Cache Behavior of Combinator Graph Reduction

PHILIP J. KOOPMAN, Jr., PETER LEE, and DANIEL P. SIEWIOREK
Carnegie Mellon University

The results of cache-simulation experiments with an abstract machine for reducing combinator graphs are presented. The abstract machine, called TIGRE, exhibits reduction rates that, for similar kinds of combinator graphs on similar kinds of hardware, compare favorably with previously reported techniques. Furthermore, TIGRE maps easily and efficiently onto standard computer architectures, particularly those that allow a restricted form of self-modifying code. This provides some indication that the conventional "stored program" organization of computer systems is not necessarily an inappropriate one for functional programming language implementations.

This is not to say, however, that present day computer systems are well equipped to reduce combinator graphs. In particular, the behavior of the cache memory has a significant effect on performance. In order to study and quantify this effect, trace-driven cache simulations of a TIGRE graph reducer running on a reduced instruction-set computer are conducted. The results of these simulations are presented with the following hardware-cache parameters varied: cache size, block size, associativity, memory update policy, and write-allocation policy. To begin with, the cache organization of a commercially available system is used and then the performance sensitivity with respect to variations of each parameter are measured. From the results of the simulation study, a conclusion is made that combinator-graph reduction using TIGRE runs most efficiently when using a cache memory with an allocate-on-write-miss strategy, moderately large block size (preferably with subblock placement), and copy-back memory updates.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*cache memories*; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*simulation*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; D.3.4 [**Programming Languages**]: Processors—*compilers, interpreters*

General Terms: Languages, Performance

Additional Key Words and Phrases: Abstract machine, combinators, graph reduction, self-modifying code

A preliminary version of this paper appeared in the *Proceedings of the IEEE 1990 International Conference on Computer Languages* (New Orleans, LA., Mar. 12-15, 1990) [20]. This research was supported in part by NASA/Goddard under contract NAG-5-1046, also 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.

Authors' addresses: P. Koopman, United Technologies Research Center, 411 Silver Lane MS 58, East Hartford, CT 06108; P. Lee and D. Siewiorek, Schoo l of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

1. INTRODUCTION

We present the results of our experiments with an abstract machine for reducing combinator graphs. The abstract machine, called TIGRE (the Threaded Interpretive Graph Reduction Engine), treats combinator graphs as self-modifying threaded programs, in a manner similar to that described by Augusteijn and van der Hoeven [2]. We have found that this method reduces combinator graphs at a rate that compares quite favorably with previously reported techniques on similar hardware [18, 19]. Furthermore, TIGRE maps remarkably easily and efficiently onto standard computer architectures, particularly those that allow a restricted form of self-modifying code. 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 [4].

This is not to say, however, that present-day computer systems are well-equipped to reduce combinator graphs. During the development of 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 [9] performed 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. This result 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.

This paper, then, is a report on our study. We begin with a brief review of combinator-graph reduction. This is followed by a description of the TIGRE abstract machine and its implementation, as well as a report on its performance and comparisons with other methods for reducing similar kinds of combinator graphs. Then, we present the results of simulating a TIGRE graph reducer running on a reduced instruction set computer with the following hardware-cache parameters varied: cache size, block size, associativity, memory update policy, and write-allocation policy.

Our simulation experiments were conducted in two stages, the first stage being an exhaustive test of selected values for all combinations of parameters. The result of this stage of experimentation, which we reported on earlier [20], showed that there were no local extrema in cache miss behavior as a function of cache design choices. In this paper we begin with 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

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

accuracy of our simulations, we compare the results with measured performance on real hardware. From the results of this study, we can conclude that combinator-graph reduction using TIGRE runs most efficiently when using a cache memory with an allocate-write-on-miss strategy, moderately large block size (preferably with subblock placement), and copy-back memory updates.

2. BACKGROUND

Implementation techniques for lazy functional programming languages such as SASL [33] and Haskell [14] have long been the subject of research. Turner [35] described a technique for implementing lazy functional languages, based on what is sometimes referred to as *SK-combinator reduction*. This idea is based on the well-known fact that all of the variables in a *λ-expression* can be abstracted by transforming it into an applicative expression involving only *combinators*. A combinator is simply (the name of) a closed λ-expression [5]. In addition to being closed, there is typically the additional stipulation that λ-abstractions not occur in the argument position of an application, as specified by the following grammar:

$$\textbf{comb} ::= \lambda v.\ \textbf{comb} \mid \lambda v.\ \textbf{exp}$$
$$\textbf{exp} ::= c \mid v \mid \textbf{exp}\ \textbf{exp} \mid (\textbf{exp})$$

where c and v range over constants and variables, respectively. (As usual, applications associate to the left.) Three combinators of particular interest are called **S**, **K**, and **I**.

$$\mathbf{S} = \lambda f.\ \lambda g.\ \lambda x.\ fx(gx)$$
$$\mathbf{K} = \lambda x.\ \lambda y.\ x$$
$$\mathbf{I} = \lambda x.\ x$$

The important property is that any λ-expression—and hence functional program—can be transformed into an expression consisting solely of applications of these combinators by a so-called "bracket abstraction" algorithm [36]. (Actually, only **S** and **K** are necessary, as **I** = **SKK**.)
The bracket abstraction of a variable v from a λ-expression e, expressed as $[v]e$, can be defined as follows:

$$[v]v = \mathbf{I}$$
$$[v]v' = \mathbf{K}v' \qquad (v \neq v')$$
$$[v]c = \mathbf{K}c$$
$$[v]ee' = \mathbf{S}([v]e)([v]e')$$
$$[v]\lambda v'.\ e = [v]([v']e)$$

where c ranges over combinator names in addition to constants.

For a practical programming language, additional combinators representing primitive operations, such as arithmetic operators, are needed for efficient execution. Other nonprimitive combinators can also be added (along with suitable extensions to the bracket abstraction algorithm) so as to reduce the size of the resultant combinator expressions.

With all of the variables abstracted from a functional program, the resulting combinator expression is easily represented as a binary tree with names of combinators appearing at the leaves and internal nodes representing applications. As a further optimization, the tree is transformed into a graph in which subgraph sharing denotes the occurrence of common subexpressions in the combinator program, cycles denote recursion and combinator definitions correspond to graph-rewrite rules. (See for example in Figure 1, the graph-rewrite rule 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 graph is rewritten according to the corresponding rewrite rule. This process is repeated on the new graphs, until finally an irreducible graph is produced, at which point program execution is complete. The consistent reduction of the left-most spines of the graphs corresponds to a normal-order reduction strategy which, in conjunction with the sharing and destructive update of pointers to subgraphs, leads to the so-called "lazy" evaluation of functional programs.

Besides the great advantage in efficiency gained from the sharing of subgraphs, 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, lazy evaluation of functional programs, even via combinator-graph reduction, is in practice much less efficient than applicative-order (or "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.

As we mentioned earlier, the most straightforward way to improve efficiency is to add more combinators, thereby reducing both the size of the graphs and the number of reductions needed for program execution. This approach was used by Turner [36], who developed what we refer to as the "Turner set" of SK-combinators. (In Section 4 we give more details about the combinators, benchmark programs and compilation model used in our experiments.) Another significantly more sophisticated development along these lines, first proposed by Hughes [15], 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. Rather than using a fixed set of combinators, 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.

The issues involved in SK-combinator reduction differ markedly from those for supercombinator reduction. Promising approaches to supercombinator

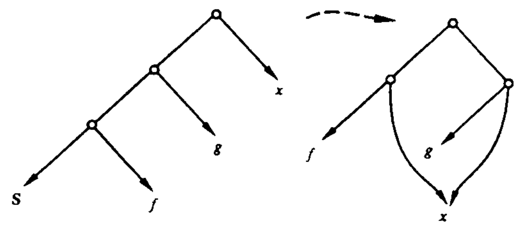
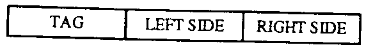


Fig. 1. Graph rewriting specified by the S combinator.

Fig. 2. Typical representation of a graph node.



reduction include TIM [10] and the "Spineless, Tagless, G-machine" [24]. Of particular importance in supercombinator reduction is the interaction between the reduction technique and compile-time analyses such as strictness and sharing analysis [1]. This interaction leads to a notion of "compiled" graph reduction, to which various optimization techniques can be applied. In this paper, we focus our attention on pure SK-combinator reduction rather than supercombinator reduction, despite the fact that supercombinator reduction is generally a more efficient way to implement lazy functional programs. We concentrate on SK reduction because we desire to study the behavior of graph reduction in the absence of influences from the compile-time analyses and other aspects of the compilation technology. In our simulation experiments, it is particularly difficult to quantify the effects of implicit assumptions about the underlying hardware (such as the size of the register set) that are necessarily made by compiled systems. We will return to this point in our concluding remarks and speculate on how supercombinator reduction might have affected the outcome of our experiments.

