Compiling for Direct Execution of Combinator Graphs on Sequential Architectures

Peter Lee
School of Computer Science
and
Philip J. Koopman, Jr.
Department of Electrical and Computer Engineering

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We have designed and implemented an abstract machine called TIGRE for combinator graph reduction. TIGRE supports an assembly language that can be easily expanded into native code for both conventional and special-purpose architectures. Furthermore, optimizations can be made over the assembly instructions that take advantage of special features of the architecture, such as long instruction pipelines and caches. The results of compile time analyses such as strictness analysis can also be usefully incorporated.

In this paper we describe the compilation of combinators into TIGRE assembly code and subsequent expansion into target-machine code. We demonstrate the retargetability of the compile-time optimizations by generating C code, MIPS R2000 assembly code, and VAX assembly code. We present benchmark results for a DECstation 3100, VAX 8800, and Sun 3/75. On the MIPS R2000, we have attained sustained rates of nearly 500,000 reductions per second for the Turner set of combinators. Supercombinators with strictness analysis improve these running times by additional factors of three to four.

1. Introduction

In recent years, the technology for compiling supercombinators has developed considerably. The work done on compiling for TIM [5] and the G-machine [8,3] in particular have provided many new insights. Perhaps the fundamental insight shared by both efforts is that supercombinators can be compiled into an abstract machine code which in turn can be expanded into efficient native machine code, thereby avoiding much of the costly interpretation of combinator strings or graphs that is normally required.

Research supported in part by the Office of Naval Research under contract N00014-84-K-0415, 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, and in part by NASA/Goddard under contract NAG-5-1046. 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.

We have also designed and implemented an abstract machine for combinator reduction called TIGRE that, like TIM and the G-machine, supports an "assembly language" that can be easily expanded into native code for both conventional and special-purpose architectures. Furthermore, optimizations can be made over the assembly instructions in order to take advantage of special features of the architecture, such as long instruction pipelines, caches, and so on. The results of compile time analyses such as strictness analysis [6,7] can also be usefully incorporated.

TIGRE is a graph reduction machine that is highly suited to the *direct execution* of combinator graphs. (The manner of this "direct execution" is briefly described below, and more fully in [9].) Although the "directness" of a reduction technique is not a quantifiable notion, we claim that the TIGRE reduction of combinators is more direct (and hence less interpretive) than either the TIM or G-machine approaches. In our experiments with TIGRE, we have attained sustained reduction rates of nearly 500,000 combinator reduction applications per second for the standard "Turner set" of combinators [2], on a MIPS R2000 processor [4], running at 16.67 Mhz. We get similarly good results for supercombinators and other architectures, including the Vax and Motorola 68000.

This paper reports on our progress thus far in developing combinator compilation techniques for TIGRE. In this effort, we have three main goals:

1. *Speed.* The compiled combinators should exploit special features of the target machine hardware as much as possible. This means that the compilation methods should allow incorporation of optimizations to take advantage of pipelines, caches, multiple processors, and so on.

   Our experiments indicate that the efficiency of graph reduction is highly dependent on effective use of a cache [10]. Thus, it is important for the compilation scheme to be able to efficiently exploit a cache if it exists.

2. *Retargetability.* It should be straightforward to retarget the implementation to various target machine architectures (including both CISC and RISC), and to various target languages.

3. *Parallelizability.* It should be possible to exploit parallel architectures in an effective way.

We have not yet explored the issue of parallelism to any great extent. Hence, this paper addresses only the issues of speed and retargetability.

We begin now with a brief description of the TIGRE abstract machine. Then, we present an assembly language for TIGRE, and show how various optimizations can be performed on combinators specified in the language. By performing optimizations on the assembly language, we hope to gain a high degree of retargetability. Next, we present the results of some benchmarks, and discuss further optimizations for supercombinators, such as strictness analysis. Finally, we conclude with our plans for future work.

2. The TIGRE Abstract Machine

The TIGRE abstract machine is described fully in [9]. Here, we provide only a brief overview.

