# VIRTUAL MEMORY ON A RECONFIGURABLE NETWORK ARCHITECTURE

Eileen Archer Allison

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-16 May 1987

TABLE OF CONTENTS

# LIST OF FIGURES

# Chapter 1

# Introduction

The Texas Reconfigurable Array Computer (TRAC) is an experimental computer that was developed at The University of Texas at Austin. It is a multiprocessor system with a dynamically reconfigurable banyan network. TRAC's interconnection network is the SW - banyan which supports a path from each apex to each base of the network while requiring only n log n switch nodes. The banyan network can dynamically partition and configure the processor, memory, and I/O resources of the system into different architectural organizations as demanded for efficient application formulation and solution [Sejnowski 80].

This unique hardware architecture poses several equally unique problems in the design and implementation of an operating system. A conceptual design for an operating system for TRAC was developed by Daniel Canas [Canas 83]. Memory management for TRAC is particularly interesting because of the spectrum of design options supported by the interconnection network. This thesis begins with the conceptual design of Canas for virtualization of physical memory, resolves and details this design, implements that portion of the TRAC operating system which virtualizes physical memory, and finally evaluates the properties of the design.

The experimental TRAC hardware has never attained the stability necessary for support of development of an operating system. It was necessary, therefore, to use a simulator of the TRAC hardware as the execution environment.

A TRAC simulator which allows the execution of TRAC binary programs has been implemented on the DEC-20 computer. The simulator operates at instruction level and has been enhanced to reflect the reconfigurability of the TRAC network. These enhancements provide a simulation of the connection between processors and memory modules, which therefore allows the execution of operating system routines on the simulator.

This thesis studics virtual memory on a reconfigurable network architecture such as TRAC. The environment of the study will be the TRAC simulator. The approach has been to strengthen the simulator to handle the execution of a page replacement algorithm, to enlarge the TRAC instruction set to provide the necessary functionality for an operating system routine, and finally to code a page replacement algorithm in TRAC assembly language and to test its performance on the TRAC simulator.

The thesis is organized as follows. Chapter 2 describes the instruction set and architecture of TRAC. Chapter 3 discusses the page management system for a reconfigurable network architecture. Chapter 4 gives the results which come from executing test programs on the TRAC simulator. Chapter 5 summarizes the results of the performance evaluation and formulates an improved capability for memory management in the TRAC architecture.

# Chapter 2

# TRAC Architecture

This chapter describes the TRAC architecture to the extent necessary to understand the operating system routine which handles page faults. Additional details can be found in the TRAC Users' Manual [Deshpande 85].

The major unique features of the TRAC computer are space sharing, reconfigurability, inter-task communication ability, varistructuring, and the fact that its design makes it a virtual machine to the user [Sejnowski 80]. Space sharing implies that independent or interacting tasks can all be running simultaneously on the same computer, as opposed to the time sharing where tasks must await their allotted time slot to execute. Reconfigurability is the ability of TRAC to dynamically partition its processors and memories under software control to obtain optimal use and minimal waste for the set of tasks to be run. The third unique feature, inter-task communication, is possible by sending packets between tasks or by tasks sharing a memory. Varistructuring is another unique feature of TRAC. It allows the execution of programs to take place on machines with different degrees of parallelism without any change to the program code.

Finally, the machine is virtual in that user programs can be oblivious of the specific set of memory and processor modules used. A given login architecture can be realized with different sets of physical resources. Memories have space-page registers which allow them to be combined in any way to form address spaces [Sejnowski 80].

Traditional paging algorithms do not take advantage of the unique features of the TRAC computer. Therefore, a unique paging algorithm has been developed to handle page faults with better performance on a reconfigurable network architecture such as the TRAC computer.

## 2.1 The Banyan Network

The most fundamental concept in the TRAC system is the use of an SW-banyan network to interconnect the set of processors with the set of resources. A banyan is represented by a graph in which nodes are divided into three types: apex, base and intermediate nodes. In TRAC, apex nodes represent processors, base nodes represent memory resources, and intermediate nodes represent switch nodes. An important property of the network is that there is a unique path between any apex (processor) and base (memory) node pair through the intermediate (switch) nodes [Sejnowski 80]. Figure 2-1 shows TRAC's 4 processor - 9 memory system built around the banyan interconnection network.

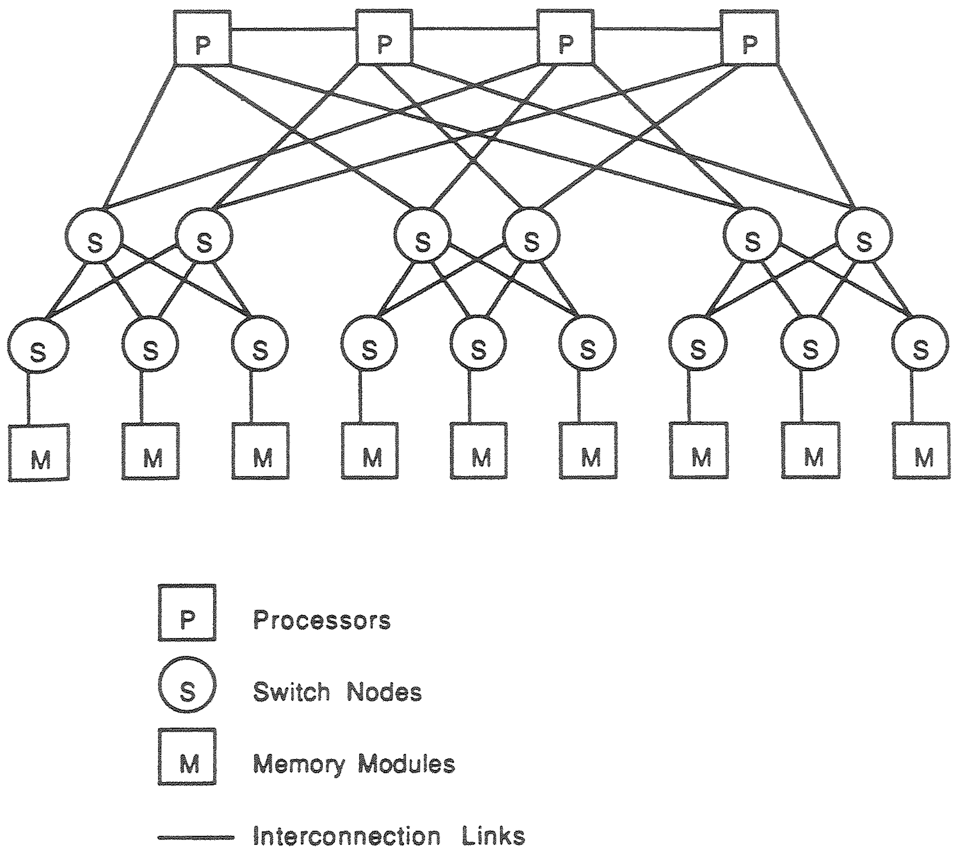


**Figure 2-1:** Banyan Network

## 2.2 Trees

A TRAC task uses the switch nodes of the network to establish three types of subtrees: data trees, instruction trees, and shared memory trees. The data tree is used as a data bus to connect a processor with appropriate memory modules (See figure 2-2). The memory modules connected to a processor using a data tree are actively attached to the processor throughout the lifetime of the data tree. These memory modules are private to the processor and cannot be shared by any other processor.

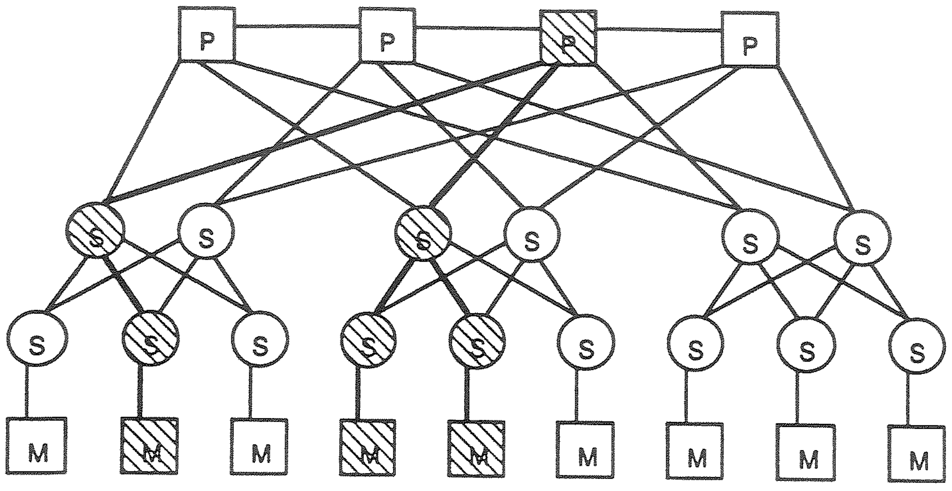


**Figure 2-2:** Data Tree

The second type of subtree, the instruction tree, is used as an instruction bus to broadcast instructions to a set of processors performing the same task (SIMD mode). The processors execute the instruction in lockstep mode. As illustrated in figure 2-3, the instruction tree connects one memory with a set of processors. Each switch node in the banyan network includes hardware for a carry-look-ahead tree node. The hardware is activated by an instruction tree link in the module. The look-ahead tree makes varistructured arithmetic possible on TRAC. This is discussed in detail later in this chapter.

