context. This is architecture and kernel dependent: For example, a field of current compiler research concerns the use of scratch pad memory instead of data cache [7] in embedded multiprocessors. Assuming that the kernel allows the threads to be preempted, the main memory must be updated before the same thread is executed again. If this is not the case, the thread may run on an other processor and used stalled data from memory. On a Sparc processor, the kernel must also define what windows are to be saved/restored by context switches, and this may have an important impact on performance and power consumption.

CPU Idle Loop

When no tasks are RUNNABLE, the CPU runs some kind of idle loop. Current processors could benefit from this to enter a low power state. However, waking up from such a state is in the order of 100 ms[8] and its use would therefore be very application dependent.

An other, lower latency solution, would be to launch an *idle* thread whose rôle is to infinitely call the `sched_yield` function. There must be one such thread per CPU, because it is possible that all CPUs are waiting for some coprocessor to complete there work. These threads should not be made RUN as long as other threads exist in the RUNNABLE list. This strategy is elegant in theory, but it uses as many threads resources as processors and needs a specific scheduling policy for them.

Our current choice is to use a more *ad-hoc* solution, in the which all idle CPUs enter the same idle loop, that is described in Algorithm 2.

This routine is called only once the scheduler lock has been acquired and the interruptions globally masked. We use the save_context function to save the current thread context, and update the register that contains the function return address to have it point to the end of the function. This action is necessary to avoid going through the idle loop again when the restore_context function is called. Once done, the current thread may be woken up again on an other processor, and therefore we may not continue to use the thread stack, since this would modify the local data area of the thread. This justifies the use of a dedicated stack (one for all processors in SMP and one per processor for distributed scheduling). The registers of the CPU can be modified, since they are not anymore belonging to the thread context.

The *wakeup* variable is a `volatile` field of the scheduler. It is needed to inform this idle loop that a thread has been made RUNNABLE. Each function that awakes (or creates) a thread decrements the variable. The *go* local variable enable each CPU to register for getting out of this loop. When a pthread function releases a mutex, signals a condition or posts a semaphore, the CPU with the correct *go* value is allowed to run the awaken thread.

This takes places after having released the semaphore lock to allow the other functions to change the threads states, and also after having enable the interruptions in order for the hardware to notify the end of a computation.

Algorithm 2 CPU Idle Loop

```
if current then
  save_context(current)
  current.return_addr_register ⇐ end of function
  stack ⇐ scheduler stack
  repeat
    go ⇐ wakeup++
    spin_unlock(lock), it_global_unmask
    while wakeup > go end while
    it_global_mask, spin_lock(lock)
    thread ⇐ elect()
  until thread exists
  restore_context(thread)
end if
end of function
```

5. Experimental setup

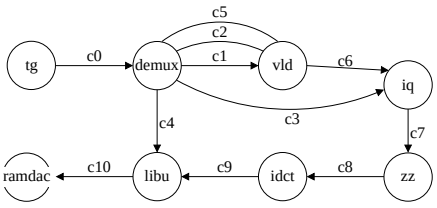

Figure 3. MJPEG application task graph.

The experiments detailed here use two applications. The first one is a multimedia application, a decoder of a flow of JPEG images (known as Motion JPEG), whose task graph is presented Figure 3. The second application is made of couple of tasks exchanging data through FIFO, and we call it COMM. COMM is a synthetic application in the which scheduler access occurs 10 times more often for a given time frame that in the MJPEG application. This allows to check the behavior of the kernel on a very system intensive applications. COMM spends from 56% to 79% of its time in kernel calls, depending on the number of processors. The architecture the applications run on is presented Figure 4, but the number of processors vary from one experiment to another. This architecture is simulated using the CASS [9] cycle true simulator whose models are compatible with SystemC. The application code is cross-compiled with **gcc** and linked with the kernel. Non disturbing profiling is performed to obtain the figures of merits of each kernel implementation.

We now want to test several implementations of our kernel. The literature defines several types of scheduler organization. We review here the three that we have retained for implementation and outline their differences.

