Chapter 4
Implementation of the TIGRE Machine

This chapter discusses the details of implementation of the TIGRE abstract machine. Section 4.1 describes the abstract machine and its assembly language. Section 4.2 describes the mapping of the TIGRE abstract machine onto different hardware platforms, including assembly language implementations for the VAX and the MIPS R2000 architectures. Section 4.3 describes the implementation of the core Turner Set combinators in TIGRE assembly language. Section 4.4 describes minimum TIGRE software support requirements.

4.1. THE TIGRE ABSTRACT MACHINE

TIGRE is defined as an abstract machine having its own assembly language. This abstract machine has an instruction set which is designed to implement efficiently the primitive operations for performing graph rewriting and graph evaluation. This chapter presents a development of TIGRE implementations in C, VAX assembly language, and MIPS R2000 assembly language. The development is shown starting from an abstract machine, through TIGRE assembly language, and then to a mapping onto real platforms at both the hardware and assembly language level.

4.1.1. Hardware Definition

Figure 4-1 shows a block diagram of the TIGRE abstract machine. As a minimum, TIGRE requires a processing unit, a scratchpad register storage space, a spine stack/subroutine return stack, memory for holding combinator definitions, and heap memory for holding the graph nodes.

TIGRE needs three memory spaces. The first memory space contains the spine stack, which is used to save pointers to the nodes in

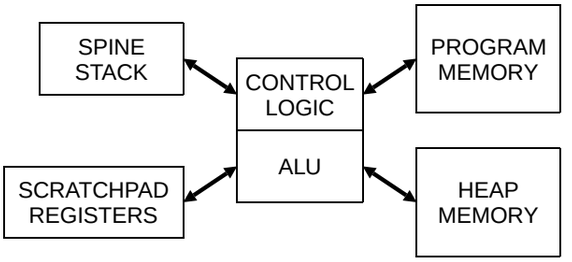

Figure 4-1. A block diagram of the TIGRE abstract machine.

the program graph that have been visited during stack unwinding. The second memory space contains the definitions for combinators (written in TIGRE assembly language). The third space contains the garbage-collected heap memory that holds the program graph. There is no prohibition of two or three of the memory spaces sharing the same hardware resources, but they are treated as separate at the abstract level to emphasize their different uses.

The processing unit must provide an ALU for arithmetic and logical operations, as well as control logic to fetch, decode, and execute instructions. The primary operation of the control logic is as a threading engine. In other words, the control logic will thread through a series of pointers representing the left spine of a graph while pushing node addresses onto the spine stack. Once a pointer into combinator memory is found, the control logic switches from threading in the heap memory space to execution in the combinator program memory space. A small amount of internal storage of intermediate values is required, which is designated as the scratchpad register space. One particularly important register is the interpretive pointer (`ip`), which is frequently called the program counter on conventional machines.

4.1.2. TIGRE Assembly Language

The threaded execution characteristics of the TIGRE abstract machine are very much like the execution characteristics for the abstract machine used by the Forth programming language (Kogge 1982, Moore 1980). Similar to a Forth abstract machine, TIGRE has two modes of operation: threading and execution of primitives. Instead of expression evaluation stack manipulation primitives used by the Forth abstract

machine, TIGRE uses graph reduction primitives which operate on the spine stack.

Forth procedure definitions primarily consist of procedure calls and stack manipulation primitives. The stack manipulation primitives *are* the assembly language of the Forth abstract machine. In a similar manner, TIGRE combinator graphs consist of pointers and combinators. The combinators are primitives of the graph reduction process (corresponding to the assembly language of the graph interpreter). A problem with both Forth and TIGRE is that the stack-oriented actions of the primitives are not directly supported by most CPU architectures. In order to solve the problem of specifying the operation of TIGRE combinators in a machine-independent way, we shall define a lower level interface than the combinator, called TIGRE assembly language.

The purpose of TIGRE assembly language is to define a low level, abstract implementation of combinators for the TIGRE abstract machine. This assembly language must have the property of concisely defining the actions required to process a particular combinator, together with the property of efficient mapping onto a variety of commercially available hardware platforms. The assembly language is defined in terms of resource names and abstract machine instructions as follows. The syntax used is a one- or two-operand register/memory based instruction format.

The resources controlled by TIGRE assembly language include the interpretive pointer `ip`, computational scratchpad registers, the spine stack, and heap memory.

- `ip` – The interpretive pointer. This register directs the threading through the graph. It must be set to point to the next node to thread through before the "thread" instruction is executed.
- `L`*n* – The left child of the node pointed to by the n^{th} element from the top of the spine stack.
- `R`*n* – The right child of the node pointed to by the n^{th} element from the top of the spine stack.
- `L+`, `R+` – These are equivalent to `L0` and `R0`, except that the spine stack pointer is auto-incremented (*i.e.* "auto-popped") after the access.
- `temp`*n* – Scratchpad node registers. The "allocate" instruction deposits addresses of newly allocated nodes into these registers.

- `Ltemp`*n*, `Rtemp`*n* – The left/right child of the node pointed to by scratchpad node register `temp`*n*.
- `result` – A temporary result register for returning the value of strict computations.
- `scratch`*n* – Other scratchpad registers needed for holding intermediate results.

There are three classes of operands:

- *i* – Immediate data.
- *n* – Node reference.
- *p* – Pointer.

