Chapter 3
Development of the TIGRE Method

This describes the TIGRE method for combinator graph reduction. Section 3.1 illustrates the conventional notion of performing graph reduction. Section 3.2 describes a faster interpretation method that uses one-bit tags. Section 3.3 describes using direct execution of combinator graphs in order to eliminate tag bits and further increase execution speed.

3.1. THE CONVENTIONAL GRAPH REDUCTION METHOD

Many early graph reducers have treated combinator graphs solely as data structures to be manipulated. This point of view leads to assumptions and implementation methods that result in significant efficiency penalties. *TIGRE* (Threaded Interpretive Graph Reduction Engine) is a graph reduction technique that views the combinator graph as a directly executable program instead of a data structure, offering significant performance improvements over the conventional approach.

Combinator graphs can be represented by binary graphs, with each node having a function cell (the left-hand side) and an argument cell (the right-hand side). However, some tagging information is also needed to identify the type of cell contents. Figure 3-1 shows that, in practice, nodes are typically represented by four fields. The first pair of fields are the tag and value of the function cell, while the second pair of fields are the tag and value of the argument cell. Figure 3-2 shows a

Figure 3-1. Basic structure of a node.

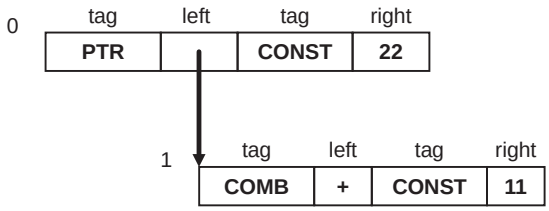

Figure 3-2. Example for expression ((+ 11) 22).

graph for the expression ((+ 11) 22) built using this typical node structure.

The problem with the tagged node representation is that every time a node is read, a case analysis must be performed to determine what to do with the contents. For example, in Figure 3-2, pointers, combinators, and literal values must be distinguished by inspecting their tag values for correct interpretation of the program graph. Conditional branches are inherently difficult and expensive operations, as can be attested to by any computer architect who has designed a pipelined processor. Hence, this case analysis is a major impediment to improving execution speed. Although only three tag types are shown in Figure 3-2, in general more tag types are used since the cost for the case analysis must be paid anyway.

One clever implementation is to select the tags to be the base value of a jump table containing addresses of action routines. Accessing a node requires a double indirection operation through the tag and jump table. This technique has been previously implemented for the G-Machine graph reduction implementation on a VAX (Peyton Jones 1987). In VAX assembly language, unwinding a node while traversing the stack requires four instructions, including this double indirect jump through the jump table:

```
movl    Head(r0),r0          # get head of node
movl    r0,-(%EP)            # push value onto spine stack
movl    (r0),r1              # get tag of node
jmp     *0_Unwind(r1)        # unwind the node
```

The G-Machine has modes of operation, the primary two of which are stack unwinding and evaluation. Each jump table has fixed locations of entries for each mode of operation, and a separate jump table is provided for each combinator. All combinator references require an

access to an empty node, which has a combinator value as its tag. Function application nodes, however, use a tag that addresses a function application "combinator" with appropriate left-hand and right-hand cells.

There is a further cost for the G-Machine of a seven-instruction sequence that must be used as a preamble to each combinator to fix up the spine stack after the stack unwinding. A representative value (reported in detail in a Chapter 6) for spine stack nodes consumed per combinator is 1.38. This means that, on average, processing nodes for stack unwinding costs:

(4 * 1.38) + 7 = 12.52 VAX instructions per combinator

This represents an overhead cost above and beyond the actual combinator execution code. Since TIGRE implements most Turner set combinators with between three and twenty VAX instructions (*e.g.,* the **S** combinator is implemented in 17 VAX instructions, while the **K** combinator is implemented in four VAX instructions), it is clear that the G-Machine approach to unwinding the spine can cause a considerable overhead. There are further problems associated with the G-Machine tag strategy. One of these is that combinators are represented by unique tag values, which can require the creation of a new jump table for each combinator added to the system.