Shared memory trees are the third type of subtree. They connect a set of processors to a single shared memory module for the purpose of sharing data (See figure 2-4). Parts of the shared memory tree are used to extend a data bus from time to time in order to share a memory [Sejnowski

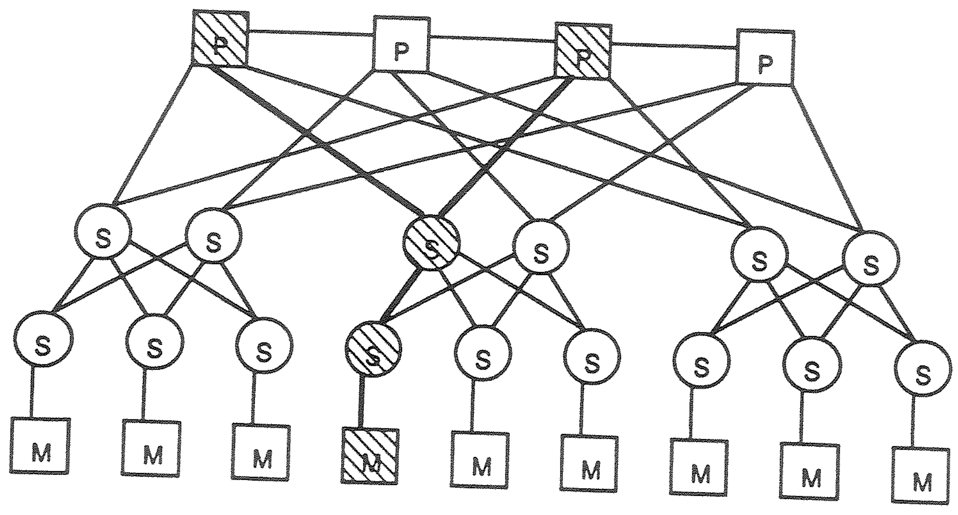


**Figure 2-3:** Instruction Tree

80]. In order to distinguish between shared trees, each shared tree is assigned a number from 0 to 3. These shared tree numbers are called **colors** in the TRAC literature.

The two instructions which are necessary to implement shared memory on TRAC are ACQR and SMS. The ACQR instruction activates the shared tree and makes the variables in the memory module available to the processor. The shared tree mechanism provides, to the processors involved, a mutually exclusive access to the memory module and thus to its storage area. To release the memory module, the processor executes the SMS instruction which sets an unowned bit in the memory module to signal the memory is unowned and has no active chain to it [Deshpande 85].

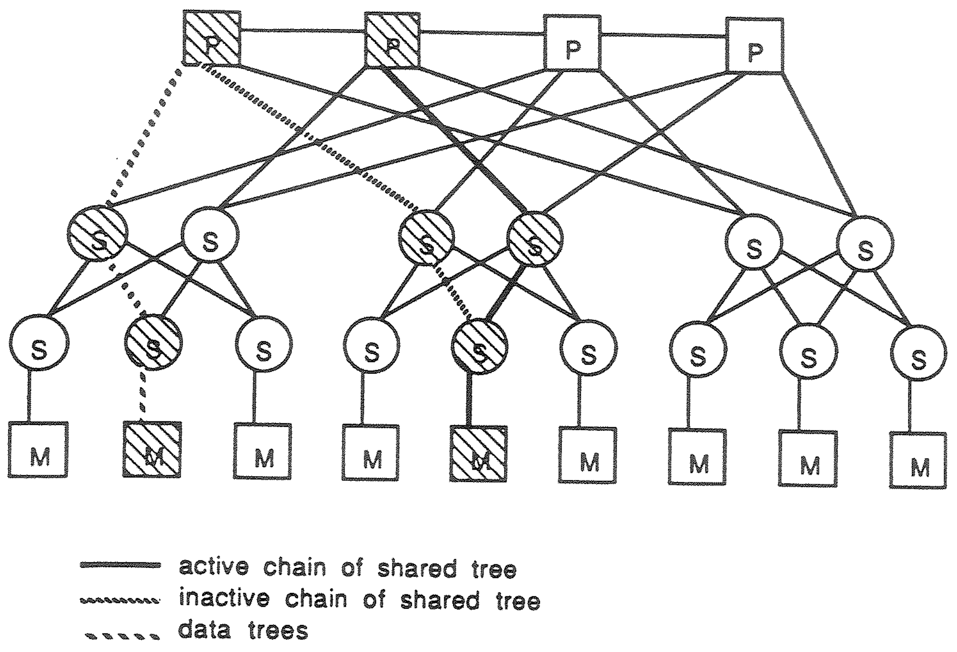


**Figure 2-4:** Shared Tree

## 2.3 Multidimensional Memory System

In an SIMD environment, the processing elements in a given task execute the same instruction but use different operands. This means that each processor needs a private memory of its own to store the operands, and all processors together need a common storage area for instructions. The TRAC architecture defines four types of storage spaces, each intended for a specific type of storage necessary during SIMD processing [Deshpande 85].

The four space types are operand, data, program and control. The first space, the operand space, is a two-dimensional space. A detailed description of two-dimensional space is given in the next section. The operations defined on this space are stack-oriented. The operand stack is used as a temporary storage area for program routines: the operands are pushed on the stack by the routines, operations are performed on the stack and results are popped thereafter [Deshpande 85].

The data space is also a two-dimensional space. The data space is a global storage area. During processing, the data is transferred from the data space to the operand stack, processed there, and then the results are transferred back to the data space. The size of the data space does not change during the execution of a process [Deshpande 85].

The program space is a one-dimensional space which stores instructions for a process. To an executing process, the program space appears as read-only memory. The instruction fetched from program space is broadcast over the instruction tree spanning the task [Deshpande 85].

The last space, the control space, is also a one-dimensional space. The control space stores all data that is common to all processors of an SIMD task. Examples of this type of data are subroutine return addresses, operand addressing offsets and operand descriptors. The user may read from and write into the control space [Deshpande 85]. The last two pages of control space are reserved for paging. The paging algorithm uses these reserved pages as page buffers.

Page faults can occur on all four of these space types.

## 2.4 Two-dimensional Memory

A processor module in TRAC is built around a byte-slice ALU with flexibly controllable carry linkages. The operation of the ALU and the carry linkages are controlled by the microcode. By controlling the carry linkages among the processors, the TRAC architecture is able to create a processing element of a larger word width. Whereas conventional processors offer general purpose registers, TRAC instructions refer only to memory-based operands and execute memory-to-memory operations. This scheme presents to the user a completely general and modular processing slice and thus provides a virtual processing resource [Deshpande 85].

The above scheme has three advantages. First, the assembly language user is oblivious to the reconfigurable processing hardware. Second, the assembly language programs are free of system specific details. And lastly, because the final configuration is made by the task, the performance of the algorithm and/or the use of the system resources can be optimized at execution time by providing varying amounts of parallelism [Deshpande 85].

The variable word-width capability is made possible through **arithmetic and logic descriptors.** These descriptors specify the properties of the operands. The micro routines that implement the instructions use the descriptors as parameters. An arithmetic and logic descriptor can be divided into three parts: the processor configuration parameters, the element descriptor, and the vector descriptor. This section describes each of these descriptors in detail [Deshpande 85].

There are two processor configuration parameters: the number of processors in a task and the number of processors per group. The number of processors in a task is specified by the parameter **p**. Since each processor can operate on one byte at a time, at most p bytes can be operated upon simultaneously. To coordinate the activities among the task processors, the information of assigned parallelism is supplied by assigning them ID's **0** through **p-1**. These processors are connected together via the carry look-ahead (CLA) tree traversing the task instruction tree. The CLA tree imposes a default direction of significance on the processors. The least significant processor is assigned ID **0**, and the more significant processor of any two is assigned the higher ID [Deshpande 85].

The second processor configuration parameter is **n**, the number of processors that will handle an operand element. This parameter is determined after the number of processors dedicated to the task is determined. The set of processors that work in unison on a single element is called a **group**. The number **n** specifies the number of processors that are grouped together via the CLA to create the multi-byte processing element which will operate on a single operand. The parameter **p** should be an integral multiple of **n** [Deshpande 85].

There are two element descriptors: slice count and most significant processor. A **group** handles one operand element at a time. The constituent processors together form a multi-byte processor with a multi-byte word width. A group is therefore able to handle an **n**-byte word at a time. This **n** byte word is called a **slice.** If a data element is larger than a slice, it is **folded** over a number of slices in the memory in a manner similar to the one used to store a double precision operand of a conventional microprocessor. The minimum number of slices needed to thus accommodate the entire element is called the **slice count (h).** The maximum size of the element accommodated in a group of size **n** and with a slice count **h**, is **n** times **h** bytes. When the size of the element is smaller than **n** times **h**, the result is **garbage bytes.** The number of garbage bytes is always less than **n** [Deshpande 85].

The parameter **f** specifies the position of the most significant byte of the element to be processed. If the processors within a group were to be numbered from 0 through n-1 corresponding to their increasing significance within the group, **f** specifies the processor which should handle the most significant byte of the element under consideration. The purpose of **f** is to flag that processor to be the handler of element sign, carry and overflow status [Deshpande 85].

