# Cache Performance of Combinator Graph Reduction

Philip J. Koopman, Jr.
Harris Semiconductor
Melbourne, Florida 32902

Peter Lee
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Daniel P. Siewiorek
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

## Abstract

The Threaded Interpretive Graph Reduction Engine (TIGRE) was developed for the efficient reduction of combinator graphs in support of functional programming languages and other applications. We present the results of cache simulations of the TIGRE graph reducer with the following parameters varied: cache size, cache organization, block size, associativity, replacement policy, write policy, and write allocation. As a check on our results, we compare the simulations to measured performance on real hardware. From the results of the simulation study, we conclude that graph reduction in TIGRE has a very heavy dependence on a write-allocate strategy for good performance, and very high spatial and temporal locality.

**Keywords:** functional programming, combinators, graph reduction, cache memory, architectural simulation.

## Introduction

During the development of the TIGRE graph reducer [1][2], the speed of graph reduction on different hardware platforms repeatedly surprised us, in some cases failing to meet expectations, and in other cases substantially exceeding predicted performance levels. For example, the DECstation 3100 system [3] (which is based on the MIPS R2000 processor chip [4]) performed 470,000 combinator reduction applications per second (RAPS) for Turner's set of SK-combinators [5][6]. This makes TIGRE, to the best of our knowledge, the fastest SK-combinator graph reducer in existence. A VAX 8800 mainframe system [7] with a faster clock rate and a wider system bus performed only 355,000 RAPS. Finally, a Cray Y-MP [8], with a clock speed ten times that of the DECstation 3100, performed only 310,000 RAPS. Another unexpected result was that a VAX mainframe implementation of TIGRE was sped up by 20% with a slight code change to circumvent the write-no-allocate cache management strategy used by that machine.

These results have prompted us to undertake a detailed study of the architectural issues affecting the efficiency of graph reduction. The purpose of the study is twofold. First, we would like to be able to predict, on the basis of the hardware architecture, what kinds of machines will best support graph reduction (and hence functional languages). Second, we would like to obtain design-tradeoff data for both custom graph reduction hardware and new reduction techniques. This is a report of the first phase of the study – the cache behavior of SK-combinator graph reduction.

## Background

The Threaded Interpretive Graph Reduction Engine (TIGRE) was developed to efficiently implement pure combinator graph reduction in support of lazy functional programming languages and other applications. The basic philosophy underlying TIGRE is the elimination of tags on data cells in order to avoid case analysis operations when accessing a graph node.

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-side children onto what is often referred to as the "spine" stack. Besides 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; the control program really ends up being an interpreter. Hence, in this scheme we seem

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

forced to accept the efficiency penalties involved with interpretation as opposed to direct execution.

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 instruction) is found.

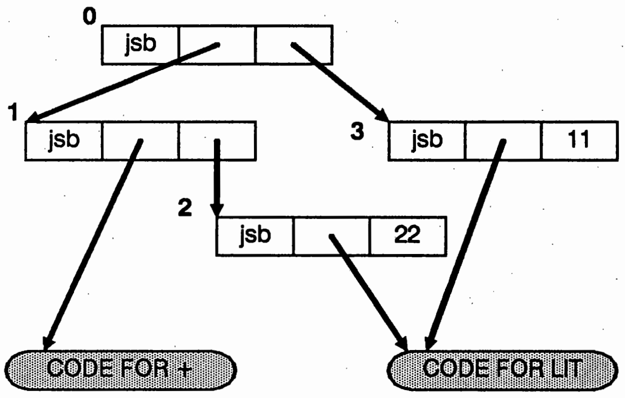

Figure 1. Example program graph for the VAX implementation of TIGRE.

Figure 1 shows a simple program graph for ((+ 22) 11) as implemented in the VAX assembly language version of TIGRE. Each node in the graph contains a VAX `jsb` subroutine call instruction as well as two subgraph pointers (a function pointer in the middle node cell, and an argument pointer in the rightmost node cell). Combinators and graph references are both represented by pointers. Literal values are implicitly tagged as data items by the fact that the function slot of a literal node always points to the LIT combinator code. With this representation scheme, there is only one explicit data type in the graph: the pointer. Hence there is only one type of node, and therefore no conditional branching or case analysis for tag interpretation is required at run time.

