Chapter 5
TIGRE Performance

Obtaining accurate and fair performance measurement data is difficult to do in any field of computing. In combinator reduction, performance measurement is further hindered by a lack of commonly used benchmark programs, a lack of statistics about program execution characteristics (at least partially because slow execution speed makes developing large programs difficult), and poor reporting methods in the available literature.

Unfortunately, solving the problem of accurate and consistent performance measurement and reporting will take time and consensus among those doing research in this area. Therefore, the results presented in this chapter reflect the limitations of available resources. Performance for TIGRE has been measured for several programs on a wide variety of platforms, which should help others publish comparisons to TIGRE.

Section 5.1 presents the results of TIGRE performance measurements on a variety of hardware platforms. Section 5.2 compares some of these measurements with published benchmark results for other combinator reduction strategies. Section 5.3 compares TIGRE execution speeds with other languages. Section 5.4 provides a brief analysis of the performance results.

5.1. TIGRE PERFORMANCE ON VARIOUS PLATFORMS

The performance of TIGRE can, in turn, only be compared to available performance measures for other machines. The available performance measures vary, and include reduction applications per second (RAPS, which may be thought of as the number of Turner Set combinators executed per second), and nfib recursions per second (the number of increments performed by the recursive nfib benchmark per second). These performance metrics are far from ideal, but are all that are available in the way of information about other implementations. The comparisons of TIGRE performance to other methods are as fair and accurate as possible, with careful attention paid to selecting an ap-

Platform	Language	Program	Time (sec)	Speed (RAPS)
DECstation 3100	Assembler	SKIFIB(23)	2.20	495000
(16.7 MHz)		FIB(23)	1.58	470000
		NFIB(23)	2.68	484000
		TAK	12.58	420000
		NTHPRIME(300)	2.60	364000
		QUEENS(20)	5.63	433000
	C	SKIFIB(23)	6.55	166000
		FIB(23)	5.08	147000
		NFIB(23)	9.02	144000
		QUEENS(20)	15.80	154000
VAX 8800	Assembler	SKIFIB(23)	2.82	387000
(22 MHz)		FIB(23)	2.10	355000
		NFIB(23)	3.55	366000
		TAK	16.07	329000
		NTHPRIME	3.91	242000
		QUEENS(20)	8.33	293000
	C	SKIFIB(23)	6.50	168000
		FIB(23)	5.01	149000
		NFIB(23)	9.13	142000
		QUEENS(20)	18.34	133000
VAXstation 3200	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	C	SKIFIB(23)	8.62	126000
(24 MHz)		FIB(23)	7.01	105000
		NFIB(23)	12.37	105000
SUN 3/75	C	SKIFIB(23)	14.62	75000
(16 MHz)		FIB(23)	12.75	58000
		NFIB(23)	22.02	59000
Cray Y-MP	C	SKIFIB(23)	3.09	352000
(167 MHz)		FIB(23)	2.40	310000
		NFIB(23)	4.25	305000
		TAK	14.69	360000
		NTHPRIME(300)	3.40	277000
RTX 2000 (10 MHz)	Forth	SKIFIB	–	450000

Table 5-1. TIGRE performance on a variety of platforms.

propriate level of compiler technology (*i.e.* whether supercombinator compilation is used), and hardware platform performance levels.

5.1.1. TIGRE Performance for the Turner Set

Table 5-1 shows the performance of TIGRE using Turner Set combinators. 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 5-2 shows source code for the smaller benchmarks along with S-expression representations for some of the compiled program graphs. The fib benchmark is a doubly recursive implementation of the Fibonacci sequence. The nfib benchmark is similar to fib, except that it returns the number of recursions taken in computing the n^{th} Fibonacci number instead of the actual Fibonacci number. Tak is a test of recursive function calls with input arguments of (18,12,6). The nthprime benchmark computes the n^{th} prime number using implicit coroutines to implement an infinite-length list of prime numbers, and performs a large number of integer divisions. The queens benchmark finds the n^{th} board solution to the 8 queens problem, with heavy use of list data structures.

Figures for the C implementation on VAX 8800, microVAX (VAXstation 3200), Sun 3/75, and Sun 3/260 used the gcc compiler (Stallman 1988) with the optimization switch turned on. Analysis of the generated code shows that there is little room for improvement by changing compilers. The DECstation 3100 C implementation uses the vendor-supplied MIPS C compiler. The DECstation 3100 assembly language implementation uses hand-scheduled assembly language to eliminate almost all of the many NOP instructions caused by load- and branch-delay slot restrictions that are present in the C compiler-generated code.