There are two vector descriptors: band count **b** and highest order group **g**. The concept of band count is similar to that of the slice count. What slice count is to an element, the band count is to a vector of elements. A vector of elements are distributed over the available groups in a task. If the number of elements in the vector is larger than the number of groups, the vector is folded into a number of **bands.** The minimum number of bands necessary to accommodate the entire vector is specified by **b** [Deshpande 85].

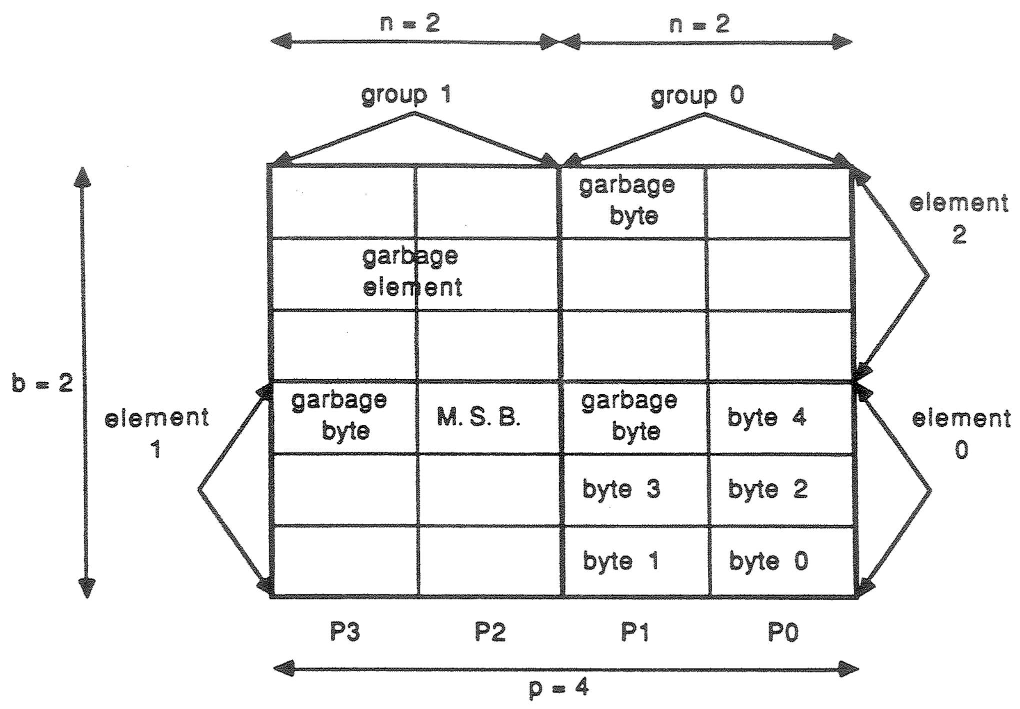
Descriptor **g** has a function similar to descriptor **f**, except that it used for describing a vector of elements. If the number of elements in the vector being considered is less than the maximum that can be accommodated, the last band will have less than **p/n** valid elements. The group that handles the highest order element of the vector is designated by **g** [Deshpande 85].

Figure 2-5 graphically captures the essence of the arithmetic and logic descriptors. Figure 2-6 shows an alternate arrangement for the same vector. The more efficient schema is figure 2-6 because it has fewer garbage bytes and fewer slices per processor.

## 2.5 Pointer Registers

Each space type has two 16-bit pointers or address registers dedicated to it. All registers are located in every memory module [Deshpande 85]. These 16-bit address registers limit the memory size of each space type to 64K bytes. The operand space and the data space are two-dimensional space types and can have four pages of each space type for each processor in the task.

The two pointers of the operand space are the T pointer and the N pointer. The T pointer points to the top, filled location in the operand stack. During arithmetic operations, the T pointer points to the element on top of the stack. In comparison, during diadic arithmetic operations, the N pointer points to the element next to the top of the stack [Deshpande 85].



h = 3 ( slices/element )
f = 0 ( Processor within group which processes MSB )
g = 0 ( group 0 processes most significant element )
b = 2 ( band count )

**Figure 2-5:** Arithmetic and Logic Descriptors

The data space has two pointers used for general purpose addressing of data space. These two address registers are the X pointer and the Y pointer [Deshpande 85].

The program space, where instructions are stored, has a P pointer which is the program counter for the task. The other pointer for the program space is the W pointer, which is used as a shadow register. The user cannot access the program space via this register [Deshpande 85].

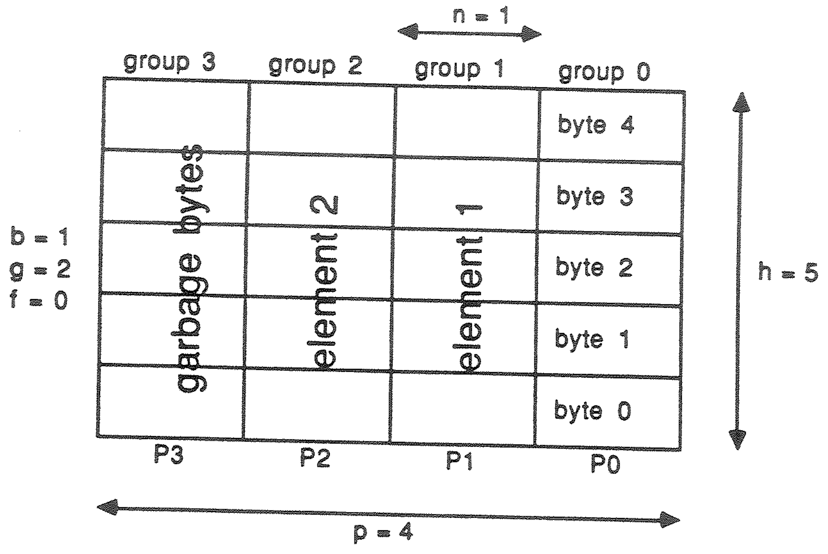


**Figure 2-6:** Alternate Storage Schema

The two pointers of the control space are the S pointer and the D pointer. The S pointer always points to the top of the control stack. Conversely, the D pointer points to the bottom of the control stack. All accesses made to the control space are made with offsets to one of these pointers [Deshpande 85].

## 2.6 Primary Memory

On the actual TRAC machine, each memory module in TRAC is 64K bytes in size, and is divided into four 16K byte pages. However, on the TRAC simulator, where the paging algorithm is implemented, the memory modules are 4K bytes in size and are divided into four 1K byte pages. Nevertheless, on both TRAC and the simulator, each page in each memory module can be any of the four memory space types.

Each page has a corresponding virtual page number register. This register has eight bits, with two bits specifying the memory space type and two bits specifying the page number. Since the address size of each space type is limited by the 16-bit address registers to 64K bytes, there can only be four pages of each space type and only two bits are needed to specify the page number.

## 2.7 Secondary Memory

In addition to primary memories, TRAC has self-managing secondary memory (SMSM). A key use of the SMSM's in the TRAC architecture is to virtualize the memory. If a processor accesses memory which is not currently in one of the memory modules attached to the processor, the SMSM is used to page out a current page in one of the modules, and then read in the desired page of memory in the now free memory module [Sejnowski 80].

The SMSM's are attached to primary memories. A primary memory need not have an SMSM attached to it, and SMSM's can be detached from one primary memory and attached to another when neither primary memory is being used. A page buffer must be reserved for paging operations in each memory module with an SMSM attached to it. This will be the last page in control space.

Two TRAC instructions which access the SMSM and are necessary for the page fault routines are WDAT and ONRD. The WDAT instruction is used to write data to the backup device. The ONRD instruction is used to read data from the backup device. The read is non-destructive.

## 2.8 Packets

Because a desired shared tree structure may be impossible to generate because of blockage by other preestablished circuits, TRAC has packets to ensure feasibility of all possible permutations of interprocessor communication circuits. The packets allow interprocessor and inter-data tree communication. Packet communication is necessary to handle a page fault when the page which caused the fault is in a shared memory currently not attached to the processor [Deshpande 85].

In packet-switching, the data is stored and forwarded by intermediate switch nodes. Each packet is eight bytes long. The first byte of the packet corresponds to the network-ID of the destination processor. The remaining seven bytes are data [Deshpande 85].

The SIP instruction is used to send a packet. The packet may be received by executing an RCVIP instruction [Deshpande 85].

## 2.9 Interrupts

An explanation of interrupts on TRAC will be helpful in understanding the page handler. Page faults occur when a physical realization in the primary memory cannot be found. They happen during the execution of an instruction, and execution cannot continue until the page fault is serviced [Deshpande 82].

When an interrupt occurs, control goes to a fixed address in the microcode where the interrupt register is tested to find the source of the interrupt. The interrupt register has eight bits. Seven of the bits signal different types of interrupts (See Figure 2-6). Bit 0, the Reset bit (RST), is set by the system wide reset signal and is equivalent to restarting the system. Bit 1 is the Page Fault Status bit (PFS). It signals three cases: a page fault, a memory protection fault, and a supervisor protection fault. Page faults cannot be masked and must be serviced as they occur. Bits 2 and 3 are the Interrupt Timer Slow and Fast bits (ITS and ITF). They indicate a timeout condition on the corresponding slow and fast timers on the processor modules. Bit 4, the Input/Output Interrupt bit (IOI), is set by the SMSM and it signals that the last search for a file has ended and that it is ready to transfer data to or from that file within a certain fixed time period. Bit 5, the Interrupting Packet Arrival bit (IPA), is set on reception of a global packet over the banyan interconnection network. And finally, the halt bit (HLT), bit 6, is set by the halt line going to each processor module. When this bit is set, the processor goes into a halt state and waits for a restart or synchronizing signal [Deshpande 82].

