A Fresh Look at Combinator Graph Reduction
(Or, Having a TIGRE by the Tail)

Philip J. Koopman, Jr.
Department of Electrical and Computer Engineering
and
Peter Lee
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

We present a new abstract machine for graph reduction called TIGRE. Benchmark results show that TIGRE's execution speed compares quite favorably with previous combinator-graph reduction techniques on similar hardware. Furthermore, the mapping of TIGRE onto conventional hardware is simple and efficient. Mainframe implementations of TIGRE provide performance levels exceeding those previously available on custom graph reduction hardware.

Introduction

We have designed and implemented a new abstract machine called TIGRE (Threaded Interpretive Graph Reduction Engine) for executing combinators in a pure graph-reduction style. TIGRE maps efficiently onto conventional hardware, providing speeds that compare quite favorably with previous combinator graph reducers. For example, the current implementation of TIGRE on a single-processor VAX 8800 can achieve over 355,000 combinator-reduction applications per second (raps) when reducing the Turner set of SK-combinators (Turner 1979b). Even a microVAX workstation executing TIGRE performs 155,000 raps. These numbers compare well with existing software implementations of graph reducers, and also with custom-built hardware such as NORMA, which has a 250,000 raps performance rating under similar benchmark conditions (Scheevel 1986). Supercombinator-based compiler optimization techniques promise to improve further the performance of TIGRE-based programs significantly beyond the current levels.

In this paper we give a description of the TIGRE abstract machine, provide code listings for portions of its implementation, report preliminary performance data, and make comparisons with other methods for combinator reduction.

Background

Turner (1979a) described a technique for implementing normal-order functional languages, sometimes referred to as SK-combinator reduction. The idea is based on the observation that all of the variables in a functional program can be removed by transforming it into a sequence of combinators which are drawn from a small, pre-defined set of combinators. With all free variables thence removed, the resulting program text becomes amenable to representation as a graph in which subgraph sharing represents the sharing of subexpressions in the program, cycles represent recursion, and combinator definitions represent graph rewrite rules. In this scheme, executing programs becomes a process of graph reduction.

Besides the great advantage in efficiency gained from sharing, the language implementation overall becomes much simpler by virtue of the fact that the usually "tricky" issues of variables and substitution are, in effect, encapsulated in a small set of simple rules for rewriting graphs. Indeed, a pure graph reducer can be implemented quite easily, and will often exhibit better performance than implementations of normal-order languages based on other approaches. Such simplicity also lends itself to direct hardware implementation, as in SKIM (Stoye 1984) and NORMA. Still, normal-order evaluation (or, more precisely, fully "lazy" evaluation) of functional programs, even via combinator-graph reduction, is in practice much less efficient than applicative-order ("eager") evaluation. Hence, a great deal of research has been directed towards improving the efficiency of combinator-based techniques. One significant development along these lines, first proposed by Hughes (1982), is the notion of "super-combinators", in which the crucial observation is made that any function can

Research supported in part by NASA/Goddard under contract NAG-5-1046, also in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No.5404, monitored by the Office of Naval Research under the same contract. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

be made into a combinator by adding extra formal parameters corresponding to the free variables appearing in the function body. Supercombinator compilation produces a set of custom-defined combinators for each program, resulting in much larger-grain reduction steps, and thus requiring fewer reductions for evaluation.

One promising development is the so-called **closure-based** execution of combinator expressions, as exemplified by TIM (Fairbairn & Wray 1987). Closure-based approaches have an advantage over previous pure graph reduction implementations in that they can be mapped easily onto conventional architectures. However, these closure-based architectures are also quite subtle and more difficult to understand than pure graph reducers.

TIGRE, A New Architecture for Pure Graph Reduction

The major sources of inefficiency in most graph reducers are the traversing of the graph's left spine (stack unwinding) and the case analysis of node tags. If these costs are reduced or eliminated, significant speedups may be possible. In this section, we shall use the mechanisms typically employed by graph reducers (for instance, the Chalmers G-Machine as described by Peyton Jones (1987)) as a starting point. Then we shall explain how TIGRE is able to avoid certain inefficiencies present in previous graph reducers.