The DECstation 3100 is a 16.67 MHz workstation using the MIPS R2000 processor (Digital Equipment Corporation 1989). The RTX 2000 is a 10 MHz 16-bit stack-based processor (Harris Semiconductor 1989). The VAX 8800 is a 22 MHz mainframe with cache memory support, a wide system bus, and high-speed emitter coupled logic (ECL) circuits (Burley 1987). Only one CPU of the two CPUs available on the VAX 8800 was used. The VAXstation 3200 is a high-end microVAX workstation. The Sun 3/260 system is a 24 MHz 68020 workstation with cache memory. The Sun 3/75 system is a 16 MHz 68020 workstation with no

```
SKIFIB:
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)))))

FIB:
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))))

NFIB:
nfib n =  1   ;   n < 2
       =  1 + nfib(n-1) + nfib(n-2)

TAK:
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 5-2. Benchmark listings.

cache memory. All measurements were taken with system load as low as possible to reduce contention for resources with other users.

In order to give an idea of potential TIGRE performance on a supercomputer, times for one processor of a Cray Y-MP are also shown in Table 5-1. The Cray Y-MP is a multiprocessor supercomputer with a 167 MHz clock. The times are not as fast as one might expect given the clock speed because TIGRE performs a large number of pointer dereferences that cannot be handled efficiently by a heavily pipelined supercomputer.

The measurements for Table 5-1 are given in Turner Set RAPS. This metric is subject to some variation because not all combinators perform the same amount of work. However, the table shows that RAPS numbers vary over relatively narrow ranges across benchmark programs on each platform. Therefore, RAPS numbers for a few varied benchmarks can probably accurately represent performance on a particular hardware platform.

5.1.2. TIGRE Performance for Supercombinator Compilation

Supercombinator compilation, combined with strictness and sharing analysis, has been shown to give up to an order of magnitude speed improvement for other combinator reduction methods (Fairbairn & Wray 1987). Of course, when the complexity of a combinator definition is not held constant, the concept of simply measuring the number of combinator reductions per second directly is not very meaningful. Therefore, all results for supercombinator reduction will be expressed in terms of absolute time and a newly created metric, normalized RAPS.

Normalized RAPS (nRAPS) shall be defined as the number of reduction applications executed by a Turner Set implementation of a program divided by the execution time of the program. Thus, for a program that makes use of a supercombinator compiler, the nRAPS rating will in general be faster than the raw supercombinator RAPS, and will accurately reflect the elapsed time speedups obtained by using supercombinators instead of Turner Set combinators. For example, a program having a Turner Set RAPS rating of 400,000 and a supercombinator nRAPS rating of 1,200,000 reflects that the supercombinator version ran exactly three time faster in terms of elapsed time than the Turner Set version. Henceforth, RAPS will always refer to Turner Set RAPS, and raw reduction rates for other methods will not use the term "RAPS".

Table 5-3 shows the results of applying two levels of supercombinator compilation optimizations to the performance on the fib and nfib benchmarks. The first level of optimization was supercombinator compilation, and the second level of optimization was strictness analysis in addition to supercombinator compilation. In both cases, factors of three to four in speedup were observed when using a supercombinator compiler. That the factor of ten speed improvement observed by others on

Platform	Language	Program	Time (sec)	Speed (nRAPS)
DECstation 3100	ASM	FIB(25)	4.07	470000
	super. ASM	FIB(25)	1.52	1258000
	super. + strict ASM	FIB(25)	0.95	2014000
	ASM	NFIB(25)	6.92	470000
	super. ASM	NFIB(25)	3.03	1073000
	super. + strict ASM	NFIB(25)	2.08	1564000

Table 5-3. TIGRE speedups using supercombinator compilation.

the same benchmarks was not observed probably indicates both that the underlying TIGRE graph traversal mechanism is very efficient (so there is less inefficiency to remove by eliminating overhead for graph traversal and invocation of combinators), and that the TIGRE supercombinator compilation technology is still rather simplistic.

The supercombinator compilation results presented were derived from hand-generated supercombinators using standard algorithms (Peyton Jones 1987).

5.2. COMPARISONS WITH OTHER METHODS

This section attempts a comparison between TIGRE performance measurements and results for the fastest known or, (in the case of Miranda) most widely available implementations of combinator reduction systems. The methods chosen are the same ones describe in Section 2.2 (previous research). In all cases, every attempt is made to present results in terms of comparable hardware and compiler technology. One disappointment was the unavailability of an Acorn RISC Machine (ARM) system for use with TIGRE, since some of the European researchers report their results for this machine.

5.2.1. Miranda

Table 5-4 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 approximately one-third to one-half the speed of assembly language, based upon results with other TIGRE platforms. It is not known, but is reasonable to assume, that Miranda uses at least some assembly language support for critical operations.

Platform	Language	Program	Time (sec)	Speed (nRAPS)
SUN 3/75	TIGRE C	FIB(23)	12.75	58000
		NFIB(20)	5.22	59000
	MIRANDA	FIB(23)	86.55	8600
		NFIB(20)	22.17	13800

Table 5-4. Performance of TIGRE versus Miranda.

5.2.2. Hyperlazy Evaluation

The reported speed for the Hyperlazy Evaluator shown in Table 5-5 is 4000 nfib recursions per second. This speed was measured on an Acorn Archimedes system (using the ARM CPU) running at 8 MHz. It is difficult to draw a direct comparison between the ARM processor and other machines, but the VAXstation 3200 (a microVAX) is probably a reasonable comparison, since both machines are rated at three million instructions per second (Connolly 1987, Pountaine 1986). In any event, Hyperlazy evaluation seems to be poorly suited to supercombinators because it suffers a combinatorial explosion in machine states with an increase in the number of combinators.

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

Table 5-5. Performance of TIGRE versus Hyperlazy evaluation.

5.2.3. The G-Machine

Absolute performance information for the G-Machine has not been made widely available. 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 a single instruction for each stack element unwound (the `jsb` instruction in the VAX implementation) and requires no preamble to its combinators.

The original G-Machine was reputed to be the most efficient graph reducer for which information is available, but is general thought to be less efficient than the TIM closure reducer (which was built, in part, upon a refinement of ideas generated from the G-Machine). Newer implementations of the G-Machine, including the spineless tagless G-Machine (Peyton Jones & Salkild 1989) have improved charac-

