15 Architectural Considerations for Combinator Graph Reduction

Philip Koopman, Jr. and Peter Lee

15.1 Introduction

Lazy functional programming languages, such as SASL [33] and Haskell [15], possess a number of theoretical properties that make them intriguing candidates for study. However, relatively little is known about how to implement these languages, despite the fact that implementation techniques have long been the subject of research. In 1979 Turner [34] described a technique for implementing lazy functional languages by a technique known as SK-combinator reduction. This idea is based on the well-known observation from combinatory logic that all of the variables in a functional program can be abstracted by transforming it into an applicative expression involving only *combinators*. A combinator is simply a closed λ-expression [5]. Three combinators of particular interest are called **S**, **K**, and **I**:

$$\mathbf{S} = \lambda f.\lambda g.\lambda x.(fx)(gx) \tag{15.1}$$

$$\mathbf{K} = \lambda x.\lambda y.x \tag{15.2}$$

$$\mathbf{I} = \lambda x.x \tag{15.3}$$

The crucial property is that any λ-expression can be transformed into an expression consisting solely of these combinators, by a so-called "bracket abstraction" algorithm [35]. (Actually, only **S** and **K** are necessary, as **I** = **SKK**. In practice, a handful of other combinators, comprising what we shall refer to as the "Turner Set," are also used in order to reduce the size of the combinator expressions.) With all variables abstracted, the resulting expression is easily represented as a binary tree with combinators appearing at the leaves and the internal nodes representing application. As a further optimization, the tree can be transformed into a graph in which the sharing of subgraphs denotes the occurrence of common subexpressions in the combinator expression, cycles denote recursion, and combinator definitions correspond to graph-rewriting rules. For example, Figure 15.1 depicts the graph rewriting corresponding to the **S** combinator.

In this scheme, then, executing programs is a process of graph reduction. The left-most "spine" of the graph is traversed until a combinator is encountered, at which point the

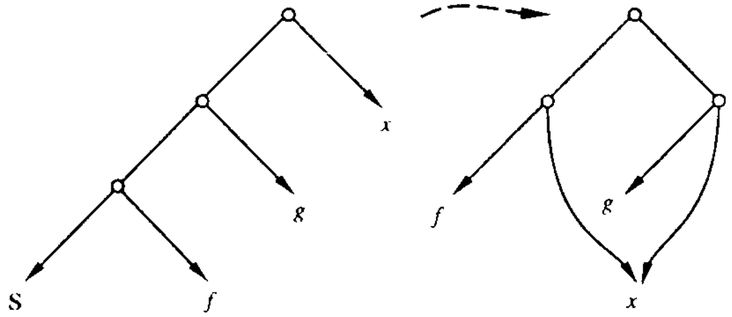

Figure 15.1: Graph rewrite corresponding to the S combinator.

graph is rewritten according to the corresponding rewrite rule. This process is consistently repeated on the new graphs until finally an irreducible graph is produced, at which point program execution is complete.

An advantage in efficiency is obtained from the sharing of subgraphs, so common subexpressions need be reduced only once. Also, the language implementation overall becomes much simpler by virtue of the fact that variable substitution is, in effect, encapsulated in a fixed set of simple rules for rewriting graphs. Indeed, a pure graph reducer can be implemented quite easily and will often exhibit better performance than implementations of lazy functional languages based on other approaches. Such simplicity also lends itself to direct hardware implementation, as in SKIM [32] and NORMA [28]. Still, normal-order evaluation (or, more precisely, lazy evaluation) of functional programs, even via combinator-graph reduction, is in practice much less efficient than applicative-order ("eager") evaluation. Lazy functional programming languages such as Haskell require lazy evaluation, so a great deal of research has been directed towards improving the efficiency of combinator-based techniques. One significant development along these lines, first proposed by Hughes [16], is the notion of *supercombinators*, in which the observation is made that any function can be made into a combinator by adding extra formal parameters corresponding to the free variables appearing in the function body. Supercombinator compilation produces a set of "tailor-made" combinators for each program, resulting in much larger-grain reduction steps and thus requiring fewer reductions for evaluation.

15.2 A Study of Architectural Considerations

In this chapter, we explore the effects of computer architecture features on the efficiency of graph reduction, and hence of lazy functional programming languages. In

particular, we present the results of our experiments with an abstract machine for reducing combinator graphs. The abstract machine, which we have called TIGRE (the **T**hreaded **I**nterpretive **G**raph **R**eduction **E**ngine), treats combinator graphs as self-modifying threaded programs, in a manner similar to that described by Augusteijn and van der Hoeven [1]. We have found that this method seems to reduce combinator graphs at a rate that compares quite favorably with previously reported techniques on similar hardware. TIGRE maps remarkably easily and efficiently onto computer architectures that allow a restricted form of self-modifying code [19,20]. This provides some indication that the conventional "stored program" organization of computer systems (the so-called "von Neumann" architecture[1]) may be more appropriate for functional programming language implementations than previously thought [3].

This is not to say, however, that present-day computer systems are well-equipped to reduce combinator graphs. During our experimentation with TIGRE, 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, a VAX 8800 mainframe system [7] with a faster clock rate and wider system bus than the DECstation 3100 [10] performed only 355,000 reduction applications per second (RAPS), compared to the DECstation's 470,000 RAPS. Further experimentation with the VAX implementation led to the discovery that its reduction rate could be increased by 20% simply by making a small change to the code to partially compensate for the write-no-allocate cache-management strategy used by that machine.