Figure 1 shows that nodes are typically represented by three words. The first word is a tag for the values in the application node. This tag value is selected so as to be an index value into an entry table containing addresses of action routines. Accessing a node requires a double indirection operation through the tag and entry table. On a VAX, unwinding a node while traversing the stack requires

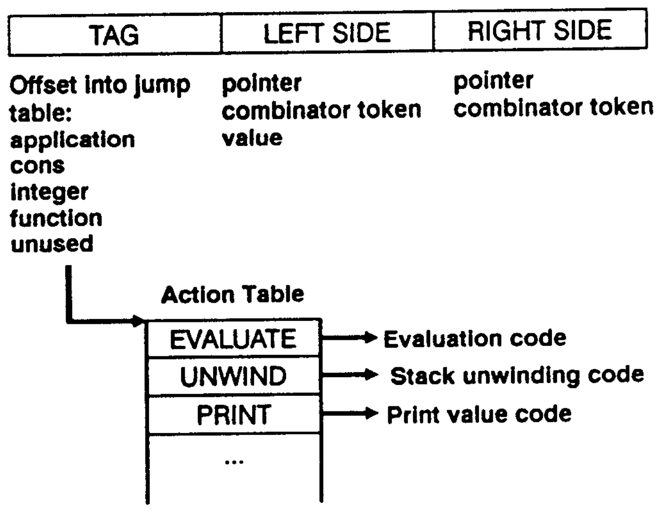

Figure 1. G-Machine tag case analysis structure (after Peyton Jones (1987))

four instructions, including this double indirect jump through the entry table (Peyton Jones 1987):

```
movl   Head(r0),r0
movl   r0,-(%EP)
movl   (r0),r1
jmp    *0_Unwind(r1)
```

One of the key points of TIGRE is the elimination of most of this overhead for traversing tree nodes during the stack unwinding process. This can best be accomplished by simply eliminating the need for tags, thereby eliminating the cost of tag interpretation. Figure 2 shows a generalized node representation which has tags associated with both the left and right-hand sides of the node. Figure 3 shows a tree for the expression ((+ 11) 22) which we shall use as a running example. The numbers above the nodes serve as labels for our discussion. Although only three tag types are shown in the example, typically more tag types are used in actual implementations.

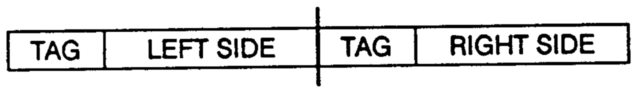

Figure 2. Basic structure of a node.

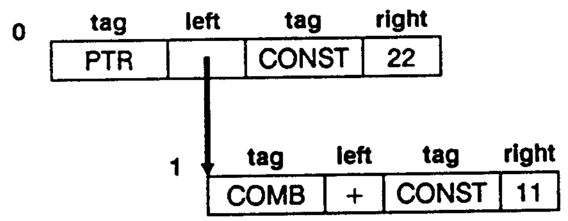

Figure 3. Example for expression ((+ 11) 22).

As a first step in eliminating tags, we shall replace the cells containing constant values by pointers to indirection nodes. (We are assuming, without loss of generality, that all data items are integers.) Figure 4 shows the result of this rewriting. Any graph can be rewritten with constant values placed in the right-hand sides of indirection nodes in a similar manner. This may appear to be wasteful, but is in fact the way graphs often exist during program execution.

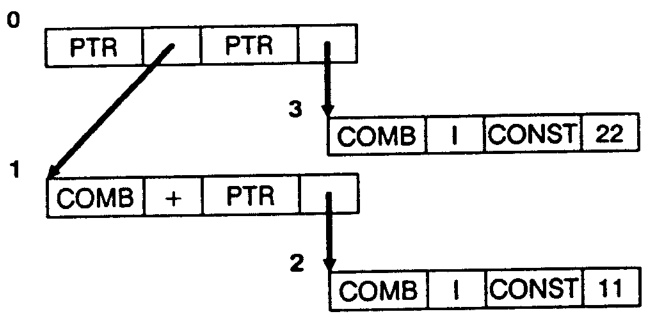

Figure 4. Example using indirection nodes for constants.

For example, the + combinator, when executed, creates an indirection node with the sum. Thus, if the 11 and 22 of Figure 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 we rename those I combinators in the left-hand side of constant nodes as LIT combinators (short for "literal value" combinators), as shown in Figure 5, 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.

The graph shown in Figure 5 now only has two tag types: combinator and pointer. We can now greatly reduce the cost of tag checking by using any number of standard tricks. For instance, all nodes and 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

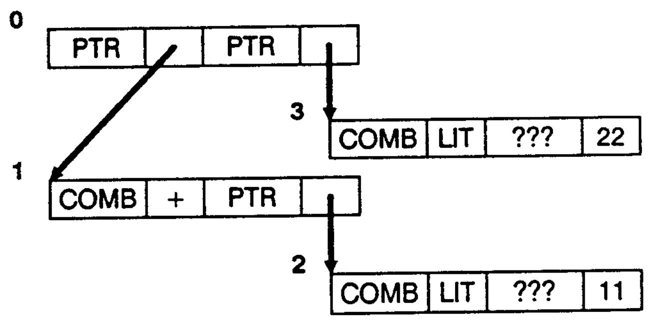

Figure 5. Example using LIT nodes instead of indirection nodes for constants.