Platform	Language	Program	Time (sec)	Speed (nfib/s)
microVAX	TIGRE ASM.	NFIB(20)	1.92	11400
	TIGRE super + strict	NFIB(20)	0.50	43800
ARM	TIM super.	NFIB(20)	1.96	11200
	TIM optimized	NFIB(20)	1.21	18100

Table 5-6. Performance of TIGRE versus TIM.

teristics created by combining concepts from graph reduction and closure reduction.

5.2.4. TIM

TIM makes use of sophisticated compiler technology for supercombinator compilation, strictness analysis, and sharing analysis.

Published performance information for TIM, shown in Table 5-6, includes a rating for nfib(20) of 1.21 seconds on an Acorn RISC Machine (Fairbairn & Wray 1987). This number includes sharing and some strictness analysis. Without such analysis (but still with the use of supercombinators), TIM executes nfib(20) in 1.96 seconds. Table 5-6 compares this performance to TIGRE running on roughly comparable hardware with Turner Set combinators, supercombinator compilation, and supercombinator with some sharing and strictness analysis (but, almost certainly, not as good an analysis as that available through the TIM compiler). From this limited data, TIGRE appears to be faster than TIM on small benchmarks.

Platform	Language	Program	Time (sec)	Speed (nRAPS)
DECstation 3200	TIGRE ASM.	FIB(23)	1.58	470000
VAX 8800	TIGRE ASM.	FIB(23)	2.10	355000
NORMA	NORMA	FIB(23)	3.1	240000

Table 5-7. Performance of TIGRE versus NORMA.

5.2.5. NORMA

At one time, NORMA was widely acknowledged to be the highest performance special-purpose hardware for combinator graph reduction. NORMA is rated by its designers at 250,000 Turner Set RAPS. Table 5-7 shows NORMA performance compared to TIGRE performance on the DECstation 3100. TIGRE is significantly faster than NORMA in terms of elapsed time and RAPS rating.

The problem with performance in NORMA is probably not the speed or capability of the hardware, but rather the underlying tag-based abstract machine implemented by the NORMA hardware.

5.3. TIGRE VERSUS OTHER LANGUAGES

In order for TIGRE to come into wide use as a research platform, it is not enough that it be faster than other methods of combinator reduction (although that is sufficient to commend its use by researchers already using functional programming languages). TIGRE should also be fast enough when compared to other programming environments that it can handle programs of the same order of complexity without undue execution speed problems. It is probably too much to hope that TIGRE will actually be faster than other methods for the general uniprocessor case, but it is reasonable to desire that TIGRE at least be within an order of magnitude of the speed of other languages. To the extent that other languages are not able to easily support lazy programming methods such as implicit coroutining, TIGRE will of course have an advantage in terms of ease of use, potential for automatic exploitation of parallelism, and perhaps speed in some cases.

Platform	Language	Program	Time (sec)	Speed (nRAPS)
SUN 3/75	TIGRE C	FIB(25)	33.38	58000
	TIGRE C	NFIB(25)	57.65	59000
	T3.0	FIB(25)	36.44	53500
	T3.0	NFIB(25)	41.01	82900

Table 5-8. TIGRE performance compared to T3.0.

5.3.1. Non-Lazy Language: T Version 3.0

T3.0 is a dialect of the Scheme language (Kranz *et al.* 1986) noted for efficient compilation and program execution. For the purposes of this discussion, T3.0 may be considered a strict functional language (*i.e.* a functional language that does not directly support lazy evaluation). Table 5-8 shows a comparison between TIGRE and T3.0 running on a Sun 3/75 in elapsed time.

This comparison should be considered rough at best, but indicates that TIGRE performance is quite close to T3.0 performance (and, TIGRE would be faster if an assembly language implementation were available for the Sun).

5.3.2. Imperative Language: MIPS R2000 C Compiler

Of course, the ultimate speed comparison is one with a mainstream imperative language such as C. If combinator reduction can (some day) be made approximately as fast as C program execution, then lazy functional programming or some other programming style that maps onto combinator reduction may become viable for widespread use in production programming. Furthermore, the potential for easy parallelization of combinator reduction methods may outweigh any speed deficiencies if the inefficiency can be limited sufficiently.

Table 5-9 shows a comparison of TIGRE and C versions of fib, nfib, and nthprime running on a MIPS R2000. The vendor-supplied R2000 C compiler was used with default optimization levels (-O2), which performs most optimizations that would help for these programs. The comparison number represents the ratio of TIGRE execution time to C execution time. Table 5-10 gives code listings for the benchmarks.

Platform	Language	Program	Time (sec)	Speed (nRAPS)
DECstation 3100	TIGRE super+strict	FIB(30)	10.58	2046000
	TIGRE super+strict	NFIB(30)	23.20	1626000
	TIGRE ASM	NTHPRIME(700)	14.93	340000
	MIPS C compiler	FIB(30)	3.27	6618000
	MIPS C compiler	NFIB(30)	5.33	7077000
	MIPS C compiler	NTHPRIME(700)	2.95	1722000

Table 5-9. TIGRE performance compared to C.

The fib and nfib benchmarks are very suitable for expression in C, and may be representative of a rather unfavorable case for comparing TIGRE against C. Nonetheless, TIGRE performance is within a factor of 5 of C performance.

The nthprime benchmark was more difficult to code in C, because it makes use of implicit coroutining. Therefore, significant modifications to the C code had to be made to arrange for a fair comparison. Hand-compilation of the supercombinator definitions for nthprime was deemed too laborious, so only a Turner Set implementation is available for this benchmark on TIGRE. Nonetheless, TIGRE is still competitive with C (and, perhaps, might be as fast as C with a supercombinator implementation running on TIGRE).

5.4. ANALYSIS OF PERFORMANCE

TIGRE executing on a MIPS R2000 is (as far as can be discerned) a very fast combinator reducer. It is faster in absolute terms than special-purpose hardware implementations of graph reduction, and is faster on comparable hardware than graph and closure reduction implementations for small benchmarks.

While the comparison of TIGRE performance to T3.0 and C performance is not comprehensive enough to be conclusive, it suggests that the TIGRE combinator reduction technique makes lazy execution of programs feasible. TIGRE appears to narrow the range of performance between fast and slow implementations of languages to less than the range of performance between commonly used hardware platforms. This makes it reasonable to study new programming languages and models of computation with respectable execution speed by using high speed workstations. Presumably, further refinements to TIGRE, especially in the area of the TIGRE compiler, will further narrow the performance gap between TIGRE and other computation models.