3.2. FAST INTERPRETIVE EXECUTION OF GRAPHS

A primary goal of TIGRE is the elimination of the tag processing overhead just described. TIGRE takes the most straightforward ap-

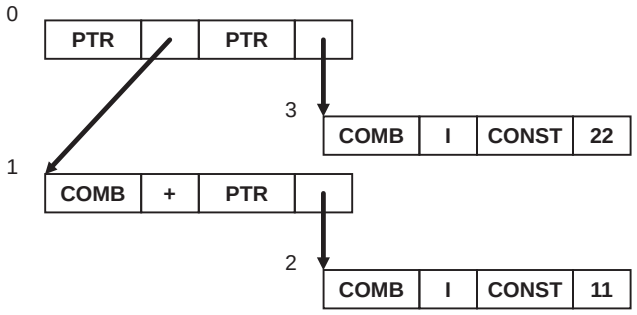

Figure 3-3. Example using indirection nodes for constants.

proach to solving the problem. Since the interpretation of tags is an expensive operation, eliminate the tags and hence the expense of processing them. This section and the next describe in a step-by-step process how this elimination of the need for tags is accomplished.

As a first step in eliminating tags, all cells containing constant values are replaced by pointers to indirection nodes having the constant value. Figure 3-3 shows the result of this rewriting on the example graph for ((+ 11) 22). Any graph can be rewritten during compilation with constant values placed in the right-hand sides of indirection nodes in a similar manner. This rewriting operation may appear to be wasteful, but is in fact the way graphs often exist during program execution. For example, the **+** combinator, when executed, creates an indirection node with the sum. Thus, if the 11 and 22 in Figure 3-3 were actually the results of previous computations, both would have been in the right-hand side of **I** nodes before being moved to the right-hand sides of nodes 0 and 1.

Now, notice that constants are only found as arguments to indirection combinators. If those **I** combinators in the left-hand side of constant nodes are renamed as **LIT** combinators (short for "literal value" combinators), as shown in Figure 3-4, the constant tag is no longer needed, since the **LIT** combinator implicitly identifies the argument as a constant value. All other special tag types can be eliminated by defining new combinators in a similar manner. In particular, variations of the **LIT** combinator can be created for different numeric data types.

The graph shown in Figure 3-4 now only has two tag types: combinator and pointer. The cost of tag checking can then be reduced by using any number of standard tricks. For instance, all nodes and

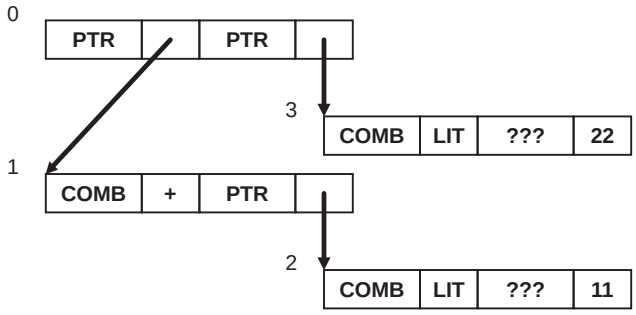

Figure 3-4. Example using **LIT** nodes instead of indirection nodes for

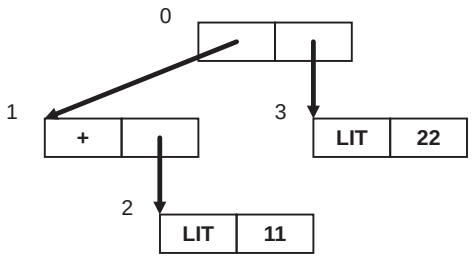

Figure 3-5. Example with tag fields removed.

therefore pointer values can be aligned on 4-byte boundaries (which improves speed or is even required on many machines). The lowest bit of a cell's contents can then be used as a one-bit tag. Figure 3-5 shows the graph rewritten in this style. It is important to note that there is still, in fact, a one-bit tag hidden within each cell value, and that the process of describing how tags are eliminated continues in the next section.

The case analysis for numeric constants has been replaced by the need to reduce **LIT** combinators (although we argue that this combinator is often present in the form of an **I** node anyway). However, we have also reduced the amount of tag checking on all other cells. This is the representation used for the C language implementation of TIGRE. Other details of TIGRE will be deferred for the moment, but in general TIGRE loops while scanning the lowest order bit of left-hand side cells to unwind the stack. When a non-pointer value is found, TIGRE then uses a case statement to jump to the correct action code.

3.3. DIRECT EXECUTION OF GRAPHS

There is an additional key insight which provides at least a twofold speedup when using assembly language on many architectures over that possible with C code alone. This insight is gained by exploiting the hardware support for graph traversal that already exists in most conventional processors.