Evaluation of a program graph is initiated by performing a subroutine call to the `jsb` node of the root of a subgraph. The machine's program counter then traverses the left spine of the graph structure by executing the `jsb` instructions of the nodes following the leftmost spine. As the subroutine calls are executed, the return-address stack accumulates references to graph nodes in the manner of a spine stack. When a slot points to a combinator, the VAX simply begins executing the combinator code, with the return-address stack providing addresses of the right-hand sides of parent nodes for the combinator's argument values.

TIGRE in no sense interprets the graph. It *directly executes* the data structure, using the hardware-provided subroutine call instruction to do the stack unwinding. When combinator bodies are reached, the arguments are popped from the return stack, the graph is rewritten, and then a jump is made 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 technique is similar to that previously reported by Augusteijn & van der Hoeven [9]. However, to our knowledge they did not conduct an in-depth architectural study of the approach.

The execution speed of TIGRE for the Turner set of SK-combinators compares favorably with previously reported combinator graph reducers, and for supercombinators it appears to be competitive with the G-machine [10] and closure reducers such as TIM [11]. Table 1 shows a summary of TIGRE performance on a number of platforms. The numbers shown for supercombinator implementations give a RAPS rating normalized to the Turner set implementations in order to reflect speedup obtained by supercombinator compilation.

## The basis for the architectural study

We conjectured that the unexpected performance variations observed among TIGRE versions were caused by hardware 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 optimal 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.

The MIPS R2000 was chosen as the implementation vehicle for the simulations for several reasons. Several different R2000-based machines are available to us for "reality checks" between simulator results and actual execution times. The R2000 is a simple architecture that is readily modeled, and information about the timing and operation of the R2000 is readily available. The R2000 processor lacks a subroutine-call instruction; however, the use of an interpretive loop instead of subroutine

Table 1.
TIGRE performance on a variety of platforms.

| Platform | Language | Combinator Set | Program | Time (sec) | Speed (RAPS) |
|---|---|---|---|---|---|
| DECstation 3100 (16.7 MHz) | ASSEMBLER | SKI Set | SKIFIB(23) | 2.20 | 495000 |
| | | Turner Set | FIB(23) | 1.58 | 470000 |
| | | | NFIB(23) | 2.68 | 484000 |
| | | | TAK | 12.58 | 420000 |
| | | | NTHPRIME(300) | 2.60 | 364000 |
| | | | QUEENS(20) | 5.63 | 433000 |
| | | Supercombinator | FIB(23) | 0.36 | 2046000 |
| | | | NFIB(23) | 0.80 | 1626000 |
| VAX 8800 (22 MHz) | ASSEMBLER | SKI Set | SKIFIB(23) | 2.82 | 387000 |
| | | Turner Set | FIB(23) | 2.10 | 355000 |
| | | | NFIB(23) | 3.55 | 366000 |
| | | | TAK | 16.07 | 329000 |
| | | | NTHPRIME | 3.91 | 242000 |
| | | | QUEENS(20) | 8.33 | 293000 |
| | | Supercombinator | FIB(23) | 1.22 | 611000 |
| | | | NFIB(23) | 0.97 | 1339000 |
| VAXstation 3200 | ASSEMBLER | SKI Set | SKIFIB(23) | 6.33 | 172000 |
| | | Turner Set | FIB(23) | 4.80 | 155000 |
| | | | NFIB(23) | 8.23 | 158000 |
| | | Supercombinator | FIB(23) | 2.77 | 269000 |
| | | | NFIB(23) | 2.15 | 605000 |
| Cray Y-MP (167 MHz) | C | SKI Set | SKIFIB(23) | 3.09 | 352000 |
| | | Turner Set | FIB(23) | 2.40 | 310000 |
| | | | NFIB(23) | 4.25 | 305000 |
| | | | TAK | 14.69 | 360000 |
| | | | NTHPRIME(300) | 3.40 | 277000 |

threading does not affect data cache access patterns, and so is irrelevant for examining data access behavior in the second half of the study.

## Phase 1: Exhaustive search of the design space

The goal of the first phase of the simulations was to use memory access traces from TIGRE and a trace-driven cache simulator to explore a wide range of values for several independent cache parameters (such as cache size, block size, and replacement policy). By simulating all possible combinations of two or three discrete values for each parameter, the performance of TIGRE over the entire cache design space was mapped. As a result, similarities and differences between the best-performing sets of parameter combinations could lend insight into what kind of cache memory organization best supports TIGRE.