```
int fib(int n)
{ int newn ;
  if ( n < 3 )  newn = 1 ;
  else          newn = fib(n-1) + fib(n-2) ;
  return(newn) ;

int nfib(int n)
{ int newn ;
  if ( n < 2 )  newn = 1 ;
  else          newn = nfib(n-1) + nfib(n-2) + 1 ;
  return(n) ;
}


int sieve_number ; /* state variable for sieve */
int every_other_number ; /* state variable for
every_other */

int every_other(int n)
  {  int newn ;
    newn = n + 2 ;
    return(newn);
  }

int filter(int n) /* returns truth flag */
  { int i  ;      /* loop counter */
    int success ; /* truth flag */
    success = TRUE ;
    i = 3 ;
    while ( i < n)
    { if ( ( n % i ) ==  0 )
        { success = FALSE ;
          break ; }
      i = every_other(i) ;
    }
    return(success);
}
```

Table 5-10. C program listings for comparison with TIGRE.

```
int sieve()
  { int newn ;
    int i ; /* loop counter */
    int success ; /* success flag */
    success = FALSE ;
    while ( ! success )
    { sieve_number = every_other(sieve_number) ;
      success = filter(sieve_number) ;
    }
    return(sieve_number);
  }

int nthprime(int n)
 { int newn;
   int i ;  /* loop counter */
   every_other_number = 3 ;
   sieve_number = 1 ;
   if ( n < 1 )
    { newn = 2 ;
      return(newn);  }
   for ( i = 1 ; i <= n ; i++)
    {  newn = sieve(); }
   return(newn);
 }
```

Table 5-10. (continued)