The core of the instruction set is as follows:

- `allocate` *i* – Allocate *i* new graph nodes, depositing pointers to them in `temp0, temp1, ..., temp`*(i-1)*.
- `mov` *n, p* – Move the contents of the operand *n* to pointer *p*.
- `pop` *i* – Pop *i* elements off of the spine stack.
- `push` *p* – Push the pointer *p* on top of the spine stack.
- `top` *p* – Replace the top of the spine stack with the pointer *p*.
- `thread` – Thread through the node pointed to by the `ip` register.
- `eval` *n* – Recursively reduce the graph rooted at the node *n*, leaving the result in the `result` register, using the spine stack to save a return address.
- `return` – Thread and pop through the top of the spine stack, restarting execution just beyond the most recent evaluation.

In two-operand instructions, the source operand is on the left, and the destination operand is on the right. In addition, there are the usual complement of instructions for conditional execution, arithmetic, and other strict operations that one would expect to be available in all assembly languages.

As an example of how graph reduction operations can be expressed in TIGRE assembly language, consider the combinator **S'**, which is defined as:

$S' \equiv \lambda c.\ \lambda f.\ \lambda g.\ \lambda x.\ (c\ (f\ x))\ (g\ x)$

or, equivalently, as

$S'\ c\ f\ g\ x \rightarrow (c\ (f\ x))\ (g\ x)$

Figure 4-2 shows the definition of **S'** graphically as well.

This code can be straightforwardly compiled into TIGRE assembler code. A simple compilation process yields the following combinator rule for **S'**:

```
; allocate heap cells
allocate 3          ; temp0, temp1, and temp2 get
                    ; the pointers to new cells.
; write values to heap cells
mov R1, Ltemp0      ; (f x)
mov R3, Rtemp0
mov R0, Ltemp1      ; (c (f x))
mov temp0, Rtemp1
```

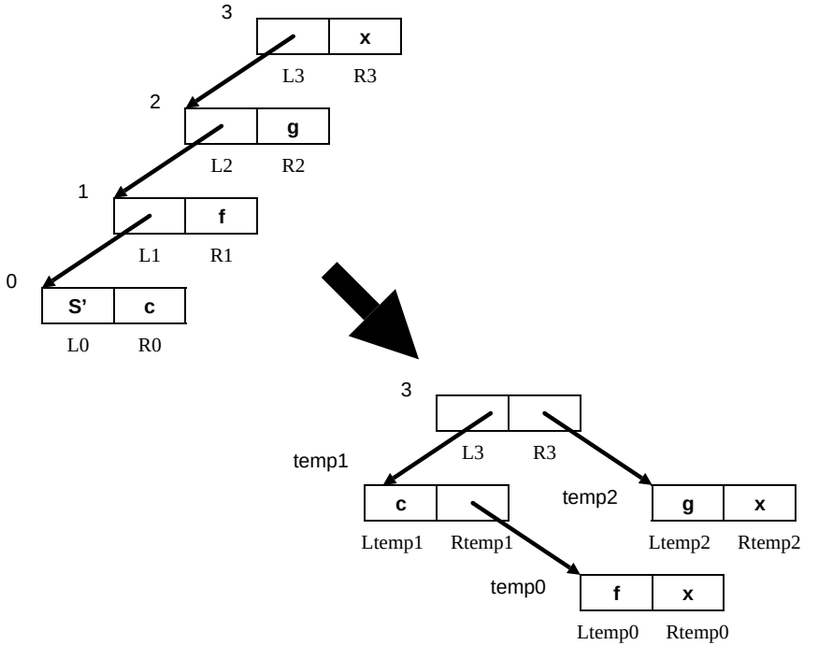

Figure 4-2. The **S'** combinator. $S'\ c\ f\ g\ x \rightarrow (c\ (f\ x))\ (g\ x)$

```
mov R2, Ltemp2       ; (g x)
mov R3, Rtemp2
; rewrite root node of subgraph
mov temp1, L3        ; (c (f x)) (g x)
mov temp2, R3
; discard spine stack pointers to nodes 0, 1, 2
pop 3
; restart threading from node temp1
mov temp1, ip
thread
```

First, three new heap cells are allocated, with pointers to them left in `temp0`, `temp1`, and `temp2`. Next, the newly allocated heap cells are written with values taken from the input parameters to the combinator. The first instruction deposits the contents of the right-hand side of the node pointed to by the second-to-top spine stack element (performing a double indirect fetch through the spine stack) into the left-hand side of the `temp0` node just allocated on the heap. The other operations are similar. Note that the notation `temp0` refers to the address of the `temp0` node, while `Ltemp0` and `Rtemp0` refer to the contents of the left- and right-hand sides of node `temp0`.

Once the newly allocated heap cells have been written with appropriate values, the root node of the subtree undergoing the **S'** graph transformation is rewritten to point to `temp1` and `temp2`. Since it is not easily decidable whether the other nodes participating in the **S'** reduction are shared by other portions of the program graph, they are simply abandoned.

This code is correct and easily generated. However, it is sub-optimal in a number of respects. For instance, redundant fetches to the spine stack and heap memory are performed. Ideally, such redundancies would be eliminated by the standard compiler or high level language available on the target platform for TIGRE. Unfortunately, it is difficult for a compiler or assembler to improve the performance of this code sequence because of all the pointer operations being performed. For example, conventional compilers given the code sequence for **S'** cannot be absolutely sure that the value represented by `R3` is not changed by a store into `Rtemp1`. In order to prove that, it would have to understand the global graph rewriting operations and spine traversals performed by the program. Experiments with the MIPS R2000 optimizing C compiler and assembler show that essentially no optimization takes place, and that most load delay slots are filled with NOP instructions.