One of the most awkward aspects of graph reduction is the need to traverse the left spine of a graph, in the process "unwinding" the right children onto what is often referred to as the "spine" stack. Besides

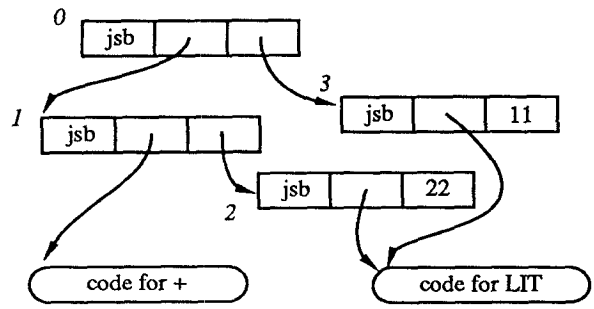

Figure 1: Representation of the combinator graph for (+ 22 11) in TIGRE

forcing one to implement a case analysis on graph-node tags, it seems also to require some kind of "control program" to control the traversal. This is unfortunate, since the program that we are actually interested in executing is essentially embedded in the graph, and so the control program really ends up being an interpreter. Hence, in this scheme we seem forced to accept the efficiency penalties involved with interpretation as opposed to direct execution.

One way to avoid this is to not bother with building graphs in the first place, but rather execute sequences of instructions that build the closures that would be represented by the left spines during a graph reduction. This is basically the approach taken by TIM, and is likely a main reason for its good efficiency. However, TIM is quite subtle, and still incurs a penalty from the fact that it must build and copy closures. In TIGRE, we seek to retain a straightforward approach to graph reduction while eliminating the interpretive overhead.

The key insight underlying TIGRE is that the graph is itself a program with two classes of instructions: pointer instructions and combinator instructions. Graph reduction then becomes a process of executing a self-modifying, threaded program which resides in the node heap. That is to say, the graph is a program that consists mainly of subroutines calls (*i.e.*, pointer instructions). One call leads to another call, which then leads to another, and so on until, finally, some other executable code (*i.e.*, a combinator definition) is found. Threaded code is not a new concept, but we are not aware of any previous applications of this idea to combinator reduction.

The important consequence for implementation on most conventional architectures is the following: *the spine stack is actually a subroutine return stack* for the threaded program. Figure 1 shows a schematic of a combinator graph in TIGRE. As control flows from node 0 to node 1, and then to the code for the + primitive, 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 a subroutine call instruction, then the return addresses that are automatically saved are the right-hand cell addresses of the spine of the graph, which is exactly the desired behavior.

When the combinator is entered, all of the combinator's arguments are available on the spine stack (which is the subroutine return stack). A pleasant aspect of this scheme is that there is now only one type of data in the graph—the pointer. Hence, there is *no conditional branching or case analysis required at*

runtime. All nodes contain either pointers to other nodes or pointers to combinator code.

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 opcode, 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. Evaluation of a program graph is initiated by performing a subroutine call to the `jsb` node at the root. (For our example, we use the the Vax `jsb` instruction which implements a fast subroutine call.) The machine's program counter then traverses the left spine of the graph structure by executing the `jsb` instructions of the nodes, each of which uses a left child subgraph pointer as its operand field. When a node points to a combinator, the machine 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 the 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-provided subroutine call instructions to do the stack unwinding. 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, in particular TIM.)

3. An Assembly Language for TIGRE

A set of combinators is thus very much like the kernel definition of a threaded execution language, such as Forth. Graph traversal is equivalent to threading, and the combinator definitions specify the actual computation. The TIGRE assembly language provides a way to specify the combinator actions in a manner consistent with this threaded-code discipline.

In terms of architecture, the TIGRE abstract machine has a spine stack, graph node heap, combinator definition memory, and a set of scratch registers. Tying these together are some control logic, an ALU, and a register for controlling the the graph traversal, which we call the interpretive pointer, or `ip`. The names of these elements of the machine are summarized below.