It was this unexpected behavior that prompted us to undertake a detailed study of the architectural issues affecting the efficiency of graph reduction, in particular the effect of hardware-cache behavior. Our study begins with the simulation of a TIGRE graph reducer running on a reduced-instruction-set computer with the following hardware-cache parameters varied: cache size, cache organization, block size, associativity, memory update policy, and write-allocation policy. The simulation proceeds in two stages, the first stage being an exhaustive test of selected values for all combinations of parameters, and shows that there are no local extrema in cache miss behavior as a function of cache design choices. At this point we take the cache design of a real machine and simulate the performance sensitivity with respect to variations of individual parameters for several programs. As a check on the accuracy of our simulations, we compare the results with measured performance on real hardware. From the results of this simulation study, we can conclude that graph reduction in TIGRE depends on a write-allocate strategy for good performance, and exhibits high spatial and temporal locality. Finally, on the basis of our experiments, we examine possible architectural changes to the MIPS R2000

[1] Actually, credit for the notion of the stored program computer should be given to Eckert and Mauchly [12, pg. 23].

processor. The changes illustrate some of the differences between graph reduction and more conventional methods for executing programs.

Before proceeding with the description of our experiments and analysis, we will first briefly review the technique of combinator-graph reduction. This will then be followed by a description of the TIGRE abstract machine and its implementation, as well as a preliminary report on its performance. Throughout this chapter we will concentrate on SK-combinator reduction. The issues involved in SK-combinator reduction differ somewhat from those for supercombinator reduction. Of particular importance in supercombinator reduction is the interaction between compile-time analyses (such as strictness analysis and sharing analysis) and the reducer. Promising approaches to supercombinator reduction include TIM [11] and the "Spineless, Tagless, G-machine" [24]. The TIGRE approach described in this paper extends naturally to supercombinator reduction. However, like Norman [22], we desire to study first the effects of architecture on pure graph reduction, thereby isolating the effects of sophisticated compiler technology.

15.3 The TIGRE Abstract Machine

The major difficulties in performing graph reduction efficiently are in traversing the left spine of the graph (referred to as "stack unwinding") and in the analysis of graph-node tags. Reduction or elimination of these costs can greatly improve performance. In this section we shall begin by describing a straightforward mechanism for graph reduction (based on the Chalmers G-Machine as described by Peyton Jones [25]). Then we shall explain how self-modifying threaded code, as employed by TIGRE, is able to avoid certain inefficiencies present in the straightforward approach.

Figure 15.2 shows that graph nodes are typically represented by three one-word fields. The first field is a tag for the values in the application node. This tag value is selected so as to be an index value into an entry table containing addresses of action routines. Accessing a node requires a double-indirection operation through the tag and entry table. On a VAX architecture, unwinding a node while traversing the stack requires four instructions, including a double-indirect jump through the entry table [25]:

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

One of the key points of TIGRE is the elimination of most of this overhead for traversing tree nodes during the stack unwinding process. This can best be accomplished simply by eliminating the need for tags, thereby eliminating the cost of tag interpretation. In the following presentation, we will eliminate the tags in several stages.

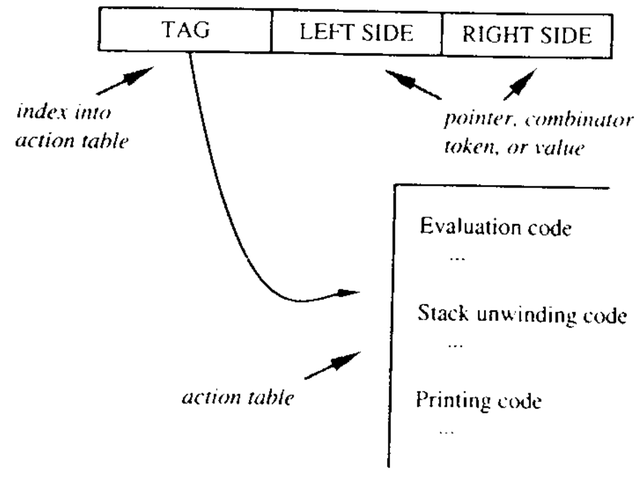

Figure 15.2: Simple scheme for graph-node tag analysis.

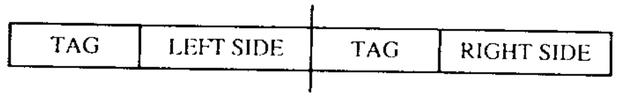

Figure 15.3: Basic structure of a graph node.

Figure 15.3 shows a generalized node representation which has tags associated with both the left-hand and right-hand fields of the node. Figure 15.4 shows a tree for the expression `((+ 11) 22)`, where `+` is the addition combinator, which we shall use as a running example. The numbers next to the nodes serve as labels for our discussion. Although only three tag types are shown in the example, typically more tag types are used in actual implementations.

As a first step in eliminating tags, we shall replace the fields containing constant values by pointers to indirection nodes (i.e., nodes involving the application of the **I** combinator).

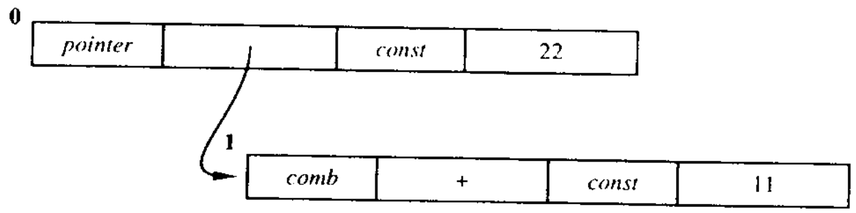

Figure 15.4: Sample graph for the expression `((+ 11) 22)`.

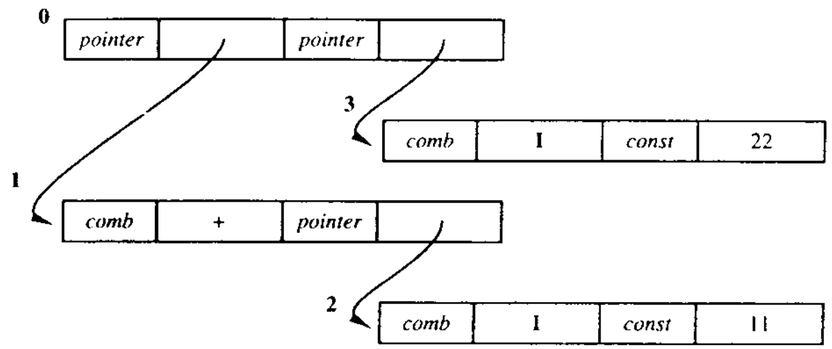

Figure 15.5: Sample graph using indirection nodes.