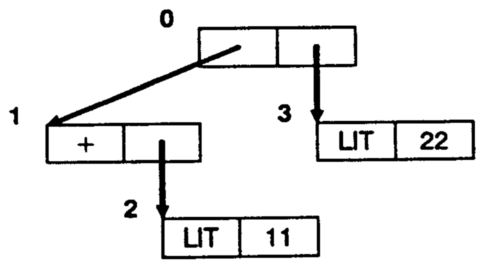

Figure 6. Example with tags removed.

contents can then be used as a one-bit tag. Figure 6 shows the graph rewritten in this style.

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.

The Key Insight

There is an additional key insight which provides approximately a twofold speedup when using VAX assembly language 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 7 is executed by traversing the leftmost spine, placing pointers to ancestor

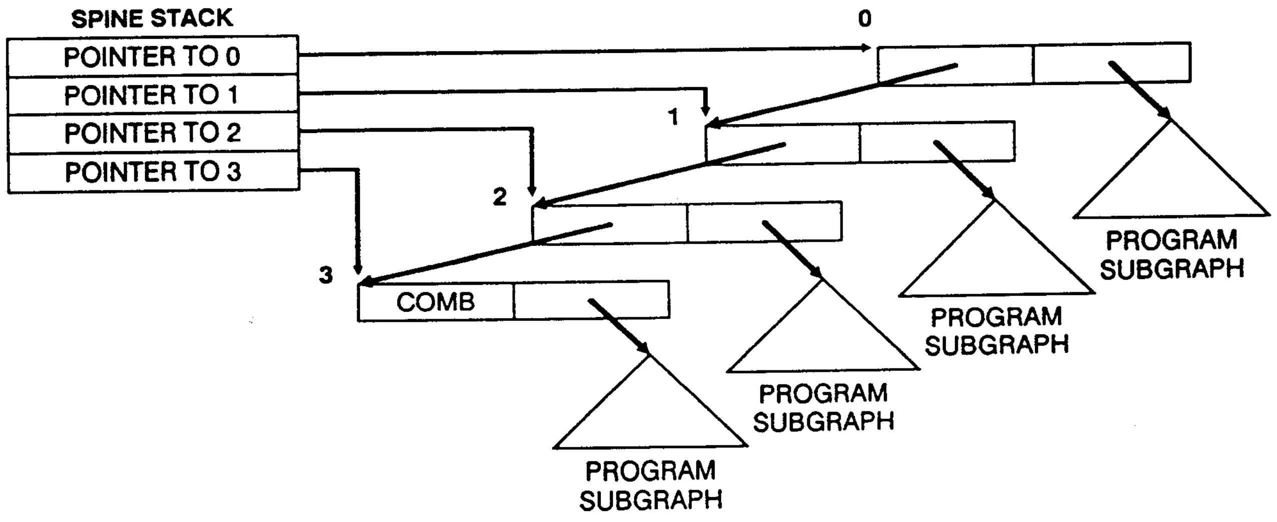

Figure 7. An example TIGRE program graph, emphasizing the left spine.

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 execution 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. The C implementation of TIGRE thus is actually a threaded code interpreter. Threaded code is not a new concept (Bell 1973), but there appear to be no previous applications of this concept to combinator graph reduction.

The key idea is that *the spine stack is actually just a subroutine return stack* for the interpreted threaded program. As control flows from node 0 to node 1 to node 2 to node 3 in the graph of Figure 7, 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 we really want saved on the stack are pointers to the right-hand sides of each node. If the left-hand sides of each node are viewed as subroutine call instructions, then the return addresses which would be automatically saved would be the right-hand cell addresses of the spine of the graph, which is exactly the desired behavior.

Combinator nodes, such as node 3, 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 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, we store a subroutine call to the combinator code, so the address of the right-hand side of node 3 in Figure 8 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 runtime*. All nodes contain either pointers to other nodes or pointers to combinator code. Figure 8 shows our 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, we can simplify matters by simply 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 9 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 only pushes the program counter onto the return address stack, as opposed to the slower subroutine call instructions which automatically allocate stack frames.)

While the graph shown in Figure 9 is simple, its operation is not necessarily obvious. Evaluation of a program graph is initiated by doing 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 following the leftmost 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 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-

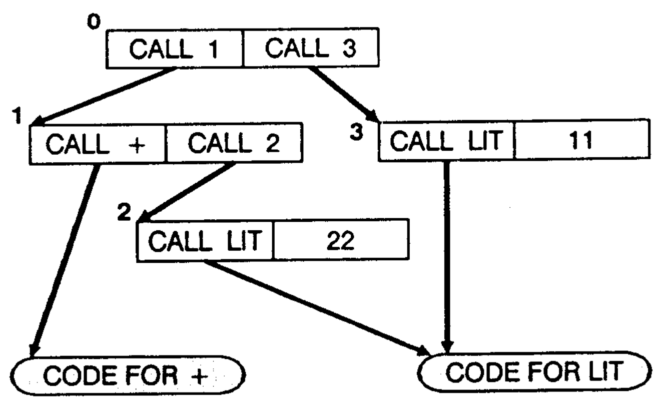

Figure 8. Example with pointers replaced by subroutine call instructions.