`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 is "auto-popped" after the access.

`temp`*n*	Scratch 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 scratch node register `temp`*n*.
`result`	A temporary result register for returning the value of strict computations.

There are three classes of operands:

i Immediate data.

n Node reference.

p Pointer.

The difference between node reference operands (*n*) and pointers (*p*) is important in operands involving the scratch node registers. For example, the operand `Rtemp1` in an *n* context refers to the right child of the node pointed to by the register `Rtemp1`. In a pointer, or *p*, context, this operand refers to the address of the right half of the cell pointed to by `Rtemp1`.

Node references may also be the names of supercombinator bodies, in which case the address of the code for the combinator is used.

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*.
`mov` *n, p*	Change the pointer *p* so that it points to *n*.
`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*	Reduce the graph rooted at the node *n*, leaving the result in the `result` register.
`return`	Thread and pop through the top of the spine stack, restarting execution just beyond the most recent evaluation.

In addition to these, there are the usual complement of instructions for conditional execution, arithmetic, and other strict operations.

4. Compiling and Optimizing TIGRE Code

The control flow, or threading, through a program is given implicitly by its graph structure. Thus, we can focus on the problem of compiling combinator definitions into TIGRE assembly code, and then expanding this into efficient target code. In order to obtain a high degree of retargetability, we have attempted to

perform as many of the optimizations as possible on the TIGRE code (as opposed to the target machine code), so that only a simple assembler need be implemented for each new target machine.

For the purposes of this section, we restrict our attention to SK-style combinators [15]. In what follows, we step through the compilation of Turner's S′ combinator. The case of supercombinators [12] is described in a later section. We consider compilation to the following targets:

- C code running on a Motorola 68020 (in a Sun 3/75), a Vax 8800, and a MIPS R2000 (in a DECstation 3100).
- Assembly code for a Vax 8800.
- Assembly code for a MIPS R2000.

Note that the Vax 8800 supports an long instruction pipeline, and both the VAX 8800 and MIPS R2000 machines support instruction and data caches. The speed of graph reduction depends heavily on the design of the caches, as described in [10]. These two machines were chosen in order to test TIGRE's efficiency on both CISC and RISC architectures.

4.1. Compiling combinators into TIGRE assembly code

The S′ combinator is defined as follows:

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

or, equivalently as

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

This code can be straightforwardly compiled into TIGRE assembly code. Our compiler produces as a first step the following code when given this combinator rule:

```
allocate 3              ; temp0, temp1, and temp2 get
                        ; the pointers to the new cells.
mov R1, Ltemp0          ; (f x)
mov R3, Rtemp0
mov R0, Ltemp1          ; (c (f x))
mov temp0, Rtemp1
mov R2, Ltemp2          ; (g x)
mov R3, Rtemp2
mov temp1, L3           ; (c (f x)) (g x)
mov temp2, R3
pop 3
mov temp1, ip
thread
```

After allocating three new nodes, each one is constructed by retrieving the appropriate arguments from either the spine stack or the scratch registers. Then, the arguments are popped off of the stack and the **ip** set to the new left spine to traverse.

As a simple optimization, our compiler reuses the top of the spine stack whenever possible, so that a node traversal can be elided. The last three instructions are thus replaced by the following:

```
mov Ltemp1, ip
pop 2
top Rtemp1
thread
```

4.2. Optimizing TIGRE assembly code

Although still nonoptimal, this code can be used without further analysis. Our TIGRE-to-VAX expansion algorithm generates the following VAX assembly code for this:

```
                                # allocate 3
[standard heap allocation/garbage collection sequence goes here...]
movl (r3), r0                   # (cache pre-touch)
movl *4(sp), (r3)               # mov R1, Ltemp0
movl *12(sp), 4(r3)             # mov R3, Rtemp0
movl *0(sp), (r4)               # mov R0, Ltemp1
movab -2(r3), 4(r4)             # mov temp0, Rtemp1
movl *8(sp), (r5)               # mov R2, Ltemp2
movl *12(sp), 4(r5)             # mov R3, Rtemp2
movab *12(sp), r0               # mov temp1, L3
movab -2(r4), -4(r0)
movab -2(r5), (r0)              # mov temp2, R3
movl (r4), r9                   # mov Ltemp1, ip
addl2 $8, sp                    # pop 2
movab 4(r3), (sp)               # top Rtemp1
jmp (r9)                        # thread
```