4.1.3. A TIGRE Compiler

Rather than trust the efficiency of compilation to external compilers of varying degrees of optimization, we have written an optimizing TIGRE compiler (Lee & Koopman 1989). The purpose of the compiler is to generate TIGRE assembly language source code when given the definition of a combinator. This compiler not only generates code such as that given for **S'** above, but also performs various optimizations to produce code that is of the same quality as hand-written assembler code for C, VAX assembly language, and MIPS R2000 assembly language. The compiler performs as many optimizations as possible at the TIGRE assembly code level, then performs a simple mapping of TIGRE assembler instructions into directly corresponding sequences of target machine instructions. Optimizations include: reusing values in registers to eliminate redundant memory accesses, consuming spine stack elements in order from the top to allow on-the-fly popping of the stack for machines with post-increment addressing modes, grouping of writes to memory for better performance on machines with wide memory write buffers, and reusing values in registers to eliminate redundant memory accesses.

4.2. MAPPING OF TIGRE ONTO VARIOUS EXECUTION MODELS

Since TIGRE is an abstract machine definition, it must be emulated on whatever hardware is actually used. Therefore, it is important that TIGRE be designed to efficiently map onto a variety of execution platforms. The following sections describe the mapping of TIGRE into

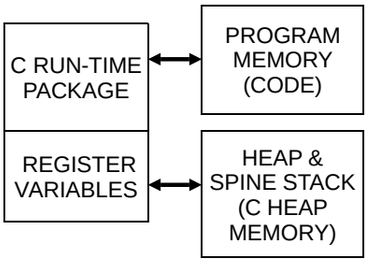

Figure 4-3. Mapping of the TIGRE abstract machine onto C.

a portable C implementation as well as assembly languages for the VAX and the MIPS R2000 processors.

4.2.1. Mapping of TIGRE Onto the C Execution Model

TIGRE can be mapped into C in a straightforward manner, but with some inherent inefficiency. In order to keep C implementations portable, TIGRE must use an interpretive loop with one-bit tag checking of cell values when performing spine unwinding.

Figure 4-3 shows the mapping of TIGRE onto a C abstract execution engine. The spine stack is a vector of 32-bit nodes allocated from the C heap space. TIGRE heap memory is likewise allocated from the C heap space (and is managed with garbage collection by TIGRE). Combinator memory corresponds to the compiled C program, which contains the combinator definitions. The scratchpad registers are implemented using register variables.

As spine nodes are unwound, addresses to the nodes are placed onto a software-managed stack. When a combinator node is found, a `switch` statement (case statement) is executed to jump to the appropriate combinator code. Many C compilers implement the `switch` statement by using a jump table, so the case analysis is reasonably efficient. The C code for **S'** is:

```
case DO_SPRIME:
   New_Node(3);
   Use_Me;
   Ltemp1 = ip = Rme ;
   Rtemp1.child = TARGET(temp0);
   Use_Parent;
   Ltemp0 = Rparent;
   Pop_Spine(2);
   Use_Me;
   Ltemp2 = Rme;
   Use_Parent;
   Rtemp2 = Rtemp0 = Rparent;
   Lparent.child = TARGET(temp1);
   Rparent.child = TARGET(temp2);
   *(spine_ptr) = temp1 + 1 ;
   continue;
```

Several macro definitions are used in C to make the code readable. `New_Node()` is a macro that expands into a heap node allocation process with a conditional call to the garbage collector in case the heap memory is exhausted. `Use_Me` is a macro that caches the top-of-stack element from the spine stack into a register variable for use with later references

to the R0 or L0 cells. The TIGRE compiler automatically invokes this macro to perform the caching just before an R0 or L0 reference is needed. Use_Parent is similar to Use_Me, except it caches the value of the second from top stack element from the spine stack for later use in referencing R1 and L1. Because many machines have a limited number of registers, the TIGRE compiler structures C code in such a way as to access only the top two stack elements at any one time, popping the stack as arguments to the combinator are consumed. Auto-incrementing access to the stack pointer is not used, because many machines do not support this addressing mode in hardware, and may therefore execute code more slowly when using post-incrementing address modes because of extra instructions generated by the C compiler.

Rme and Lme in the C code correspond to R0 and L0. Rparent and Lparent in TIGRE assembly language similarly correspond to R1 and L1. The "TARGET" notation generates a reference to the address of a heap node. The ".child" notation is used to satisfy type checking requirements of the C compiler, since heap cells may contain pointers, combinator values, or integer constants.

The C code generated by the TIGRE compiler is nearly identical to hand-tuned C code. The hand-tuned C code was developed by iteratively examining the assembly language output of a VAX C compiler and changing the C source code to improve efficiency. The result is that, on a VAX, the C code generated for a particular combinator is as close as is possible to the VAX assembler expression of that combinator within the limit of the capabilities of the C language. Unfortunately, C is unable to explicitly express indirect jumps, "light-weight" subroutine calls (that do not save stack frames), direct subroutine return stack manipulations, and other convenient implementation mechanisms for TIGRE threading operations. For this reason, C implementations of TIGRE typically run two or three times slower than assembly language implementations on the same hardware.