3. THE TIGRE ABSTRACT MACHINE

The major difficulties in performing graph reduction efficiently are in traversing the graph's left spine (sometimes referred to as "stack unwinding") and in the case 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 2 shows that 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, unwind-

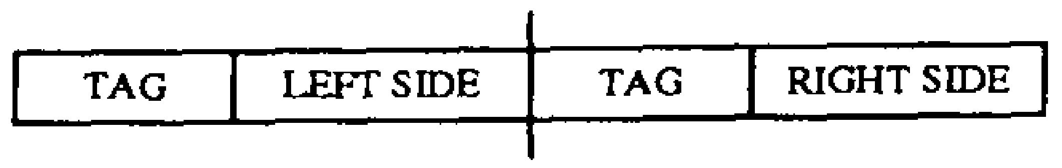


Fig. 3. Generalized representation of a graph node.

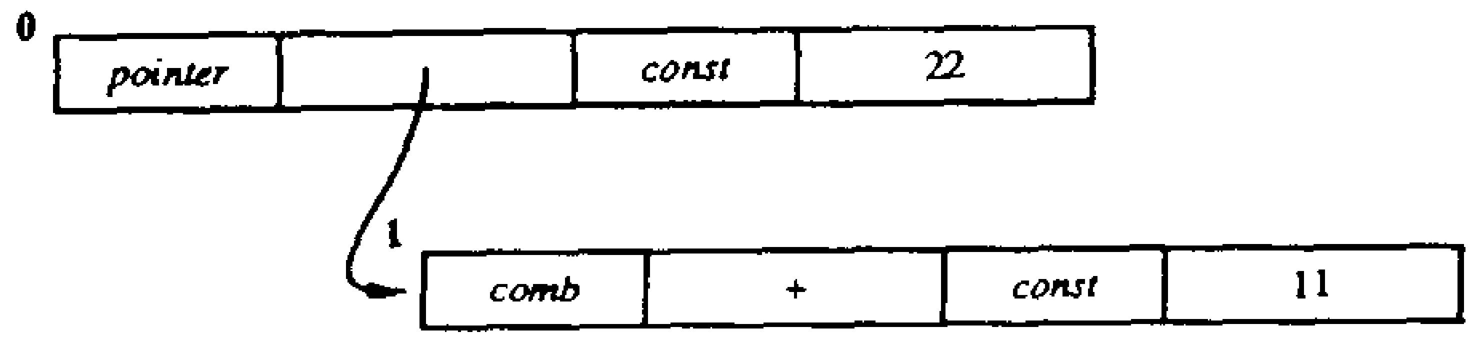


Fig. 4. Example graph for the expression ((+ 11) 22).

ing a node while traversing the spine requires four instructions, including this double-indirect jump through the entry table [25]:

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

One of the key points of TIGRE is the elimination of most of this overhead for traversing graph nodes during the 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 eliminate the tags in several stages.

Figure 3 shows a generalized node representation which has tags associated with both the left-hand and right-hand side fields of the node. Figure 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 kinds of tags are shown in the example, typically more tag types are used in actual implementations.

As a first step in eliminating the tags, we replace the fields containing constant values by pointers to indirection nodes (i.e., nodes that apply the I combinator). Figure 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 reduction. 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 of indirection nodes. If these **I** combinators in the left-hand sides of constant nodes are renamed as

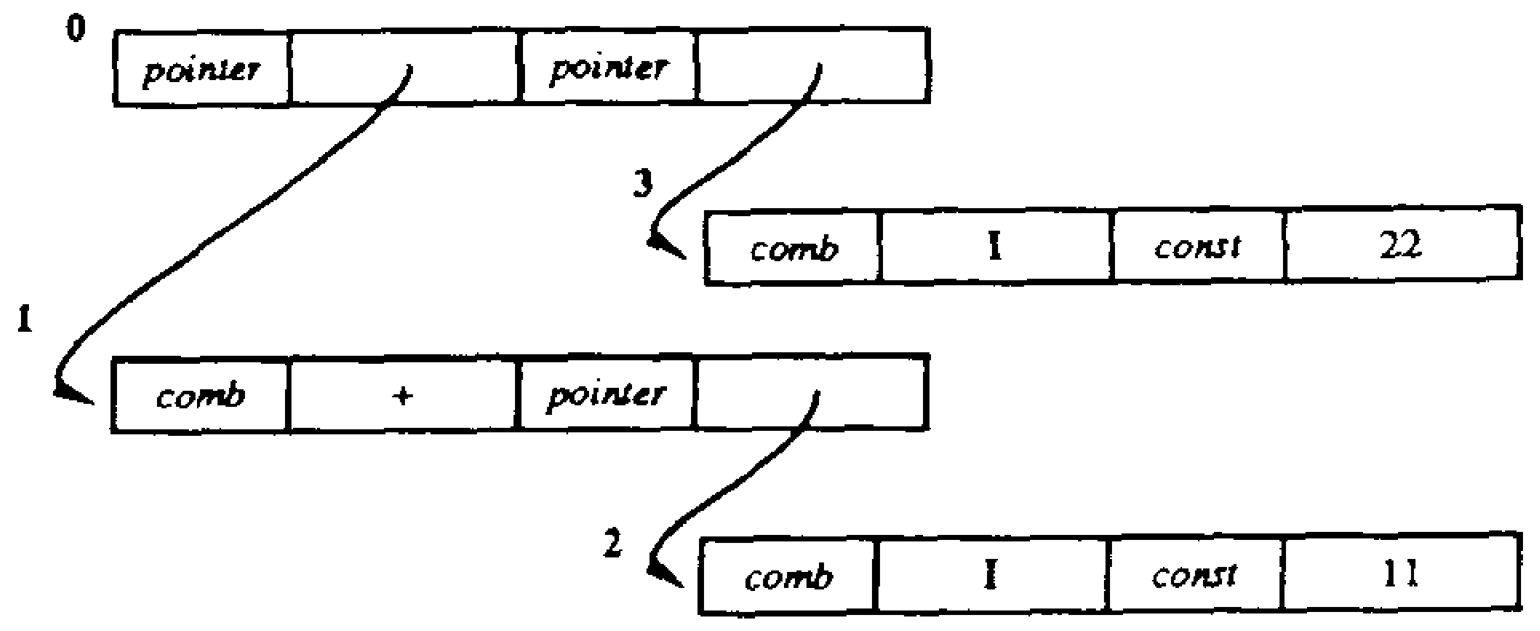


Fig. 5. Example graph with indirection nodes.

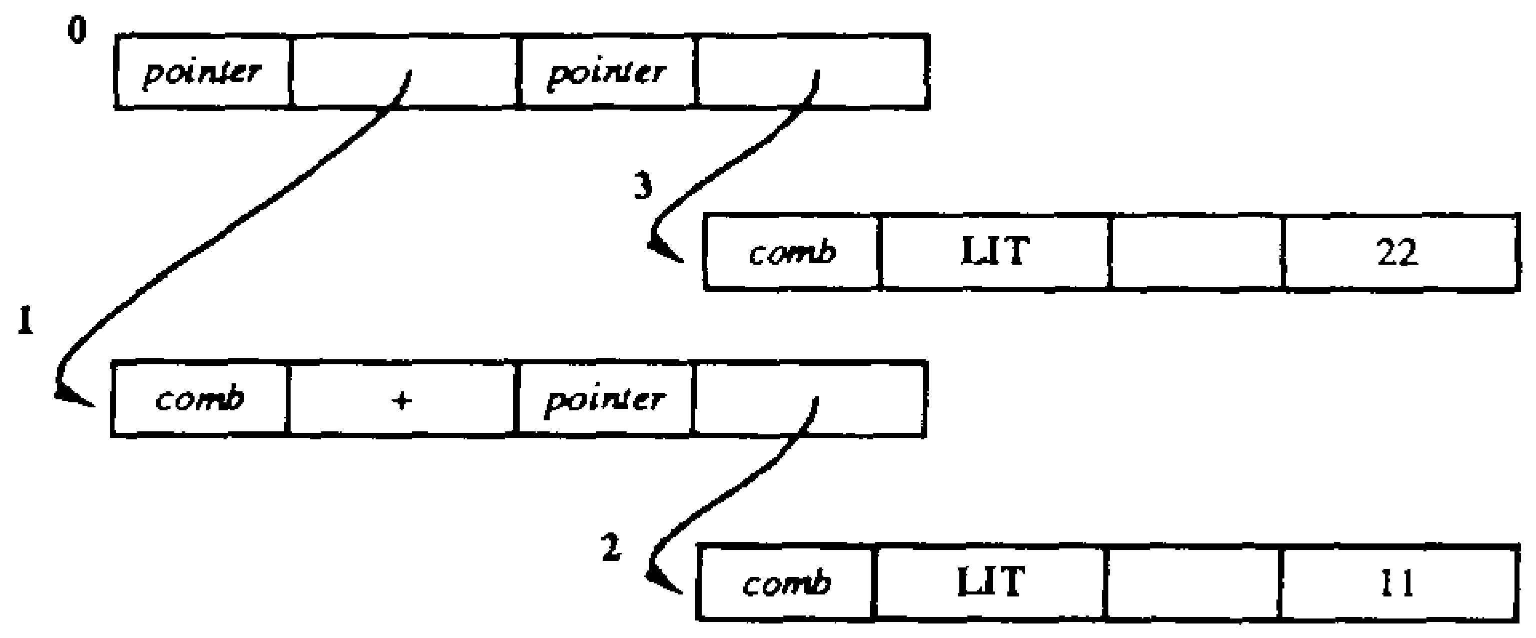


Fig. 6. Example graph with LIT nodes.