The dineroIII trace-driven cache simulator program [12] was used. 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. [13] and Smith [14] 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 was chosen for the exhaustive design space search. A large enough heap was used to avoid the need to simulate garbage collection.

Table 2 shows the best sixteen configurations based on 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). Traffic ratio is the number of 32-bit transfers on the data bus from the combination of

Table 2.
The sixteen best cache configurations.

| CACHE SIZE | CACHE SPLITTING | BLOCK SIZE | ASSOCIA-TIVITY | REPLACE POLICY | WRITE POLICY | WRITE ALLOCATE? | MISS RATIO | TRAFFIC RATIO |
|---|---|---|---|---|---|---|---|---|
| 64K | UNIFIED | 16 | 4 WAY | LRU | COPY | YES | 0.0096 | 0.0767 |
| 64K | SPLIT | 16 | 4 WAY | LRU | COPY | YES | 0.0096 | 0.0768 |
| 64K | UNIFIED | 16 | 4 WAY | LRU | THRU | YES | 0.0096 | 0.1609 |
| 64K | SPLIT | 16 | 4 WAY | LRU | THRU | YES | 0.0096 | 0.1610 |
| 16K | UNIFIED | 16 | 4 WAY | LRU | COPY | YES | 0.0097 | 0.0773 |
| 16K | UNIFIED | 16 | 4 WAY | LRU | THRU | YES | 0.0097 | 0.1612 |
| 64K | UNIFIED | 16 | 4 WAY | FIFO | COPY | YES | 0.0098 | 0.0776 |
| 16K | SPLIT | 16 | 4 WAY | LRU | COPY | YES | 0.0098 | 0.0777 |
| 64K | SPLIT | 16 | 4 WAY | FIFO | COPY | YES | 0.0098 | 0.0779 |
| 16K | SPLIT | 16 | 4 WAY | LRU | THRU | YES | 0.0098 | 0.1615 |
| 64K | UNIFIED | 16 | 4 WAY | FIFO | THRU | YES | 0.0098 | 0.1615 |
| 64K | SPLIT | 16 | 4 WAY | FIFO | THRU | YES | 0.0098 | 0.1617 |
| 64K | SPLIT | 16 | DIRECT | – | COPY | YES | 0.0101 | 0.0795 |
| 64K | SPLIT | 16 | DIRECT | – | THRU | YES | 0.0101 | 0.1627 |
| 64K | SINGLE | 16 | DIRECT | – | COPY | YES | 0.0102 | 0.0799 |
| 64K | SINGLE | 16 | DIRECT | – | THRU | YES | 0.0102 | 0.1632 |

Total data reads = 0.1585, Total Data writes = 0.1224, Total Instruction reads = 0.7191
1042523 MIPS R2000 instructions, 1449863 memory accesses, 37480 combinators.

cache misses and writes of modified cache contents to memory, normalized to the total number of accesses.

Each simulation run involved a total of 1449863 memory accesses, 1042523 of which were instruction reads. 71.9% of all memory accesses were instruction reads, 15.9% were data reads, and 12.2% were data writes. To avoid the possibility of misleading results caused by an insufficiently large simulation data set size, the simulation was rerun on several data points from various regions of the simulated design space with a data set ten times as large (created by running `skifib` with a larger input). These expanded simulations yielded essentially identical results.

Some obviously desirable characteristics can be inferred from Table 2. 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, 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 has a slightly better miss ratio than a split cache because the interpretive program was quite small. Thus, a unified cache gives more total cache memory for the data portion of the program. However, split caches work better in practice, since most RISC processors depend on the extra bandwidth available from a split cache scheme for high performance.

From the data in these tables, we conclude that 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 is the optimal strategy.

## Phase 2: Parametric analysis

The initial exhaustive search of the design space gave a good starting point for determining the optimal 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 cache design obtained from the first phase of the simulation study 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. Therefore, the parametric analysis simulations modeled 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, was: 32K byte data-cache size, 16 byte block size, 4-way set associative, LRU replacement, copy-back, and write-allocate. 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.

**The importance of a write-allocate strategy**

Table 3 shows the results of varying the write allocation policy. We have found that this design parameter is more important by far than any of the other parameters, 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 systems running conventional software.