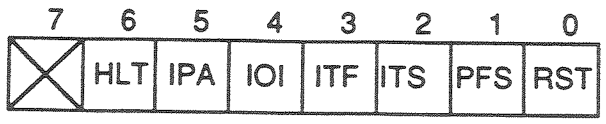


**Figure 2-7:** The Interrupt Register

Examining the interrupt register determines the type of interrupt which in turn dictates what microcode is then executed. For all types of interrupts, the microcode saves the status in a reserved page in data space. During a page fault, the machine status must be saved in a fixed location on a page that will always be in primary memory. For this purpose, page zero of data space has been reserved. In order to maintain consistency, all other interrupts also store the status in page zero of data space [Deshpande 82].

The status information that must be saved and the address at which it is stored are determined by the type of the interrupt [Deshpande 82]. During a page fault interrupt, the following information is saved: the ALU registers, the status bits on the TRAC processor module, the condition codes, the last microcommand that caused the page fault, the last microaddress, the last opcode, and the X and P (program counter) pointers [Deshpande 82].

In addition to storing status information, page zero of the data space stores addresses for the software service routines for interrupts. The address of the page fault handler is stored in two bytes of data space: the high byte of the routine is in address 2 and the low byte is in address 3 [Deshpande 82].

A return from interrupt (RTI) instruction has been added to the TRAC instruction set given in the TRAC Users' Manual [Deshpande 85]. This instruction is necessary to implement a page fault interrupt. Program counter, register X, loop counter and other values saved on the stack when the interrupt occurred are restored. RTI then returns control to the interrupted program.

# Chapter 3

# Paging Algorithm

This chapter describes a paging algorithm designed by Daniel Canas for a reconfigurable network architecture (RNA) such as TRAC [Canas 83]. This algorithm is implemented and tested on the TRAC simulator. The results of the tests are described in Chapter 4.

The paging algorithm for an RNA is complicated by the unique features of such an architecture and by the desire for optimum parallelism. Because TRAC is intended to support high degrees of parallel processing, the memory virtualization mechanisms must avoid serialization due to paging [De Groot 81].

Three features of TRAC which directly augment the paging problem are shared memory, reconfigurability, and two-dimensional memories. Shared memory complicates the paging problem because two situations instead of one can cause a task executing in lockstep mode to encounter a page fault. Either the page in question is stored in a shared memory module currently not attached to the task, or the page is stored in a backup device [Canas 83]. Therefore, when a page fault occurs, the shared memory tables must be searched to determine which type of fault has occurred.

Reconfigurability causes additional problems because a task does not know the topology of its configuration until load time. Different execution requests for the same task may execute on different hardware configurations. Processors may or may not have access to backup devices where its pages are stored [Canas 83]. Therefore, paging tasks must be created, so that a task can have virtual memory on any given configuration.

The third complication to the paging problem is two-dimensional memory. When a page fault occurs on a two-dimensional memory space, more than one page must be replaced. That is, more than one page must be read into primary memory, and more than one page must be written to secondary memory [Canas 83].

Section 1 gives an overview of the paging algorithm for a page fault when the page in question is stored in a backup device. Sections 2, 3, and 4 describe specific parts of the paging algorithm for this type of fault in more detail. Section 2 describes breaking up the task-wide instruction tree. Section 3 discusses the actual page transfer, and Section 4 describes recreating the task. Section 5 describes how a shared memory fault is handled by the paging algorithm.

## 3.1 Overview of the Paging Process

If the fault is not a shared memory fault, then the page in question is located in a backup device. What follows is an overview of the paging algorithm to handle this type of fault.

All processors of an SISD or SIMD task on TRAC execute in lockstep. Each memory access for data is always to the same virtual page in each column and to the same address within a page. Consequently, each time a page fault occurs, all processors within the task fault at the same time. If the page fault occurs on a two-dimensional memory type, then each processor must perform a paging operation on its column of the memory [De Groot 81]. If the memory type is one-dimensional, only one processor, specifically the processor at the root of the instruction tree, need perform a paging operation.

When a physical memory page is to be swapped out, the backup device to which the page is to be written must be determined. If both the page-out and the backup device belong to the same processor, that processor can output the page to the backup device. This operation is called a **local write**. If, however, the page-out belongs to one processor and the backup device to which the page is to be written belongs to another processor, then the two processors must cooperate to output the page. This operation is called a **non-local write**. The operations **local read** and **non-local read** are similarly defined [De Groot 81].

While loading a task, two restrictions must be imposed. First, a page buffer must be reserved in each memory module which has an SMSM and which will be page 3 of Control Space (CS-3). Second, each processor that is not a paging task must have a page buffer for non-local reads and writes. Page 2 of control space has been reserved for this purpose. Both these pages are required for the transfer of pages among processors [Canas 83]. Pages 2 and 3 of control space were chosen to be page buffers ad hoc because a one-dimensional space is needed for paging.

Eight instructions have been added to the TRAC instruction set to move bytes into and out of the page buffers. MsB moves bytes from one of the four spaces to the page buffer. The **s** can be O, P, C, or D for operand, program, control or data space, respectively. MBs moves bytes from the buffer to the memory space specified in the instruction by **s**.

When paging two-dimensional memory, one page from each column must be replaced by each processor. Maximal paging performance obviously occurs when all processors perform their own paging operations concurrently. Therefore, the task must be broken up so that processors can discontinue lockstep execution mode and begin operating independently. Each processor performs its own paging code. The paging code executed depends on the following three attributes:

1. Is the processor a paging task?;
2. Is the processor the owner of the page to be replaced in main memory?;

3. Is the page which caused the fault a one-dimensional or two-dimensional memory space?

The processors execute the paging code asynchronously and can complete at different times. Upon completion of all the page replacements, the processors must resynchronize and once again enter lockstep execution mode [De Groot 81].

The creation of paging tasks implies the deletion of the task-wide instruction tree. Page in/out is accomplished among two processors which must be connected by an instruction tree. Therefore, these trees must be created and deleted dynamically at the request of the paging tasks [Canas 83]. For this purpose, the instructions DINST and CINST have been added to the TRAC instruction set given in the TRAC Users' Manual [Deshpande 85]. DINST deletes an instruction tree and CINST creates an instruction tree. Both of these instructions are used repeatedly to create and delete instruction trees between the paging task and a selected processor from the group.

Thus, the three steps of the paging process are:

1. break up task;
2. perform page transfer;
3. recreate task.

The following sections describe each of these steps in detail.

## 3.2 Breaking Up Task

As mentioned in the previous section, the first step of the paging algorithm is breaking up the task into smaller tasks which will execute in parallel. Processors which do not have access to backup devices are not able to replace pages in their memory modules. Therefore, paging tasks are created in order to transfer pages in these cases. A paging task is defined as the smallest execution unit capable of performing page-in and page-out operations. In practice a paging task is a one processor task with access to a backup device [Canas 83]. Paging tasks will execute in parallel unless they share the same backup device, in which case they must compete for the use of the device.

A paging task must perform the paging operations for all processors which have their pages stored in the backup device and which do not have access to the device. A paging group is defined as a group of processors whose pages are stored in the same backup device, but only one of whose processors has access to the device. Therefore, a paging task must perform all paging operations for the paging group. A paging group can consist of only one processor, being the paging task itself, or it can consist of one paging task and several processors [Canas 83].

Before the task-wide instruction tree is broken up, a hardware bit called the processor resident monitor bit is saved so that the original task can be reconstructed when paging is finished. Two instructions, RPRM and SPRM, have been added to the TRAC instruction set [Deshpande 85] to manipulate this hardware bit. RPRM reads the bit and stores its value in the least significant bit position on top of the operand stack. SPRM sets the bit with the least significant bit of the value on top of the operand stack.

The page which caused the fault is determined by searching the page tables. The page to be swapped out is determined by executing an LRU instruction. In order to implement the paging algorithm, this instruction was added to the instruction set outlined in the TRAC Users' Manual [Deshpande 85]. This instruction determines which page in primary memory was least recently used (LRU) and pushes the virtual page number onto the operand stack. Page zero of each space type and the page buffers are exempt from paging out. The page type of the page selected to be replaced is the same as the page type on which the fault occurred.

A semaphore register is present at the memory module which is at the root of the task's instruction tree. This semaphore is used to count the number of processors which have completed their paging operations [Canas 83]. The semaphore is set to the number of processors. When the semaphore reaches a value of zero, all processors have completed their paging code. Two instructions have been added to the TRAC instruction set [Deshpande 85] to implement the semaphore. These two instructions are GSEM and SSEM. GSEM reads the semaphore and pushes the value on top of the operand stack. SSEM pops the value on top of operand stack into the semaphore register.