LIT combinators (short for "literal value" combinators), as shown in Figure 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 (for example, **FLIT** for floating point constants) in a similar manner.

The graph shown in Figure 6 now only has two tag types: combinator and pointer. At this point, a number of standard techniques can be used to reduce tag-checking costs. For instance, all nodes and therefore pointer values can be aligned on four-byte boundaries. (On some machines, this improves speed or is even required.) The lowest bit of a cell's contents can then be used as a one-bit tag.

The case analysis for numeric constants has been replaced by the need to reduce **LIT** combinators, but the amount of tag checking on all other cells has been reduced. This representation is used in interpreted implementations of graph reduction. For example, in the C programming language, the reducer can be programmed to unwind the stack by looping while scanning the low-order bit of left-hand side cells. When a nonpointer value is encountered, a case statement can then be used to jump to the appropriate graph-rewriting code. This technique corresponds precisely to threaded code interpretation [6], and is used by the version of TIGRE which is implemented in C.

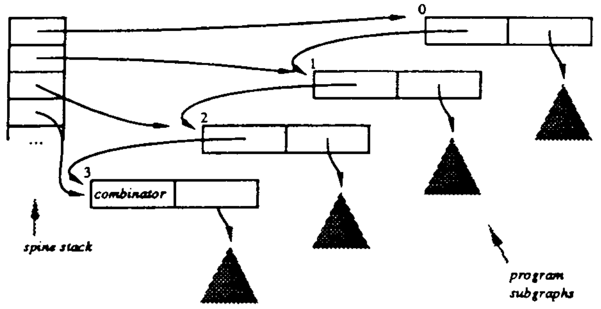


Fig. 7. Graph spine traversal.

3.1 Self-Modifying Threaded Code

There is an additional key insight which provides an approximately two-fold speedup in the execution speed of graph reduction. This is gained by exploiting the hardware support for graph traversal that already exists in many conventional processors.

The generic graph shown in Figure 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 a threaded program. As control flows from node 0 to node 1 to node 2 to node 3 in the graph of Figure 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 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 code to be executed, so the actual code addresses of the combinator action routines can be stored

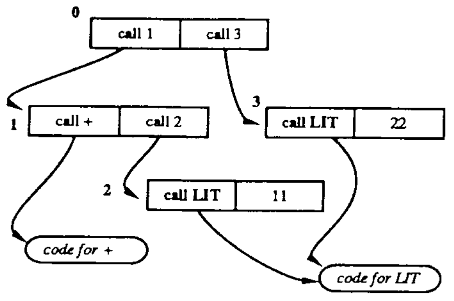


Fig. 8. Example graph as self-modifying code.

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. Then, the combinator will have all its arguments pointed to by the spine stack (which is now 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 8 shows our running example of ((+ 1 1) 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 reduction engine.

In such self-modifying implementations of TIGRE, graph nodes are represented by triples of 32-bit cells instead of pairs of 32-bit cells as found in interpreted implementations.[2] 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 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 th e program counter onto the return address stack, as opposed to the slower funct ion call instructions which automatically allocate stack frames.)

[2]There are associated costs with dedicating a third cell per node to increase speed. Aside from the obvious memory space growth, three-word cells do not align well with cache blocks of 8 or 16 bytes. However, the high spatial locality of graph reduction (which is discussed in Section 5.3) gives high probability that it is desirable to have adjacent nodes resident in cache simultaneously, minimizing wasted cache capacity caused by misalignment. A third cell is commonly used in other graph reducers to speed up tag-checking; we simply use that same node for storing a subroutine call instruction instead (and eliminate tag checking).

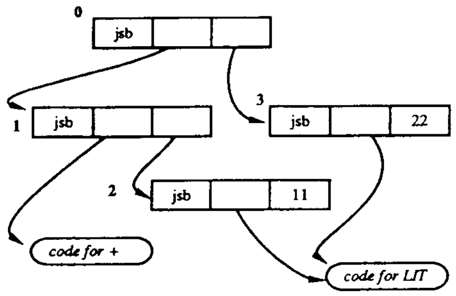


Fig. 9. Example graph on a VAX.

Evaluation of a program graph is initiated by performing a subroutine call to 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. When a node 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 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 can be 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).

The processor is in no sense interpreting the graph. It is *directly executing* the data structure and 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.

3.2 Implementing TIGRE

The availability of a fast subroutine-call instruction on most modern architectures makes the TIGRE technique usable, in theory, on most computers. In practice, however, there are issues having to do with modifications of the instruction stream that make the approach difficult to implement on some machines. In particular, machines with split instruction and data caches typically do not directly support self-modifying code because the instruction

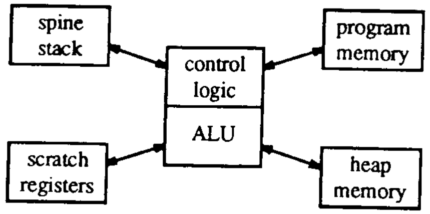


Fig. 10. Block diagram of the TIGRE abstract machine.

```
DO_S:                            ;  (((S f) g) x) -> ((f x) (g x))
 need2(r8,r7)                    ; allocate 2 cells from heap
 movl *(sp)+,r6                  ; pointer to f into r6, popping spine stack
 movl r6,(r8)                    ; store f to build (f x)
 movl *(sp),(r7)                 ; store g to build (g x)
 movl 4(sp),r10                  ; move pointer to pointer to x in r10
 movl (r10),4(r7)                ; store x in (g x)
 movl (r10),4(r8)                ; store x in (f x)
 movab -2(r8),-4(r10)            ; rewrite subgraph root to point to (f x)
 movab -2(r7),(r10)              ; rewrite subgraph root to point to (g x)
 movab 4(r8),(sp)                ; repoint top spine stack element to (f x)
 jmp(r6)                         ; start evaluating f

DO_I:                            ;  (I f) -> f
 movl *(sp),r6                   ; move pointer to f into r6
 movab 4(sp),sp                  ; pop single argument from spine stack
 jmp (r6)                        ; "fall-through" to execute f

DO_K:                            ;  ((K f) x) -> f (note similarity to DO_I)
 movl *(sp),r6                   ; move pointer to f into r6
 movab 8(sp),sp                  ; pop BOTH arguments from spine stack
 jmp (r6)                        ; "fall-through" to execute f
```

Fig. 11. VAX-TIGRE implementation of S, K, and I combinators.

cache does not update modified cache-resident instructions. These problems can be viewed as the result of inappropriate trade-offs in system design (at least for the application of graph reduction) rather than 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 efficient graph reduction makes good use of self-modifying techniques.

Figure 10 shows a block diagram of a TIGRE abstract machine. As a minimum, TIGRE requires a processing unit (with ALU and control logic), a spine stack/subroutine return stack, a small collection of registers for holding temporary values, program memory for holding combinator definitions, and heap memory for holding the graph nodes. In the VAX implementation,

```
DO_IF:                     ;  (((IF x) f) g)  ->  (I f)  (if x true)
                           ;                  ->  (I g)  (if x false)
 movl *(sp)+,r6            ; pointer to x into r6, popping spine stack
 jsb (r6)                  ; evaluate x, result in r11
 movl 4(sp),r10            ; pointer to root of subgraph in r10
 tstl r11                  ; test truth of r11
 jeql L39
 movl *(sp),(r10)          ; if true, overwrite with f (else leave g)
L39:  $DO_I,-4(r10)        ; place I in left side
 movl (r10),r6             ; place either f or g in r6
 addl2 $8,sp               ; pop last 2 arguments from spine stack
 jmp (r6)                  ; jump to f or g

DO_LIT:                    ;  (LIT n)  -> leave n in register r11
 movl *(sp)+,r11           ; load right-hand-side into r11
 rsb                       ; end of evaluation - result is in r11

DO_PLUS:                   ;  ((+ x) y)  -> (LIT z)
 movl *(sp)+,r6            ; get address of x in r6
 jsb (r6)                  ; evaluate x, leaving result in r11
 movl *(sp),r6             ; get address of y in r6
 pushl r11                 ; stash value of x
 jsb (r6)                  ; evaluate y, leaving result in r11
 addl2 (sp)+,r11           ; recover value of x from stack, compute z
 movl (sp)+,r9             ; pop address of root of subgraph into r9
 movl $DO_LIT,-4(r9)       ; place LIT combinator in left-hand side
 movl r11,(r9)             ; place result z in right-hand side
 rsb                       ; end of evaluation - result is in r11
```

Fig. 12. VAX-TIGRE implementation of some primitive operation combinators.

the stack memory, combinator memory, and graph memory all reside in the same memory space. Figures 11 and 12 give sketches of the VAX assembly code implementing 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.