4.2.2. Mapping of TIGRE Assembly Language Onto a VAX

Since the VAX has a lightweight subroutine call instruction (jsb), TIGRE can map very efficiently onto the VAX architecture. As shown in Figure 3-8, each heap node consists of a triple of cells, with the first cell containing a VAX jsb instruction. The VAX then executes self-modifying graph code, using the hardware-supported stack pointer register as the spine stack pointer. Jumps to combinators are accomplished by simply having a pointer to the combinator code resident in a heap cell.

Figure 4-4 shows how the TIGRE abstract machine maps onto a VAX 8800. The spine stack and heap memory both reside in main program memory. The combinator memory is a sequence of VAX assembly instructions that resides in a different memory segment (at least under the UNIX operating system), but shares the same physical memory. Since the VAX 8800 has a single cache memory, all three TIGRE memory spaces share the same cache. The VAX hardware registers are used as the TIGRE scratchpad registers.

The following is optimized VAX assembly code for the **S'** combinator, commented with the corresponding TIGRE assembly code:

```
# r3 is temp0, r4 is temp1, r5 is temp2, r9 is ip
movl (r3), r0         # cache pre-touch
movl *(sp)+, r9       # mov R+, ip /* R+ pops R0 */
movl r9, (r4)         # mov ip, Ltemp1
movab -2(r3), 4(r4)   # mov temp0, Rtemp1
movl *(sp)+, (r3)     # mov R1, Ltemp0
movl *(sp), (r5)      # mov R2, Ltemp2
movl 4(sp), r7        # mov R3, Rtemp2
movl (r7), r8
movl r8, 4(r5)
movl r8, 4(r3)        # mov R3, Rtemp0
movab -2(r4), -4(r7) # mov temp2, L3
movab -2(r5), (r7)    # mov temp2, R3
movab 4(r3), (sp)     # top Rtemp1
jmp (r9)              # thread
```

In the VAX 8800 code, the first instruction performs a dummy read to accomplish cache pre-touch, which partially defeats the write-no-allocate behavior of the cache memory on that machine (the reason for this is discussed in Chapter 6). The `movl` instructions use the double-

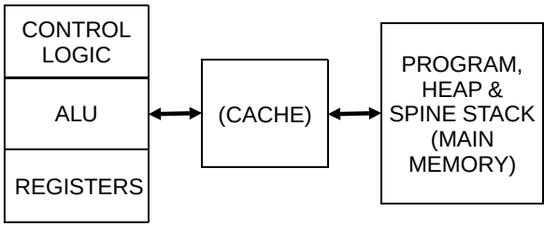

Figure 4-4. Mapping of the TIGRE abstract machine onto a VAX 8800.

indirect addressing capability of the VAX architecture to implement efficiently many TIGRE assembler instructions with a one-to-one correspondence.

The `movab` instructions are used to write pointers to new nodes into heap cells. Since the VAX uses a subroutine-threaded interpretation of TIGRE, two bytes (the size of a `jsb` opcode) must be subtracted from each address to point to the `jsb` in front of each heap node. The `jsb` instruction places the address of the right-hand side of a node onto the spine stack, so an offset of -4 is used to access the left-hand side of a heap node. Finally, the thread instruction is simply a jump to the `jsb` opcode of the next heap node to be unwound onto the spine stack.

It should be noted that writing into the instruction stream is not necessarily safe on a high-end VAX architecture. In the case of VAX 8800 code, a dummy write instruction must be added to the end of some combinators (**S'** is not one of them) in order to flush a write buffer, forcing updating of resident cache elements, which in turn forces updating of the instruction prefetch buffer. However, with this one programming note, self-modifying TIGRE code runs perfectly on the VAX 8800. A non-self-modifying version of TIGRE on a VAX can be designed which uses an interpretive loop to perform stack unwinding instead of subroutine call instructions, but executes at slower speed.

4.2.3. Mapping of TIGRE Assembly Language Onto a MIPS R2000

The MIPS R2000 processor does not support a subroutine call instruction. Furthermore, the R2000 has split instruction and data caches, with no updates of the instruction cache for bus writes. This means that

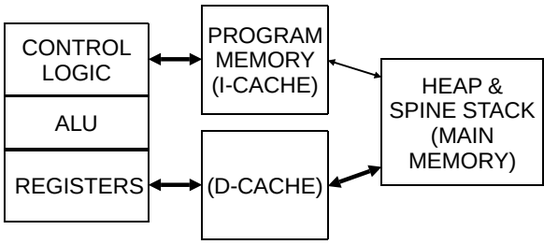

Figure 4-5. Mapping of the TIGRE abstract machine onto a MIPS

self-modifying code is not practical on an R2000. So, the R2000 implementation of TIGRE uses a five-instruction interpretive loop to perform stack unwinding, and does a jump to the combinator code when the highest order bit (which is a one-bit tag) of a cell value is set.

