Chapter 6
Architectural Metrics

This chapter discusses architectural metrics for the execution of TIGRE. Section 6.1 describes the cache behavior of TIGRE, including two methods of exploring the effects of varying cache parameters on TIGRE performance. Section 6.2 discusses a performance of the DECstation 3100, and uses cache simulation results to predict how cache management changes to the 3100 would affect performance. Section 6.3 describes measurements of heap memory use and stack memory use.

6.1. CACHE BEHAVIOR

During the benchmarking of TIGRE on various platforms, it became apparent that unexpected variations in performance (both good and bad) were taking place. These variations were eventually conjectured to be caused by implementation differences among platforms, especially with regard to cache management policy. In order to better understand the operation of TIGRE, a set of cache simulations was run to measure TIGRE's use of cache memory.

The first simulation experiment was an exhaustive exploration of a number of cache design parameters to search for the best combination. An exhaustive search was performed to avoid the pitfalls of hill-climbing search strategies that may become trapped at local extrema. The second simulation experiment examined the sensitivity of performance to changes in individual parameters.

6.1.1. Exhaustive Search of the Cache Design Space

An enumeration of possible combinations of parameter variations for cache design were simulated, using memory access traces of TIGRE and a trace-driven cache simulator program. The goal of the set of simulations was to explore a range of values for several independent cache parameters (such as cache size, block size, memory write policy, and replacement policy), and then pick two or three different values for each parameter. By then simulating performance on all possible combina-

tions of these parameter values, the performance of TIGRE across the entire cache design was mapped. As a result, similarities and differences between the best-performing sets of parameter combinations could lead insight into what kind of cache memory organization best supports TIGRE.

The DineroIII cache simulator program was used (Hill 1984). The simulation parameters varied were: cache size (64K and 16K bytes), cache organization (unified and split), block size (also known as line size, of 4, 8, and 16 bytes), associativity (direct-mapped and 4 way set associative), replacement policy (LRU and FIFO), write policy (write-through and copy-back), and write allocation (allocate on write miss, and no allocation on write miss). Kabakibo et al. (1987) and Smith (1982) provide more information on cache management strategies and terminology.

All meaningful combinations of parameters were run (some combinations, such as varying replacement policy on a direct-mapped cache, are meaningless). The split caches divide the available cache memory evenly between instruction and data caches, as is commonly done on real systems (*e.g.* a split 64K cache allocates 32K each to the instruction cache and data cache).

The fib(16) benchmark using the SKI combinator set (consisting of the combinators **S**, **K**, **I**, **+**, **-**, **<**, **IF**, **1**, **2**, **3**, and **LIT**) was chosen for the exhaustive design space search. The SKI set was chosen instead of the Turner Set for initial study because it was believed that the SKI set has worse performance on conventional architectures (*i.e.* it "breaks" architectures more effectively). A large enough heap was used to avoid the need to simulate garbage collection.

Table 6-1 shows the simulation results for the program skifib(16). The primary ranking is by miss ratio, which has a strong effect on program running time. Miss ratio is the number of memory accesses that result in cache misses normalized to the number of total accesses (*e.g.* 0.3000 would represent a 30% miss ratio). The secondary ranking is by bus traffic ratio. Traffic ratio is the number of words transferred on the data bus from the combination of cache misses and writes of modified cache contents to memory, normalized to the total number of accesses.

Each simulation run involved a total of 1449864 memory accesses, 1042523 of which were instruction reads, and 407341 of which were data accesses. 71.9% of all memory traffic was instruction accesses, 15.9% was memory reads, and 12.2% was memory writes. To avoid the possibility of misleading results because of an insufficiently large simulation data set size, the simulation was rerun on several data points

S = CACHE SIZE (BYTES)
U = CACHE ORGANIZATION (UNIFIED / SPLIT)
B = BLOCK SIZE (BYTES)
A = ASSOCIATIVITY (DIRECT-MAPPED / 4-WAY SET)
R = REPLACEMENT POLICY (LRU / FIFO)
T = MEMORY UPDATE POLICY (THRU / COPY-BACK)
W = WRITE ALLOCATE? (YES/NO)
1042523 INSTRUCTION, 407341 DATA ACCESSES

MISS	S	U	B	A	R	T	W	TRAFFIC
0.0096	64K	U	16	4	L	C	Y	0.0767
0.0096	64K	S	16	4	L	C	Y	0.0768
0.0096	64K	U	16	4	L	T	Y	0.1609
0.0096	64K	S	16	4	L	T	Y	0.1610
0.0097	16K	U	16	4	L	C	Y	0.0773
0.0097	16K	U	16	4	L	T	Y	0.1612
0.0098	64K	U	16	4	F	C	Y	0.0776
0.0098	16K	S	16	4	L	C	Y	0.0777
0.0098	64K	S	16	4	F	C	Y	0.0779
0.0098	16K	S	16	4	L	T	Y	0.1615
0.0098	64K	U	16	4	F	T	Y	0.1615
0.0098	64K	S	16	4	F	T	Y	0.1617
0.0101	64K	S	16	D	–	C	Y	0.0795
0.0101	64K	S	16	D	–	T	Y	0.1627
0.0102	64K	U	16	D	–	C	Y	0.0799
0.0102	64K	U	16	D	–	T	Y	0.1632
0.0104	16K	U	16	4	F	C	Y	0.0810
0.0104	16K	U	16	4	F	T	Y	0.1642
0.0105	16K	S	16	4	F	C	Y	0.0819
0.0105	16K	S	16	4	F	T	Y	0.1644
0.0129	16K	S	16	D	–	C	Y	0.0962
0.0129	16K	S	16	D	–	T	Y	0.1739
0.0192	64K	U	8	4	L	C	Y	0.0766
0.0192	64K	S	8	4	L	C	Y	0.0767
0.0192	64K	U	8	4	L	T	Y	0.1609
0.0192	64K	S	8	4	L	T	Y	0.1609
0.0193	16K	U	8	4	L	C	Y	0.0770
0.0193	16K	U	8	4	L	T	Y	0.1611
0.0194	16K	S	8	4	L	C	Y	0.0773
0.0194	16K	S	8	4	L	T	Y	0.1612
0.0195	64K	U	8	4	F	C	Y	0.0773

Table 6-1. Cache performance simulation results for TIGRE on a MIPS R2000.