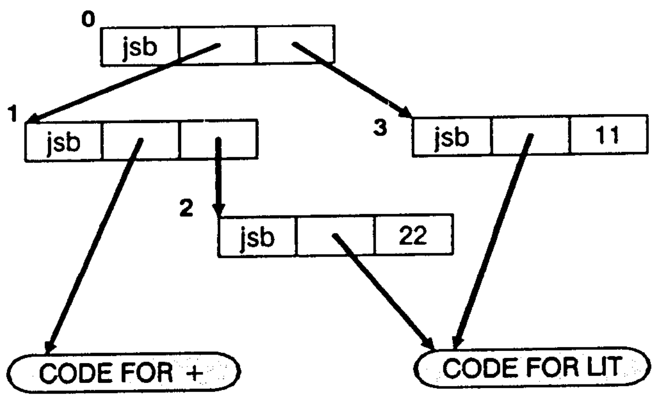

Figure 9. VAX implementation of a TIGRE expression.

provided subroutine call instructions to do the stack unwinding.

Briefly, TIGRE uses an interpretive pointer to do subroutine call operations down the left spine of the graph. When combinators are reached, they pop their arguments from the return stack, perform graph rewrites, and then jump to the new 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, but rather, the addresses are consumed from the return stack by the combinators. (This seems to be a characteristic of other combinator reducers as well).

Implementing TIGRE

The availability of a reasonably quick subroutine call instruction on most modern architectures makes the TIGRE technique applicable, in theory, to most computers. In practice, there are some 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.

Figure 10 shows a block diagram of the TIGRE abstract machine. As a minimum, TIGRE requires a processing unit (with ALU and control logic), a spine stack/subroutine return stack, a small collection of registers for holding temporary values (or a second stack for data manipulation), memory for holding combinator definitions, and heap memory for holding the graph nodes. In the VAX implementation, the stack memory, combinator memory, and graph memory all reside in the same memory space.

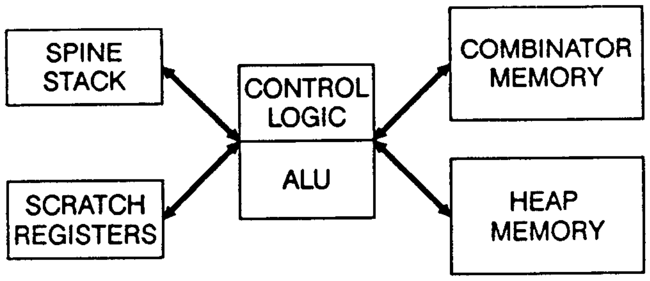

Figure 10. A block diagram of the TIGRE abstract machine.

Table 1 sketches two implementations of TIGRE for the SKI combinator set. The first column lists TIGRE pseudocode which is the basis for a TIGRE assembly language (for supercombinator definitions) currently under development (Lee & Koopman 1989). The second column lists C code to implement the pseudo-operations. The C macros `Rme`, `Lme`, `Rparent`, and `Lparent` perform indirect reads and writes to heap memory through the top two nodes of the spine stack. The third column lists the corresponding VAX assembly language implementation of TIGRE. The VAX code presented is a simple version written for clarity. The VAX implementation actually in use has various small optimizations to eliminate redundant memory reads and better exploit the pipeline of high-end VAX processors.

It is important to realize that traversing the leftmost spine is nearly free, while rewriting the graph to perform reductions is an expensive operation. This leads to some novel design decisions, one of which is the implementation of "projection" combinators, such as I and K. These combinators do not modify the graph at all, but rather redirect the flow of control of the graph evaluation, popping elements from the return stack as they execute.

The K and I combinators are implemented in an almost identical manner, revealing K and I as two specializations of the projection set of combinators which simply drop a number of parent nodes while performing an indirection operation on the topmost node on the spine stack. This optimization may degrade garbage collection performance by leaving subtrees attached to a K node when they would have otherwise been abandoned, but our experience thus far has been that the speedup realized by avoiding graph rewrites more than makes up for this inefficiency.

Another important realization is that TIGRE uses the same primitive functions over and over again to implement combinators. Only a few primitives such as "fetch the right-hand value of the parent node" are needed to implement the entire Turner Set (Turner 1979b) of combinators. An assembly language of similar primitives can be used to defined supercombinators for TIGRE, even on a special purpose hardware version, with only a minimal set of machine operations.

TIGRE Performance

The TIGRE architecture was first explored using Forth, a language noted for its stack manipulation facilities. Since then, TIGRE has been implemented in C and VAX assembler. The C version has been run on a variety of platforms. The VAX assembler versions have been run on different members of the VAX family, and show significant improvements over the output of an optimizing C compiler. These improvements are in large part due to the inability of C to express manipulations of the subroutine return stack.