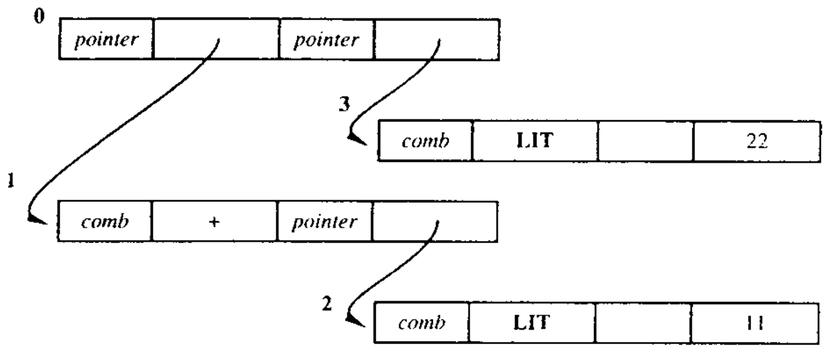

Figure 15.6: Sample graph using LIT nodes.

Figure 15.5 shows the result of this rewriting. Any graph can be rewritten so that constant values are placed in indirection nodes, and in fact this is a standard technique in graph rewriting [34]. For example, the + combinator, when executed, creates an indirection node with the sum. This allows the fields of the root node of the original graph to be overwritten by the result of the graph rewrite.

Now, constant values are only found as arguments to indirection combinators. If we rename the **I** combinators in the left-hand sides of constant nodes as **LIT** combinators (short for "literal value" combinators), as shown in Figure 15.6, the constant tag is no longer needed, since the **LIT** combinator implicitly identifies the argument as a constant value. All other special tags, including tags for other numeric types, can be eliminated by defining new combinators (e.g., **FLIT** for floating point constants) in a similar manner.

The graph shown in Figure 15.6 now only has two tag types: combinator and pointer. At this point, standard techniques can be used to reduce tag-checking costs. (See Chapter 1 for a complete discussion of this point.) For instance, all nodes and therefore pointer values can be aligned on 4-byte boundaries. The lowest bit of a cell's contents can then be used as a one-bit tag. In this case, the tag analysis for numeric constants has been replaced by the need to reduce **LIT** combinators. However, the amount of tag checking on all other cells has been reduced. This is the representation used for interpreted implementations of TIGRE. For instance, in the C-based implementation, TIGRE loops while scanning the lowest order bit of left-hand side cells to unwind the stack. When a non-pointer value is found, TIGRE then uses a `case` statement to jump to the correct action code.

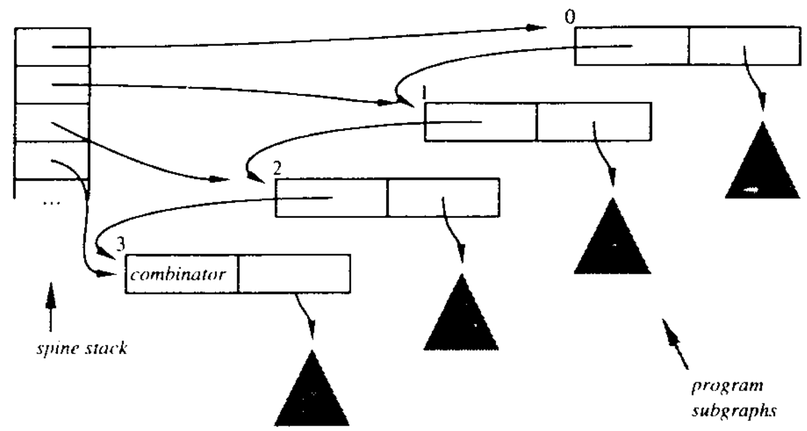

Figure 15.7: Sample graph with the spine stack.

15.4 Self-modifying Threaded Code

There is a key insight which provides approximately a twofold speedup in the execution speed of TIGRE. This is gained by exploiting the hardware support for graph traversal that already exists in most conventional processors.

The generic graph shown in Figure 15.7 is executed by traversing the leftmost spine, placing pointers to ancestor nodes onto a stack (the so-called "spine stack"). When a combinator is encountered in the graph, some code to carry out the graph rewrite is executed. The data structure is controlling the execution of the program. Another, more insightful, way to view this is that the data structure is itself a program with two instruction types: pointer and combinator. Then graph reduction is essentially a process of interpreting a self-modifying threaded program that happens to reside in the node heap. In other words, the graph is a program that consists mainly of calls to subroutines. These subroutines then contain calls to other subroutines, and so on until, finally, some other executable code, which performs a graph rewrite, is found.

The key idea is that *the spine stack is actually a subroutine return stack* for the threaded program. As control flows from node 0 to node 1 to node 2 to node 3 in the graph of Figure 15.7, these nodes are stored on the spine stack. Eventually, a rewriting of the graph involving the right-hand side fields of these nodes will be performed. So, what is actually needed on the stack are pointers to the right-hand side fields of each node. If the left-hand sides of each node are viewed as subroutine call instructions, then the return addresses which would be automatically saved on the return stack would be the addresses of the right-hand fields of the spine of the graph, which is exactly the desired behavior.