MISS	S	U	B	A	R	T	W	TRAFFIC
0.0195	64K	S	8	4	F	C	Y	0.0775
0.0195	64K	U	8	4	F	T	Y	0.1614
0.0195	64K	S	8	4	F	T	Y	0.1614
0.0198	64K	S	8	D	–	C	Y	0.0783
0.0198	64K	S	8	D	–	T	Y	0.1620
0.0199	64K	U	8	D	–	C	Y	0.0786
0.0199	64K	U	8	D	–	T	Y	0.1623
0.0206	16K	U	8	4	F	C	Y	0.0800
0.0206	16K	S	8	4	F	C	Y	0.0805
0.0206	16K	S	8	4	F	T	Y	0.1636
0.0206	16K	U	8	4	F	T	Y	0.1636
0.0228	16K	S	8	D	–	C	Y	0.0873
0.0228	16K	S	8	D	–	T	Y	0.1680
0.0232	16K	U	16	D	–	C	Y	0.1347
0.0232	16K	U	16	D	–	T	Y	0.2152
0.0309	64K	U	4	4	L	C	Y	0.0581
0.0309	64K	S	4	4	L	C	Y	0.0582
0.0309	64K	U	4	4	L	T	Y	0.1533
0.0309	64K	S	4	4	L	T	Y	0.1533
0.0310	16K	U	4	4	L	C	Y	0.0583
0.0310	16K	U	4	4	L	T	Y	0.1534
0.0311	16K	S	4	4	L	C	Y	0.0585
0.0311	16K	S	4	4	L	T	Y	0.1535
0.0312	64K	U	4	4	F	C	Y	0.0586
0.0312	64K	S	4	4	F	C	Y	0.0587
0.0312	64K	U	4	4	F	T	Y	0.1537
0.0312	64K	S	4	4	F	T	Y	0.1537
0.0314	64K	S	4	D	–	C	Y	0.0589
0.0314	64K	S	4	D	–	T	Y	0.1538
0.0315	64K	U	4	D	–	C	Y	0.0590
0.0315	64K	U	4	D	–	T	Y	0.1540
0.0325	16K	S	4	4	F	C	Y	0.0606
0.0325	16K	S	4	4	F	T	Y	0.1550
0.0327	16K	U	4	4	F	C	Y	0.0604
0.0327	16K	U	4	4	F	T	Y	0.1551
0.0336	16K	S	4	D	–	C	Y	0.0621
0.0336	16K	S	4	D	–	T	Y	0.1560
0.0415	16K	U	8	D	–	C	Y	0.1233
0.0415	16K	U	8	D	–	T	Y	0.2054
0.0551	64K	U	16	4	L	C	N	0.1080
0.0551	64K	U	16	4	L	T	N	0.1608

Table 6-1. (continued).

MISS	S	U	B	A	R	T	W	TRAFFIC
0.0552	64K	S	16	4	L	C	N	0.1081
0.0552	16K	U	16	4	L	C	N	0.1082
0.0552	16K	S	16	4	L	C	N	0.1084
0.0552	64K	S	16	4	L	T	N	0.1609
0.0552	16K	U	16	4	L	T	N	0.1610
0.0552	16K	S	16	4	L	T	N	0.1611
0.0553	64K	U	16	4	F	C	N	0.1086
0.0553	64K	S	16	4	F	C	N	0.1086
0.0553	64K	S	16	4	F	T	N	0.1613
0.0553	64K	U	16	4	F	T	N	0.1613
0.0554	64K	S	16	D	–	C	N	0.1091
0.0554	64K	S	16	D	–	T	N	0.1616
0.0555	64K	U	16	D	–	C	N	0.1092
0.0555	64K	U	16	D	–	T	N	0.1619
0.0558	16K	U	16	4	F	C	N	0.1109
0.0558	16K	S	16	4	F	C	N	0.1110
0.0558	16K	S	16	4	F	T	N	0.1631
0.0558	16K	U	16	4	F	T	N	0.1633
0.0559	16K	U	4	D	–	C	Y	0.0840
0.0559	16K	U	4	D	–	T	Y	0.1783
0.0567	16K	S	16	D	–	C	N	0.1132
0.0567	16K	S	16	D	–	T	N	0.1648
0.0654	64K	U	8	4	L	C	N	0.0969
0.0654	64K	U	8	4	L	T	N	0.1499
0.0655	64K	S	8	4	L	C	N	0.0969
0.0655	16K	U	8	4	L	C	N	0.0970
0.0655	16K	S	8	4	L	C	N	0.0970
0.0655	64K	S	8	4	L	T	N	0.1499
0.0655	16K	U	8	4	L	T	N	0.1499
0.0655	16K	S	8	4	L	T	N	0.1500
0.0656	64K	U	8	4	F	C	N	0.0972
0.0656	64K	S	8	4	F	C	N	0.0973
0.0656	64K	S	8	4	F	T	N	0.1502
0.0656	64K	U	8	4	F	T	N	0.1502
0.0657	64K	S	8	D	–	C	N	0.0974
0.0657	64K	S	8	D	–	T	N	0.1503
0.0658	64K	U	8	D	–	C	N	0.0974
0.0658	64K	U	8	D	–	T	N	0.1504
0.0662	16K	S	8	4	F	C	N	0.0987
0.0662	16K	S	8	4	F	T	N	0.1513
0.0663	16K	U	8	4	F	C	N	0.0987

Table 6-1. (continued).