TIGRE OPERATION	C CODE	VAX ASSEMBLER

I:
ip = Rme ;
pop(1) ;
THREAD ;
	case DO_I:
	ip = Rme ;
	spine_stack += 1 ;
	continue;
		DO_I:
		movl *(sp),r6
		movab 4(sp),sp
		jmp (r6)

K:
ip = Rme ;
pop(2) ;
THREAD ;
	case DO_K:
	ip = Rme ;
	spine_stack += 2 ;
	continue ;
		DO_K:
		movl *(sp),r6
		movab 8(sp),sp
		jmp (r6)

S:
ALLOCATE(2) ;
ip = Rme ;
Ltemp1 = Rme ;
pop(1) ;
Ltemp2 = Rme ;
Rtemp2 = Rparent ;
 	
Rtemp1 = Rparent ;
Lparent = temp1 ;
Rparent = temp2 ;
pop(1) ;
push(right_addr(temp1));
THREAD ;
	case DO_S:
	need2(temp1,temp2) ;
	Ltemp1 = ip = Rme ;
	
	spine_stack += 1 ;
	Ltemp2 = Rme ;
	Rtemp2 = Rtemp1 = Rparent ;
	
	Lparent = temp1 ;
	Rparent = temp2 ;
	*(spine_stack) = temp1+1
	
	continue ;
		DO_S:
		need2(r8,r7)
		movl *(sp)+,r6
		movl r6,(r8)
		
		movl *(sp),(r7)
		movl 4(sp),r10
		movl (r10),4(r7)
		movl (r10),4(r8)
		movab -2(r8),-4(r10)
		movab -2(r7),(r10)
		movab 4(r8),(sp)
		
		jmp(r6)

IF:
result = Eval(Rme)
 	
pop(1) ;
 	
if (result is true)
 	
Rparent = Rme ;
endif
Lparent = addr(I) ;
ip = Rparent ;
pop(2) ;
THREAD ;
	case DO_IF:
	result = Eval(Rme)
	
	spine_stack += 1 ;
	
	if (result)
	
	{ Rparent = Rme ; }
	
	Lparent = DO_I ;
	ip = Rparent ;
	spine_stack += 2 ;
	continue ;
		DO_IF:
		movl *(sp)+,r6
		jsb (r6)
		
		movl 4(sp),r10
		tstl r11
		jeql L39
		movl *(sp),(r10)
		
		L39: $DO_I,-4(r10)
		movl (r10),r6
		addl2 $8,sp
		jmp (r6)

LIT:
result = Rme ;
pop(1) ;
return(result) ;
	case DO_LIT:
	result = Rme ;
	spine_stack += 1 ;
	return(result) ;
		DO_LIT:
		movl *(sp)+,r11
		
		rsb

PLUS:
result = Eval(Rme)
 	
pop(1) ;
result = result + Eval(Rme);
 	
Lme = addr(LIT) ;
 	
Rme = result ;
pop(1) ;
return(result) ;
	case DO_PLUS:
	result = Eval(Rme)
	
	pop(1) ;
	result += Eval(Rme);
	
	Lme = DO_LIT ;
	
	Rme = result ;
	pop(1) ;
	return(result) ;
		DO_PLUS:
		movl *(sp)+,r6
		jsb (r6)
		
		movl *(sp),r6
		pushl r11
		jsb (r6)
		addl2 (sp)+,r11
		movl (sp)+,r9
		movl $DO_LIT,-4(r9)
		movl r11,(r9)
		
		rsb

THREAD:
while (ip not a combinator)
{ push(right_addr(ip)) ;
ip = Lme ; }
jump(ip);
	/* Threader loop */
	while (ip < MAX_COMB)
	{*(--spine_stack)= ip+1;
	ip = Lme ; }
	switch(ip)
		(Implicit threading)
		jsb Lme

Table 1. Code listing for some TIGRE combinators.

Table 2 shows the performance of TIGRE. Simple stop-and-copy garbage collection (Baker 1978) is used. The allocated heap space is small enough to force several dozen garbage collection cycles in order to represent fairly the average cost of garbage collection. No sharing analysis or other optimizations beyond compiling to the Turner Set of combinators has been used. Table 3 shows source code for the benchmarks along with S-expression representations for some of the compiled program graphs.