Figure 4-5 shows how the TIGRE abstract machine maps onto a MIPS R2000. The combinator memory resides in the instruction cache (the TIGRE kernel is small enough to fit entirely into cache), while the stack memory and graph memory reside in a combination of the data cache and main memory. The code for the **S'** combinator below is scheduled to eliminate pipeline breaks caused by the one-clock load delay slot of the R2000. Comments are given in C code instead of TIGRE assembler to make the buffering of pointers to the top and second spine elements more obvious.

```
    # $16 = spine stack pointer
    # $18 = temp0,  $19 = temp1,  $20 = temp2
    # $21 = buffer for top of spine stack
    # $17 = buffer for second on spine stack
$DO_SPRIME:
   NEWNODE3 ;           # allocate 3 cells

   lw     $21, 0($16)   # Use_Me ;
   lw     $17, 4($16)   # Use_Parent;
   lw     $10, 0($21)   # Ltemp1  = ip  = Rme ;
   sw     $19, 4($18)   # Rtemp1.child=TARGET(temp2);
   sw     $10, 0($18)
   lw     $8, 0($17)    # Ltemp2 = Rparent ;
   addu   $16, $16, 8   # Pop_Spine(2) ;
   sw     $8, 0($19)
   lw     $21, 0($16)   # Use_Me ;
   lw     $17, 4($16)   # Use_Parent;
   lw     $8, 0($21)    # Ltemp3 = Rme ;
   sw     $18, -4($17)  # Lparent.child=TARGET(temp1);
   lw     $9, 0($17)    # Rtemp2 = Rtemp3 = Rparent ;
   sw     $8, 0($20)
   sw     $9, 4($19)
   sw     $9, 4($20)
   sw     $20, 0($17)   # Rparent.child=TARGET(temp3);
   addu   $8, $18, 4    # *(temp_spine) = temp1+1;
   sw     $8, 0($16)    # moved to branch delay slot
   b      $THREAD            # continue;
```

With the R2000 assembly language, it becomes apparent that the combinator definition for **S'** is simply a long sequence of memory loads and stores. This corresponds closely to the notion of performing a graph rewrite, which is simply copying values between memory locations.

4.2.4. Translation to Other Architectures

The availability of a reasonably quick subroutine call instruction on many architectures makes the TIGRE technique applicable, in theory, to most computers. In practice, there are issues having to do with modifications of the instruction stream that make the approach difficult to implement on some machines. It should be emphasized, however, that these problems are the result of inappropriate (for the current application) tradeoffs in system design, not the result of any inherent limitation of truly general-purpose CPUs. Inasmuch as graph reduction is a self-modifying process, it is not surprising that a highly efficient graph reduction implementation makes use of self-modifying techniques. One could go as far as to say that the extent to which graph reducers use self-modifying code techniques reflects the extent to which they efficiently implement the computation being performed.

4.3. TIGRE ASSEMBLER DEFINITIONS OF COMBINATORS

The previous section explored the mapping of TIGRE onto a high level language, a Complex Instruction Set Computer (CISC) architecture, and a Reduced Instruction Set Computer (RISC) architecture. The following subsections describe the different classes of combinators needed for implementing TIGRE efficiently, and give example combinator implementations in TIGRE assembly language.

4.3.1. Non-Strict Combinators

The Turner Set of combinators includes two types of non-strict combinators: projection combinators and simple graph-rewriting combinators. Each type has a different implementation strategy.

4.3.1.1. 1-Projection Combinators

The 1-projection combinators, **I** and **K**, are combinators that jump to the right-hand side of the heap node referred to by the topmost element on the spine stack, discarding one or more other references to heap nodes on the spine stack. They have the general form:

W a b c d ... → a

In TIGRE, these 1-projection combinators are implemented by jumping to the subgraph pointed to by **a**, while popping references to

the other inputs **b**, **c**, **d**, and so on. No graph rewriting is performed, but rather a simple "fall-through" flow of control operation is performed.

For the **I** combinator, this strategy results in remarkably simple code:

```
mov R0, ip
pop(1)
thread
```

The simplicity of this code results in great speed. It also eliminates a conditional analysis that would otherwise be required to decide whether the I node is at the top of a subtree, in which case the address of the parent cell may not be available for rewriting in the TIGRE evaluation scheme.

The **K** combinator is defined in TIGRE as:

```
mov R0, ip
pop(2)
thread
```

Thus, we see that the operation of I and K are almost identical if viewed in the proper manner.

In other graph reducers, the **K** combinator rewrites a node to eliminate the reference to the second input argument. In TIGRE, the K combinator simply pops the reference from the spine stack, eliminating it from the dynamic execution history of the program (but not from the static tree structure). In fact, any 1-projection combinator that takes *n* input arguments may be defined as:

```
mov R0, ip
pop(n)
thread
```

With this method, space is temporarily lost in the heap to subgraphs that would have been abandoned as garbage with a projection combinator that did graph rewrites. With the TIGRE projection combinator scheme, such subgraphs cannot be reclaimed until the subtree owning the reference to the **K** combinator is itself abandoned However, in practice, defining **K** to perform "fall-through" operations results in measurably improved overall performance (yielding approximately a 5% overall program speed improvement for the Fibonacci benchmark discussed in Chapter 5 using the **SKI** subset of the Turner Set). A secondary space consideration is that **I** nodes themselves take up heap space that might be reclaimed, but this problem can be overcome by

using a garbage collector that performs **I**-node shorting (Peyton Jones 1987) if necessary. Similarly, **K**-node shorting could be added to the garbage collector if desired.

4.3.1.2. Simple Graph Rewriting Combinators

The Turner Set also includes other non-strict combinators which perform simple graph rewriting functions. These combinators are **S**, **B**, **C**, **S'**, **B***, and **C'**. All are similar in definition to the **S'** example already discussed.

4.3.2. Strict Combinators

Strict combinators require that some or all of their arguments be evaluated before the combinator can produce an answer. TIGRE can use totally strict combinators that perform computations and return results, and can use partially strict combinators, primarily for conditional branching.