MISS	S	U	B	A	R	T	W	TRAFFIC
0.0663	16K	U	8	4	F	T	N	0.1515
0.0666	16K	S	8	D	–	C	N	0.0992
0.0666	16K	S	8	D	–	T	N	0.1517
0.0679	16K	U	16	D	–	C	N	0.1580
0.0679	16K	U	16	D	–	T	N	0.2100
0.0696	64K	U	4	4	L	C	N	0.0785
0.0696	64K	S	4	4	L	C	N	0.0785
0.0696	64K	U	4	4	L	T	N	0.1403
0.0696	64K	S	4	4	L	T	N	0.1403
0.0697	16K	U	4	4	L	C	N	0.0785
0.0697	16K	S	4	4	L	C	N	0.0786
0.0697	16K	U	4	4	L	T	N	0.1404
0.0697	16K	S	4	4	L	T	N	0.1404
0.0698	64K	U	4	4	F	C	N	0.0787
0.0698	64K	S	4	4	F	C	N	0.0787
0.0698	64K	U	4	4	F	T	N	0.1405
0.0698	64K	S	4	4	F	T	N	0.1405
0.0699	64K	S	4	D	–	C	N	0.0788
0.0699	64K	U	4	D	–	C	N	0.0788
0.0699	64K	S	4	D	–	T	N	0.1405
0.0699	64K	U	4	D	–	T	N	0.1405
0.0705	16K	S	4	4	F	C	N	0.0796
0.0705	16K	S	4	4	F	T	N	0.1411
0.0706	16K	U	4	4	F	C	N	0.0796
0.0706	16K	U	4	4	F	T	N	0.1412
0.0707	16K	S	4	D	–	C	N	0.0798
0.0707	16K	S	4	D	–	T	N	0.1412
0.0861	16K	U	8	D	–	C	N	0.1379
0.0861	16K	U	8	D	–	T	N	0.1906
0.0932	16K	U	4	D	–	C	N	0.1020
0.0932	16K	U	4	D	–	T	N	0.1636

Table 6-1. (continued).

from various regions of the chart with a data set ten times as large (created by running skifib with a larger input). These expanded simulations yielded identical results.

Some obviously desirable characteristics appear from an inspection of Table 6-1. The write allocation policy should be set to write-allocate, and the block size should be set to 16 bytes for good performance. There is relatively little difference among the miss ratios given near the

beginning of the table, indicating that some of the design parameters, including the cache size, have little effect on performance.

Details of the cache simulation results showed that a unified cache is slightly better than a split cache because the interpretive program was quite small. Thus, a unified cache gives more total cache memory with which to work for the data portion of the program. However, split caches will be considered to be more desirable, since most RISC processors require the extra bandwidth available from a split cache scheme. From the data in these tables, a cache design of 64K bytes, split I/D cache (giving 32K bytes each for program and data caches), 16 byte blocks, 4-way set associative, LRU replacement, copy-back, and write-allocate was chosen as the most desirable strategy based on the results of this first experiment.

6.1.2. Parametric Analysis

The initial exhaustive search of the design space gave a good starting point for determining the optimum cache design parameters. But, there was no precise indication of the sensitivity of the performance to variation in the parameters. For this reason, a second set of cache simulations was conducted to measure the performance effects of changing the parameters.

For this second set of simulations, the above-mentioned desirable cache design was used as a baseline. Individual parameters were then altered, one at a time, across a wide range to observe performance trends. The first set of simulations confirmed that the instructions needed to run the combinator reducer were almost immediately loaded into cache and stayed in cache throughout the program execution.

parameter	value		
cache organization	split I/D (32K bytes each)		
associativity	4-way set associative		
replacement policy	LRU		
memory update policy	copy-back		
write allocation	write allocate		
characteristic	**SKI set**	**Turner Set**	**Super+Strict**
miss ratio	0.0341	0.0300	0.0528
traffic ratio	0.2721	0.2209	0.4223

Table 6-2. Baseline for parametric analysis.

Therefore, the parametric analysis simulations modelled only the data accesses of the programs, and collected statistics for just the data cache (assuming a split I/D cache scheme). The baseline configuration, against which sensitivity to change was measured, is shown in Table 6-2. The benchmark program run was fib(18), with data collected for three implementations of the program: the **SKI** combinator set, the Turner Set, and supercombinator compilation with strictness analysis.

6.1.2.1. Write Allocation

Table 6-3 shows the results of varying the write allocation policy. This design decision is more important by far than any of the other design tradeoffs, with very poor cache hit ratios of 76% to 85% awaiting the user of a machine which incorporates a write-no-allocate policy. A 95% or higher cache hit ratio is generally considered desirable for most systems running conventional software.

The reason for the extreme sensitivity to write-allocation policy lies with the use of heap nodes. Graph reduction allocates nodes from a garbage-collected heap frequently during program execution. As heap nodes are allocated, the addresses of the new cells are generated without accessing heap memory (using a stop-and-copy garbage collection algorithm). After heap nodes are allocated, graph data is first written to the heap, then read back from it for further reduction operations. The first time the node is written, a cache miss is generated. A write-allocate strategy will load the node into the cache, while a write-no-allocate strategy will simply write the node value into main memory. The problem comes on the subsequent read of this node, which typically happens within a few hundred clock cycles. A write-no-allocate policy will experience a second cache miss, while a write-allocate policy will usually get a cache hit on the previously written element. This second cache miss with a write-no-allocate policy significantly degrades performance. The effect becomes even more pronounced when a long sequence

	MISS RATIOS		
Allocation Strategy	SKI set	Turner Set	Super+Strict
write allocate	0.0341	0.0300	0.0528
write no allocate	0.1914	0.1522	0.2433

Table 6-3. TIGRE performance with varying cache write allocation strategy.

of writes (each generating a cache miss) is performed in succession before the first read, as can happen when performing a sequence of graph rewrites on a small portion of the program graph.

The Turner Set numbers showed the least degradation from using write-no-allocate because it does not create a large number of superfluous nodes as the **SKI** set does (by using the **B** and **C** combinators instead of **S** and **K** combinations). But, the Turner Set does have a large number of redundant reads of elements for intermediate graph rewriting that are eliminated by the supercombinator approach, so the supercombinator version shows even more degradation in performance from using a write-no-allocate strategy.

As an example of the importance of this range of cache performances, a DECstation 3100 class machine would be 18000 RAPS slower on the Turner Set version using a write-no-allocate strategy. On a mainframe or other large processor, cache miss delays may be considerably longer, increasing this delay further. For instance, on a VAX 8800 a write-no-allocate strategy may account for a 75000 RAPS speed degradation.

The difference in write allocation policy partially explains the result that a VAX 8800 mainframe was outperformed by a DECstation 3100 workstation (376000 RAPS for the VAX 8800 compared to 475000 RAPS for the DECstation 3100). The VAX 8800 uses a write-no-allocate policy, while the DECstation 3100 uses a write-allocate policy. This greatly increases the cache miss ratio for the VAX. The difference in miss ratios is exacerbated by the fact that mainframes tend to have a longer memory access time (and therefore cache miss penalty) than workstations.