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

MISS RATIOS

| Allocation Strategy | SKI set | Turner set | Super. |
|---|---|---|---|
| write allocate | 0.0341 | 0.0300 | 0.0528 |
| write no allocate | 0.1914 | 0.1522 | 0.2433 |

The reason for the extreme sensitivity to write-allocation lies with the allocation of heap nodes. 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). 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 several hundred clock cycles. A write-no-allocate policy will experience a second cache miss, while a write-allocate policy will 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 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 data 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 less efficient sequences of the S and K combinators). But, the Turner set does have a large number of redundant accesses to elements for intermediate graph rewriting that are eliminated by the supercombinator approach, so the supercombinator version shows marked degradation in performance from using a write-no-allocate strategy.

**Strong spatial locality means large block size**

Figure 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 and Turner Set methods, and up to 8K bytes (the limit to block size given 4-way set associativity) for the supercombinator method. 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 collection and heap allocation schemes, 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-block size based on the miss ratios alone. For conventional programs, this decision would be

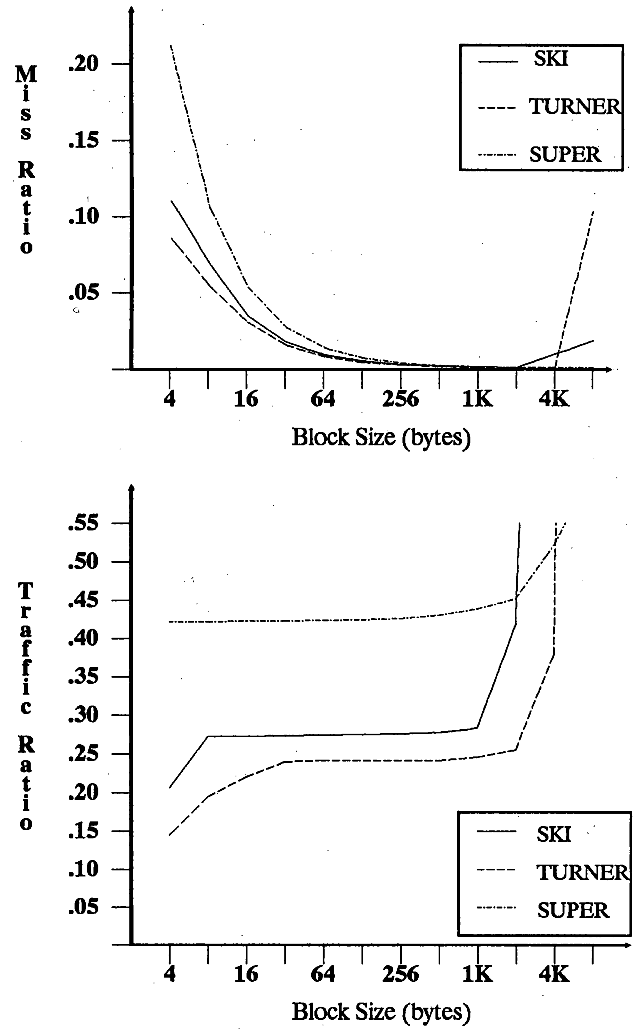

Figure 2. TIGRE performance with varying cache block size.

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-block size is entirely reasonable for this application.

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 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 observed temporal locality bodes well for virtual memory access behavior. Since most translation lookaside buffers are limited in size (for example, 64 entries addressing 4K bytes each on a MIPS R2000), good spatial locality is important to 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 collections were done for these simulation runs).

**High temporal locality means small cache size**

Figure 3 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

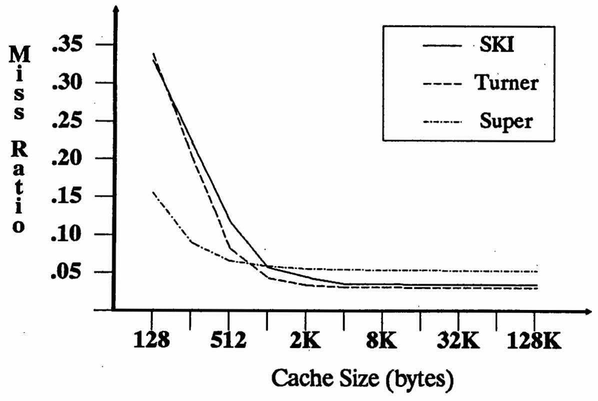

Figure 3. TIGRE performance with varying cache size.