The heap allocator puts the addresses of the new nodes into VAX machine registers `r3`, `r4`, and `r5`. The VAX's call/return stack, `sp`, is used for the spine stack. The TIGRE `ip` register is maintained in VAX register `r9`. The purpose of the initial instruction, "`movl (r3), r0`," is to defeat the write-no-allocate cache policy used by some VAX computers. This optimization is discussed in detail in [10].

Although workable, quite a bit of improvement is still possible. By building a dependency graph of the node and register assignments, the compiler can sort the references to the spine, obtaining the following:

```
allocate 3
mov R0, Ltemp1
mov Ltemp1, ip
mov R1, Ltemp0
mov R2, Ltemp2
mov R3, Rtemp0
mov temp0, Rtemp1
mov R3, Rtemp2
mov temp1, L3
mov temp2, R3
pop 2
top Rtemp1
thread
```

For sorting purposes, the `ip` register is treated as a spine reference which lexicographically precedes `R0`. On a VAX, this optimization enables the use of auto-incrementation on the spine stack pointer. On other architectures, it permits the use of a small set of registers to cache spine references.

Next, transitive data-movement chains involving TIGRE ALU registers (such as `ip`) as final targets are rewritten so as to improve register use. This takes advantage of target-machine register transfers which are typically faster than memory-to-memory transfers. Thus, the second and third instructions are rewritten as follows:

```
mov R0, ip
mov ip, Ltemp1
```

Finally, an attempt is made to arrange for writes to the left and right sides of nodes to happen in immediate succession (but not at the expense of upsetting the spine stack access pattern). This improves memory bus behavior on some machines with wide write buffers.

```
allocate 3
mov R0, ip
mov ip, Ltemp1
mov temp0, Rtemp1
mov R1, Ltemp0
mov R2, Ltemp2
mov R3, Rtemp2
mov R3, Rtemp0
mov temp1, L3
mov temp2, R3
pop 2
top Rtemp1
thread
```

4.3. Expanding TIGRE assembly code into target-machine code

At this point, we introduce target-machine-dependent optimizations. For a VAX or other architectures with auto-incrementing address modes, the optimizer introduces auto-incrementation:

```
allocate 3
mov R+, ip
mov ip, Ltemp1
mov temp0, Rtemp1
mov R+, Ltemp0
mov R0, Ltemp2
mov R1, Rtemp2
mov R1, Rtemp0
mov temp1, L1
mov temp2, R1
top Rtemp1
thread
```

This is then expanded into the following optimized VAX code:

```
movl (r3), r0           # (cache pre-touch)
movl *(sp)+, r9         # mov R+, ip
movl r9, (r4)           # mov ip, Ltemp1
movab -2(r3), 4(r4)     # mov temp0, Rtemp1
movl *(sp)+, (r3)       # mov R1, Ltemp0
movl *0(sp), (r5)       # mov R2, Ltemp2
movl 4(sp), r7          # mov R3, Rtemp2
movl (r7), 4(r5)
movl (r7), 4(r3)        # mov R3, Rtemp0
movab (r7), r0          # mov temp1, L3
movab -2(r4), -4(r0)
movab -2(r5), (r0)      # mov temp2, R3
movab 4(r3), (sp)       # top Rtemp1
jmp (r9)                # thread
```