Note that this code does not explicitly check for the case that an insufficient number of arguments are supplied to a combinator. Such a situation is treated as an exceptional condition which can be trapped in a number of ways. Our current implementation places several pointers to exception-trapping code beneath the base of the spine stack when the stack is initialized. Thus, any combinator that attempts to evaluate arguments that have not been supplied will generate a trap, invoking the exception-handling code. An alternative method is to use the virtual-memory hardware to protect the memory page at the base of the spine stack. Then, a stack underflow will generate a page trap, which could be handled simply by returning the weak-head normal form graph.

In TIGRE, traversing the leftmost 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 11 do not modify the graph at all, but rather redirect the flow of control of the graph evaluation, thereby 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 garbage collection performance by, for example, leaving subgraphs 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.

4. MEASURING THE PERFORMANCE OF TIGRE

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). Table I gives some benchmark results for the various versions of TIGRE on several hardware platforms. In addition to total CPU time, a speed rating in terms of reduction-applications-per-second (RAPS) is given. Each graph rewriting induced by the execution of a combinator counts as a reduction application.

The C implementation, which we use on Sun workstations and the Cray Y-MP, uses threaded interpretation as described in Section 3. The version written in VAX assembler uses the jsb instruction to perform self-modifying threaded execution of the graph as discussed in Section 3.1. The MIPS R2000 assembly-language implementation, for use on the DECstation 3100 workstation, uses a carefully written threaded interpretive loop instead of direct execution. This is done because the instruction set does not provide a full subroutine-call instruction, and also the instruction cache is not automatically updated when code is modified.

In all of the implementations, simple stop-and-copy garbage collection [11] is used. Every benchmark is run with a heap memory which is small enough that several dozen garbage-collection cycles are required. As might be expected, the versions of TIGRE implemented in assembly language are significantly faster than those versions implemented in C.

The various TIGRE implementations have been run on a variety of hardware platforms. The DECstation 3100 is a 16.7 MHz MIPS R2000-based workstation. The VAX 8800 is a 22 MHz mainframe. The VAXstation 3200 is a high-end microVAX workstation. Cray Y-MP [26] is a vectorized supercomputer that has a fast scalar processing unit. The Sun 3/260 system is a 24 MHz Motorola 68020 workstation with cache memory. The Sun 3/75 system is a 16 MHz 68020 workstation with no cache memory. The C implementation on the Sun 3/75 and Sun 3/260 workstations is compiled by the GNU C compiler [31] with the optimization switch turned on.

The benchmark programs, excerpts of which are shown in Figure 13, are written in a polymorphic functional language which is essentially a simple

Table I. Benchmark Results for TIGRE

Platform	Program	Time (sec)	Speed (RAPS)
DECstation 3100 (16.7 MHz)	SKIfib(23)	2.20	495000
	Fib(23)	1.58	470000
	NFib(23)	2.68	484000
	Tak	12.58	420000
	NthPrime(300)	2.60	364000
	8Queens(20)	5.63	433000
VAX 8800 (22 MHz)	SKIfib(23)	2.82	387000
	Fib(23)	2.10	355000
	NFib(23)	3.55	366000
	Tak	16.07	329000
	NthPrime(300)	3.91	242000
	8Queens(20)	8.33	293000
VAXstation 3200	SKIfib(23)	6.33	172000
	Fib(23)	4.80	155000
	NFib(23)	8.23	158000
Cray Y-MP (167 MHz)	SKIfib(23)	3.09	352000
	Fib(23)	2.40	310000
	NFib(23)	4.25	305000
Sun 3/260 (24 MHz)	SKIfib(23)	8.62	126000
	Fib(23)	7.01	105000
	NFib(23)	12.37	105000
Sun 3/75 (16.7 MHz)	SKIfib(23)	14.62	75000
	Fib(23)	12.75	58000
	NFib(23)	22.02	59000

To obtain these benchmark results, the C implementation of TIGRE was used for the Sun and Cray timings. Self-modifying assembly code was used for the VAX timings. The DECstation timings were obtained from a threaded interpretive implementation written in MIPS R2000 assembler.

subset of Haskell [14]. Our compiler performs polymorphic type inference, followed by translation to the intermediate language FLIC [22]. The FLIC terms are then compiled into combinator graphs. The combinator graph output produced for the Fib benchmark program is shown in Figure 14. No strictness or sharing analysis, or any other kind of optimizations beyond compiling to a fixed set of combinators, are used. The precise set of combinators and the bracket abstraction algorithm used in our compiler are described by Jones [25, Ch. 16]. Included in this set of combinators, in addition to **S**, **K**, and **I**, are **B**, **C**, **S′**, **B***, and **C′**. Because most of these combinators were first proposed by Turner [36], we refer to this as the "Turner set" of combinators. Furthermore, a complete collection of primitive operation combinators as specified in the FLIC language are also used.

The Fib(n) benchmark program is the standard recursive program for computing the nth Fibonacci number. SKIfib(n) is the same program, but compiled using only the **S**, **K**, and **I** combinators (in addition to FLIC primitives). The Nfib(n) program is the commonly reported benchmark that tallies the number of recursions used in computing a Fibonacci number. Tak (heavy use of function calls), NthPrime(n) (computation of the nth prime),

```
fib 0 = 0
fib 1 = 1
fib (n+2) = fib (n+1) + fib n

nfib 0 = 1
nfib 1 = 1
nfib (n+2) = 1 + nfib (n+1) + nfib n

tak x y z { not (y<x) } = z
tak x y z = tak (tak (x-1) y z) (tak (y-1) z x) (tak (z-1) x y)

nthprime n =
  nth n (2:sieve [3,5..])
  where
   sieve (p:r) = p:(sieve [x|| x <- r, with x mod p == 0])

eightqueens n =
  nth n (queens n)
  where
   queens 0 = [[]]
   queens (n+1) = [q:b|| b <- queens n, q <- [1..8], with safe q b]

   safe q b = forall [not (checks q b i)|| i <- index b]

   checks q b i = (q == b||i) | (abs(q-b!!i) == i+1)
```

Fig. 13. Excerpts of lazy functional programs used in benchmark studies. For our benchmarks, we use a simple polymorphic language based on Haskell.

and 8Queens(n) (computation of n solutions to the 8-queens problem) are versions of other commonly used benchmarks.

We now attempt to make some comparisons with two other techniques for reducing similar kinds of combinator graphs. Comparisons of this type are fraught with peril because of varying execution platforms and operating system environments. Nonetheless, we can draw a few rough conclusions about the performance of the TIGRE approach.

4.1 Hyperlazy Evaluation

Hyperlazy evaluation [21] focuses on only the three basic combinators **S**, **K**, and **I**. This permits implementing combinator-graph reduction that is lazy at two levels. It provides for lazy function evaluation and also lazy updating of the graph in memory by using registers to pass small portions of the tree between combinators.

The Hyperlazy evaluation scheme attempts to deal with common sequences of graph manipulation operations not by creating more complicated combinators, but rather by implementing a finite state machine that remembers the sequence of the last few combinators that have been executed. This finite state machine enforces a discipline of maintaining outputs of a combinator

```
#1   <#2,CONST_NIL>          #28  <#29,#7>
#2   <CONS,#3>               #29  <BSTAR,#30>
#3   <#4,#7>                 #30  <I,INT_PLUS>
#7   <I,#8>                  #18  <S,#19>
#8   <#9,#16>                #19  <#20,#24>
#16   <I,#17>                #20  <#21,#22>
#17   <#18,#25>              #22  <#23,#24>
#25   <#26,#37>              #24  <LIT,1>
#37   <#38,#39>              #23  <C,INT_EQ>
#39   <#40,#36>              #21  <CPRIME,IF>
#40   <C,INT_MINUS>          #9  <S,#10>
#38   <B,#7>                 #10  <#11,#15>
#26   <S,#27>                #11  <#12,#13>
#27   <#28,#31>              #13  <#14,#15>
#31   <#32,#24>              #15  <LIT,0>
#32   <#33,#34>              #14  <C,INT_EQ>
#34   <#35,#36>              #12  <CPRIME,IF>
#36   <LIT,2>                #4  <#5,#6>
#35   <C,INT_MINUS>          #6  "fib"
#33   <CPRIME,#30>           #5  <TUPLE_n,2>
```

Fig. 14. Graph output of the lazy functional language compiler for Fib. This figure shows the graph output produced for the Fib(n) program. Each node is represented by a node identifier (which is an integer) along with the identifier or combinator for each of the left and right children.

sequence in designated registers for use by the next combinator in the state sequence. Implementing the finite state machine involves performing a case analysis at the end of each combinator to jump to the next state based on the next combinator executed from the graph.