Fortunately, there is a work-around available for existing architectures that have a write-no-allocate policy. To reduce the effects of the problem on the VAX 8800, TIGRE executes a dummy read (*i.e.* a memory read, the results of which are discarded) each time a heap cell is allocated. This dummy read improves overall program Turner Set performance on skifib by approximately 20%, despite the overhead of executing extra instructions to perform the memory reads. This performance increase comes about because a read miss, a write hit, and a read hit are significantly quicker on a VAX 8800 than a write miss followed by a read miss.

Graph reduction makes extremely heavy use of a garbage-collected heap, so the effectiveness of write-allocation on cache miss ratios is quite pronounced. However, the need for a write-allocate cache policy when using garbage-collected heaps extends beyond the graph reduction domain. Since a heap, by its very nature, is used in a write-followed-

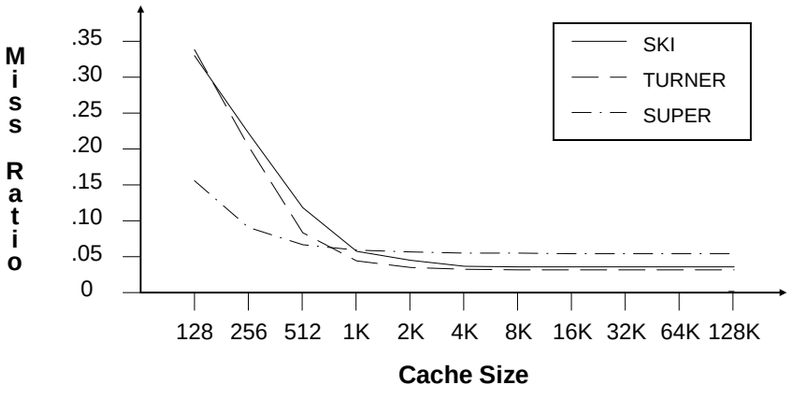

Figure 6-1. TIGRE performance with varying cache size.

by-read manner, a write-allocate cache policy is important to support any system that uses a heap.

6.1.2.2. Cache Size

Figure 6-1 shows the results of varying cache size over a range of 128 bytes to 128K bytes. Since most newer designs tend to use large cache memories to improve performance (with 64K bytes in a data cache often the minimum acceptable amount for a RISC implementation), it is surprising to see that performance for all three implementations stays at approximately 95% to 98% hit ratio with a cache as small as 2K bytes, which corresponds to only 256 graph nodes. This suggests that combinator graph reduction has much better temporal locality than conventional programs. This temporal locality may be due in part to a high infant mortality rate among allocated heap nodes.

As already noted, the supercombinator implementation strategy is more efficient at accomplishing tree rewritings (because one combinator reduction can rewrite a large section of the graph). This efficiency is reflected in a somewhat higher cache miss ratio, since fewer redundant memory accesses are used. The **SKI** and Turner Set numbers might be considered diluted by multiple accesses to intermediate result graph nodes that decrease the miss ratio, but also slow down program execution speed. Because of this effect, the data reported in this section should not be used to compare the merits of **SKI** reduction,

Turner Set reduction, and supercombinator reduction against each other, but rather to understand the effects of varying design parameters given a certain graph reduction strategy.

High temporal locality suggests that generational garbage collection techniques (Appel et al. 1988) may be useful with combinator graph reduction. It is possible that frequent changes of pointers in older generations may negate any advantages that might be gained using generational techniques , but this issue has not been explored in detail.

A word of caution on the interpretation of the cache size data collected here is in order. The benchmarks used are rather small a certain sense. They access a large amount of heap space, so it cannot be said that the programs are too small to exercise a large cache. However, only a few thousand heap nodes are actually active (*i.e.*, not garbage) at any given time during a computation, so it might be argued that good performance of small caches is a factor of running small test programs. This issue is revisited by simulating larger programs in Section 6.2.

6.1.2.3. Block Size

Figure 6-2 shows the results of varying block size over a range of 4 bytes to 8K bytes. The cache miss ratio decreases up to a cache size of 2K bytes for the **SKI** method, and up to 8K bytes (the limit to block size given 4-way set associativity) for the other methods. This suggests very strong spatial locality. This spatial locality is probably due to the fact that short-lived cells are allocated from the heap space in sequential memory locations (this sequentiality is an inherent property of compacting garbage collectors, such as the stop-and-copy garbage collector used by TIGRE).

One could, at first glance, decide to build a machine with a 2K byte cache line size based on the miss ratios alone. For conventional programs, this decision would be unwise, because the traffic ratio (the number of words of data moved by the system bus) usually increases dramatically with an increased block size. This heavy traffic can slow a system down by greatly increasing the time required to refill a cache block after a miss. With combinator graph reduction, this effect is much less pronounced. The traffic ratio does not increase appreciably until the block size is between 1K and 4K bytes in size. So, a machine with a 256 byte or 512 byte cache line size is entirely reasonable for this application.

The strong difference in traffic ratio between a block size of 4 bytes and 8 bytes on the **SKI** set and Turner Set implementation is worthy of