4.3.2.1. Totally Strict Combinators

Totally strict combinators implemented in TIGRE include literal values, unary arithmetic/logic operations and binary arithmetic/logic operations. All of these operations are distinguished by the fact that they are strict, and by the fact that they all return a value to a calling function.

Evaluation of a subgraph in TIGRE is accomplished by performing a subroutine call to a subtree to be evaluated. In interpreted threaded versions, this constitutes a subroutine call to a threading loop, which accomplishes the same purpose. Non-strict combinators do not evaluate any of their arguments, but also do not leave any pointers to their arguments on the spine stack. So, what happens during program execution is that non-strict combinators continually rewrite the program graph without causing a permanent buildup of entries on the spine stack. When a combinator executes, the compilation process has guaranteed that exactly enough parameters are on the spine stack to perform its function. When a combinator that returns a result is completed, it can remove its own inputs from the spine stack and be *guaranteed* that the top element of the spine stack is always a return address to the function that invoked the subtree evaluation.

When a strict combinator requires an input to be evaluated, it performs a subroutine call to perform the evaluation, then resumes execution when the evaluation of the subtree is completed, accepting the

result of the evaluation in the "result" register of the TIGRE abstract machine.

As an example of a combinator that returns a result, consider **LIT**. **LIT** takes one input, which is a constant value in the right-hand side of a node, and returns that value in the result register:

```
mov R0, result
pop 1
return
```

The **+** combinator is an example of a combinator that evaluates arguments and returns a result. Simple TIGRE assembler code for the **+** combinator is:

```
/* evaluate first argument */
mov R+, ip
evaluate
      /* recursive call to evaluation function*/
push(result)        /* save result on stack */
                    /* evaluate second argument */
mov R0, ip
evaluate
      /* recursive call to evaluation function*/
mov pop(1), scratch0   /* capture first result */
add scratch0, result   /* sum in result */
/* re-write answer as LIT node */
mov DO_LIT, L0
mov result, R+
return
```

The **+** combinator first calls the right-hand side of the top node on the stack, which evaluates the first argument to **+** using a subroutine call. When the subgraph is eventually reduced, a combinator within the evaluated subtree will return a value in the result register. This value is pushed onto the spine stack for safe-keeping, and the second argument to **+** is evaluated. The first argument is popped back off the spine stack, and the result is computed to be transferred back to the routine that called the **+** combinator. The **+** combinator also rewrites the node which was the parent of the node containing the **+** combinator, so that if the subtree is shared the evaluation need only be performed once.

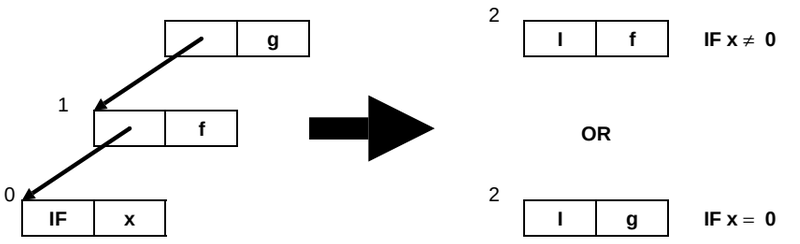

Figure 4-6. The **IF** combinator. IF x f g → (I f) OR (I g)

4.3.2.2. Partially Strict Combinators

The only partially strict combinator in the Turner Set is the **IF** combinator, shown in Figure 4-6. The **IF** combinator evaluates its first argument, then selects the second argument if the first argument is true (non-zero), or selects the third argument if the first argument is false (zero).

The code for **IF** is:

```
    mov R+, ip
    evaluate            /* evaluate first argument */
    bzero IF_B
                        /* TRUE - select first input */
    mov R0, R1
IF_B:                   /* FALSE - leave second input */
    mov R1, ip
    mov DO_I, L1
    pop 2
    thread
```

The code evaluates the first argument. Then, if the first argument is true (non-zero), the second argument is used to overwrite the third argument, otherwise the third argument is left in place. Finally, an **I** combinator is placed in the left-hand side of the third argument node, converting the node to a jump to the selected subtree. While the **IF** combinator could be implemented so as not to rewrite graphs, in the style of the projection combinators, the overhead involved in repeatedly evaluating the first argument probably outweighs the savings possible from not rewriting the graph.

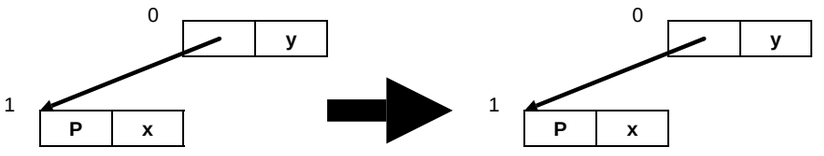

Figure 4-7. The **P** combinator.

4.3.3. List Manipulation Combinators

The Turner Set includes definitions for two list manipulation combinators: **P** and **U**. **P** is the "pairing" combinator, which works much like a "cons" operation in LISP. Figure 4-7 shows the **P** transformation, which protects a pair of subtrees from being evaluated, and returns a pointer to the structure of paired elements. A succession of **P** combinators may be used to build data lists or other data structures.

Figure 4-8 shows the **U** transformation, which performs an "unpair" operation. The **U** combinator is guaranteed by the compilation process to always have a **P**-protected subtree as its second expression. In effect, the **U** combinator is used to peel away the protection afforded to a pair by the **P** combinator.