One problem with this approach is a combinatorial explosion in the number of states (and therefore the number of code fragments to handle these states) as the length of the "memory" of the system is increased or as the number of combinators that is recognized by the system is increased. In the actual system, the C combinator was used in addition to **S**, **K**, and **I** since it resulted in significant efficiency improvements.

The reported speed for the Hyperlazy Evaluator shown in Table II is 4,000 nfib recursions per second. This speed was measured on an Acorn Archimedes system (a RISC system) running at 8 MHz. It is difficult to make a direct comparison between the ARM RISC processor and machines available to us, but the VAXstation 3200 is probably the closest match among the machines we tested. In any event, TIGRE seems to compare favorably against the Hyperlazy approach.

4.2 Norma

The Normal Order Reduction MAchine (NORMA) [28] is a special-purpose machine built specifically for reducing combinator graphs. It was at one time considered to be the fastest graph reducer in existence. Among NORMA's

Table II. Performance of TIGRE versus Hyperlazy Evaluation

Platform	Language	Combinator Set	Time (sec)	Speed (nfib/sec)
VAXstation 3200	TIGRE (assembler)	Turner	1.92	11400
ARM	Hyperlazy	SKI	—	4000

Table III. Performance of TIGRE versus NORMA

Platform	Language	Combinator Set	Time (sec)	Speed (nRAPS)
DECstation 3100	TIGRE (assembler)	Turner	1.58	472000
NORMA	NORMA	Turner	3.1	240000

features are a 370-bit-wide microinstruction, five cooperating processors, a 64-bit-wide memory bus, a 64-bit-wide hardware spine stack and extensive use of semicustom chips to optimize performance. A highly structured node representation is used that includes five tag fields in addition to two data fields. NORMA also uses some of its processors to perform garbage collection operations and heap allocation in parallel with node processing and arithmetic operations. The machine reduces the Turner set of combinators.

NORMA is rated at approximately 250,000 RAPS. Table III shows NORMA performance compared to TIGRE performance. TIGRE is significantly faster than NORMA for Fib(23) both in elapsed time and in RAPS when using the same combinator set. Furthermore, TIGRE has additional advantages in that it can be easily ported to new machines and easily extended to handle new combinators.

4.3 Conclusions about the Performance of TIGRE

Based on these comparisons, TIGRE appears to be a reasonably fast combinator-graph reducer. And because TIGRE is relatively simple, it can be easily ported to new machines. Of course, the benchmark programs used in our comparisons are unrealistic in their size and complexity (with the possible exception of the exact real arithmetic benchmark, described in the next section), and our selection is rather limited, due to the unavailability of a standard benchmark suite. However, despite the small size of the programs, it is worth noting that they generally consume very large amounts of heap memory, and so the behavior relative to heap accesses is far from trivial. At any rate, the good performance results we have obtained, even for this limited set of benchmarks, are a strong preliminary indicator that the TIGRE method is worth pursuing.

5. THE BASIS FOR THE ARCHITECTURAL STUDY

During the development and testing of TIGRE on several different machines, we found it quite difficult to predict the speed of graph reduction on any particular machine, despite knowing roughly how the various machines compared with respect to conventional C programs. We conjectured that the unexpected variations in performance were caused by hardware implementation differences among platforms, especially with regard to cache organization and management. In order to attain a better understanding of this effect, a set of cache simulations was run to measure TIGRE's use of cache memory.

Our measurements showed the TIGRE implementation on the DECstation 3100 to be the fastest (despite the need for an interpretive threading loop). Thus, we decided to use this machine's cache configuration as the starting point for our investigation. This approach gives a starting point based on a real system from which we can examine how variations in cache organization affect graph-reduction performance.

The DECstation 3100 has split instruction and data caches. During execution of combinator-graph reduction, 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 SK-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 concentrate our simulation efforts on the data cache performance.

Because 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 sort 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 and the instruction set of the CPU being used (which affect instruction cache access) from the mechanics of graph reduction (which appears as accesses to the data cache).

The DineroIII trace-driven cache simulator program was used [13]. The starting point for the simulations, based on DECstation 3100 characteristics, were the following: split instruction-cache and data-cache organization (with only the data-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. [17] and Smith [30] provide more information on cache management strategies and terminology.

5.1 Parametric Analysis

In earlier experimentation, we performed exhaustive simulations of the cache design space for a single program [20] and showed that unexpected parameter interactions and nonglobal extrema in performance are unlikely across the full search space. We therefore explore variations in single cache parameters rather than use an exhaustive search, thus avoiding excessive consumption of computational resources. The cache simulations we report here were performed by varying individual parameters, one at a time, across a wide range.

```
type Bignat = [Int]

data Sign = Pos | Neg
type Bignum = (Sign, Bignat)

type Bigrat = (Bignum, Bignum)
type Homography = (Bigrat, Bigrat)

{- A term in a continued fraction -}
data CFterm = Z Bignum | H Homography
type Bigreal = [CFterm]

{- Some interesting real numbers. -}

{- Sqrt 2 -}
sqrt2 = [ Z(bignum i) || i <- 1:[2,2..] ] :: Bigreal

{- The transcendental constant e -}
e = [ Z(i) || i <- two:(rest 0) ] :: Bigreal
    where { rest n = one:(add (mult two (bignum n)) two):one:
                     (rest (n+1)) }

{- Compute the homography obtained by absorbing the first
 - n+1 terms of the continued fraction. -}
real n []              = [H((zero,one),(one,zero))] :: Bigreal
real n [Z(z)]          = [H((z,one),(one,zero))]
real n (r as [H(?)])  = r
real 0 r               = r
real (n+1) (x0:x1:r)   = real n ((H(hom2 x0 x1)) : r)

hom2 (Z z) (Z z') =
   (qnorm (add (mult z z') one, z'), (z, one))
hom2 (Z z) (H((n,d),(n',d'))) =
   (qnorm (add (mult n z) d, n), qnorm (add (mult n' z) d', n'))
hom2 (H((n,d),(n',d'))) (Z z) =
   (qnorm (add (mult n z) n', add (mult d z) d'), (n, d))
hom2 (H((n,d),(n',d'))) (H((n1,d1),(n1',d1'))) =
   (qnorm (add (mult n n1) (mult n' d1), add (mult d n1) (mult d' d1)),
    qnorm (add (mult n n1') (mult n' d1'), add (mult d n1') (mult d' d1')))
```

Fig. 15. Excerpts from a program for exact real arithmetic.

We use the same benchmark programs described earlier, but with somewhat smaller inputs so as to keep the memory traces down to a manageable size. In addition, we add a fairly large, 600-line program that implements exact real arithmetic based on continued fractions, as described by Vuillemin [37]. Excerpts of this program, which uses relatively complex data structures, are shown in Figure 15. The inputs to the benchmark programs are as follows: Fib(19), NthPrime(90), Tak(18 12 6), 8Queens(3), and Real(10 e) (which computes an expansion of the constant *e*).

Complete program executions are simulated in all cases except for Tak, which is terminated at 10 million data references. Overall, the traces range

Table IV. Performance for Baseline Cache Organization: 64K-Byte Data Cache, 4-Byte Block Size, Direct Mapped, Write-Through, Write Allocate

	Fib	NthPrime	Tak	8Queens	Real
cache miss ratio	0.1124	0.1600	0.1658	0.1675	0.2614
bus traffic ratio	0.5274	0.5339	0.6115	0.6047	0.7288

from 1.3 million to 3.7 million data references in length, not counting Tak's 10 million data references. Spot checks of significantly longer cache traces show no substantive variations in the results. The garbage collector is not traced, and in any case is not invoked at all for any of the runs except in Tak. An assumption of sequential allocation from the heap space as performed by our stop-and-copy garbage collection scheme is still reflected in these simulations.

Table IV summarizes the results of simulating the baseline cache configuration. Two important characteristics emerge from the simulation. The cache miss ratio (fraction of cache accesses experiencing a cache miss) is a relatively high 11% to 26% 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 0.53 and 0.73. 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 show that varying the cache parameters can have a dramatic effect on both the cache miss ratio and the bus traffic ratio.

5.2 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, thereby 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.