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 may have better temporal locality than conventional programs. This temporal locality may be due in part to a high infant mortality rate among allocated heap nodes.

High temporal locality suggests that generational garbage collection techniques [15] may be useful with combinator graph reduction, 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 the good performance observed for small caches is due to running small test programs. Final resolution of this question will have to wait until a diverse body of large programs is available for measurement. This is particularly important for the measurements involving supercombinator compilation.

**Write-through policy**

Table 4 shows the results on miss ratio and traffic ratio for a write-through versus copy-back memory update 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 block 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

Table 4.
TIGRE performance with varying memory update strategy.

| | MISS RATIO / TRAFFIC RATIO | | |
|---|---|---|---|
| Memory Update | SKI set | Turner set | Super. |
| copy-back | 0.0341<br>0.2721 | 0.0300<br>0.2209 | 0.0528<br>0.4223 |
| write-through | 0.0341<br>0.5721 | 0.0300<br>0.5431 | 0.0528<br>0.6849 |

can swamp a bus, causing memory-bandwidth performance limitations. 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 data memory accesses, which could correspond to 1.7 clock cycles. Clearly, a copy-back policy is desired to limit the effects of bus saturation.

The supercombinator implementation has even worse bus-write characteristics. This is caused by a higher 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 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 generate significant demands on bus bandwidth.

**Associativity & Replacement Policy**

Simulation results of varying the associativity of the cache from direct-mapped (1-way associative) to 8-way associative showed a variation in miss ratios of less than 0.2%. 2-way set associative seems to bring a slight performance improvement over direct-mapped, but beyond that there is no discernible advantage to adding cache sets.

Simulation results of varying the replacement policy for the cache similarly showed variations of less than 0.1% in miss ratios. Least Recently Used (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 more than 1-way associativity.

Neither associativity nor replacement policy seem to matter much for combinator graph reduction.

## The optimal cache strategy

Based on the analysis of 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 a 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 between 16-byte and 4-byte block sizes represents approximately a 44000 RAPS (nearly 10%) speed penalty for a DECstation 3100 class machine.

**Comparison with actual measurements**

Cache simulation results are an important architectural design 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 was made between the results of a simulation of the DECstation 3100 and the results of actual program execution. The DECstation 3100 has a split cache with 64K bytes in each cache, a block size of 4 bytes, direct-mapped organization, and uses write-through memory updating with write-allocate cache management. [3]

Simulation indicates that for `skifib`, the 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) 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 at least 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 475000 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 are actually in use at any given time, so a copying garbage collector must typically copy just 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 [16]), 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:

475000 * 1.0106 = 480000 RAPS

This rate is 15% slower than the 554000 RAPS predicted raw reduction rate. Some of this 15% discrepancy is due to the overhead of cache cold starts on a multiprogrammed operating system. The rest of the discrepancy is probably caused by bursts of traffic to the write buffer, which stalls the processor when full.

## The potential of special-purpose hardware

### DECstation 3100 as a baseline

We have described various implementation methods and performance data for TIGRE. This section uses those data points to propose architecture and implementation features which could be used to speed up the execution of TIGRE. The reason for examining such features is to determine the feasibility of constructing special-purpose hardware, or, if construction of special-purpose hardware is not attractive, the features that should be selected when choosing standard hardware to execute TIGRE.

Since the best measured performance for TIGRE was for the MIPS R2000 assembly language implementation, the approach used for examining processor features to support TIGRE will be made in terms of incremental modifications to the MIPS R2000 processor. This approach will give a rough estimate for the potential performance improvement, while maintaining some basis in reality. For the purposes of the following performance analysis, the characteristics of the SKI implementation of the `fib` benchmark executing on the DECstation 3100 shall be used.

Since TIGRE has been shown to have some unusual cache access behavior, the first area for improvement that will be considered is changing the arrangement of cache memory. Then, improvements in the architecture of the R2000 will be considered.