- Symmetric Multiprocessor SMP. There is a unique scheduler shared by all the processors and protected

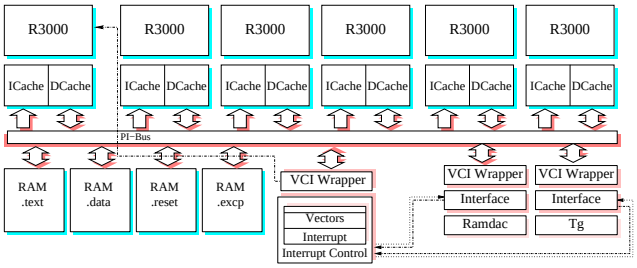

Figure 4. Target architecture simulated using cycle true models.

by a lock. The threads can run on any processor, and thus migrate. This allows to theoretically evenly distribute the load on all CPUs at the cost of more cache misses,

- Centralized Non SMP NON_SMP_CS. There is a unique scheduler shared by all processors and protected by a lock. Every thread is assigned to a given processor and can run only on it. This avoid task migration at the cost of less efficient use of CPUs cycles (more time spend in the CPU idle loop),
- Distributed Non SMP NON_SMP_DS. There are as many schedulers as processors, and as many locks as schedulers. Every thread is assigned to a given processor and can run only on it. This allows a better parallelism by replicating the scheduler that is a key resource.

In both non SMP strategies, load balancing is performed so as to optimize CPU usage, with a per task load measured on a uniprocessor setup.

In all cases, the spin locks, mutex, conditions and semaphores are shared, and there is a spin lock per mutex and semaphore.

Our experimentation tries to give a quantitative answer to the choice of scheduler organization.

6. Results

The table 1 indicate the code size for the three versions of the scheduler. The NON_SMP_DS strategy grows dynamically of around 80 bytes per processor, whereas there is no change for the other ones.

Organization	SMP	NON_SMP_CS	NON_SMP_DS
Code size	7556	9704	10192

Table 1. Kernel code size in byte.

The Figure 5 plots the execution time of the MJPEG application for 48 small pictures. The SMP and NON_SMP_CS

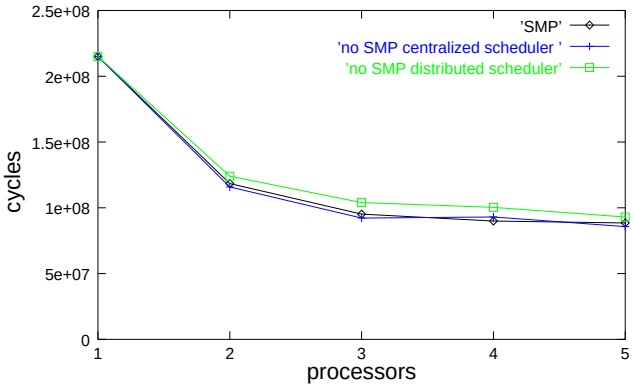

Figure 5. Execution times of the MJPEG application

approaches are more than 10% faster than the NON_SMP_DS one. The Figure 6 shows the time spent in the CPU Idle

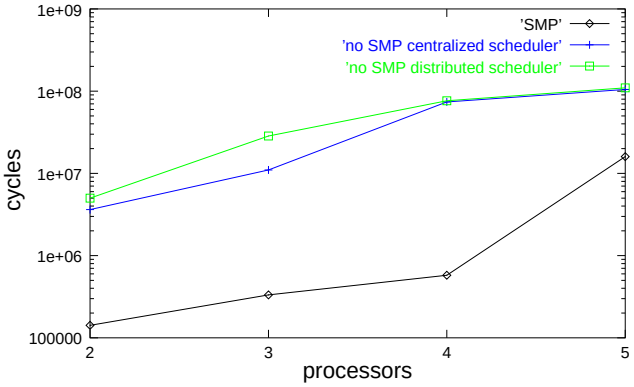

Figure 6. Cycles spent in the CPU Idle Loop.

Loop. We see that the SMP kernel spends more than an order of magnitude less time in the Idle loops than the other strategies. This outline its capacity to use the CPU cycles more efficiently. However, task migration has a high cost in terms of cache misses, and therefore, the final cycle count is comparable to the other ones. It shall be noted that the SMP interest might become less clear if shared memory latency access increases too much. The NON_SMP_DS strategy is more complicated from an implementation point of view. This is the reason why it is less efficient in this case.

Our second application does not exchange data between processors, and the performances obtained are plotted on Figure 7. The benefit of having one scheduler per processor is very sensitive here, and visible on the NON_SMP_DS results. The only resource shared here is the bus, and since the caches are big enough to contain most of the application data, the application uses the processors at about full power.