Combinator nodes, such as node 3 in Figure 15.7, contain some sort of token value that invokes a combinator. At some point during program execution, this value will have to be resolved to an address for a piece of graph-rewriting code to be executed, so the actual code addresses of the combinator action routines can be stored instead of token values. In fact, a subroutine call to the combinator code can be stored, so that the address of the right-hand side of node 3 will be pushed onto the spine stack, and the combinator will have all its arguments pointed to by the spine stack (i.e., the subroutine return stack). A pleasant side effect of this scheme is that there is now only one type of data in the graph: the pointer. Hence there is only one type of node, and therefore *no conditional branching or case analysis is required at runtime*. All nodes contain either pointers to other nodes or pointers to combinator code. Figure 15.8 shows our running example of ((+ 11) 22) compiled using this scheme. Since all node values (except the right-hand sides of **LIT** nodes) are subroutine call instructions, we can simplify matters by saying that each field contains a pointer that is interpreted as a subroutine call by the TIGRE execution engine.

In the typical TIGRE implementation, graph nodes are represented by triples of 32-bit cells. The first cell of each triple contains a subroutine call instruction while the second and third cells of the triple contain the left-hand and right-hand sides of the node, respectively. The hardware's native subroutine calling mechanism is used to traverse the spine, using the subroutine return stack as the spine stack. Figure 15.9 shows the example graph as it appears in the VAX assembly-language implementation of TIGRE. (Note that the `jsb` is the fast VAX subroutine call instruction which only pushes the program counter onto the return address stack, as opposed to the slower function call instructions which automatically allocate stack frames.) Thus, threaded code interpretation [6] is performed by the C version of TIGRE.

Evaluation of a program graph is initiated by doing a subroutine call to the `jsb` node of the root of a graph. The machine's program counter then traverses the left spine of the graph structure by executing the `jsb` instructions of the nodes following the leftmost spine. When a node points to a combinator, the VAX simply begins executing

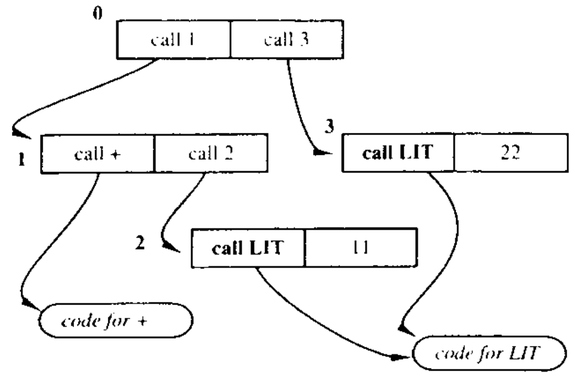

Figure 15.8: A TIGRE graph with only subroutine call nodes.

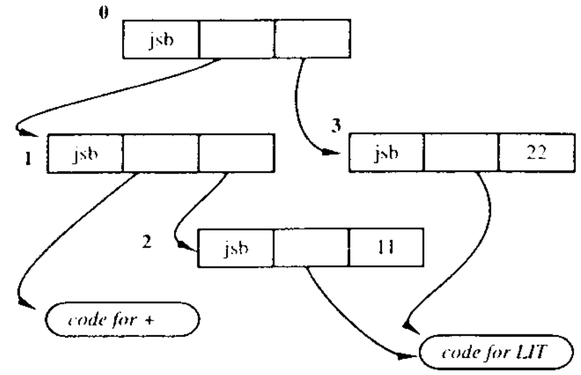

Figure 15.9: The VAX representation of a TIGRE graph.

the combinator code, with the return address stack providing addresses of the right-hand sides of parent nodes for the combinator argument values. When graph nodes are rewritten, only the pointer values (which are 32 bits in size on a VAX) need be rewritten. The `jsb` opcode is initialized upon acquisition of heap space and thereafter never modified.

TIGRE performs subroutine call operations down the left spine of the graph. When combinators are reached, they pop their arguments from the return stack, perform graph rewrites, and then jump to the new subgraph to continue traversing the new left spine. The use of the return stack for graph reduction is different than for "normal" subroutines in that subroutine returns are never performed on the pointers to the combinator arguments but rather, the addresses are consumed from the return stack by the combinators. (This seems to be a characteristic of other combinator reducers as well.)

```
DO_S:
need2(r8,r7)
movl *(sp)+,r6
movl r6,(r8)
movl *(sp),(r7)
movl 4(sp),r10
movl (r10),4(r7)
movl (r10),4(r8)
movab -2(r8),-4(r10)
movab -2(r7),(r10)
movab 4(r8),(sp)
jmp (r6)

DO_K:
movl *(sp),r6
movab 8(sp),sp
jmp (r6)
```

```
DO_I:
movl *(sp),r6
movab 4(sp),sp
jmp (r6)

DO_IF:
movl *(sp)+,r6
jsb (r6)
movl 4(sp),r10
tstl r11
jeql L39
movl *(sp),(r10)
L39: $DO_I,-4(r10)
movl (r10),r6
addl2 $8,sp
jmp (r6)
```

```
DO_LIT:
movl *(sp)+,r11
rsb

DO_PLUS:
movl *(sp)+,r6
jsb (r6)
movl *(sp),r6
pushl r11
jsb (r6)
addl2 (sp)+,r11
movl (sp)+,r9
movl $DO_LIT,-4(r9)
movl r11,(r9)
rsb
```

Figure 15.10: VAX code listings for some TIGRE combinators.

The processor is in no sense interpreting the graph. It is *directly executing* the data structure, using the hardware-provided subroutine call instructions to do the stack unwinding. In our experiments, we have found that this technique exhibits performance that compares favorably with other approaches described in the literature.

15.5 Implementing TIGRE

The availability of a fast subroutine-call instruction on most modern architectures makes the TIGRE technique applicable, in theory, to most computers. In practice, however, there are issues having to do with modifying the instruction stream that make the approach difficult to implement on many machines. These problems can be viewed as the result of inappropriate tradeoffs (for the application of graph reduction) in system design, and not the result of any inherent limitation of truly general-purpose CPUs. Inasmuch as graph reduction is a self-modifying process, it is not surprising that a highly efficient graph reduction implementation makes use of self-modifying techniques.