For C code output, auto-incrementation is not as useful, since many architectures do not have auto-incrementation, so instead the compiler attempts to reorder stack pops so that all references to the spine stack are no deeper than one below the top of stack.

```
allocate 3
mov R0, ip
mov ip, Ltemp1
mov temp0, Rtemp1
mov R1, Ltemp0
pop 2
mov R0, Ltemp2
mov R1, Rtemp2
mov R1, Rtemp0
mov temp1, L1
mov temp2, R1
top Rtemp1
thread
```

This allows references to the spine stack to be cached by two target-machine registers (which we refer to as "me" and "parent"). The expansion to C is as follows:

```
case DO_SPRIME:
New_Node(3);                        /* allocate 3 */
Use_Me;                             /* mov R0, ip
                                       mov ip, Ltemp1 */
Ltemp2 = ip = Rme;
Rtemp2.child = TARGET(temp1);       /* mov temp0, Rtemp1 */
Use_Parent;                         /* mov R1, Ltemp0 */
Ltemp1 = Rparent;
Pop_Spine(2);                       /* pop 2 */
Use_Me;                             /* mov R0, Ltemp2 */
Ltemp3 = Rme;
Use_Parent;                         /* mov R1, Rtemp2
                                       mov R1, Rtemp0 */
Rtemp3 = Rtemp1 = Rparent;
Lparent.child = TARGET(temp2);      /* mov temp1, L1
Rparent.child = TARGET(temp3);      /* mov temp2, R1 */
*(spine_ptr) = temp2+1;             /* top Rtemp1 */
continue;                           /* thread */
```

Here, the C macros `Use_Me` and `Use_Parent` cache the spine stack references into registers for subsequent faster access. Note that since C does not permit programs to access the call/return stack, an interpretive loop must be used. Hence the use of C's `case`/`continue` construct.

This version of the TIGRE code is also used for the expansion to MIPS R2000 assembly code.

```
                                    /* allocate 3 */
[standard heap allocation/garbage collection sequence goes here...]
/* $16 = Spine stack pointer
   $18 = temp0
   $19 = temp1
   $20 = temp2
 */
lw      $21, 0($16)      /* mov R0, ip */
lw      $10, 0($21)
sw      $10, 0($19)      /*  mov ip, Ltemp1 */
lw      $17, 4($16)      /* mov R1, Ltemp0 */
lw      $8, 0($17)
sw      $8, 0($18)
addu    $16, $16, 8      /* pop 2 */
lw      $21, 0($16)      /* mov R0, Ltemp2 */
lw      $9, 0($21)
sw      $9, 0($20)
lw      $17, 4($16)      /* mov R1, Rtemp2 */
lw      $8, 0($17)
sw      $8, 4($20)
sw      $8, 4($18)       /* mov R1, Rtemp0 */
sw      $18, 4($19)      /* mov temp0, Rtemp1 */
sw      $19, -4($17)     /* mov temp1, L1 */
sw      $20, 0($17)      /* mov temp2, R1 */
addu    $8, $19, 4       /* top Rtemp1 */
sw      $8, 0($16)
b       $THREAD          /* thread */
```

One drawback of the MIPS R2000 for TIGRE implementation is that it does not provide a single-instruction subroutine call. This forces the use of a five-instruction interpreter loop to traverse the spine.

5. Benchmarks

Our experiments indicate that combinators compiled by the method outlined in the previous section are quite efficient, in most cases outperforming even our own hand-written combinator implementations.

5.1. MIPS R000 benchmarks

The following table lists the results of benchmarks run on the MIPS R2000. All timings are given in seconds.

MIPS R2000	asm	hand asm	C	hand C	Sun 3/75 T3.0
fib(25)	4.07	4.10	12.70	13.17	36.44
nfib(25)	6.92	6.97	22.62	23.25	41.01
tak	12.42	12.43	26.28	27.70	10.12
nthprime(300)	2.60	2.65	6.33	6.53	—

The garbage collector used is based on a simple stop-and-copy algorithm [1]. The "asm" and "C" columns give the run times for the combinators compiled as described in the previous section. The "hand asm" and "hand C" columns show the timings for hand-coded combinators. The fib, nfib, and tak programs are the standard benchmarks. The nthprime program computes the 300^{th} prime number by enumeration of infinite lists. In all cases, the Turner set of SK combinators was used.