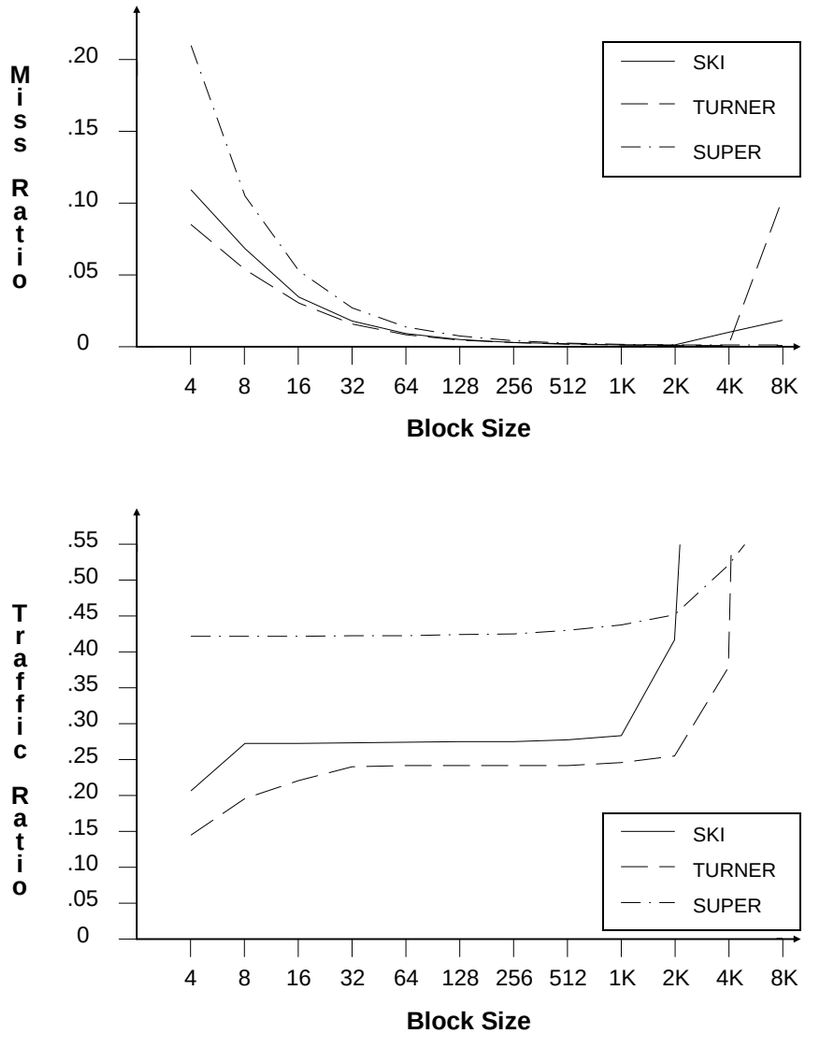

Figure 6-2. TIGRE performance with varying cache block size.

comment. This increase is caused by the fact that left-hand sides of nodes are seldom accessed after the stack unwinding operation, whereas right-hand sides are frequently accessed to first retrieve a value (perhaps using a recursive evaluation call), then to write that value. Thus, a cache which must retrieve both halves of a node wastes time fetching left-hand side values which will never be used in the case that the cache miss occurs on an access to a right-hand side. The traffic ratio on the supercombinator implementation is almost identical for the 4 byte and 8 byte block size cases, because that particular code only had one argument to evaluate for the supercombinator. The supercombinator body fetched and evaluated the argument as soon as the supercombinator body was entered, avoiding cache misses.

The fact that the miss ratio stays very low until the cache block size increases to within a factor of between four and sixteen of the total cache size gives further insight into the behavior characteristics of graph reduction. The code in this experiment tends to access approximately four to sixteen regions of memory at a time, since the miss ratio begins to climb when the 32K byte cache can hold fewer than sixteen cache blocks. This suggests excellent temporal locality.

The excellent temporal locality observed bodes well for virtual memory management behavior. Since most translation lookaside buffers are limited in size (for example, 64 entries addressing to 4K bytes each on a MIPS R2000), good spatial locality is important limit the number of TLB misses. At a second level, good spatial locality also limits thrashing of virtual memory pages between main memory and secondary storage devices. The result is that combinator graph reduction seems to provide excellent virtual memory behavior even without the use of compacting techniques (since no garbage collection was done for this example).

Block sizes of greater than 16 bytes are seldom seen in practice, because most conventional programs do not have enough spatial locality to support very large block sizes. Another consideration is that a large block size usually requires a very wide memory access bus to provide the data to fill the block. But, the significant performance increases possible with this application give strong incentive to consider large block sizes.

6.1.2.4. Associativity

Table 6-4 shows the results of varying the associativity of the cache from direct mapped (1-way associative) to 8-way associative. 2-way set

	MISS RATIOS		
Associativity	SKI set	Turner Set	Super+Strict
direct mapped	0.0358	0.0317	0.0536
2-way set	0.0341	0.0300	0.0528
4-way set	0.0341	0.0300	0.0528
8-way set	0.0341	0.0300	0.0528

Table 6-4. TIGRE performance with varying cache associativity.

associative seems to bring a slight performance improvement, but beyond that there is no discernible advantage to adding cache sets.

Many systems use direct mapped caches because they are simpler to build and can be more easily made to run at high speeds (Przybylski *et al.* 1988). The cost of using such a direct mapped cache is a only a 0.17% additional miss ratio penalty, so such a performance tradeoff of using direct mapped caches seems desirable.

6.1.2.5. Replacement Policy

	MISS RATIO		
Replacement Policy	SKI set	Turner Set	Super+Strict
LRU	0.0341	0.0300	0.0528
FIFO	0.0347	0.0305	0.0531
RANDOM	0.0349	0.0306	0.0531

Table 6-5. TIGRE performance with varying cache replacement policies.

Table 6-5 shows the results of varying the replacement policy for the cache. LRU replacement was found to be the best by a small margin. In the original simulation with both program and memory sharing a unified cache, LRU replacement was more important, since it prevented the program words from being flushed from the cache when using multi-way associativity. In a separated data cache, the performance improvement is smaller.

	MISS RATIO / TRAFFIC RATIO		
Memory Update	SKI set	Turner Set	Super+Strict
copy back	0.0341/0.2721	0.0300/0.2209	0.0528/0.4223
write through	0.0341/0.5721	0.0300/0.5431	0.0528/0.6849

Table 6-6. TIGRE performance with varying cache write-through strategy.

6.1.2.6. Write-Through Policy

Table 6-6 shows the results on miss ratio and traffic ratio for a write-through versus copy-back management policy. The cache miss ratios are the same, as expected, since this policy does not affect whether misses occur. However, the bus traffic generated for the write-through method is significantly higher than for copy-back. This can cause severe problems with system performance, even on a uniprocessor.

With a write-through policy with a line size of 16 bytes (4 words), 14.3% of data cache accesses for the **SKI** implementation generate a bus transaction. This is manageable on most systems. Unfortunately, it is more common for processors to have narrower buses to memory, with most microprocessors supporting only a 4-byte bus. In this case, a memory bus access would be generated on average on 57.2% of data accesses, which can easily swamp a bus, causing memory-bandwidth performance limitations even on uniprocessors. This bus overloading takes place because a microprocessor bus can only sustain a data transfer every 4 to 8 clock cycles, whereas a 57.2% bus access rate demands bandwidth corresponding to a transfer for every 1.7 clock cycles. Clearly, a copy-back policy is desired to limit the effects of bus saturation. Even if cache line size is reduced, similar bus write saturation effects are possible.