Figure 15.10 gives a sketch of the VAX-assembler implementation of the combinators for the SKI combinator set. This code is a simple version written for clarity. The VAX implementation actually in use has various small optimizations to eliminate redundant memory reads and better exploit the pipeline of high-end VAX mainframe systems.

In TIGRE, traversing the left spine is typically less expensive than rewriting the graph. This leads to some novel design decisions, one of which affects the implementation of "projection" combinators such as **I** and **K**. The implementations of these combinators as shown in Figure 15.10 do not modify the graph at all, but rather redirect the flow of control of the graph evaluation, popping elements from the return stack as they execute. **K** and **I** are two instances of the set of projection combinators which simply drop a number of parent nodes while performing an indirection operation on the topmost node on the spine stack. This optimization may degrade memory usage by, for example, leaving subtrees attached to a **K** node when they would have otherwise been abandoned, but our experience thus far has been that the speedup realized by avoiding graph rewrites more than makes up for this inefficiency.

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

15.6 TIGRE performance

TIGRE has been implemented in C, VAX assembler, and MIPS R2000 assembler. (An initial exploration was carried out in Forth, a language noted for its support of threaded code.) The C version uses a threaded interpretation as previously discussed. The VAX assembler version uses the `jsb` instruction to perform subroutine-threaded execution of the graph. The MIPS R2000 version uses a carefully written threaded interpretive loop, because the architecture does not have a subroutine call instruction.

Figure 15.11 gives some performance figures for TIGRE. Simple stop-and-copy garbage collection is used. The allocated heap space is small enough to force several dozen garbage collection cycles in each benchmark. No sharing analysis or other optimizations are used. The assembler versions show significant improvements over versions compiled with an optimizing C compiler. The Fib benchmark program is the standard recursive Fibonacci function. The Nfib program is the commonly reported benchmark that tallies the number of recursions used in computing a Fibonacci function. Tak, NthPrime, and 8Queens are lazy functional versions of other well-known benchmarks.

The DECstation 3100 is a 16.7 MHz MIPS R2000-based workstation. The VAX 8800 is a 22 MHz mainframe. The Cray Y-MP [26] is a vectorized supercomputer that has a fast scalar processing unit. The Sun 3/75 system is a 16 MHz 68020 workstation with no cache memory. For the C implementation on the Sun 3/75 workstation, the

Platform	Language	Combinators	Benchmark	Time (sec)	Speed (NRAPS)
DECstation 3100 (16.7 MHz)	assembler	SKI	Fib(23)	2.20	495000
		Turner	Fib(23)	1.58	470000
			NFib(23)	2.68	484000
			Tak	12.58	420000
			NthPrime(300)	2.60	364000
			8Queens(20)	5.63	433000
		supercombinators	Fib(23)	0.36	2046000
			NFib(23)	0.80	1626000
VAX 8800 (22 MHz)	assembler	SKI	Fib(23)	2.82	387000
		Turner	Fib(23)	2.10	355000
			NFib(23)	3.55	366000
			Tak	16.07	329000
			NthPrime(300)	3.91	242000
			8Queens(20)	8.33	293000
		supercombinators	Fib(23)	1.22	611000
			NFib(23)	0.97	1339000
Cray Y-MP (167 MHz)	C	SKI	Fib(23)	3.09	352000
		Turner	Fib(23)	2.40	310000
			NFib(23)	4.25	305000
			Tak	14.69	360000
			NthPrime(300)	3.40	277000
Sun 3/75 (16 MHz)	C	SKI	Fib(23)	14.62	75000
		Turner	Fib(23)	12.75	58000
			NFib(23)	22.02	59000

Figure 15.11: Some TIGRE performance results.

GNU C compiler [31] was used with the optimization switch turned on. Reduction-rate figures for supercombinator implementations were normalized to approximate Turner Set measurements as follows. First, we measured the speedup of the supercombinator version over a Turner Set version on the same platform. Next, this speedup was used to scale the Turner Set reduction rate to arrive at the normalized supercombinator reduction rate (NRAPS).

The reduction rates measured for TIGRE compare quite favorably with those reported for other approaches to graph reduction, including Hyperlazy evaluation [22], the G-Machine [2,17,25], the Spineless, Tagless G-machine [24], TIM [11,36], and NORMA [28]. Due to differences in hardware platforms and compiling technology, direct comparisons are difficult to make. The most direct comparison can be made with NORMA,

which is a special-purpose computer dedicated to Turner-set combinator reduction. The reported reduction rate for NORMA is 250,000 RAPS.

15.7 The basis for the architectural study

During the implementation and performance measurements of TIGRE, we noted unexpected results on different platforms. We conjectured that the unexpected performance variations observed among TIGRE implementations were caused by hardware differences among platforms, especially with regard to cache organization and management. In order to better understand the operation of TIGRE, a set of cache simulations was run to measure TIGRE's use of cache memory.

Since our measurements on several hardware platforms have shown the DECstation 3100 to be the fastest available TIGRE implementation (despite the need for an interpretive threading loop), we used this machine's cache configuration as our starting point. This approach gives a starting point based on a real system. From this starting point, we examined how variations in cache organization affect program performance. (A study of cache behavior based on exhaustive search simulation techniques is described by Koopman et al. [21]).

The DECstation 3100 has a split instruction and data cache. During combinator-graph reduction execution, the instruction cache holds code to execute primitives of an abstract machine. The data cache contains the actual combinator graph, which is the abstract machine program being executed. Since the kernel of code required for graph reduction is small, previous simulations showed that the instruction cache on this machine experiences essentially a 100% hit ratio after the cache becomes filled with combinator code. Therefore, we concentrated our simulation efforts on the data cache performance.