Table V 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 increasing by a factor of 1.64 (harmonic mean) 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 many current garbage collection techniques). 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,

Table V. Cache Miss Ratios with Varying Write-Allocation Strategy (64K-Byte Data Cache, 4-Byte Block Size, Direct Mapped, Write-Through)

Allocation strategy	Fib	NthPrime	Tak	8Queens	Real
write allocate	0.1124	0.1600	0.1658	0.1675	0.2614
write no allocate	0.2036	0.2322	0.2945	0.2905	0.3929

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 before the first read. This can happen when performing a sequence of graph rewritings 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 increase overall performance by up to 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 heavily uses a heap.

5.3 Block Size: Strong Spatial Locality Means Larger Block Size

Figures 16 and 17 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 1024 bytes. The cache miss ratio for all programs decreases up to a cache size of 256 bytes. This suggests strong spatial locality.

This strong spatial locality is probably due to three different effects. The first effect is that heap nodes are allocated from the memory space in sequential memory locations, thus giving a high degree of locality for many write operations. This locality would, of course, be absent in garbage collection schemes that do not perform sequential allocation of heap space.

The read miss ratio curves in Figure 16, which show the cache miss ratio for only read memory accesses, show a secondary effect causing spatial

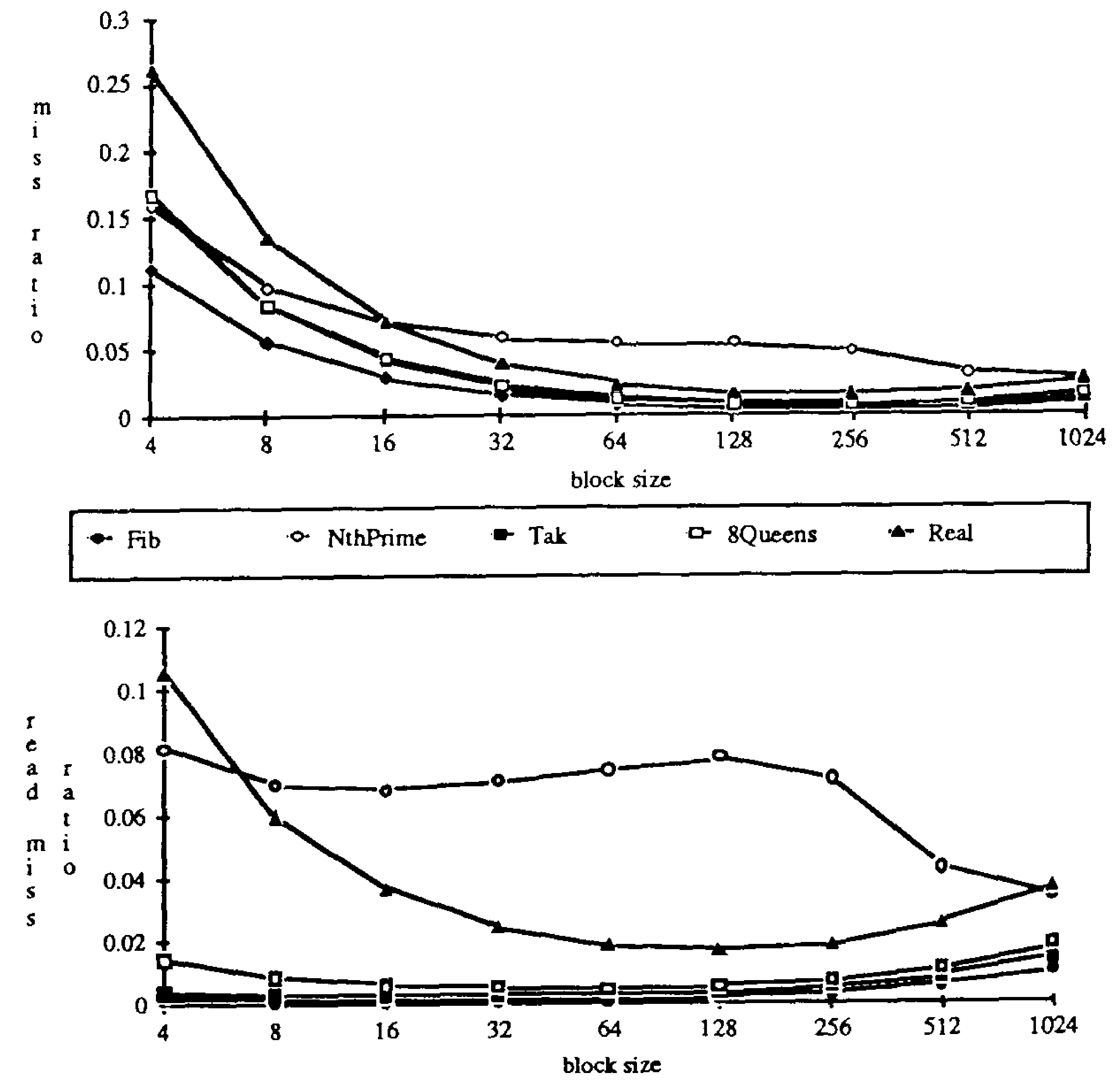


Fig. 16. Cache miss ratios with varying block size. (64K-byte data cache, direct mapped, write-through, write allocate).

locality. The read miss ratio reaches a minimum in the range of 16 to 128 bytes. This indicates that there is good spatial locality in references to previously allocated data. (The sharp drop past 512 bytes in the NthPrime test may be caused by accesses to structures spaced apart by a stride of 512 to 1024 bytes. This explanation is consistent with the high traffic ratio, discussed below). For the Real program, the large value of 128 bytes also reflects the fact that the relatively complex data structures tend to cluster together data in memory.

A third reason for spatial locality is that the spine stack is used to access elements in a first-in last-out manner.

One could, at first glance, decide to build a machine with a 16-byte to 256-byte cache block size based on the overall miss ratios alone. For conventional programs, this decision could be unwise, because the bus traffic ratio can increase dramatically with an increased block size, especially for a data cache. 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 32 and 128 bytes in size. So a machine with a 32-byte cache block size appears to be entirely reasonable for this application.

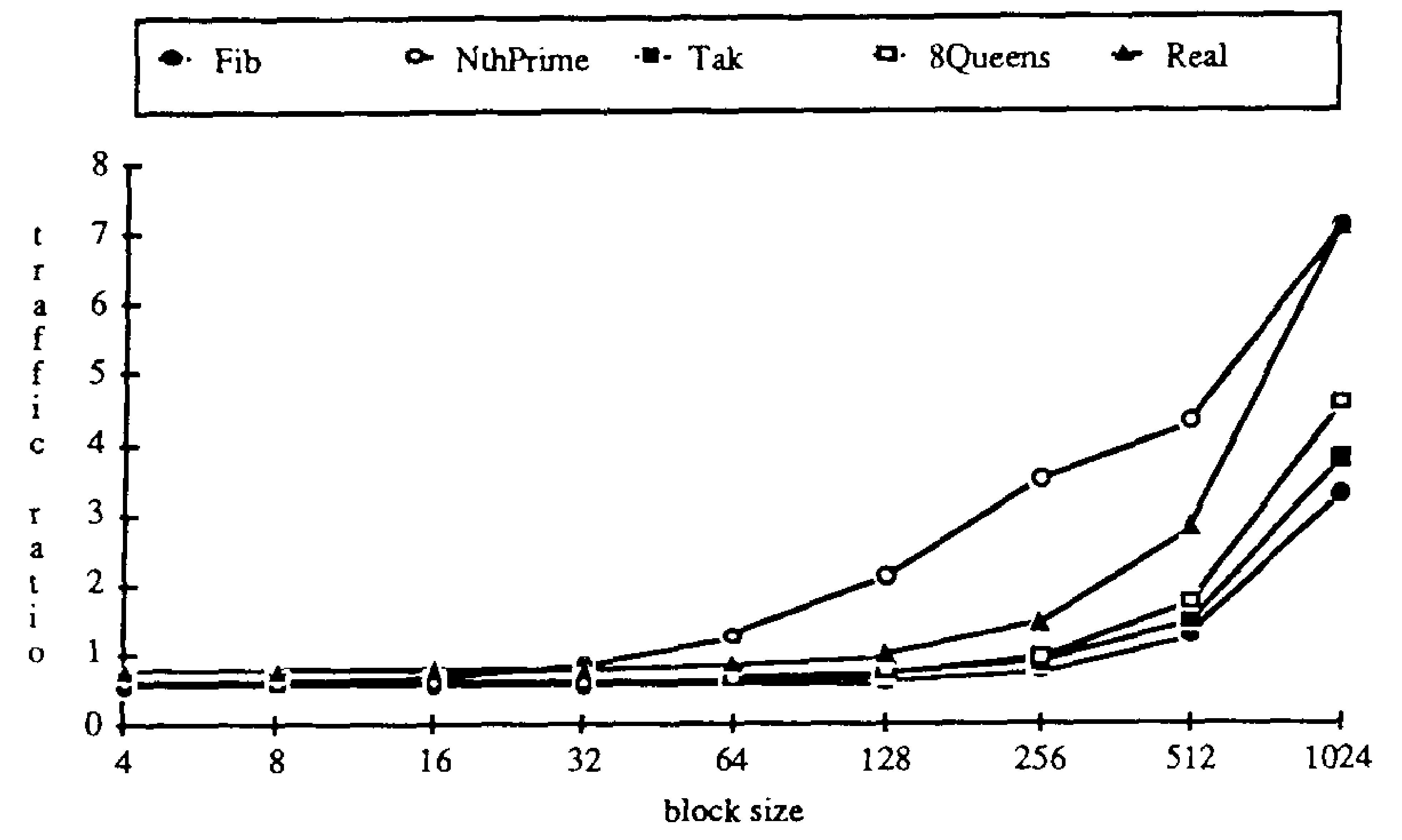


Fig. 17. Traffic ratios with varying block size. (64K-byte data cache, direct mapped, write-through, write allocate).