Platform	Language	Program	Time (sec)	Speed (RAPS)
VAX 8800	ASSEMBLER	SKIFIB(23)	2.82	387000
		FIB(23)	2.10	355000
		NFIB(23)	3.55	366000
		TAK	16.07	329000
	C	SKIFIB(23)	6.50	168000
		FIB(23)	5.01	149000
		NFIB(23)	9.13	142000
microVAX	ASSEMBLER	SKIFIB(23)	6.33	172000
		FIB(23)	4.80	155000
		NFIB(23)	8.23	158000
	C	SKIFIB(23)	13.12	83000
		FIB(23)	10.75	69000
		NFIB(23)	19.16	68000
SUN 3/260 24 MHz	C	SKIFIB(23)	8.62	126000
		FIB(23)	7.01	105000
		NFIB(23)	12.37	105000
SUN 3/75 16 MHz	C	SKIFIB(23)	14.62	75000
		FIB(23)	12.75	58000
		NFIB(23)	22.02	59000

Table 2. TIGRE performance measurements.

<u>SKIFIB:</u>

```
fib n      = 1 ; n < 3
           = fib(n-1) + fib(n-2)
```

```
((S ((S ((S (K IF)) ((S <) (K 3))))
(K 1))) ((S ((S (K +)) ((S (K CYCLE))
((S -) (K 1)))))) ((S (K CYCLE))
((S -) (K 2))))))
```

<u>FIB:</u>

```
fib n      = 1 ; n < 3
           = fib(n-1) + fib(n-2)
```

```
((S (((S' IF) ((C ) 3)) (K 1)))
(((S' +) ((B CYCLE) ((C -) 1)))
((B CYCLE) ((C -) 2))))
```

<u>NFIB:</u>

```
nfib n     = 1 ; n < 2
           = 1 + nfib(n-1) + nfib(n-2)
```

<u>TAK:</u>

```
tak x y z  = z   ; not (y < x)
           = tak (tak(x-1) y z) (tak(y-1) z x)
                                 (tak (z-1) x y)
```

Table 3. Benchmark listings.

Figures for the C implementation on the VAX 8800, microVAX (VAXstation 3200), Sun 3/75, and SUN 3/260 used the gcc compiler (Stallman 1988) with the optimization switch turned on. The VAX 8800 is a 22 MHz mainframe with fast cache memory. The VAXstation 3200 is a high-end microVAX workstation. The Sun 3/75 system is a 16 MHz 68020 workstation with no cache memory. The Sun 3/260 system is a 24 MHz 68020 workstation with cache memory.

We shall now take these and other results for TIGRE performance and attempt comparisons with other important combinator evaluation strategies described in the literature. Comparisons of this type are often fraught with peril because of varying execution platforms and operating system environments. Nonetheless, we shall attempt as fair a comparison as the data currently available to us permits.

Miranda

The Miranda (Turner 1985) system is a straightforward commercial implementation of a lazy functional programming language that is based on reduction of the Turner Set of combinators.

Table 4 shows the measured performance of Miranda on a Sun 3/75 compared to TIGRE performance (in C) on exactly the same machine. It should be noted that the C performance of TIGRE can be expected to be a approximately half the speed of assembly language, based upon results with the VAX architecture.

Platform	Language	Program	Time (sec)	Speed (RAPS)
SUN 3/75	TIGRE C	FIB(23)	12.75	58000
		NFIB(20)	5.22	59000
	MIRANDA	FIB(23)	86.55	7300
		NFIB(20)	22.17	7000

Table 4. Performance of TIGRE versus Miranda.

Hyperlazy Evaluation

Hyperlazy evaluation (Norman 1988) concentrates on only the three basic combinators S, K, and I. This permits implementing combinator graph reduction that is lazy at two levels. It provides for lazy function evaluation, and also lazy updating of the graph in memory by using registers to pass small portions of the tree between combinators.

The hyperlazy evaluation scheme attempts to deal with common sequences of graph manipulation operations not by creating more complicated combinators, but rather by implementing a finite state machine that remembers the sequence of the last few combinators that have been executed. This finite state machine enforces a discipline of maintaining outputs of a combinator sequence in designated registers for use by the next combinator in the state sequence. Implementing the finite state machine involves performing a case analysis at the end of each combinator to jump to the next state based on the next combinator executed from the graph.

Problems with this finite state machine approach include a combinatorial explosion in the number of states (and therefore the number of code fragments to handle these states) as the length of the "memory" of the system is increased or as the number of combinators that is recognized by the system is increased. In the actual system, the C combinator was used in addition to S, K, and I since it resulted in significant efficiency improvements.

The reported speed for the Hyperlazy Evaluator shown in Table 5 is 4000 nfib recursions per second. This speed was measured on an Acorn Archimedes system (a RISC system) running at 8 MHz. It is difficult to draw a direct comparison between the ARM RISC processor and machines available to us, but the microVAX is probably a reasonable comparison. In any event, Hyperlazy evaluation seems to be poorly suited to supercombinators because it suffers a combinatorial explosion with an increase in the number of combinators.