Since graph reduction may be thought of as an interpretive process of executing a program expressed as a data structure, the data cache is actually the cache of prime importance. In this situation, the instruction cache is acting as a kind of microcode memory for storing code to execute primitive operations, and the data cache actually contains both the interpreted code (the program graph) and the program data. The approach of studying only the data cache has the added advantage of largely decoupling the particulars of the abstract-machine implementation (which affect instruction cache access) from the mechanics of graph reduction (which appears as accesses to the data cache).

In order to carry out the simulations, memory-access traces from TIGRE were used as input to the DineroIII trace-driven cache-simulator program [13]. In the first phase of the experimentation, an exhaustive exploration of a number of cache design parameters were tried in order to find the best combinations. An exhaustive search was performed in order to avoid the pitfalls of hill-climbing strategies that may become trapped at local extrema.

	Fib	NthPrime	8Queens	Real	Tak
cache miss ratio	0.1434	0.1768	0.1554	0.1595	0.1912
bus traffic ratio	0.5854	0.6262	0.5942	0.5971	0.6478

Figure 15.12: Cache performance for the baseline organization: 64K data cache, 4-byte block size, direct-mapping, write-through, and write-allocate.

On the basis of this exploration, and on the basis of DECstation 3100 characteristics, we arrived at the following "optimum" cache design parameter values: split I- and D-cache organization (with only the D-cache simulated), 64K byte data cache size, 4-byte block size, direct-mapped organization, write-through memory updates, and allocation on cache write miss. Kabakibo et al. [18] and Smith [30] provide more information on cache management strategies and terminology.

15.8 Parametric analysis

For the second phase of the experimentation, 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), 8Queens (the 8-queens problem), Real (infinite precision real arithmetic), and Tak (a program that tests recursive function calls). In all cases, between one and two million data memory accesses were simulated, with accesses to a heap space at least 320K bytes in size.

Figure 15.12 summarizes the results of simulating the baseline cache configuration. Two important characteristics emerge from the simulation. The cache miss ratio (percentage of cache accesses experiencing a cache miss) is a relatively high 14% to 19% for all the programs. Furthermore, the bus traffic ratio (the average number of 4-byte words transferred on the memory bus per cache access) is between 0.58 and 0.65. As a result, graph-reduction programs generate memory references in excess of DECstation 3100 available bus bandwidth (this is discussed in detail in a later section). We shall show that varying the cache parameters can decrease both the cache miss ratio and the bus traffic ratio dramatically.

15.8.1 Write allocation: The importance of a write-allocate strategy

A cache is said to perform write allocation when a memory write that generates a cache miss copies the data being written into a newly allocated cache block (allowing subsequent reads and writes to that address to achieve cache hits). A write-no-allocate policy does not write the data to cache, but instead transfers the data directly to memory.

Allocation strategy	Fib	NthPrime	8Queens	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

Figure 15.13: Cache miss ratios with varying write-allocation strategy.

Figure 15.13 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 cache miss ratios almost doubling when a write-no-allocate policy is used.

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 (when using a many current garbage collection algorithms). 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. A write-no-allocate policy will experience a second cache miss, while a write-allocate policy will often get a cache hit on the previously written element (as long as no intervening memory reference has bumped the node out of cache). 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.

As an example of the importance of this range of cache performances, the VAX 8800 mainframe uses a write-no-allocate strategy in managing its cache. This strategy is commonly used to simplify the cache control logic on machines with large cache block sizes. This strategy, combined with the longer latency for a cache miss than that found on the DECstation 3100, accounts for most of the performance difference between the two machines. In order to increase the VAX 8800's speed, our graph reduction code performs a dummy memory read (i.e. a memory read, the results of which are discarded) each time a group of heap cells is allocated. This forces allocation of a cache line before the initial write to the heap cell, and can increases overall performance

by 20% despite the overhead of executing extra instructions to perform the memory reads.

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 probably extends beyond the graph reduction domain. Since a heap, by its very nature, is used in a write-followed-by-read manner, a write-allocate cache policy is likely to be important to support any system that uses a heap.

15.8.2 Block size: Strong spatial locality means larger block size

Figure 15.14 shows the results of varying block size (the number of bytes in the smallest allocated unit of memory in the cache) 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. This suggests very strong spatial locality. This spatial locality is probably due to the fact that heap nodes are allocated from the heap space in sequential memory locations.

One could, at first glance, decide to build a machine with a 256 byte cache block size based on the miss ratios alone. For conventional programs, this decision could be unwise, because the bus traffic ratio (the number of words of data moved by the system bus) often 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 128 and 256 bytes in size. So, a machine with a 128 byte cache block size appears to be entirely reasonable for this application.

Large block sizes are seldom seen in practice, because most conventional programs do not have enough spatial locality to justify very large block sizes. But, the significant performance increases possible with this application give strong incentive to consider large block size. Even a block size of 16 elements brings dramatic reductions in the miss ratio.

15.8.3 Write through policy

A write-through cache transmits modified data to system memory every time the processor performs a store operation. A copy-back cache buffers the data in cache until it must be flushed to make the cache block available for another address. If multiple writes are performed to a single address, a copy-back cache eliminates the requirement to use memory bus bandwidth for all but the ultimate write.