The generic graph shown in Figure 3-6 is executed by traversing the leftmost spine, placing pointers to ancestor nodes onto a spine stack. When a combinator is encountered in the graph, some code to implement the combinator is executed. The data structure is controlling the exe-

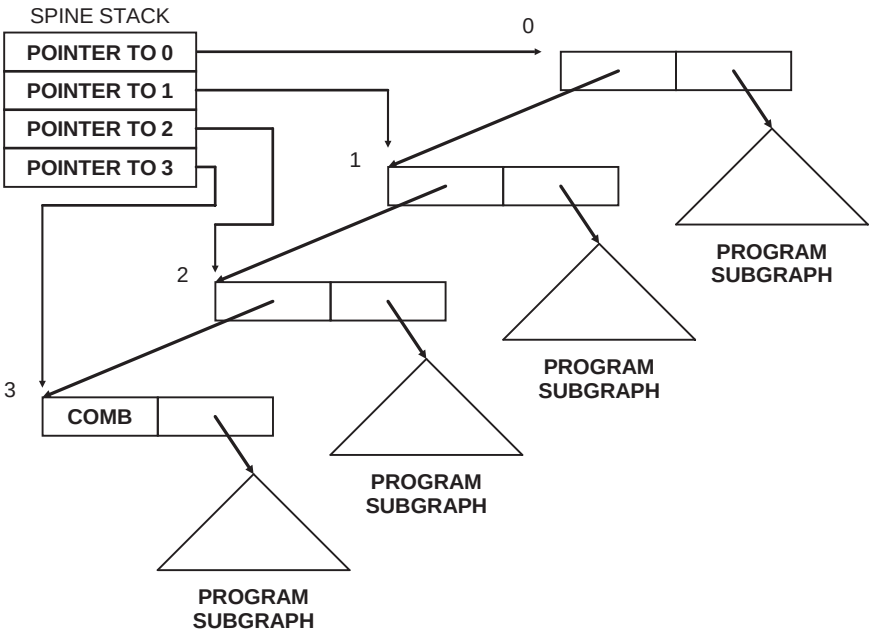

Figure 3-6. An example TIGRE program graph, emphasizing the left

cution of the program. Another, more insightful, view is that the data structure is itself a program with two instruction types: pointer and combinator. Then graph reduction is essentially a process of interpreting a threaded program that happens to reside in the node heap. In other words, the tree is a program that consists mainly of calls to subroutines. These subroutines then contain calls to other subroutines, and so on until, finally, some other executable code is found. Thus, threaded code interpretation (Bell 1973) is used to implement the C version of TIGRE.

Now, consider the operations used to unwind the spine of this threaded program graph. As each spine element is unwound, a pointer into the program graph (which may be called pc), is pushed onto the spine stack, then the contents of pc are replaced with the contents of the memory location pointed to by pc as in the following sequence:

```
push(pc)
pc ← [pc]
```

However, it is equally valid to specify that a pointer to the right-hand side of the graph is pushed onto the spine stack, while still accomplishing the same purpose of retaining pointers to spine elements. If pointers are assumed to be four bytes in size, the spine unwinding operation becomes:

push(pc+4)
pc ← [pc]

But, this exactly corresponds to the actions of a subroutine call instruction (assuming that the pc points to a 32-bit word in memory containing the subroutine call instruction's address field when the sequence is started).

The key idea is that *the spine stack is actually just a subroutine return stack for the threaded program*. As control flows from node 0 to node 1 to node 2 to node 3 in the graph of Figure 3-6, pointers to these nodes are stored on the spine stack. These pointers will eventually be used to access the right-hand side values of the ancestor nodes as arguments to a combinator, so what really is desired is, in fact, to save the pointers to the right-hand sides of each node on the spine stack. Thus, saving pc+4 on the return stack is actually more efficient than saving just pc.

Combinator nodes, such as node 3 in Figure 3-6, contain some sort of token value that invokes a combinator. At some point during program execution, this value will have to be resolved to an address for a piece

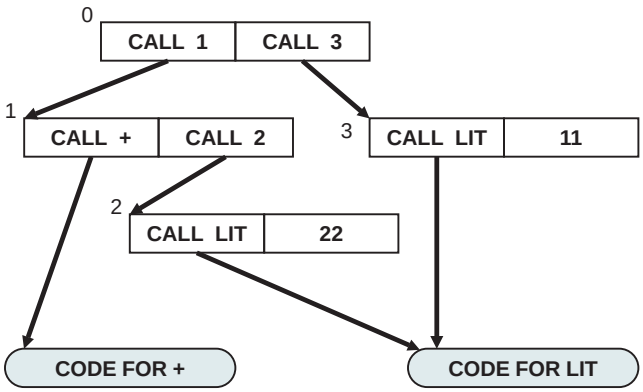

Figure 3-7. A TIGRE program graph with only subroutine call pointers.