## Improvements in cache management

### Copy-back cache

The most obvious limitation of the DECstation 3100 cache is that it uses a write-through cache. This caused the limiting performance factor to be bus bandwidth for memory write accesses, instead of instruction read or data read miss ratios. A simple improvement, then, is to employ a copy-back cache. A cache simulation of `fib` for the DECstation 3100 shows that this reduces the data cache traffic ratio from 0.5461 to 0.2078, removing the bus bandwidth as the limiting factor to performance. This reduces the execution time of an average combinator from 30.09 clock cycles (the bus bandwidth-limited performance) to 29.47 clock cycles (the cache hit ratio-limited performance).

### Increased block size

A second parameter of the cache that could be improved is the block size. TIGRE executes well with a large block size, so increasing the cache-block size from 4 bytes to, say 256 bytes, should dramatically decrease the cache miss ratio, but would suffer from the limited width of the memory bus. Using a wide bus-write buffer with a 4 byte cache-block size can capture many of the benefits of a large block size, and reduce bus traffic. A write buffer width of 8 bytes (one full graph node) can be utilized efficiently by a supercombinator compiler to get a very high percentage of paired 4-byte writes to the left-and right-hand sides of cells when updating the graph.

However, even if a very sophisticated cache mechanism were used to reduce cache misses to the absolute minimum possible (ideally, 0.0000 miss ratio), the speedup possibilities are somewhat small. This is because only 1.65 clock cycles of the 29.47 clock cycles per combinator are spent on cache misses to begin with.

## Improvements in CPU architecture

The opportunities for improvement by changing the architecture of the R2000 are somewhat more promising than those possible by modifying the cache management strategy. In particular, it is possible to significantly increase the speed of stack unwinding and performing indirections through the stack elements.

### Stack unwinding support

The one serious drawback of the R2000 architecture for executing TIGRE is the lack of a subroutine call instruction. The current TIGRE implementation on the R2000 uses a five-instruction interpretive loop for performing threading (*i.e.* stack unwinding). Since 1.37 stack unwind operations are performed per combinator, this represents

6.85 instructions which, assuming no cache misses, execute in 6.85 clock cycles.

But, there is a further penalty for performing the threading operation through graphs with the R2000. A seven-instruction overhead is used for each combinator to perform a preliminary test for threading, and to access a jump table to jump to the combinator code when threading is completed. (One of these instructions increments a counter used for performance measurement. It can be removed for production code, as long as measuring the number of combinators executed is not important.) This imposes an additional 7.00 clock cycle penalty on each combinator.

So, the total time spent on threading is 13.85 clock cycles per combinator. It takes three clock cycles to simulate a subroutine call on the R2000:

```
# store current return address
   sw      $31, 0($sp)
# subroutine call
   jal     subr_address
# branch delay slot instruction follows
# decrement stack pointer
   addu    $sp, $sp, -4
```

so it is reasonable to assume that a hardware-implemented subroutine call instruction could be made to operate in three clock cycles. Thus, if the instruction cache were made to track writes to memory (permitting the use of self-modifying code), a savings of 10.85 clock cycles is possible. One important change to the instruction set would be necessary to allow the use of subroutine call instructions – the subroutine call instruction would have to be defined to have all zero bits in the opcode field (so that the instruction could be used as a pointer to memory as well). An alternate way to implement a subroutine call with a modifiable address field is to define an indirect subroutine call that reads its target address through the data cache, eliminating the need to keep the instruction cache in synch with bus writes.

**Stack access support**

An important aspect of TIGRE's operation is that it makes frequent reference to the top elements on the spine stack. In fact, 4.61 accesses to the spine stack are performed per average combinator. Most of the load and store instructions that perform these stack accesses can be eliminated by the use of on-CPU stack buffers that are pushed and popped as a side effect of other instructions.

For spine-stack unwinding, two of the three instructions used to perform a subroutine call could be eliminated with the use of hardware stack support, leaving just a single `jal` instruction to perform the threading operation at each node. Of course, the R2000 has a built-in branch delay slot, so it probably not the case that the actual time for the threading operation could be reduced to less than two clock cycles. But, the second clock cycle could be used to allow writing a potential stack buffer overflow element to memory.

Of the 4.61 instructions that access the spine stack, the threading technique just described may be used to eliminate the effect of 1.37 of the instructions per combinator. The remaining 3.24 instructions can also be eliminated by introducing an indirect-through-spine-stack addressing mode to the R2000. All that would be required is to access the top, second, and third element of a spine-stack buffer as the source of an address instead of a register. A simple implementation method could map the top of stack buffer registers into the 32 registers already available on the R2000. This gives a potential savings of 3.24 clock cycles, since explicit load instructions from the spine stack need not be executed when performing indirection operations.