The Table 2 shows the number of cycles necessary to perform the main POSIX function using our SMP kernel on

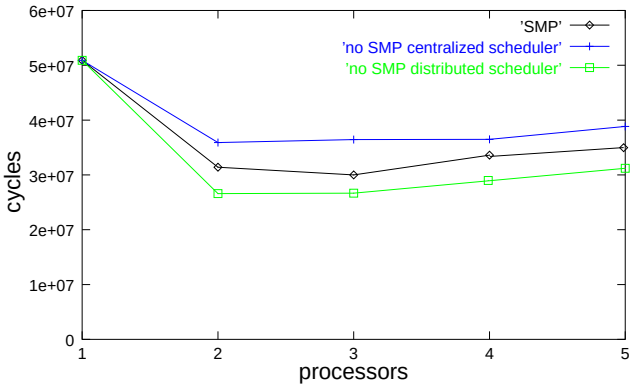

Figure 7. Execution times of the COMM application.

the target architecture. Theses values have been obtained by

Operations	Number of processors			
	1	2	4	6
Context Switch	172	187	263	351
Mutex Lock (acquired)	36	56	61	74
Mutex Unlock	30	30	31	34
Mutex Lock (suspended)	117	123	258	366
Mutex Unlock (awakes a thread)	-	191	198	218
Thread Creation	667	738	823	1085
Thread Exit	98	117	142	230
Semaphore Acquisition	36	48	74	76
Semaphore Release	36	50	78	130
Interrupt Handler (†)	200	430	1100	1900

Table 2. Performance of some kernel functions in numbers of cycle.

performing the mean for over 1000 calls. For the interrupts (†), all processors were interrupted simultaneously.

7. Conclusion and future trends

This paper relates our experience in implementing the POSIX threads for a MIPS multiprocessor based around a VCI interconnect. Compared to kernels for on-board multiprocessors, that have been extensively studied in the past, our setup uses a generic interconnect for which the exact interconnect is not known, the CPUs use physical addresses, and the latency to access shared memory is much lower.

The implementation is a bit tricky, but quite compact and efficient. Our experimentations have shown that a POSIX compliant SMP kernel allowing task migration is an acceptable solution in terms of generality, performance and memory footprint for SoC.

The main problem due to the introduction of networks on chip is the increasing memory access latency. One of our goal in the short term is to investigate the use of latency hiding techniques for these networks. Our next experiment concerns the use of a dedicated hardware for semaphore and pollable variables that would queue the acquiring requests and put to sleep the requesting processors until a change occurs to the variable. This can be effectively supported by the VCI interconnect, by the mean of its request/acknowledge handshake. In that case, the implementation of `pthread_spin_lock` could suspend the calling task. This could be efficiently taken care of if the processors that run the kernel are processors with multiple hardware contexts, as introduced in [10].

The SMP version of this kernel, and a minimal C library, is part of the Disydent tool suite distributed under GPL at `www-asim.lip6.fr/disydent`.

References

[1] VSI Alliance. *Virtual Component Interface Standard (OCB 2 2.0)*, August 2000.

[2] T. R. Halfhill. Embedded market breaks new ground. *Microprocessor Report*, January 2000.

[3] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[4] T. P. Baker, F. Mueller, and V. Rustagi. Experience with a prototype of the POSIX 'minimal realtime system profile. In *Proceedings 11th IEEE Workshop on Real-Time Operating Systems and Software. RTOSS '94*, pages 12–16, Seattle, WA, USA, 1994.

[5] OSEK/VDX. *OSEK/VDX Operating System Specification 2.2*, September 2001. http://www.osek-vdx.org.

[6] A. S. Tanenbaum. *Distributed Operating Systems*, chapter 6.3, pages 315–333. Prentice Hall, 1995.

[7] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Design Automation Conference*, pages 628–633, New Orleans, LA, June 2002.

[8] J. Montanaro et al. A 160mhz 32b 0.5w cmos risc microprocessor. In *ISSCC Digest of Technical Papers*, pages 214–215, February 1996.

[9] Frédéric Pétrot, Denis Hommais, and Alain Greiner. Cycle precise core based hardware/software system simulation with predictable event propagation. In *Proceeding of the* 23rd *Euromicro Conference*, pages 182–187, Budapest, Hungary, September 1997. IEEE.

[10] Jr. R. H. Halstead and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *15th Annual International Symposium on Computer Architecture*, pages 443–451, June 1988.