The "T3.0" column shows the timings for the same programs compiled and executed by the T3.0 system on a Sun 3/75. T is a dialect of the Scheme language [11], and hence for these purposes is essentially an strict functional language. (Since the nthprime program depends on laziness, we have not reported a T3.0 timing for this benchmark.) It is interesting to note that the combinator-based fib and nfib benchmarks implemented in MIPS assembler run much faster than the equivalent T3.0 programs (although T is running on a slower machine), and that the tak benchmark is quite competitive. In the next section, we present timings for supercombinator-based benchmarks which put TIGRE in an even more favorable light.

The MIPS assembler version averages 440,000 combinator reduction applications per second (raps), with rates on selected benchmarks reaching over 510,000 raps. This compares quite favorably to the reported reduction rate of 250,000 raps for NORMA [13], a special-purpose machine for reduction of Turner set combinator graphs. To our knowledge, NORMA was the fastest graph reducer in existence.

For the C implementation of the combinators on the MIPS R2000, TIGRE exhibits 161,000 raps. In general, we have observed that the assembly code implementations of the combinators run two to three times faster than the C implementatons.

Besides raps, another "standard" measure for combinator reducers is the number of recursive nfib calls that can be made per second. TIGRE executes approximately 35,000 nfibs per second (nps) on the MIPS R2000, when reducing the Turner set combinators. This compares well with the reported nps rating for TIM of 11,200 on an Acorn RISC machine, when reducing supercombinators. With additional strictness information, TIMs rating improves to 18,100 nps. When supercombinators and strictness analysis are used in the TIGRE version of nfib (see the next section), then TIGRE is able to achieve a rating of 117,000 nps on the MIPS R2000. We estimate that the MIPS R2000 is about three times faster than the Acorn RISC machine.

5.2. VAX 8800 and Sun 3/75 benchmarks

For completeness, we present the assembler[1] and C timings for the VAX 8800, as well as C, Miranda [14], and T3.0 timings on a Sun 3/75.

[1]Note that the VAX assembler timings presented in this table were derived from hand-crafted assembly code, due to a software problem with the compiler. This will be updated in the final paper.

VAX 8800 and Sun 3/75	VAX asm	VAX C	Sun C	Sun Miranda	Sun T3.0
fib(25)	5.45	13.60	33.08	227.15	36.44
nfib(25)	9.31	28.08	69.40	398.45	41.01
tak	16.01	32.53	78.97	138.55	10.12
nthprime(300)	3.91	7.41	17.42	135.55	—

The VAX assembly code implementation averages 308,000 raps, whereas the C VAX implementation averages 139,000 raps. The Sun C implementation gives 57,000 raps, with Miranda exhibiting 7,100 raps on the same machine. The wide variance between the Miranda and TIGRE performance for different benchmarks is likely due to the more sophisticated compilation techniques employed by the Miranda compiler.

6. Supercombinators and Strictness Analysis

Supercombinators greatly increase the granularity of reductions, thus leading to faster program execution. TIGRE is capable of reducing supercombinators, with the main added complication being that strict combinators (such as + and **if**) should usually be reduced inline, or "open-coded." This essentially amounts to writing a code generator for a strict sublanguage which, at the time of this writing, is still under development.

However, we have managed to compile and expand some TIGRE specifications of supercombinators by hand into both C and assembly language on the MIPS R2000. The following table gives the benchmark results with these supercombinator implementations.

MIPS R2000	SK asm	super asm	strict asm	SK C	super C	strict C	Sun 3/75 T3.0
fib(25)	4.07	1.52	0.95	12.70	6.02	3.42	36.44
nfib(25)	6.92	3.03	2.08	22.62	11.95	7.17	41.01

The "SK asm" and "SK C" columns repeat the timings given in the previous section for Turner set reduction on the MIPS R2000. The "super asm" and "super C" columns give timings for supercombinator reductions in the assembler and C implementations of the supercombinators, respectively. Finally, the "strict asm" and "strict C" columns show the timings for supercombinators compiled with strictness information.