The supercombinator implementation has even worse bus write characteristics. This is caused by a difference in the percentage of bus write operations, since supercombinator code does less graph traversing (and hence fewer reads) per combinator. This effect is exacerbated by the fact that supercombinator compilation reduces the redundancy of computations, resulting in fewer instances of repeated overwriting of nodes. This, in turn, limits the effectiveness of the copy-back strategy (which is attenuates bus write traffic only to the extent that nodes are written more than once while the node is resident in the cache memory). Thus, with supercombinators it is even more important to use a copy-

back strategy, but even this strategy is likely to make significant demands on bus bandwidth.

6.1.3. A Desirable Cache Strategy

Based on the findings of these simulations, a cache design which minimizes complexity and cost while achieving reasonable performance would have the following characteristics: cache size of 16K bytes each for split instruction and data caches, 16 byte block size, direct mapped, write-allocate, and copy-back. This cache configuration was simulated to have a 98.94% hit ratio overall for the **SKI** method (96.24% data hit ratio, and 99.99+% instruction hit ratio), and a traffic ratio of 0.0827 words transferred on average per memory access.

Unfortunately, even though data prefetching or sub-block filling could efficiently support a block size of 16 bytes, most microprocessors in workstations support block sizes of 4 bytes. The same cache configuration with a 4 byte block size was simulated to have a 96.80% hit ratio overall (92.13% data hit ratio, and 99.99+% instruction hit ratio) with bus traffic of 0.0599 words transferred on average per memory access. This difference of 2.14% in cache hit ratio represents approximately a 44000 RAPS (nearly 10%) speed penalty for a DECstation 3100 class machine.

6.2. PERFORMANCE OF REAL HARDWARE

Section 6.1 discussed a search for a good cache strategy without regard to commercially available hardware, and using a single benchmark program (mostly because of the impracticality of collecting data for the large number of simulation runs required for multiple benchmark programs). This section reports a set of simulations with a slightly different viewpoint: a group of benchmark programs are measured, using the DECstation 3100 design characteristics as a starting point for parametric analysis. We shall be seeing another instance of using a DECstation 3100 as a basis for comparison in Chapter 7.

6.2.1. Simulation Results for a DECstation 3100

The curves and ratios measured for the DECstation 3100 are different from those shown in Section 6.1, even for the fib benchmark, since the base case for the DECstation 3100 is different than that used in Section 6.1. However, we shall see that even with a different starting point, the

parameter	value
cache organization	split I/D (64K bytes each)
associativity	direct mapped
replacement policy	n/a
memory update policy	write-through
write allocation	write allocate

characteristic	Fib	NthPrime	Queens	Real	Tak
miss ratio	0.1434	0.1768	0.1554	0.1595	0.1912
traffic ratio	0.5854	0.6262	0.5942	0.5971	0.6478

Table 6-7. Baseline for DECstation 3100 analysis.

general relationships between performance and changes in cache design parameters will, for the most part, hold true. For that reason, the results in this section may be considered a double-check on the previously discussed results.

For these cache simulations, individual parameters were altered, one at a time, across a wide range to observe performance trends. The benchmark programs run were: fib (recursive Fibonacci calculation), nthprime (a prime number generator), queens (the N-queens problem), real (infinite precision real arithmetic), and tak (a program that tests recursive function calls). All programs used the Turner Set of combinators. In all cases, between one and two million data memory accesses were simulated, with accesses to a memory range of at least 320K bytes.

The DECstation 3100 has a split cache with 64K bytes in each cache, a block size of 4 bytes, direct mapped organization, and uses a write-through strategy with write-allocate management. (Digital

	MISS RATIOS				
Allocation Strategy	Fib	NthPrime	Queens	Real	Tak
write allocate	0.1434	0.1768	0.1554	0.1595	0.1912
write no allocate	0.2405	0.3099	0.2669	0.2848	0.3271

Table 6-8. Performance with varying cache write allocation strategy.

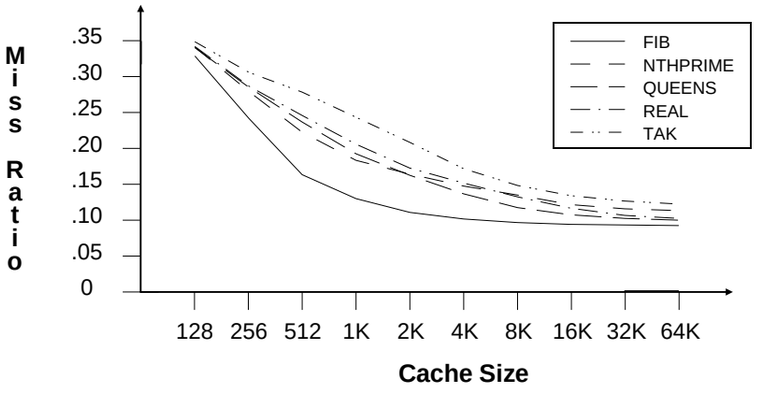

Figure 6-3. Cache performance with varying cache size.

Equipment Corporation 1989) Table 6-7 summarizes the results of simulating the baseline cache configuration of the DECstation 3100.

Two important characteristics emerge from the simulation. The cache miss ratio is a relatively high 14% to 19% for all the programs. Furthermore, the bus traffic ratio is between 0.58 and 0.65. As a result, graph reduction programs generate memory references in excess of DECstation 3100 available bus bandwidth.

Table 6-8 shows the results of varying the write allocation policy. These results show that using a write-no-allocate strategy significantly increases cache misses compared to the write-allocate case.

	MISS RATIOS				
Associativity	Fib	NthPrime	Queens	Real	Tak
direct mapped	0.1434	0.1768	0.1544	0.1595	0.1912
2-way set	0.1425	0.1724	0.1515	0.1530	0.1858
4-way set	0.1425	0.1724	0.1514	0.1530	0.1857
8-way set	0.1425	0.1724	0.1513	0.1530	0.1857

Table 6-9. Cache performance with varying cache associativity.