After the semaphore is set to the number of processors in the task, a DINST instruction is executed to delete the task-wide instruction tree. Each processor then begins executing the following page transfer algorithm independently.

## 3.3 Page Transfer

Each processor must test for six different cases and must handle them accordingly. Each processor will satisfy only one case and execute the code for that case. These six situations and how to manage them are described in this section. The first four situations exist when the fault occurs on a one-dimensional space and the last two cases are for a fault on a two-dimensional space. The task-wide instruction tree has been deactivated at this point, and each processor executes this code independently. The questions that each processor must ask to determine the paging code to execute are outlined in Figure 3-1.

### 3.3.1 Paging Task Owns Page

In this case, the fault has occurred on a one-dimensional space. The processor owns the page-out and is a paging task. The processor performs its own local page transfer. If the page-out is "dirty", it is moved to the page buffer and then written to the SMSM. The page-in is then transferred from the backup device to the page buffer and on to the former position of page-out in primary memory. Figure 3-2 is a detailed picture of how the operating system handles a local page transfer.

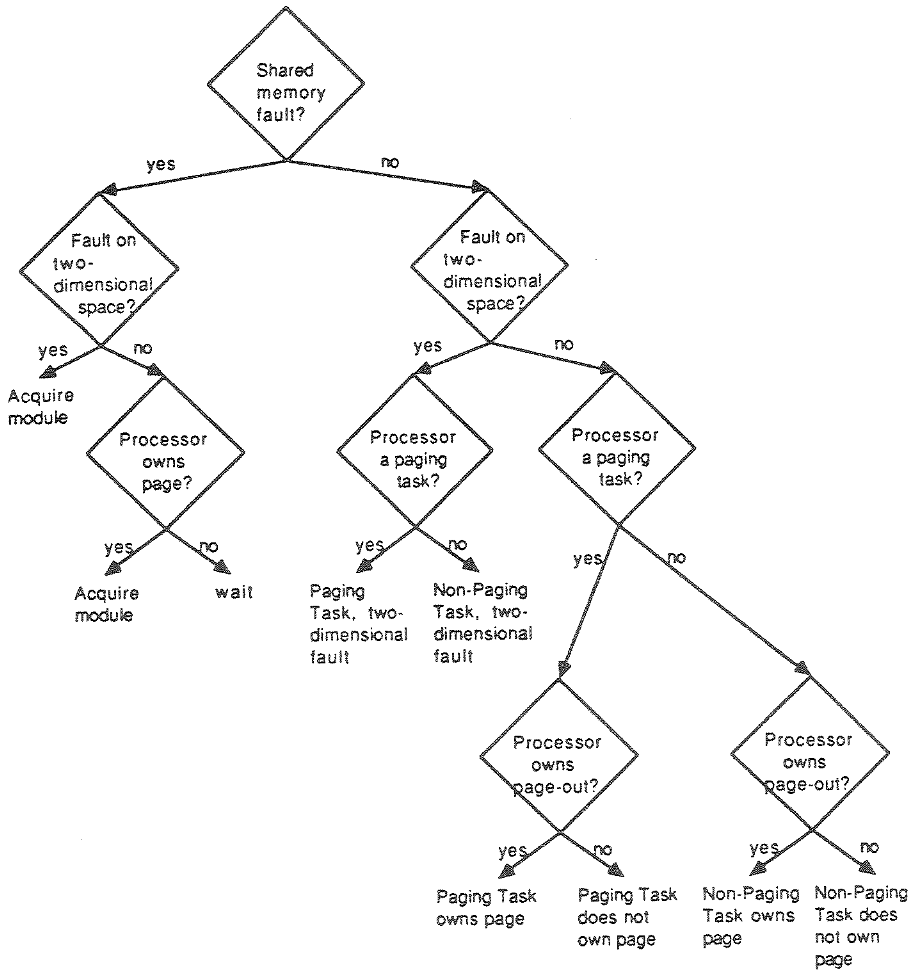


**Figure 3-1:** Decision Tree

The page table is updated and the paging transfer is complete.

### 3.3.2 Non-Paging Task Owns Page

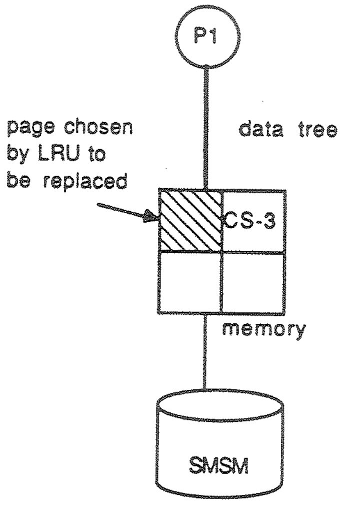
The fault in this case is also on a one-dimensional space. The processor again owns the page-out but in this case is not a paging task. Therefore, a non-local transfer must be done.

The page-out is moved to the page that is reserved in control space for non-local transfers, that is, CS-2. The processor waits for an instruction tree to be created between itself and the paging task. Once the tree is created, the two processors cooperate to execute the code for a non-local transfer.

In a non-local transfer, page-out is moved from CS-2 of the processor which owns the task to the page buffer, CS-3, of the paging task. From there, the page is moved to the SMSM. The page-in is read from the SMSM to CS-3 of the paging task. It is them moved to CS-2 of the non-paging task and onward to the position in primary memory from which page-out came. At this point, the processor has completed its page transfer, and the page table is updated.

Nonlocal transfers must go through two page buffers on TRAC because a byte cannot move from an SMSM to a primary memory other than the one to which it is attached in one clock cycle. Processor 1 in Figure 3-3 depicts a processor which is not a paging task but which owns the page-out.

1) Page fault occurs. Page chosen by LRU to be replaced is owned by processor 1. Processor 1 is a paging task.



2) Move page-out to CS-3.

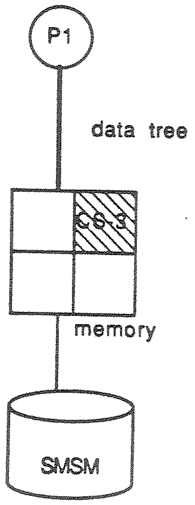
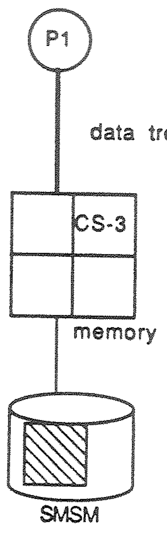


**Figure 3-2:** Local Page Transfer

3) Write page-out to backup device.



4) Read page-in from backup device.

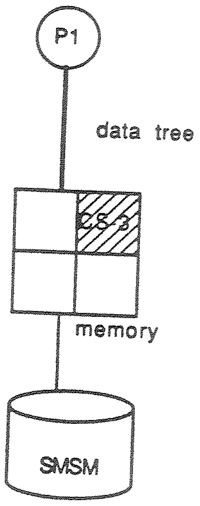


Figure 3-2, continued

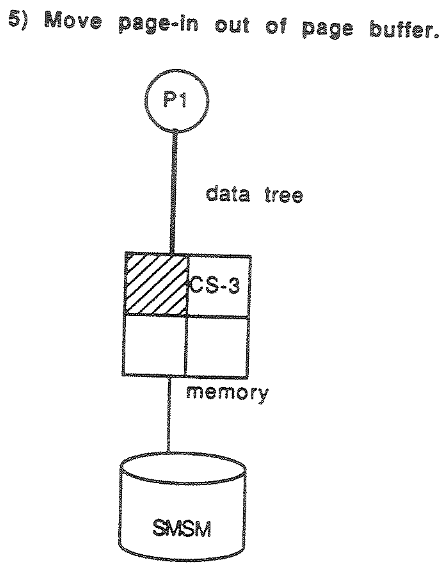


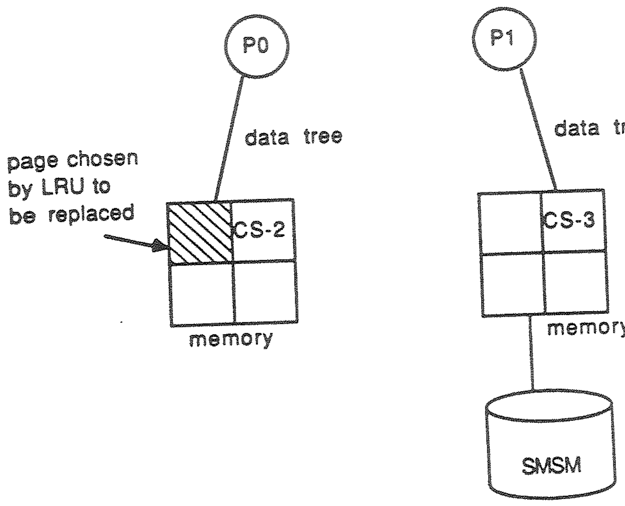
Figure 3-2, concluded

### 3.3.3 Paging Task of Owner of Page

Once again, the fault is on a one-dimensional space in this case. The processor does not own the page-out itself but is the paging task of the owner of the page-out. Therefore, this processor must perform a non-local transfer for the processor which does own the page.

In order to do the non-local transfer, this processor executes the CINST instruction to establish an instruction tree with the processor that owns the page. These two processors then execute the nonlocal transfer outlined in Section 3.3.2. Processor 0 in Figure 3-3 depicts this case. The two processors are cooperating to execute the page transfer.