Even given the above data, there is a serious inefficiency inherent in using a large cache block size with a program that frequently allocates heap space. This inefficiency is caused by the fact that when a write miss occurs the cache must read in data for all words of the block not being written in order to ensure valid data for the entire block, assuming the use of a write-allocate policy. For example, with a block size of 128 bytes, a write miss caused by the first write of a heap cell will cause 124 bytes of data to be read in to fill the block. But, this data (assuming the usual case that the write miss was at the beginning of a cache block during sequential allocation) is uninitialized (since it is in the unallocated portion of the heap) and is soon written over with new heap data.

There are several ways to overcome this inefficiency. One way is to provide an instruction that allocates a cache block but does not initialize the data. Another way that is likely to be more commonly available is to make use of a subblock placement feature in the cache memory. A cache is said to have subblock placement if portions of a single cache block have associated with them individual data-valid flag bits. This permits the cache controller to manage the cache in large blocks, thus reducing resources required for tag storage, while still allocating individual words of data within each cache block. For our discussion, we shall assume a common subblock size of one 4-byte word.

With subblock placement a write operation that results in a cache miss causes the cache block to be allocated but left uninitialized, with all data-valid flag bits set to false. Then, the single word being written is placed into this block and the corresponding data-valid flag bit is set true. Subsequent writes to this block will simply set the data-valid flag bit on appropriate subblocks. Reads to this block will check the data-valid flag bit in order to determine whether a cache hit has taken place.

Subblock placement alone solves the problem of superfluous data transfer when performing initializing writes. However, it does not take advantage of locality on cache read misses, since only single subblocks are loaded on read misses. Therefore, a better strategy to use combines subblock placement with prefetching on cache read misses. A simple prefetching strategy is to fill the entire cache block from memory when a read miss is experienced, but do no memory fetching when a write miss is experienced.

Figure 18 shows the read miss ratios for individual 4-byte words and traffic ratios using subblock placement combined with a block-fill prefetch strategy for read misses. The data in this graph only extends to a block size of 128 bytes because of a cache simulator design limitation of 32 subblocks per block. The overall miss ratio for cache blocks is identical to that experienced when not using subblock placement (because the blocks are the same size).

The read miss ratios are higher with subblock placement than without it. This is caused by the fact that the usually wasted read-back of data performed without subblock placement can occasionally bring in useful data when a write miss occurs while updating a suspension data structure. The higher read miss ratio may, however, be more than offset by the decrease in the traffic ratio caused by suppression of these same read-backs. The read miss ratios are lower for a block size of 8 bytes than for the same programs with a block size of 4 bytes. (Tak is an exception, but its miss ratios are so small as to be unimportant to this argument.) This indicates that, since cache write misses often have much lower penalties (often zero) than cache read misses, the read miss ratios dictate a block size of 8 bytes and offer a performance advantage over a block size of 4 bytes if prefetching with block filling is used. The flatness of the traffic ratio curves is likely due to the elimination of superfluous reads during initialization of heap storage.

The choice of only 8-byte block sizes for use with subblock placement might seem at first to diminish somewhat the case for large block sizes shown when subblocks were not used. However, there are significant design considerations beyond the abilities of available cache simulators that can yield improved performance with larger block sizes, with or without subblock placement. One consideration is the use of burst transfer modes that can make transmission of a single block quite a bit less expensive than the transmission of two blocks of half the size.

Another consideration is that on multiprocessor systems (even if running a

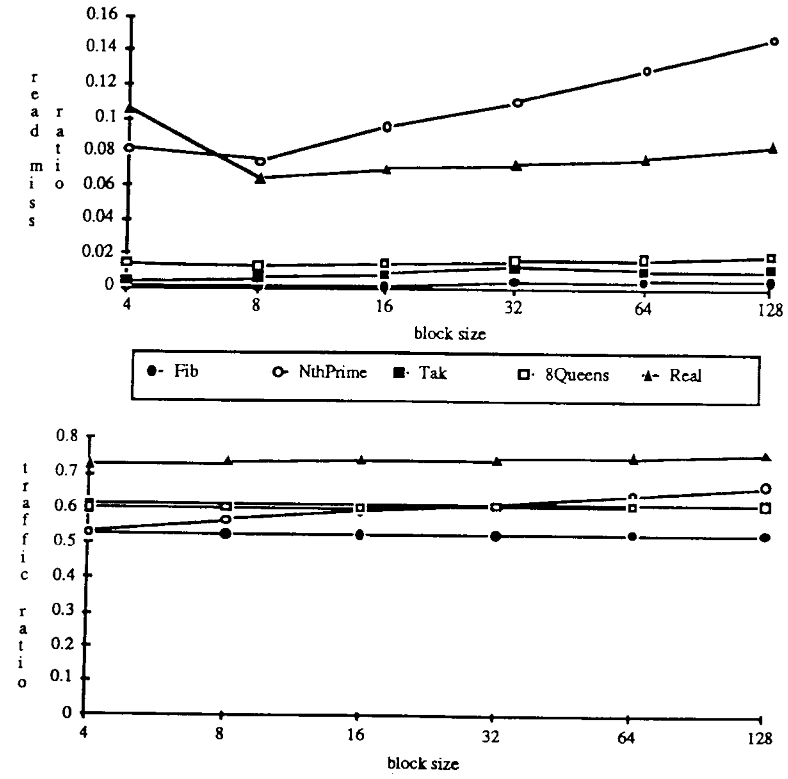


Fig. 18. Cache performance using subblock placement and a block-fill prefetch strategy with varying block size. (64K-byte data cache, direct mapped, write-through, write allocate, 4 byte subblock size.

serial version of graph reduction), cache write misses can cause overhead for invoking cache coherency protocols. Because the absolute number of write misses far exceeds the number of read misses in even the subblock placement data, any nonzero penalty for write block misses, which are still lowest in the 256-byte block size range, can quickly overwhelm advantages gained from using smaller block sizes to minimize the ratio of read misses.

Under all conditions it appears that a block size of 8 bytes brings reductions in the miss ratio. Depending on the specific trade-offs between read and write cache miss penalties and whether subblock placement is available, some machines will benefit from block sizes of up to 256 bytes.

Moderate block sizes are commonly seen in mainframe systems. (For example, the VAX 8800 has a 16-byte bus). Large block sizes are seldom seen

Table VI. Cache Traffic Ratios with Varying Write-Through Strategy (64K-Byte Data Cache, 4-Byte Block Size, Direct Mapped, Write Allocate)

Memory update	Fib	NthPrime	Tak	8Queens	Real
write through	0.5274	0.5339	0.6115	0.6047	0.7288
copy back	0.2241	0.2711	0.3300	0.3274	0.4689

in practice, because most conventional programs do not have enough spatial locality to justify very large block sizes. However, moderate-to-large block sizes will become more commonly available as they are used to conserve chip area with CPUs having on-chip cache. The significant performance increases possible with this application give strong incentive to consider a system having moderate-to-large block sizes.

5.4 Write-Through Policy: Copy-Back Reduces Bus Traffic

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.

Table VI shows the traffic ratio for a write-through versus copy-back management policy. The cache miss ratios are the same since this policy does not affect whether misses occur. However, the bus traffic generated for the write-through method is significantly higher than for copy-back (a harmonic mean factor of 1.88). This is caused by the fact that a very high percentage of memory accesses are memory writes. (Between 37% and 47% of memory references were writes on the programs simulated). This can cause severe problems with system performance by causing memory bus saturation (this actually happened on the DECstation 3100, as discussed later).

One of the promises of combinator-graph reduction is simple parallel program execution. Since many multiprocessors are built with a common memory bus, bus traffic is a prime consideration in predicting the limits to parallel processing performance. Because 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.