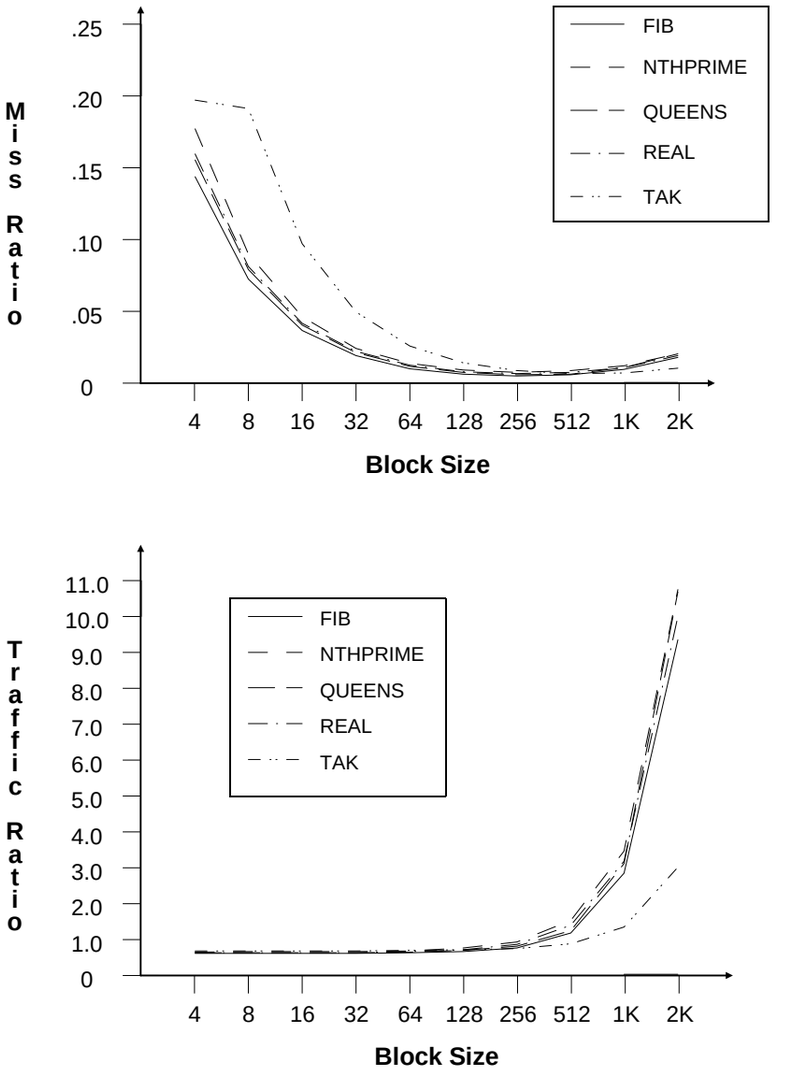

Figure 6-4. Performance with varying cache block size.

Figure 6-3 shows the results of varying cache size over a range of 128 bytes to 64K bytes. While different programs show different degrees of temporal locality, the curves suggest that increases in cache size beyond 64K will not significantly change the miss ratio. So, conventional hardware platforms seem to be adequate with respect to cache size. The example of fib given in Section 6.1 was slightly misleading, as shown by Figure 6-3. Fib has a miss ratio curve that dips significantly below the curves for other programs. Based on Figure 6-3, one could speculate that the required cache size varies with the number of active, or "live" nodes in the program graph. Results from larger benchmarks would be required to confirm this speculation.

Figure 6-4 shows the results of varying block size over a range of 4 bytes to 2K bytes. The cache miss ratio for all programs decreases up to a cache size of 256 bytes. The traffic ratio starts increasing noticeably at between 128 and 256 bytes, suggesting that a block size of 128 bytes would be advantageous.

Table 6-9 shows the results of varying the associativity of the cache from direct mapped (1-way associative) to 8-way associative. 2-way set

	TRAFFIC RATIO				
Memory Update	Fib	NthPrime	Queens	Real	Tak
write through	0.5854	0.6262	0.5942	0.5971	0.6478
copy back	0.2863	0.3507	0.3063	0.3123	0.3769

Table 6-10. Cache performance with varying cache write-through strategy.

associative seems to bring a slight performance improvement, but beyond that there is little or no advantage to adding cache sets.

Table 6-10 shows the traffic ratio for a write-through versus copy-back management policy. The cache miss ratios are the same since this policy does not affect whether misses occur. However, the bus traffic generated for the write-through method is significantly higher than for copy-back.

To sum up, the simulations in this section corroborate the results of Section 6.1, except that the results from Section 6.1 slightly overstated the maximum advantageous cache block size, and exaggerated the temporal locality of the programs.

6.2.2. Comparison with Actual Measurements

Cache simulation results are an important architectural tool. However, there is always the question of whether the results of such simulations correspond to the "real world". In order to establish some confidence in the simulation results, a comparison will be made between the results of a simulation of the DECstation 3100 and the results of actual program execution.

Simulation indicates that for skifib, the MIPS R2000 processor executes 27.82 instructions per combinator reduction application (on average). The R2000 also performs 33.95 memory reads (including both instruction reads and data reads) accesses per combinator reduction application, which when multiplied by a simulated miss ratio of 0.0097, gives 0.33 cache read misses per combinator reduction. The DECstation 3100 has a cache read miss latency of 5 clock cycles, resulting in a cost of 1.65 clock cycles per combinator because of cache misses. This, when added to the 27.82 cycle instruction execution cost (27.82 instructions at one instruction per clock cycle), yields an execution time of 29.47 clock cycles per combinator.

The DECstation 3100 has a cost of zero clock cycles for a cache write miss, so long as the write buffer does not overflow. With an average of 4.74 writes (at 6 clock cycles per write) plus 0.33 cache miss reads (at 5 clock cycles per read) per combinator, a total of 30.09 clock cycles is needed per combinator to provide adequate memory bandwidth for the write-through strategy.* This is somewhat longer than the 29.47 clock cycle instruction execution speed, leading to the conclusion that the DECstation 3100 implementation of TIGRE is constrained by memory bandwidth.

As a result of this analysis, we calculate the simulated execution speed of the DECstation 3100 to be 30.09 clock cycles per combinator. At 16.67 MHz, this translates into a speed of 554000 RAPS between garbage collections.