1) Page fault occurs. Page chosen by LRU to be replaced is owned by processor 0. Processor 1 is paging task of processor 0.



2) Move page-out to page buffer

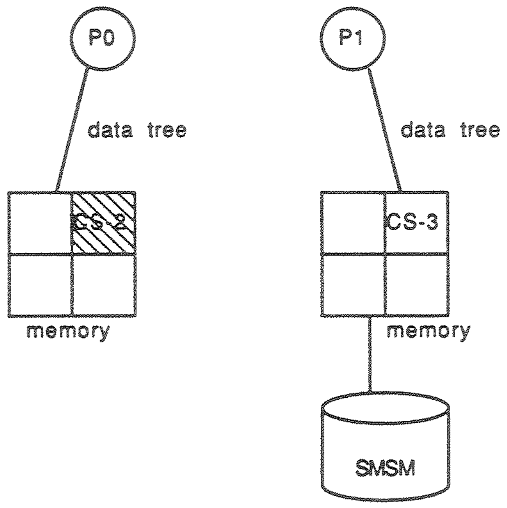
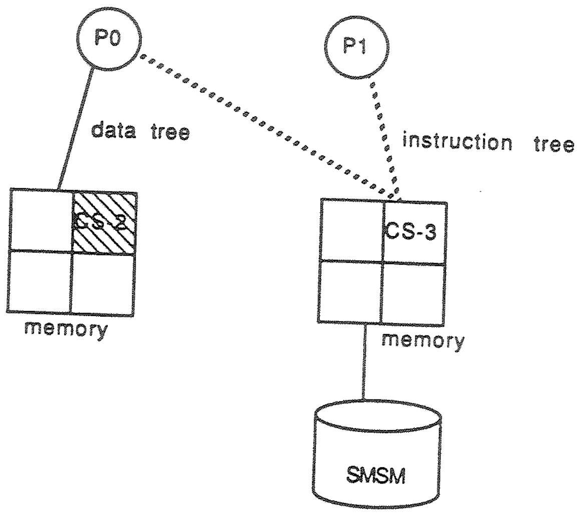


**Figure 3-3:** Nonlocal Page Transfer

3) Processor 1 establishes instruction tree between processor 0 and itself.



4) Move page from page buffer in processor 0 to page buffer in processor 1.

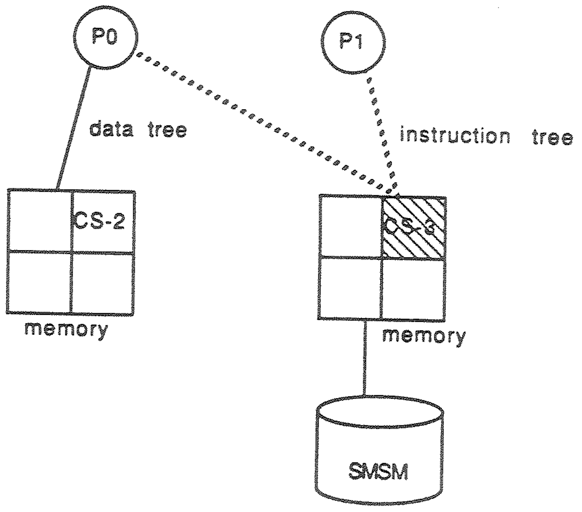
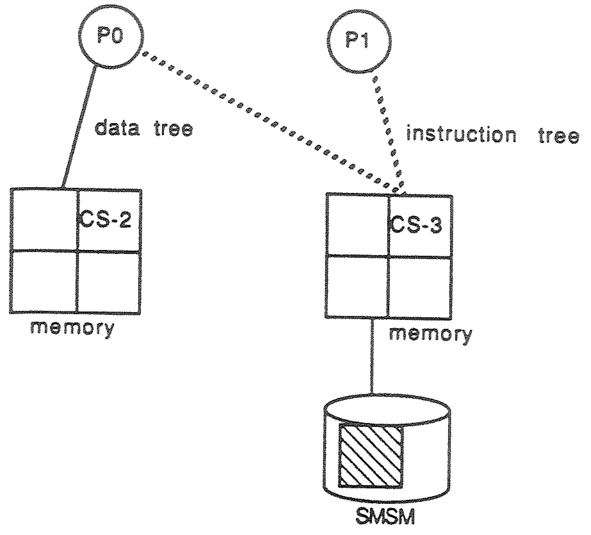


Figure 3-3, continued

5) Write page to back up device.



6) Read page-in from backup device to CS-3.

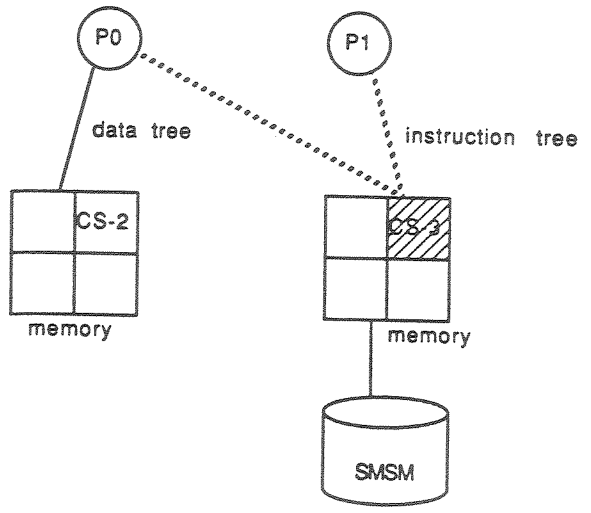
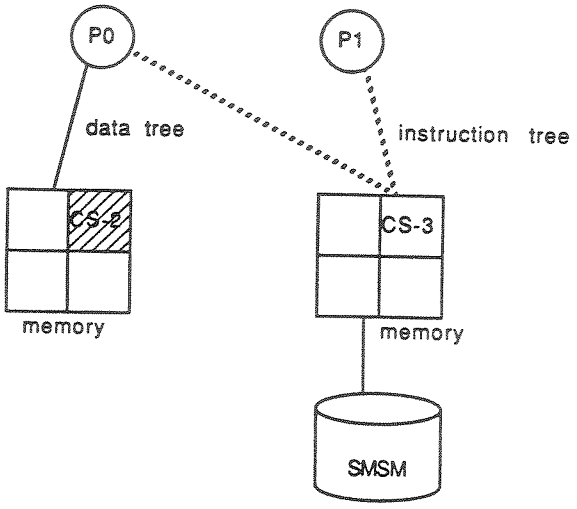


Figure 3-3, continued

7) Move page to CS-2.



8) Move page out of page buffer.

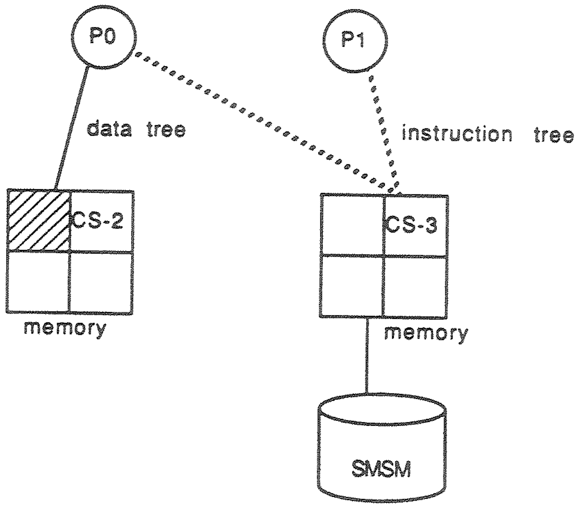


Figure 3-3, continued

9) Break up instruction tree between processors.

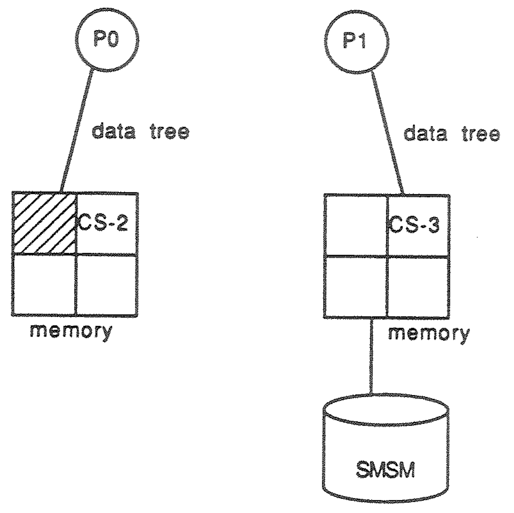


Figure 3-3, concluded

### 3.3.4 Processor Not Needed in Page Fault

This is the final case for a one dimensional fault. The processor does not own the page-out nor is it the paging task of the owner of the page-out. Therefore, this processor need not perform a page transfer.

### 3.3.5 Non-Paging Task for Two-Dimensional Fault

The last two page fault cases which must be dealt with occur on a two-dimensional memory space. In these cases, each processor will have a page transfer. In this case, the processor is not a paging task whereas in Section 3.3.6, the processor is its own paging task.