Figure 15.15 shows the traffic ratio for a write-through versus copy-back management

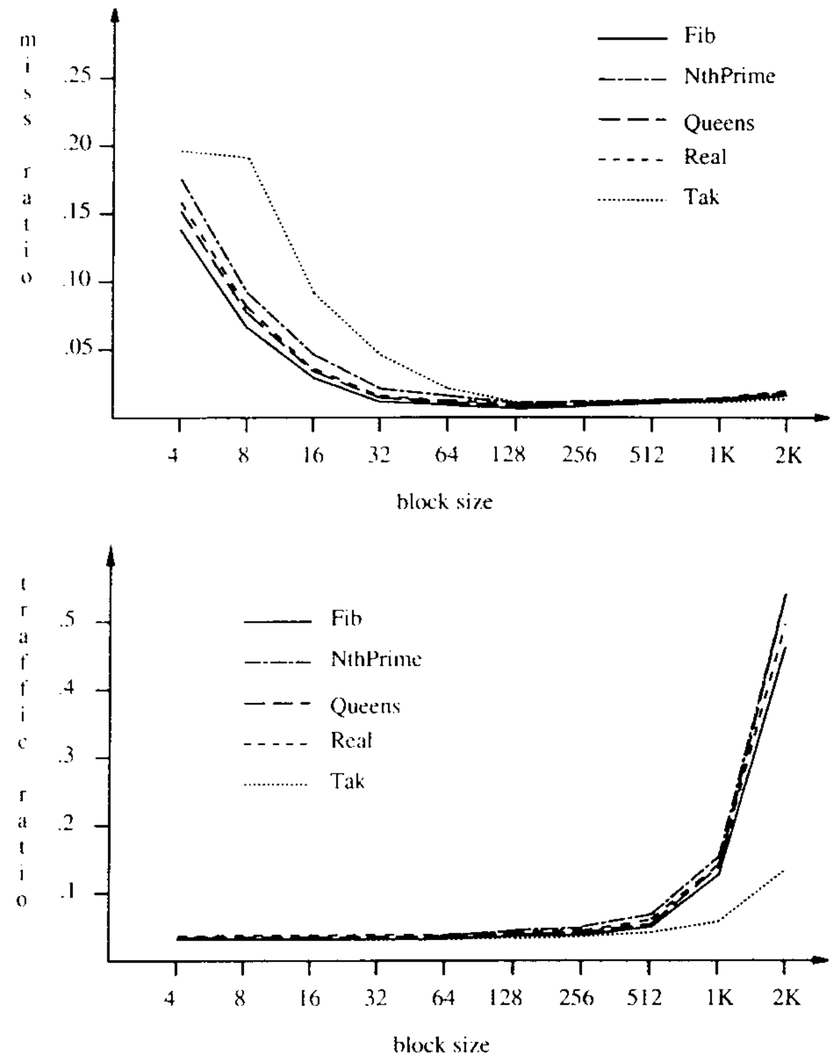

Figure 15.14: Cache performance with varying block size

Memory update	Fib	NthPrime	8Queens	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

Figure 15.15: Cache traffic ratios with varying write-through strategy.

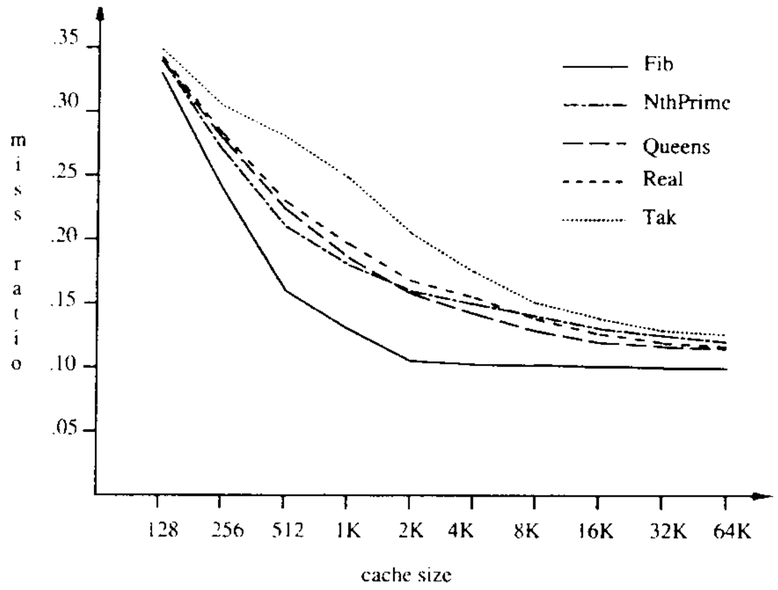

Figure 15.16: Cache performance with varying cache size

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. This is caused by the fact that a very high percentage of memory accesses are memory writes (between 44% and 46% of memory references were writes on the programs simulated). This can cause severe problems with system performance by causing memory bus saturation.

Since one of the promises of combinator-graph reduction is simple parallel program execution, and since many multiprocessors are built with a common memory bus, bus traffic is a prime consideration in predicting the limits to parallel processing performance. Since graph reduction causes a high number of memory writes, use of copy-back cache is highly desirable to avoid bus saturation for a multiprocessor system. However, even with copy-back cache the bus traffic is reduced by less than a factor of two, indicating that a multiprocessor using a common data bus could have a severe bus bandwidth bottleneck.

15.8.4 Cache size

Figure 15.16 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

Associativity	Fib	NthPrime	8Queens	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

Figure 15.17: Cache miss ratios with varying associativity.

ratio. So, conventional hardware platforms which typically have more than 16K bytes of cache seem to be adequate with respect to cache size.

15.8.5 Associativity

Figure 15.17 shows the results of varying the associativity of the cache from direct mapped (1-way associative) to 8-way associative. 2-way set associative seems to bring a slight performance improvement, but beyond that there is little or no 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 [27]. The miss ratio penalty of using such a direct mapped cache over a set associative cache is quite small, so the performance tradeoff of using direct mapped caches seems desirable.

15.8.6 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.

Simulation indicates that for Fib, 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 combined instruction and data cache 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 using the write-through memory updating policy) 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 and Koopman [29]), and noting that an average combinator takes 30.09 clocks, this gives a penalty of:

$$0.00136 * 13/30.09 \text{ (clocks per combinator)}$$

$$= 0.06\%$$

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

$$475000 * 1.0106 = 480000 \text{ RAPS}$$

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 simulators available to us did not permit exploring the behavior of a write buffer, but an examination of the code shows that write buffer stalls are likely.