**Double word stores**

TIGRE is usually able to write cells in pairs, with both the left-and right-hand cells of a single node written at approximately the same point in the code for a particular combinator. Thus, it becomes attractive to define a "double store" instruction format. Such an instruction would take two source register operands (for example, an even/odd register pair), and store them into a 64-bit memory doubleword. If the processor were designed with a 64-bit memory bus, such a "double store" could take place in a single clock cycle instead of as a two-clock sequence. The savings of using 64-bit stores is 0.895 clock cycles per combinator for the SKI implementations of `fib`, and 1.192 clock cycles per combinator for the Turner set implementation of `fib` (measured by instrumenting TIGRE code to count the opportunities for these stores as the benchmark program is executed). Support of 64-bit memory stores would speed up supercombinator definitions even more, since the body of supercombinators often contains long sequences of node creations. For example,

Table 5.
Summary of possible performance improvements.

| cumulative optimizations | clocks per combinator |
|---|---|
| current implementation | 30.09 |
| copy-back cache | 29.47 |
| 100% cache hit ratio | 27.82 |
| subroutine call + self-modifying code | 16.97 |
| hardware stack for `jal` | 15.60 |
| hardware stack indirect addressing | 12.36 |
| 8-byte store instructions | 11.47 |

the supercombinator implementation of `fib` can make use of 1.33 64-bit stores per combinator.

Table 5 summarizes the efficiency improvements that may be gained through the cache and processor architecture changes just discussed. Nearly a three-fold speed improvement is possible over the R2000 processor with just a few architectural changes.

## Further work

We recognize the fact that our benchmarks are not very realistic. Larger benchmarks are required, as well as more benchmarks based on supercombinators rather than the simple SK-combinators. Unfortunately, we have been hindered by the unavailability of good benchmark suites. We are working to develop a good range of benchmark programs.

## Results

We have found that an efficient cache for combinator graph reduction has several unusual characteristics, including: a very strong dependence on the write-allocate strategy, very modest cache size requirements, and the ability to effectively use very large block sizes.

The results of this research should help users of combinator graph reduction select commercial machines which will perform efficiently. They may also influence the course of design of special-purpose graph reduction hardware in the future.

## References

[1] Koopman, P. (1989) *An Architecture for Combinator Graph Reduction*, Ph.D. Dissertation, Carnegie Mellon University.

[2] Koopman, P. & Lee, P. (1989) A Fresh Look at Combinator Graph Reduction. In *Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland OR, June 21-23, SIGPLAN Notices*, 24(7), July 1989, 110-119.

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

[4] Kane, G. (1987) *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ.

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

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

[7] Burley, R. (1987) An overview of the four systems in the VAX 8800 family. *Digital Technical Journal*, 4:10-19, February.

[8] Pittsburgh Supercomputer Center (1989) *Facilities and Services Guide*, Pittsburgh PA.

[9] Augusteijn, A. & van der Hoeven, G. (1984) Combinatorgraphs as self-reducing programs. University of Twente, the Netherlands.

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

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

[12] Hill, M. D. (1984) Experimental evaluation of on-chip microprocessor cache memories, *Proc. Eleventh Int. Symp. on Computer Architecture*, Ann Arbor, June.

[13] Kabakibo, A., Milutinovic, V., Silbey, A. & Furht, B. (1987) A survey of cache memory in modern microcomputer and minicomputer systems. In: Gajski, D., Milutinovic, V., Siegel, H. & Furht, B. (eds.) *Tutorial: Computer Architecture*, IEEE Computer Society Press, pp. 210-227.

[14] Smith, A. J. (1982) Cache memories, *ACM Computing Surveys*, 14(3):473-530, September.

[15] Appel, A., Ellis, J. & Li, K. (1988) Fast garbage collection on stock multiprocessors. In *Proceedings of the Conference on Programming Language Design and Implementation, Atlanta*, June.

[16] Siewiorek, D. & Koopman, P. (1989) *A Case Study of a Parallel, Vector Workstation: the Titan Architecture*, Academic Press, Boston. In Press.