Since the processor is not a paging task in this case, it cannot perform its own paging operations. The processor moves the page-out to the reserved page in control space, that is, CS-2. This converts the two-dimensional space to a one-dimensional space. At this point, the processor waits for an instruction tree to be created between itself and the paging task. The paging task creates the instruction tree.

The two processors must then work together to execute a non-local transfer. The paging task moves page-out from CS-2 of the non-paging task to CS-3 of the paging task and then onward to the SMSM. Then, the paging task reads page-in from the SMSM to its CS-3 and on to CS-2 of the non-paging task. The page-in is then moved from CS-2 to the position in primary memory from which page-out came. The page table is updated and the instruction tree between the paging task and the owner of page-out is deleted. This ends the page transfer for this case. Figure 3-3 depicts how the operating system handles a nonlocal transfer. Processor 1 is the nonpaging task described above.

### 3.3.6 Paging Task for Two-Dimensional Fault

The final case occurs when a paging task has a fault on a two-dimensional space. The processor first performs a local transfer to manage its own page fault. Its page-out is moved to the page buffer and then written to the SMSM (if the page is "dirty") and its page-in is read to the page buffer from the SMSM. The page-in is then moved from the page buffer to its position in primary memory. The processor updates the page table for this page transfer. Figure 3-2 illustrates how a local transfer such as this is executed on TRAC.

The processor then must perform non-local transfers for each processor in its group as described in Section 3.3.5. The processor executes the CINST instruction to form an instruction tree between itself and a selected processor in its grcup.

As outlined in Section 3.3.5, the paging task and the non-paging task cooperate to form a non-local transfer and then the instruction tree between them is deleted with the DINST instruction. Processor 0 in Figure 3-3 depicts this case.

## 3.4 Task Resynchronization

After each processor handles its appropriate case as detailed in the previous section, it decrements the semaphore register. If the semaphore register is still greater than zero after decrementing, then the processor executes a WAIT instruction and waits for the other processors to finish their respective paging code. WAIT has been added to the instruction set [Deshpande 85] for this purpose. When the semaphore register reaches a value of zero, all paging operations are completed. At this time, the task-wide instruction tree must be recreated.

Before the task was broken up, a hardware bit called the processor resident monitor bit was saved so the original task could be reconstructed when paging is finished. Therefore, the processor resident monitor bit is now restored to its original value and the task-wide instruction tree is recreated with the CINST instruction. At this point, lockstep execution is resumed and control can be returned to the executing task [Canas 83].

## 3.5 Shared Memory Fault

Page faults can be of two types on TRAC. Either the page which caused the fault can be in a shared memory module currently not attached to the processor, or the page can be in a backup device. The previous sections described a fault when the page is in a backup device. This section describes a shared memory fault.

When a page fault occurs, the processor at the root of the instruction tree (the task head) will search the shared memory page table. The table contains an entry for every page stored in a shared memory module along with the description of its memory space and the color of the shared memory module in which it resides [Canas 83].

If the page causing the fault is found in the table, then the shared memory of the specified color must be acquired. If the page in question is not in the table, then the task head will proceed as explained previously [Canas 83].

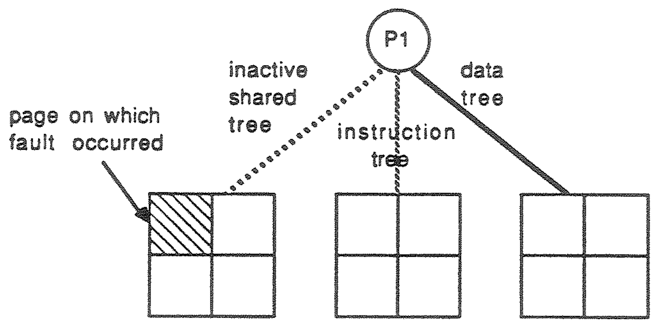
Pages with the same page number of different processors of the task may or may not be stored in shared memories of the same color. If they are not the same color, more than one memory will need to be acquired [Canas 83].

In order to avoid deadlock situations, the acquisition of shared modules is under the control of the Job Monitor (JM). This means that the processor which is the task head must send requests for the shared modules to the JM. To handle this situation, each processor of the task will be requested to send the color of the shared memory it needs to the task head. The task head will in turn send a request for all shared memory needed to the job monitor [Canas 83].

At this point the task head is waiting for a message from the JM instructing it to acquire the shared memories [Canas 83]. When the JM sends the acquire message, the task head will in turn send a message to the other processors of the task. Each processor will issue an acquire instruction for its shared memory module. The task is then resynchronized [Canas 83]. Figure 3-4 illustrates a processor before and after handling a shared memory fault.

After the task halts or executes 1000 instructions, whichever comes first, a time-out interrupt will occur and the shared memory module(s) will be released. The number 1000 was chosed ad hoc. The fewer instructions executed before releasing the memory, the greater the overhead is because the memory module may need to be acquired again. However, if too many instructions are executed before releasing, other tasks may be starved of the shared memory module.

1) Processor 1 has page in inactive shared tree.



2) Processor 1 acquires memory module.

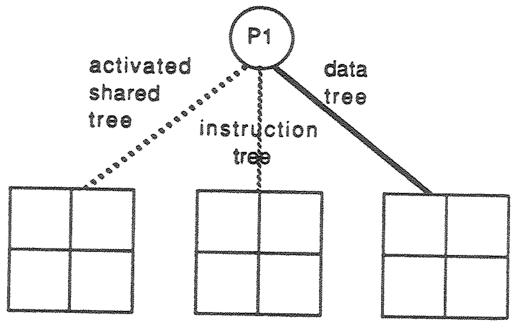


**Figure 3-4:** Shared Memory Fault

By acquiring the shared memory when a shared memory fault occurs and by releasing the shared memory after a specified time period, the operating system can virtualize the shared memory to writers of TRAC software.

# Chapter 4

# Results

The paging algorithm outlined in the previous chapter was coded in TRAC assembly language. The paging code was then executed on the TRAC simulator. This chapter describes the results of some test programs which have page faults handled by the paging code.

The purpose of the implementation is to show that this paging algorithm can indeed provide virtual memory and virtual shared memory on a reconfigurable network architecture. Therefore, in order to concentrate on the paging code rather than the test programs which generate page faults, the test code is very simple.

The results of the tests are given by telling the number of instructions that are executed by the operating system as it handles a particular type of page fault. These results are not exact numbers because time spent searching tables and determining space type may vary by a few instructions. However, these results are very close approximations.

The TRAC loader loads only page zero of each space in main memory when the program is initially loaded. Therefore, the first access to any page other than page zero will definitely cause a fault. All processors need a page reserved for paging in order to execute the paging algorithm. Each paging task processor needs a page CS-3 to execute the paging code. Each non-paging task processor needs page CS-2 to perform the paging code. These requirements for page buffers and the fact that page zero of each space is always in main memory necessitate a certain number of memory modules in order to have virtual memory on the current TRAC architecture. Because the LRU instruction always replaces a page with a page of the same space type (i.e., operand, data, program, or control), at least four pages must be available for a one processor task to have paging. These four pages are in addition to page buffers and page zero of each space. If **p** is the number of processors in a task, 2+(2***p**) is the least number of pages necessary for paging. This represents one page to be used for paging control space, one page to be used for paging program space, and a page for each processor for both operand and data space.

A two-processor task requires at least five memory modules for paging. A three-processor task requires seven memory modules, and a four-processor task requires nine memory modules. These numbers are given to illustrate that even with virtual memory, a certain number of resources are required in order to execute a program on a reconfigurable architecture. Without virtual memory, a one-processor task which only uses page zero of each space can execute with only one memory module.

In addition to the memory requirements, virtual memory also requires access to at least one backup device. Therefore, the necessary memory modules and backup devices must be available for a task to execute with virtual memory on TRAC.

## 4.1 One-dimensional Faults

The test program for a one-dimensional fault is a simple program which pushes register values onto the control stack enough times to cause a fault in the control space. The register values are then popped off the control stack into the appropriate register.

When this test program is executed on the TRAC simulator, the paging task takes approximately 120 instructions to execute its paging algorithm if no page must be written to the backup device. Two of these instructions are very time-consuming. One of the costly instructions reads an entire page from the backup device to the page buffer, and one of them moves a whole page out of the page buffer.

If a page must be written out to the backup device in order to have an available frame in main memory, the processor executes about 250 instructions. In this case, there are four very time-consuming instructions. The same two instructions as before read the page from the backup device, but two more instructions move page-out out of main memory. One instruction moves the page-out to a page buffer, and one instruction writes the page on the backup device.

The processors which do not own the page and which do not perform a page transfer still must search the page table to determine if they own the page which caused the fault. When a one-dimensional fault occurs, all processors in the task which are not the paging task execute approximately 100 instructions.

## 4.2 Two-dimensional Faults

The test program for a two-dimensional fault is a program which has a large vector. The vector occupies more than one page of memory in data space. Some accesses to the vector will therefore cause page faults. The test program simply assigns values to different positions in the vector, thus causing page faults.

When this test program is executed on the TRAC simulator, each processor executes approximately 245 instructions to perform a page transfer, if no page must be written to the backup device. Once again, two of these instructions are very costly. One expensive instruction reads page-in from the backup device and into the page buffer, and the other moves page-in out of the page buffer.