of code to be executed, so the assembler version of TIGRE simply stores the actual code addresses of the combinator action routines instead of token values. In fact, if a subroutine call to the combinator code is stored, the address of the right-hand side of node 3 will be pushed onto the spine stack, and the combinator will have all its arguments pointed to by the spine stack (which is now the subroutine return stack). A pleasant side effect of this scheme is that there is now only one type of data in the graph: the pointer. Hence there is only one type of node, and therefore *no conditional branching or case analysis is required at run-time*. All nodes contain either pointers to other nodes or pointers to combinator code. Figure 3-7 shows the running example of ((+ 11) 22) compiled using this scheme. Since all node values (except the right-hand sides of **LIT** cells) are subroutine call instructions, matters are simplified by saying that each cell contains a pointer that is interpreted as a subroutine call by the TIGRE execution engine.

At a more detailed implementation level, TIGRE graph nodes can be implemented as triples of 32-bit cells. The first cell of each triple contains a subroutine call instruction while the second and third cells of the triple contain the left- and right-hand sides of the node, respectively. The hardware's native subroutine calling mechanism is used to traverse the spine, using the subroutine return stack as the spine stack. Figure 3-8 shows the example graph as it appears in the VAX assembly language implementation of TIGRE. (Note that the jsb is the fast VAX subroutine call instruction, which pushes only the program counter onto

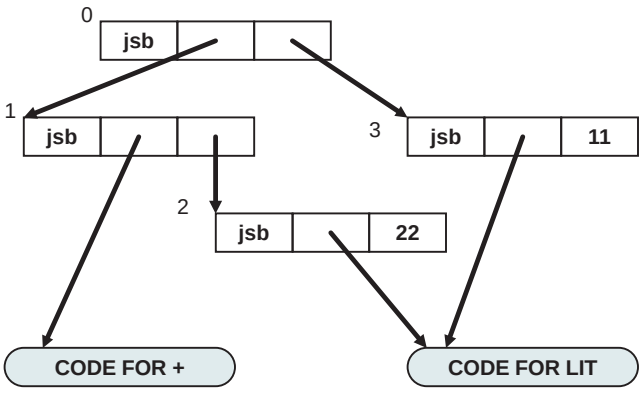

Figure 3-8. VAX assembly language implementation of a TIGRE ex-

the return address stack, as opposed to the slower procedure call instructions which automatically allocate stack frames.)

While the graph shown in Figure 3-8 is simple, its operation is not necessarily obvious. Evaluation of a program graph is initiated by performing a subroutine call to the jsb node of the root of a subgraph. The machine's program counter then traverses the left spine of the graph structure by executing the jsb instructions of the nodes forming the spine. When a node points to a combinator, the VAX simply begins executing the combinator code, with the return address stack providing addresses of the right-hand sides of parent nodes for the combinator argument values. When graph nodes are rewritten, only the pointer values (which are 32 bits in size on a VAX) need be rewritten. The jsb opcode is initialized upon initial acquisition of heap space and thereafter never modified.

The processor is in no sense interpreting the graph. It is *directly executing* the data structure, using the hardware-provided subroutine call instructions to do the stack unwinding. However, the jsb opcode does provide a tag of sorts for pointer cells (since jsb is associated with cells in the heap whereas combinator code uses other opcodes). Thus, TIGRE can be thought of as a method for using the decoding circuits already existing in conventional hardware to perform hardware-based execution of tagged data. As noted previously, this use of jsb instructions to speed up graph reduction was previously reported by Augusteijn & van der Hoeven (1984).

A summary of TIGRE's operation, then, is that an interpretive pointer is used to execute subroutine call operations down the left spine of the graph. When combinators are reached, they pop their arguments from the subroutine return stack, perform graph rewrites, and then jump to the newly reduced subgraph to continue traversing the new left spine. The use of the return stack for graph reduction is slightly different than for "normal" subroutines in that subroutine returns are never performed on the pointers to the combinator arguments which have been pushed onto the return stack. Instead, the addresses are consumed from the return stack by the combinators. (This use of the stack to provide arguments seems to be a characteristic of other combinator reducers as well).

TIGRE uses the same primitive functions over and over again to implement the combinators. Only a few primitives such as "fetch the right-hand value of the parent node" are needed to implement the entire Turner Set of combinators. This suggests the possibility of a simple assembly language that can be used to defined supercombinators for TIGRE, even on a special-purpose hardware version, with only a mini-

mal set of machine operations. A definition of the TIGRE abstract machine and its assembly language are given in the next chapter.