For the MIPS assembler implementation, the reduction rate averages 479,000 raps. That the reduction rate does not fall off due to the extra complexity involved in each reduction is somewhat surprising to us. However, this might be simply be an artifact of these simple benchmarks.

7. Conclusions

We still have much to learn, especially with regard to supercombinator compilation. It is our expectation that many of the techniques that have been developed for supercombinator compilation for TIM and the G-machine will be applicable to TIGRE as well. In essence, the work we have presented here is a

machine-level, or architectural, study of combinator graph reduction. The higher-level concerns that are more directly related to the semantics of lazy functional languages, and that have been studied so deeply by others, have yet to be explored for TIGRE.

Still, we are quite encouraged by our results thus far, in particular TIGRE's good performance on the VAX 8800 and the MIPS R2000, both of which have very different architectures. This indicates that TIGRE is amenable to efficient implementation on many conventional architectures, and that optimizations on TIGRE abstract machine code may be designed and implemented in a target-machine independent manner. The comparisons against the T3.0 compiler give some initial indications that a supercombinator reduction rate of about 500,000 raps will allow lazy functional programming languages to be roughly the same speed as strict languages. Testing on larger benchmark programs is required in order to verify this, however.

We find the architecture of TIGRE to be simple and easy to understand. Furthermore, its implementation is straightforward and amenable to conventional compiler optimizations. We have hopes that this simplicity will make it amenable to implementation on parallel architectures.

8. References

[1] Andrew W. Appel, John R. Ellis, and Kai Li. *Real-time Concurrent Collection on Stock Multiprocessors.* Technical Report CS-TR-133-88, Princeton University, Princeton, March 1988. Also in Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation, Atlanta, June 1988.

[2] David A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2):67–270, 1979.

[3] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-machine. In *Proceedings of the 1988 Conference on Lisp and Functional Programming, Snowbird*, pages 244–258, ACM, 1988.

[4] Digital Equipment Corporation. *DECstation 3100 Technical Overview (EZ-J4052-28).* Digital Equipment Corporation, Maynard, Massachussetts, 1989.

[5] Jon Fairbairn and Stuart Wray. Tim: a simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture, Portland*, pages 34–45, Springer-Verlag, 1987.

[6] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 300–314, ACM, January 1985.

[7] Paul Hudak and Jonathan Young. *A Set-Theoretic Characterization of Function Strictness in the Lambda Calculus.* Research Report YALEU/DCS/RR-391, Department of Computer Science, Yale University, July 1985.

[8] Thomas Johnsson. The G-machine: an abstract machine for graph reduction. In *Proceedings of the SERC Declarative Programming Workshop at UCL*, Springer-Verlag, 1983.

[9] Philip Koopman and Peter Lee. A fresh look at combinator graph reduction. In *SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon*, ACM Press, June 1989. To appear.

[10] Philip J. Koopman, Daniel P. Siewiorek, and Peter Lee. Cache performance of combinator graph reduction. In *Submitted to the 1989 Conference on Functional Programming and Computer Architecture*, Springer-Verlag, Berlin, 1989.

[11] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: an optmizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, Palo Alto*, pages 219–233, ACM Press, June 1986.

[12] R. J. M. Hughes. Super-combinators: a new implementation method for applicative languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh*, ACM, August 1982.

[13] Mark Scheevel. NORMA: a graph reduction processor. In *Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming, Cambridge*, pages 212–219, ACM, August 1986.

[14] David A. Turner. Miranda: a non-strict functional lanugage with polymorphic types. In *Functional Programming Languages and Computer Architecture*, Springer-Verlag, Berlin, September 1985.

[15] David A. Turner. A new implementation technique for applicative languages. *Software–Practice and Experience*, 9:31–49, 1979.