An obvious way to implement the **U** combinator is to have it interpret the **P**-protected subtree to locate and extract the two list subtrees. Unfortunately this process is slow. It is further complicated by the fact that un-rewritten projection combinators (**I** and **K**) and nodes may be lingering between the **U** combinator and the **P** combinator, introducing case analysis situations into the tree traversal process.

The way TIGRE implements the **U** combinator efficiently is to recursively call the **P** subgraph (using an evaluation call) and let it evaluate itself. The value returned from the **P** combinator is defined to be a pointer to the parent node of the node having the **P** combinator cell (node 0 in figure 4-7):

```
# result is value of second from top spine stk el.
mov address_of(R1), result
      # short out projection combinators
mov address_of(L0), L1
pop 2
return
```

The returned `result` is simply the contents of the second-to-top-most spine stack entry (which points to the parent node of the **P** combinator node). The left-hand side of this parent node is rewritten with a pointer to the **P** combinator node to eliminate any potential projection combinators in the path. This rewriting is in preparation for the **U** node making a traversal of the subtree later. It is important to note that the value returned by the **P** combinator is not necessarily the same as the value used by the **U** combinator subtree to access the **P** subtree, since additional projection combinators may interfere there as well. A secondary use of the **P** node which is supported by this method is the use of **P** to return pointers to unevaluated lists for performing list equality comparisons.

The **U** combinator expects that its second argument will be a pointer to a tree which reduces to a **P** combinator subtree. The value returned from the **P** combinator points to the root of the subtree, whose right-hand side contains one of the subtrees needed by **U** to build its

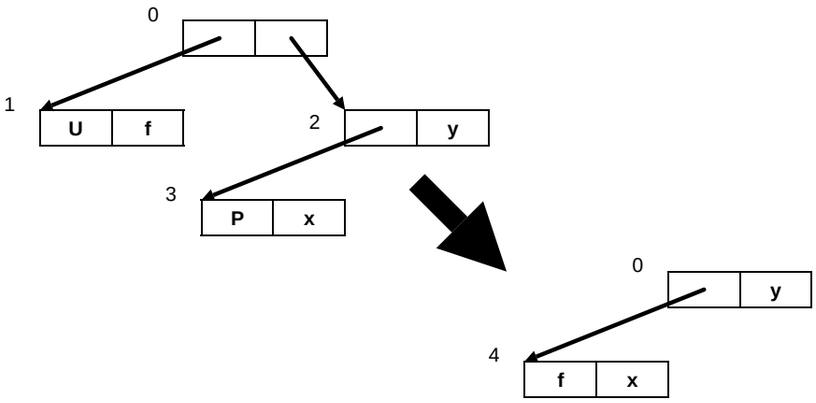

Figure 4-8. The **U** combinator.

result. A single indirection performed by **U** on the left-hand side of this root node is guaranteed to give access to the other subtree reference required by **U**, since **P** has shorted-out any intermediate projection combinators. The code for **U** is as follows:

```
mov R1, ip
evaluate
```

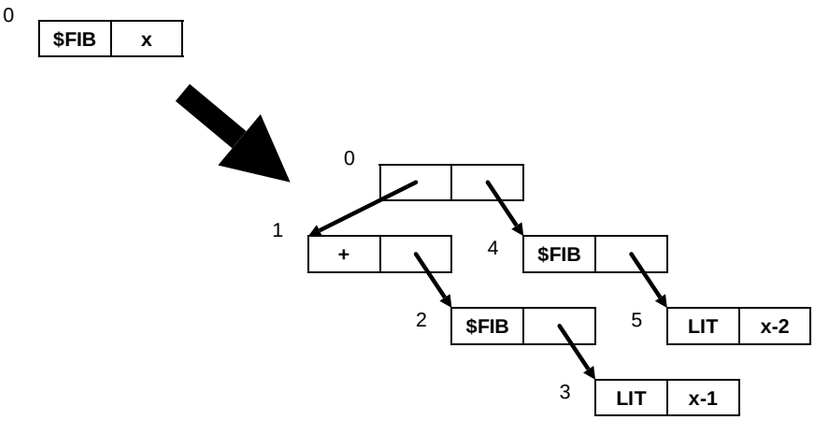

Figure 4-9. The **$FIB** supercombinator.

```
allocate 1
mov R0, ip
mov ip, Ltemp0
mov Rresult, R1
mov Lresult, result
mov Rresult, Rtemp0
top (Rtemp0)
thread
```

The allocation is performed after the evaluation, because the evaluation may disrupt the contents of any heap node pointer registers, and may trigger a garbage collection cycle.

4.3.4. Supercombinators

The compilation of supercombinator definitions for TIGRE is supported by the same set of primitives used to implement the Turner Set. Hand-compiled supercombinator code shows that TIGRE can readily take advantage of supercombinator compilation with sharing and strictness analysis.

One example of supercombinator compilation is the "fib" benchmark, which recursively computes the n^{th} Fibonacci number. Since the definition of fib is itself a supercombinator, then a single graph rewrite for the combinator **$FIB** may be defined as shown in Figure 4-9.