Platform	Language	Program	Time (sec)	Speed (nfib/sec)
microVAX	TIGRE ASM.	NFIB(20)	1.92	11400
ARM	Hyperlazy	NFIB(?)	?	4000

Table 5. Performance of TIGRE versus Hyperlazy

The G-Machine

The G-Machine (Augustsson 1984, Johnsson 1984, Peyton Jones 1987) is a graph reducer that uses supercombinators to increase execution speed. The idea is that in most combinator reduction schemes, traversing the graph tree, performing case analysis on node tags, and performing case analysis to determine which combinator to execute are all quite expensive. Therefore, using supercombinators will speed up the system, since supercombinators reduce the number of nodes traversed and the number of combinators executed. The G-Machine is representative of the most sophisticated graph reducers developed.

A novel idea introduced by the G-Machine is the use of macro instructions to synthesize sequences of machine instructions for executing combinators. Each supercombinator is built using a sequence of G-code instructions, which are then expanded by a macro assembler into the assembly language of the target system.

Absolute performance information for the G-Machine is not presently available to us. However, there are some indicators that may be used for comparison with other implementations. The code listed by Peyton Jones (1987) for the stack unwinding operation indicates that four VAX instructions (including a doubly indirect jump with offset) are executed for each node traversed. Also, seven VAX instructions must be executed as a preamble to combinators to rearrange the stack. TIGRE uses only a single instruction for each stack element unwound (the `jsb` instruction in the VAX implementation) and requires no preamble to its combinators.

TIM

The Three Instruction Machine (TIM) (Fairbairn & Wray 1987, Wray & Fairbairn 1988) is an evolution beyond the G-Machine graph reducer into the realm of closure reducers. An important realization is that graph reducers must produce suspensions to accomplish lazy evaluation. Pointers to these suspensions are stored in the ancestor nodes to a combinator in the tree. As the left spine is traversed, the stack accumulates pointers to the ancestor nodes, forming a list of pointers to the suspension elements. TIM goes a step further, and copies the top stack elements to a memory location so that they form a closure. This closure is simply a tuple of elements forming a vector of data in the memory heap.

The driving force behind TIM is to make closures inexpensive to create and manipulate. But, since the cost of traversing the spine is not free, and since the cost of manipulating graphs is not free, TIM also uses supercombinators to reduce the number of closures that must be created and manipulated. Costs are greatly reduced by executing code that pushes pointers directly onto a stack instead of traversing a graph that incurs overhead for each node to accomplish this same building of a pointer list on the stack. An important efficiency consideration with TIM is that it first builds closures by pushing elements onto a stack, then moves the top stack elements into a closure created from heap memory. This is equivalent in cost to a context switch (where a set of registers are copied out to memory when switching tasks) for each invocation of a combinator (Wray 1988).

The closure building on top of a stack is roughly analogous to a machine using a set of register windows. This is not an accident. TIM is the result of an evolution of software techniques that have transformed the representation of the combinator graph reduction problem from one of interpreting a combinator graph to one of executing sequences of inline code using register windows which contain groups of arguments. In other words, TIM shows how the graph reduction problem can be made to fit conventional hardware and software techniques. Since TIM is optimized for the use of conventional software and hardware techniques, it is unlikely that TIM performance can be significantly improved by the use of any special purpose hardware, beyond that available in a well designed general purpose RISC processor. TIM is probably the most advanced combinator reduction scheme thus far described.

Published performance information for TIM, shown in Table 6, includes a rating of nfib(20) as 1.21 seconds on an Acorn RISC Machine (Fairbairn & Wray 1987). This number includes sharing and some strictness analysis (Hudak & Goldberg 1985). Without such analysis (but still with the use of supercombinators), TIM executes nfib(20) in 1.96 seconds. This is compared to TIGRE performance with Turner Set combinators on a microVAX of 1.92 seconds, which indicates rough comparability in performance. However, TIGRE may improve significantly when used with supercombinators, as discussed later.

Platform	Language	Program	Time (sec)	Speed (nfib/sec)
microVAX	TIGRE ASM.	NFIB(20)	1.92	11400
ARM	TIM supercomb.	NFIB(20)	1.96	11200
	TIM optimized	NFIB(20)	1.21	18100

Table 6. Performance of TIGRE versus TIM.

NORMA

The Normal Order Reduction MAchine (NORMA) (Scheevel 1986) is widely acknowledged to be the highest performance special purpose hardware for combinator graph reduction. Among NORMA's features are a 370-bit wide microinstruction, five cooperating processors, a 64-bit wide memory bus, a 64-bit wide hardware spine stack, and extensive use of semicustom chips to optimize performance. NORMA uses a highly structured node representation that includes five tag fields in addition to two data fields. NORMA also uses some of its processors to perform garbage collection operations and heap allocation in parallel with node processing and arithmetic operations. NORMA uses the Turner Set of combinators to accomplish graph reduction.

NORMA is rated at 250,000 raps. This is approximately 25 times faster than the C-based interpreter run on a VAX 11/780 previously used by the NORMA development group. Table 7 shows NORMA performance compared to TIGRE performance on a high-end VAX system. TIGRE is significantly faster than NORMA for FIB(23) in both elapsed time and raps.