The paging operation takes a processor about 380 instructions when a page must be written to the SMSM. In this case, four instructions take a great deal of time. The two instructions which read the page in are the same, plus an instruction which moves page-out to the page buffer and an instruction which writes the page buffer onto the backup device.

For a two-dimensional fault, some of the processors may share a backup device. Therefore, the two processors must compete for use of the backup device. Only one processor at a time can own the backup device. The other processors must wait. When two processors share a backup device, the required time to handle the page fault can almost double since only one processor at a time can execute its page transfer. In the best case, no processors share an SMSM.

## 4.3 Shared Memory Fault

A shared memory fault occurs when a page is accessed that is in a shared tree not currently owned by the processor. This type of fault can also occur on a one-dimensional space or a two-dimensional space.

The test program for a one-dimensional fault accesses page two of program space, which is in a shared memory tree. A programmer might want to keep a long routine which is used by many tasks in a shared memory module so that the routine is not duplicated in main memory.

When the fault occurs on a one-dimensional space, the task need not be broken up. The fault requires about 60 instructions to acquire the memory module. None of these instructions is especially time-consuming either, although there may be some delay if the shared memory module is currently owned by another task. However, this delay would occur whether shared memory is virtualized or not.

The test program for a two-dimensional fault in shared memory has a large vector which is shared by two tasks. A task with a two-dimensional fault must be broken up so that more than one memory can be acquired. Each processor executes approximately 60 instructions in order to acquire the memory module. None of these instructions takes very long to execute, but as with the one-dimensional space, a processor may have to wait for its memory module to be released before it can acquire the memory.

The interrupt routine which releases the shared memory modules after a specified amount of time is 30 instructions. This routine has no time-consuming instructions and will never have delays because it is simply releasing a shared memory.

## 4.4 Summary of Results

This chapter has given results of test programs run on the TRAC simulator. These test programs generate page faults and shared memory faults. The results are given by telling how many instructions are executed by the operating system to handle the fault. The paging code for a page fault is very time-consuming. However, the paging code for a shared memory fault has few instructions and does not take much more time than an explicit request by a TRAC program to acquire the memory module. The interrupt routine which releases the shared memory does not require much more time than an explicit release instruction. Virtualized shared memory has the advantage of not requiring the TRAC programmer to know the details of shared memory.

# Chapter 5

# Conclusion

The purpose of the paging algorithm outlined in this thesis is to provide virtual memory and virtual shared memory on a reconfigurable network architecture such as TRAC. TRAC was originally formulated as a high capacity scientific computer which would provide efficient application and formulation [Sejnowski 80]. Because one of the main goals of a parallel architecture is to optimize performance, one of the main goals of the operating system should also be to optimize performance. This goal can be accomplished in the operating system by optimizing use of the resources (processors, memories, I/O resources, backup devices, etc.) and by having as much parallelism as possible.

One way the paging algorithm optimizes the use of resources is by deleting the task-wide instruction tree and by creating paging groups. This allows all of the processors which have access to a backup device to perform their page transfers in parallel.

However, two situations exist which decrease the amount of parallelism that can be achieved by the creation of paging tasks. One such situation occurs when blockage in the network prevents a processor from being its own paging task and performing its own page transfers. The blockage prevents the creation of a shared tree between a processor and the backup device, so a more time-consuming nonlocal transfer must be performed.

The second situation occurs when one or more paging tasks shares a backup device. These processors will have to compete for the memory. Only one processor at a time can acquire the memory with the backup device and perform its page transfers. This problem may be solved by increasing the number of SMSM's in the configuration, so that each processor has exclusive access to one SMSM. However, sometimes this is not physically or economically feasible.

This paging algorithm is most efficient when each processor in the task is a paging task and each processor has exclusive access to a backup device. This will mean that each processor needs to perform only one page transfer, and that page transfer will be local.

Even in the optimum case, however, the algorithm takes so much time execute a page transfer that this algorithm would only be feasible if it were implemented **efficiently** in hardware.

One way to optimize performance of this paging routine in hardware is to use pipeline processing. Pipeline is a technique of decomposing a sequential process into subprocesses with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments [Mano82]. A pipeline can be visualized as a collection of processing segments through which information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data has passed through all segments. In pipelines, several computations can be in progress at once [Mano82].

In the paging algorithm, an obvious use of pipelines occurs when writing the page-out to the backup device. One processing segment can move the page-out to the page buffer and one processing segment can write the page buffer to the SMSM. Figure 5-1 depicts this use of pipelines.

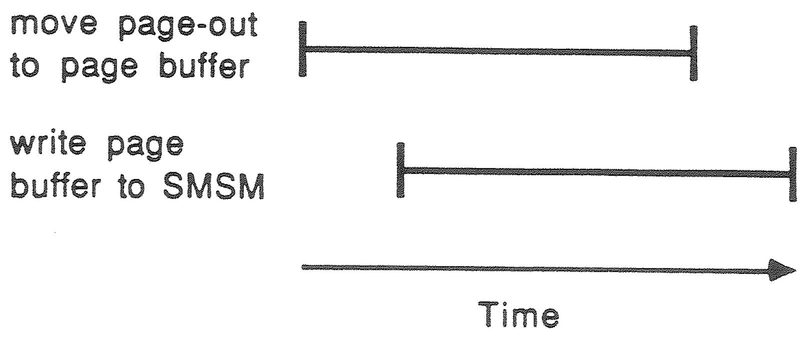


**Figure 5-1:** Pipeline Processing During Write

In addition, pipelines could be used to read the page-in from the backup device. As bytes of page-in are read from the backup device into the page buffer, some bytes can be moved out of the page buffer into their final page in main memory. Figure 5-2 illustrates pipeline processing on the read operation.

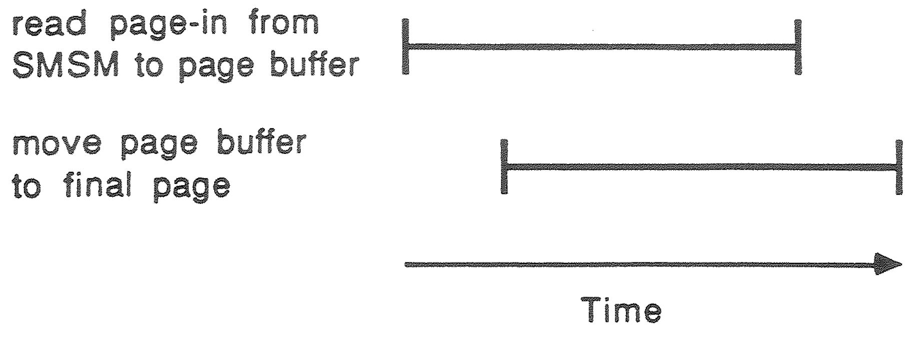

read page-in from SMSM to page buffer
move page buffer to final page
Time

**Figure 5-2:** Pipeline Processing During Read

Because these page transfer operations are the bulk of the paging code, pipelining these operations should almost half the time needed to handle a page fault.

Virtual shared memory is another operating sytem issue discussed and tested in this thesis. The results obtained from testing this idea showed it took relatively few TRAC assembly language instructions to implement, and none of the instructions was especially time-consuming. Once again, one might consider putting this operating system routine in hardware to make it more efficient; however, it seems feasible in software as well. Shared memory provides a way for processors to share large segments of memory. By virtualizing shared memory, writers of TRAC programs can be oblivious to the hardware configuration and yet write powerful software. Virtual shared memory is a feature unique to TRAC and seems to be a worthwhile characteristic.

This thesis has described the TRAC hardware to the extent necessary to understand the paging algorithm. Next, the paging algorithm designed by Daniel Canas for a reconfigurable network architecture such as TRAC was described. This algorithm was resolved and implemented on the TRAC simulator and results of the tests were given. Finally, an analysis of these results was given along with possible improvements.

The purpose of this thesis is to prove the concept of virtual memory and virtual shared memory on TRAC. Parallel computing gives rise to many interesting design issues in hardware as well as in software. Because of the importance of virtual memory in all systems, it will continue to be an important issue in parallel computing for many years.

# Bibliography

| | |
|---|---|
| [Canas 83] | Canas, D. Operating Systems for Reconfigurable Network Architectured Systems: The Node Kernel. 1983. |
| [De Groot 81] | De Groot, D., W. Hunt, Jr., J. C. Browne. Virtual Memory Management for Network Architectured Varistructured Computers. 1981. |
| [Deshpande 82] | Deshpande, S. Interrupt Structure for TRAC. 1982. |
| [Deshpande 85] | Deshpande, S., M. Sejnowski, et al. TRAC Users' Manual. 1985. |
| [Mano 82] | Mano, M. M. *Computer System Architecture.* Prentice Hall, 1982. |
| [Sejnowski 80] | Sejnowski, M. C., G. J. Lipovski, et al. An Overwiew of the Texas Reconfigurable Array Computer. *NCC* :631-641, 1980. |

## VITA

Eileen Archer Allison was born in Amarillo, Texas on July 17,1962, the daughter of Dr. and Mrs. Richard K. Archer. She graduated from The University of Texas at Austin with a Bachelor of Arts in Computer Sciences in December, 1984. In January 1985, she entered The Graduate School of The University of Texas at Austin.

Permanent Address: 1406 Windsor #202
Austin, Texas 78703

This thesis was typed by the author.