15.9 The potential of special-purpose hardware

15.9.1 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. While it is understood that adding complexity to a RISC architecture may be undesirable (because, for example, it may reduce the maximum clock frequency for existing instructions), this is a way to obtain an approximation of potential benefits.

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. 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.

15.10 Improvements in cache management

15.10.1 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).

15.10.2 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 128 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) could probably be utilized efficiently by a supercombinator compiler to get a 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 theoretical minimum (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.

15.11 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.

15.11.1 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.[2]

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. This implementation is likely to be more desirable for split-cache systems.

15.11.2 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 requires the use of delayed branches, 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

[2]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.

cumulative optimizations	clock cycles 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 indirect addressing	12.36
8-byte store instructions	11.47

Figure 15.18: Possible performance improvements to the MIPS R2000 for TIGRE.

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.

15.11.3 Double-word stores

TIGRE is often 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 double-word. 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, the supercombinator implementation of Fib can make use of 1.33 64-bit stores per combinator.

Figure 15.18 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. This is a significant speedup, but probably does not justify the production of a special CPU for uniprocessor implementations. Rather, the results should indicate desirable

architectural features that should be sought when selecting a standard RISC platform for combinator-graph reduction.

15.12 Results

We have described an abstract machine for combinator-graph reduction, and shown that it has good performance compared to other graph reducers and closure reducers. Using TIGRE, we have performed architectural simulations that show it has unusual execution characteristics, including: a very strong dependence on a write-allocate strategy for efficient execution, a high degree of spatial locality, and a high proportion of memory writes to total memory accesses. Thus, a system which will execute these programs efficiently should ideally have a write-allocate cache with copy-back memory updating, and a relatively large block size of at least 16 or 32 bytes. Since the combination of copy-back updating with write-allocation requires additional complexity in control logic, this combination is not likely to appear without evidence to suggest that it is useful for some applications. This study is a piece of evidence in that vein.

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.

15.13 Further work

Our simulated programs are restricted to small benchmarks. The measurements of TIGRE behavior are limited by a lack of extensive software support. Additionally, large lazy functional programs are difficult to find and we have been hampered by a lack of extensive compiler support. Therefore, we plan to repeat the experiments on TIGRE when a larger software base is available. Also, it would be revealing to compare TIGRE against other abstract machines using a comprehensive benchmark suite with comparable implementations and compilers. This would not only improve understanding of the strengths and weaknesses of TIGRE, but also of common architectural requirements for combinator reduction techniques in general.

A problem with parallel graph reduction in the past has been one of practicality. If individual graph reduction processors do not execute within a factor of 100 times the speed of a uniprocessor running an imperative language, there seems little point in building a 100-processor system. TIGRE and other fast combinator reducers (such as TIM) make it feasible to consider the design of a parallel graph reduction engine that can potentially run programs more quickly than a uniprocessor using imperative languages. While TIM makes parallel closure reduction machines such as the Grip project

[23] practical, TIGRE may make parallel pure graph reduction viable. Graph reduction appears to be a more obvious program manipulation technique than other methods, and therefore *may* allow better insight into parallel execution issues.

References

[1] A. Augusteijn and G. van der Hoeven. (1984) Combinatorgraphs as self-reducing programs.

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

[3] J. Backus. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. of the ACM*, August 1978, **21**(8) 613-641

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

[5] H. P. Barendregt. (1981) *The Lambda Calculus: Its Syntax and Semantics*, Elsevier, New York.

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

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

[8] G. Burn, S. Peyton Jones, and J. Robson. (1988) The spineless G-Machine. In: *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird UT, July 25-27.

[9] H. B. Curry and R. Feys. (1968) *Combinatory Logic, Volume 1*, North-Holland.

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

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

[12] J. Hennessy and D. Patterson. (1990) *Computer Architecture: a quantitative approach*, Morgan Kaufmann Publishers, San Mateo, CA.

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

[14] P. Hudak and B. Goldberg. (1985) Serial combinators: "optimal grains of parallelism. In *Conference on Functional Programming Languages and Computer Architecture, Nancy*, Springer Verlag, pages 382-399.

[15] P. Hudak, P. Wadler, et al. (1990) *Report on the Programming Language Haskell, Version 1.0*, Research Report YALEU/DCS/RR-777, April.

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

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

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

[19] P. Koopman. (1990) *An Architecture for Combinator Graph Reduction*, Academic Press, Boston.

[20] P. Koopman and P. Lee. (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.

[21] P. Koopman, P. Lee, and D. Siewiorek. (1990) "Cache Performance of Combinator Graph Reduction". In: *1990 International Conference on Computer Languages*, March 12-15, 39-48.

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

[23] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. (1987) GRIP - a high-performance architecture for parallel graph reduction. In: Kahn, G. (ed.) *Functional Programming Languages and Computer Architecture, Portland OR, September 14-16*, Springer-Verlag, 98-112

[24] S. L. Peyton Jones and J. Salkild. (1989) The spineless tagless G-machine. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture, London*, pp. 184-201, September.

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

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

[27] S. Przybylski, M. Horowitz, and J. Hennessy. (1988) Performance tradeoffs in cache design. In *The 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii, 30 May - 2 June*, IEEE Computer Society Press, pp. 290-298

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

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

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

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

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

[33] D. A. Turner. (1976) *SASL Reference Manual*, University of St. Andrews.

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

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

[36] S. C. Wray and J. Fairbairn. (1988) Non-strict languages (197) programming and implementation (draft) October 16, 1988.

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

Topics in Advanced Language Implementation

Edited by

PETER LEE

The MIT Press
Cambridge, Massachusetts
London, England

©1991 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Topics in advanced language implementation / edited by Peter Lee.

p. cm.

Includes bibliographical references and index.

ISBN 0-262-12151-4

1. Programming languages (Electronic computers) I. Lee, Peter, 1960– .

QA76.7.T65 1991

005.13—dc20 91-9709

CIP