The idea behind the supercombinator definition is to eliminate the need for combinations of graph rewriting combinators such as **S**, **B**, and **C**. In the case of **$FIB**, the TIGRE assembler code is:

```
    allocate 5
    /* evaluate argument */
    mov R0, ip
    evaluate
    cmp result,3
    bge FIB_CONT
    /* result less than 3, return 1 */
    mov 1, result
    return

FIB_CONT:
    /* result not less than 3, recurse */
    dec result          /* decrement result for N-1 */
    mov DO_LIT, Ltemp0
    mov result, Rtemp0
    mov DO_$FIB, Ltemp1
    mov temp0, Rtemp1
    mov DO_PLUS, Ltemp2
    mov temp1, Rtemp2
    dec result          /* decrement result for N-2 */
    mov DO_LIT, Ltemp3
    mov result, Rtemp3
    mov DO_$FIB, Ltemp4
    mov temp3, Rtemp4
    mov temp2, L0
    mov temp4, R0
    mov temp2, ip
    thread
```

From this code, it may be seen that **$FIB** is able to implement efficiently the desired behaviors of a supercombinator. If the input argument to **$FIB** is evaluated to be less than 3, then a 1 value is returned without updating the graph at all (this is because compilation analysis shows that the recursive calls to **$FIB** cannot be shared, so graph updating is of no value).

If the input argument to **$FIB** is 3 or greater, then a new graph is built to hold the recursion structures. No stacking or other memory mechanism is required explicitly by TIGRE to remember the fact that two recursive evaluations are taking place for each evaluation of **$FIB**, since the program graph captures all essential information. Note that the values (x-1) and (x-2) are pre-computed and stored in LIT nodes.

4.4. SOFTWARE SUPPORT

The TIGRE graph reducer cannot live in an isolated environment. It requires various pieces of support software for proper operation. This software will be briefly discussed for the sake of completeness.

4.4.1. Garbage Collection

First and foremost, TIGRE needs an efficient garbage collector. Graph reduction tends to generate huge amounts of garbage. The heap manager must therefore support efficient allocation and quick garbage collection. Several methods of performing this task are available, such as mark/sweep garbage collection, stop-and-copy collection, and generational collection (Appel et al. 1988). The TIGRE implementation currently uses stop-and-copy collection, because it gives significant speedups over mark/sweep collection, yet is easy to implement.

Since the garbage collector must be able to discriminate combinator references from pointers, most implementations of TIGRE use a one-bit tag that is set to indicate a combinator reference. The garbage collector can then follow references to pointers until it sees a combinator when performing copying or marking. This one-bit tag adds no additional execution overhead, however, since it may be ignored by the execution engine if subroutine threading is in use. As an example of an actual implementation, the VAX assembler version of TIGRE aligns all combinator definitions on odd byte boundaries so that the lowest bit of a reference to a combinator is always 1. `Jsb` instructions in the heap are aligned on odd 16-bit boundaries, causing pointers to heap cells to have the lowest bit set to 0. The garbage collector can use this alignment information to distinguish pointers from combinators, but the `jsb` instructions at run time ignore the information available, since it is not needed. An alternate method that does not require an explicit tag bit is to perform an address range check to see whether a pointer points to an element in the heap space. On the VAX, the lowest bit was used as a tag because the VAX architecture supports a branch-on-lowest-bit instruction.

A problem with stop-and-copy garbage collection, or any garbage collector that performs relocation of elements, is that the contents of the spine stack must be updated whenever elements are moved. This process of updating the spine stack is relatively quick, but it does increase code complexity and is subject to subtle bugs.

Because of the complexity inherent in directly updating the spine stack, a different method for coping with heap cell relocation has been

found. The method used by TIGRE is to simply throw away the spine stack after a garbage collection, and restart graph reduction at the root node of the tree. This method is guaranteed to work, because the program graph is continually updated by graph rewrites to reflect the current state of the computation. So, the information in the spine stack is redundant, since it is guaranteed to match the path down the left spine of the graph. This means that the spine stack information can be regenerated simply by re-interpreting the graph.

This method of throwing the spine stack contents away after each garbage collection has been implemented successfully. It eliminates the likely chance of a bug in the stack relocation algorithm. The cost of regenerating the spine stack seems to be roughly comparable to relocating the spine stack (no measurable speed difference was detected on trial runs). And, the concept of discarding the spine stack brings to light the fact that a processor evaluating a graph need only have one word of state (the graph root pointer) in order to have access to the entire state of the computation. This economy of state representation may prove crucial in efficiently implementing parallel processing versions of TIGRE.

4.4.2. Other Software Support

Software support is also needed to read the TIGRE graph from a file, build it in memory, and print a graph out of memory for debugging purposes. These functions are supported by C procedures that call the TIGRE interpreter as required.

The TIGRE input file parser is perhaps the most interesting of these three functions. The TIGRE parser takes two input modes. The first input mode is S-expression notation, which takes parenthesized binary graph expressions such as:

```
((S ((S (K +)) I)) I)
```

which implements the doubling function. Integer constants may be included freely in the graph description, and the parser will automatically create **LIT** nodes as the program is parsed. S-expression notation has the advantage of being readily understood by humans.

The second input mode for the TIGRE parser is sets of triples. Triples are a more powerful method of representing a graph, since S-expression notation has difficulty expression sharing and cycles. A triple file for the doubling function might look like:

```
0 #1 I
1 S  #2
2 #3 I
```

```
3 S   #4
4 K   +
```

where the first column identifies an integer node number (with 0 defined as the root of the graph). Combinators appear as their name, while integer constants (which have automatically created **LIT** nodes) appear as just a number. A hash mark followed by a number indicates a pointer reference. The middle of the three symbols in a line is the left-hand side of a node, while the third of the three symbols is the right-hand side of a node.