Platform	Language	Program	Time (sec)	Speed (RAPS)
VAX 8800	TIGRE ASM.	FIB(23)	2.10	355000
NORMA	NORMA	FIB(23)	3.1	250000

Table 7. Performance of TIGRE versus NORMA.

Analysis

We can see that even preliminary results are quite promising. The VAX 8800 implementation of TIGRE is (as far as is known to us) faster than all other reported combinator graph reducers. TIGRE is close enough in performance on comparable hardware to TIM, which is reputed to be the fastest closure reducer, that the outcome of a comparison between TIM and TIGRE is unclear.

Further Work

One of the problems with using the Turner set of combinators is that the grain size of computations is small. We are implementing supercombinators on TIGRE. Preliminary results of this work show that even the simplest of supercombinator compilation schemes gives a factor of two or more speedup over Turner Set combinators. The definition of a TIGRE assembly language is allowing us to create a fully automatic supercombinator compiler. Further sharing and strictness analysis will probably result in more dramatic speed increases for those program which do not require fully lazy evaluation. Results of applying advanced compiler techniques to TIGRE are in preparation (Lee & Koopman 1989).

A significant amount of work remains to be done in the performance analysis area. Performance metrics other than simple reduction applications per second are desirable. We feel that it is important to distinguish the effects of different abstract machines, hardware, and compiler technology on performance. Since the combinator reduction techniques used by TIGRE are quite different from the typical code executed by general purpose processors, an architectural study of how design tradeoffs in conventional machines influence suitability for combinator graph reduction is also forthcoming (Koopman, Siewiorek & Lee 1989).

Conclusions

Through its simplicity, TIGRE gives fundamentally better insight into the operation of graph reducers. As evidence of this, TIGRE is able to match or improve upon the execution speed of all previous reported combinator reduction schemes. With relatively simple complier optimizations, TIGRE has the potential to become faster than closure reducers. One thing is clear: there is still enough unknown area in the graph reducer design space to prevent abandoning graph reducers in favor of closure reducers.

References

Augustsson, L. (1984) A compiler for lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin*, pp. 218-27, August.

Baker, H. (1978) List processing in real time on a serial computer. *Communications of the ACM*, 21(4) 280-294, June.

Bell, J. (1973) Threaded code. *Communications of the ACM*, 16(6) 370-372, June.

Fairbairn, J. & Wray, S. (1987) TIM: A simple, lazy abstract machine to execute supercombinators. In Kahn, G. (ed.), *Proceedings of the Conference on Functional Programming and Computer Architecture, Portland*, pp. 34-45, Springer Verlag, 1987.

Hudak, P. & Goldberg, B. (1985) Serial combinators. In *Conference on Functional Programming Languages and Computer Architecture, Nancy*, Springer Verlag, pp. 382-399.

Hughes, R. J. (1982) Supercombinators: a new implementation method for applicative languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh*, ACM, August 1982.

Johnsson, T. (1984) Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction, Montreal*, pp. 58-69, June.

Koopman, P., Siewiorek, D. & Lee, P. (1989) Cache performance of combinator graph reduction. Submitted to the *1989 Conference on Functional Programming Languages and Computer Architecture*.

Lee, P. & Koopman, P. (1989) Compiling for direct execution of combinator graphs. Submitted to the *1989 Conference on Functional Programming Languages and Computer Architecture*.

Norman, A. C. (1988) Faster combinator reduction using stock hardware. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird Utah*, pp. 235-243, ACM, July.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*, Prentice-Hall, London.

Scheevel, M. (1986) NORMA: A graph reduction processor. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, Cambridge Massachusetts*, pp. 212-219, ACM, August.

Stallman, R. (1988) GNU Project C Compiler. In *UNIX Programmer's Manual*, on-line system documentation, Unix version 4.3.

Stoye, W. (1984) *The Implementation of Functional Languages using Custom Hardware*, Technical Report No. 81, University of Cambridge Computer Laboratory.

Turner, D. A. (1979a) A new implementation technique for applicative languages. *Software - Practice and Experience*, 9(1) 31-49, January.

Turner, D. A. (1979b) Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2) 67-270.

Turner, D. A. (1985) Miranda - a non-strict functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture, Nancy*, Springer Verlag, pp. 1-16.

Wray, S. C. & Fairbairn, J. (1988) Non-strict languages — programming and implementation (draft), October 16, 1988.

Wray, S. C. (1988) Private communication, October 24.

SIGPLAN
NOTICES
Volume 24
Number 7
July 1989

acm
PRESS

Proceedings of the

SIGPLAN '89 Conference on

Programming Language Design and Implementation

Portland, Oregon
June 21–23, 1989

Sponsored by the
Special Interest Group on Programming Languages,
Association for Computing Machinery