When actually executing the skifib benchmark, the DECstation 3100 performed approximately 495000 reduction applications per second (RAPS) including garbage collection time. Garbage collection overhead was measured at approximately 1%. This rather low cost is attributed to the fact that a small number of nodes actually in use at any given time, so a copying garbage collector must typically copy just

---

* Actually, it may be worse than this steady-state rate, since bursts in memory accesses may overflow the write buffer, causing even more stalling, but this is difficult to measure without detailed simulation of the actual system involved.

a few hundred nodes for each collection cycle on the benchmark used. Virtual memory overhead can be computed based on a 0.0091 miss ratio for a block size of 4K bytes, with 6.67 data access per combinator, giving a computed virtual memory miss ratio of 0.00136 per combinator. Assuming 13 clock cycles overhead per TLB miss (based on an 800 ns TLB miss overhead for a MIPS R2000 with a 16 MHz clock as reported by Siewiorek & Koopman (1989)), and noting that an average combinator takes 30.09 clocks, this gives a penalty of:

0.00136 * 13 / 30.09 (clocks per combinator) = 0.06%

Together with the 1% garbage collection overhead, this 1.06% overhead predicts a raw reduction rate of:

495000 * 1.0106 = 500000 RAPS

This rate is 11% slower than the 554000 RAPS predicted raw reduction rate.

A major portion of the discrepancy is probably caused by bursts of traffic to the write buffer, which stalls the processor under many conditions. The rest of the discrepancy is due to subtle system overheads such as interference between memory refresh operations and cache misses, as well as cache cold starts on a multiprogrammed operating system.

6.3. DYNAMIC PROGRAM BEHAVIOR

Although cache performance is a crucial part of overall TIGRE performance on conventional hardware, there are two other areas of performance measurement that deserve attention. One area is in the use of heap memory, and the other area is access to spine stack memory.

6.3.1. Heap Memory Use

The previous discussions of cache memory behavior have covered most of the important points about heap use. There are two points left to

	NODES ALLOCATED PER COMBINATOR		
	SKI set	Turner Set	Super+Strict
nodes / combinator	0.737	0.731	1.000

Table 6-11. TIGRE use of heap memory.

cover: the rate at which heap nodes are consumed, and the importance of choosing a garbage collection technique.

Table 6-11 shows the average number of heap nodes allocated per combinator for the fib benchmark. The **SKI** implementation continually executes the **S** combinator, which allocates two nodes of heap memory each time it is used. **K** combinators are then used in many instances to discard one of the two newly created heap nodes. The Turner Set number, therefore, is similar, since the **B** and **C** combinators each allocate one node, but are used to replace pairs of **S** and **K** combinators, maintaining the ratio of heap nodes allocated per combinator relatively constant. The supercombinator implementation uses more heap nodes per combinator, because there are fewer "noise" combinators (*e.g.* **I**, **K**) executed.

The number of heap nodes consumed per combinator is quite large when one considers that, at a Turner Set execution rate of 450000 combinators per second, approximately a third of a million heap nodes (and therefore, eventually, cells to be garbage collected) are generated each second. This is a heap space consumption rate of 2.63M bytes per second on the DECstation 3100. One way of understanding this number is that all physical memory in a minimum configuration system (8M bytes) can be consumed by heap allocation in approximately three seconds. Alternately, the heap node consumption rate is equal to 66% of the maximum I/O transfer rate of the system. In other words, heap allocation is a significant system load.

In order to alleviate the demands of heap allocation on the system, an appropriate garbage collection technique is important. This technique must be efficient both at allocation and collection. The original implementation of TIGRE used a mark/sweep garbage collector. Upgrading to a stop-and-copy garbage collector resulted in a speed improvement of 130000 RAPS on the VAX 8800 assembly language implementation. This significant speedup was caused not only by the fact that stop-and-copy tends to execute fewer instructions, but also by the fact that the mark/sweep method sweeps through the heap space, flushing cache memory. The stop-and-copy algorithm is much better at preserving locality of reference to the heap, and so results in better performance.

One design challenge with using a stop-and-copy garbage collector, or any garbage collector that relocates heap elements, is maintaining consistency of the state of the computation with the relocated elements. When performing a stop-and-copy garbage collection, the spine stack must examined and modified so that the spine stack elements reflect the new locations of nodes in the graph. This can be accomplished with

a fixup routine after each garbage collection. A harder problem (and, one which does not appear easy to solve at first thought) is that some references to **P** combinators are used to check for equality between data structures. Thus, there is a mechanism for the address of a list (returned by **P**) to be propagated as a data value in the system. Unfortunately, if a garbage collection happens between two accesses to the same **P** combinator, it is possible that an equality test will improperly fail, since one of the values being checked contains a stale reference to a pre-collection address.

The stale reference problem with the **P** combinator is difficult to solve with a fixup routine. Fortunately, values from **P** combinators are ensured by the compilation process to never be used in any computation except equality checking. TIGRE solves this stale reference problem (and, the spine stack relocation problem as well) in a very simple way. TIGRE simply throws away the contents of the spine stack at each garbage collection. Since the entire state of a computation is contained in the program graph, the spine stack contains only redundant information (and it is, in fact, maintaining consistency between the redundant representations of information which is the source of the problem). So, after each garbage collection cycle, TIGRE performs a "warm start" of graph execution, restarting the spine unwinding from the root of the program graph. Measurements have been unable to detect any difference in execution time between this warm start technique and a spine stack fixup technique. Of course, there is the important difference that the spine stack fixup technique alone doesn't work for some programs.

	ACCESSES PER COMBINATOR		
	SKI set	Turner Set	Super+Strict
top of stack write	2.18	2.54	1.83
top of stack read	1.13	1.54	1.33
second on stack read	0.74	1.08	0.66
third on stack read	0.55	0.35	0.33
spine unwinds	1.37	1.38	1.00
total stack accesses	4.61	5.50	5.33

Note: top of stack writes include one write per spine node unwound.

Table 6-12. TIGRE use of stack memory for fib.

6.3.2. Stack Memory Use

Table 6-12 summarizes the stack access characteristics of TIGRE. These results were obtained by annotating a simulation of MIPS R2000 assembly language with data indicating load and store instructions that accessed spine stack elements. The number of spine unwinds per combinator are included in the number of top of stack writes.

From simulation results discussed earlier, an average Turner Set combinator performs 10.87 data memory accesses. As Table 6-12 shows, approximately half of these accesses (5.50) are to the spine stack. This access pattern is consistent with the notion that most accesses to heap memory are indirected through the spine stack, causing frequent pairings of accesses to the spine stack with accesses to heap memory.

Another point of interest is that most accesses are to the topmost stack element. Furthermore, accesses deep into the stack are infrequent (and, for the programs measured, no accesses exceeded a depth of three). This raises the possibility of a small, perhaps even single-element, stack buffer register for improved performance.