5.5 Cache Size

Figure 19 shows the results of varying cache size over a range of 128 bytes to 256K bytes. While different programs show different degrees of temporal locality, the curves suggest that very large cache sizes will not significantly change the miss ratio. So conventional hardware platforms are probably adequate with respect to cache size.

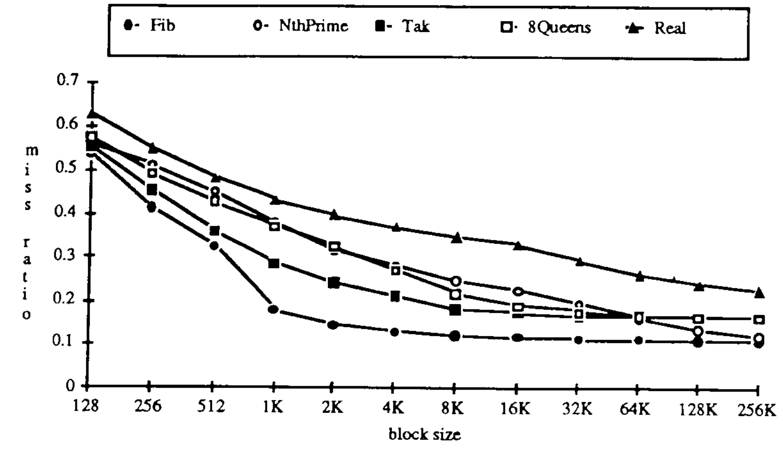


Fig. 19. Cache performance with varying data cache size. (4-byte block size, direct mapped, write-through, write allocate).

Table VII. Cache Miss Ratios with Varying Associativity (64K-Byte Data Cache, 4-Byte Block Size, Write-Through, Write Allocate)

Associativity	Fib	NthPrime	Tak	8Queens	Real
direct mapped	0.1124	0.1600	0.1658	0.1675	0.2614
2-way set	0.1113	0.1447	0.1635	0.1596	0.2368
4-way set	0.1113	0.1340	0.1634	0.1595	0.2299
8-way set	0.1113	0.1259	0.1634	0.1595	0.2273

5.6 Associativity

Table VII shows the results of varying the associativity of the cache from direct mapped (1-way associative) to 8-way associative. Multiway set associativity seems to bring modest performance improvement, but not enough to be a decisive design factor.

There is a case in which higher associativity would bring important benefits. If heap allocation were done in a nonsequential manner (due to the use of a noncompacting garbage collector), then there would be increased risk of conflicts between freshly allocated data and consequent performance degradation due to increased cache misses. In this case, higher levels of associativity would provide increased insurance against sets of heap data that accidentally map into the same block of cache.

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 trade-off of using direct mapped caches seems desirable.

5.7 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 results indicate that for the SKIfib benchmark, the MIPS R2000 processor executes, on average, 27.82 instructions per combinator reduction application. It 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, as long as the write buffer does not overflow. With an average of 4.74 writes (at 6 block 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 554,000 RAPS between garbage collections.

When actually executing the SKIfib benchmark, the DECstation 3100 performed approximately 495,000 reduction applications per second 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, thereby 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

$$\frac{0.00136 \times 13}{30.09 \text{ (clocks per combinator)}} = 0.06\%$$

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

$$495{,}000 \times 1.0106 = 500{,}000 \text{ RAPS}.$$

This rate is 11% slower than the 554,000 RAPS predicted raw reduction rate. Some of this 11% 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 can stall the processor under certain circumstances. 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 quite likely.

6. RESULTS

We have described an abstract machine for combinator-graph reduction. In comparisons against other reducers for combinator graphs based on a fixed set of combinators, we find that TIGRE shows good performance. Our architectural simulations show that TIGRE exhibits unusual execution characteristics, including the following: 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 for executing 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 bytes (preferably using subblock placement with a block-fill read miss strategy). 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.

In an earlier report [20], we described how the results of our simulation experiments could be used to guide the design of a RISC processor for SK-combinator graph reduction. Although good improvements in speed are possible, they are not great enough to warrant the effort of building a new processor. However, users and developers of reducers for SK-combinator graphs can use the results of our experimentation to aid in selecting a hardware platform that will perform well. Furthermore, the TIGRE approach is reasonably fast and simple to implement on a variety of architectures, and so provides a relatively easy way to obtain good efficiency for lazy functional programs.

7. FURTHER WORK

A great deal of further research is required to develop a full understanding of the architectural considerations in practical functional programming. In particular, we require a more extensive and realistic set of benchmarks, as well as a new set of simulation experiments based on compiled supercombinator reduction.

Although our set of benchmark programs are fairly standard with respect to the literature in functional programming language implementation, it must be realized that it is an extremely limited set. With the exception of the Real(n) program, the benchmarks are unrealistically small and simple, and they fail to use laziness in an interesting way. (The Real(n) program, on the other hand, uses relatively complex infinite data structures, and compiles into a combinator graph containing over 18,000 nodes.) Still, all of our programs allocate large amounts of heap memory, and so we suspect that their behavior relative to the data cache is not so different from larger "production" programs. Our experience with the Real(n) seems to bear this out, although many more experiments are needed.

With regard to compiled supercombinator reduction, such as that used in TIM [10], it is clear that practical functional programming systems must be based on supercombinators rather than Turner set combinators. We have put off doing such experiments for several reasons. First, the performance of compiled supercombinator reduction is highly dependent on the level of compiling technology used. Optimizing compilers necessarily make implicit assumptions about the underlying hardware. As an initial study, we thought it prudent to study graph reduction in the absence of influences from such assumptions. Second, compile-time analyses such as strictness analysis [1] have the effect of shifting a larger portion of the emitted "code" to the instruction cache. This is in contrast to the situation in our cache simulation experiments, in which the instruction space is relatively small and fixed (certainly small enough to fit entirely within the instruction cache on many machines). Thus, experiments using compiled supercombinator reduction would need to take into account the level of compiler optimizations, since they will tend to increase instruction-cache usage in order to decrease data-cache usage.

It is likely, of course, that the results of simulation experiments in the context of super-combinator reduction will be different from those presented in this paper. We believe it will be instructive to examine such differences. Or, the results may be similar but less pronounced. Some pronounced effects in TIGRE may appear as only small quirks in data gained from supercombinator-based experiments. With the TIGRE results in hand, there will be less danger of overlooking such quirks, which may still have important consequences for system efficiency.

We are presently extending TIGRE to a method called SuperTIGRE, which will accommodate a form of compiled supercombinator reduction. The main difference in SuperTIGRE from TIGRE is that the graph representation is altered to accommodate variable-sized graph nodes (called "environments"), as in TIM and the G-machine [24]. This necessitates the use of a TIM-like "take" operation, but with compile-time analyses used to avoid take's whenever possible. We hope to report on more simulation experiments with SuperTIGRE in the future. Conducting controlled experiments in this setting

Table VIII. Performance of TIGRE versus TIM

Platform	Language	Time (sec)	Speed (nfib/sec)
VAXstation 3200	TIGRE	1.92	11400
ARM	TIM	1.96	11200

is somewhat of a challenge, as the effects of compile-time optimizations must be accounted for in addition to the choice of cache-management policies.

With SuperTIGRE, it will also be possible to make comparisons with modern functional language implementations such as Lazy ML [3] and Clean [34]. There are still challenges here, as the level of compiling technology among the various systems is different. Still, some amount of direct comparison should be possible. For example, for extremely simple programs such as NFib(n), we can already make a rough comparison with the TIM approach [10, 38]. In TIGRE we can treat the definition of NFib(n) itself as the definition of a combinator and then implement (in a naive manner) this "NFib" combinator in a TIGRE reducer. This then corresponds to supercombinator reduction without strictness analysis. The result of this approach, along with the timing reported for TIM by Wray and Fairbairn [38], is given in Table VIII. This gives us some hope that TIGRE graph reduction can compete with compiled graph reduction approaches.

ACKNOWLEDGMENTS

The authors gratefully acknowledge Eric Schnarr, and especially Chris Okasaki, who implemented the Haskell-subset compiler used in some of our experiments. The authors would also like to thank the reviewers, who suggested many improvements to this paper, including the addition of several interesting experiments.

REFERENCES

1. Abramsky, S., and Hankin, C., Eds. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.
2. Augusteijn, A., and van der Hoeven, G. Combinatorgraphs as self-reducing programs. Tech. Rep. Univ. of Utrecht, 1984.
3. Augustsson, L. A compiler for lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Austin, Tex., Aug. 1984), pp. 218-227.
4. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM. 21*, 8 (Aug. 1978), 613-641.
5. Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, New York, 1981.
6. Bell, J. Threaded code. *Commun. ACM 16*, 6 (June 1973), 370-372.
7. Burley, R. An overview of the four systems in the VAX 8800 family. *Digit. Tech. J. 4* (Feb. 1987), 10-19.
8. Burn, G., Peyton Jones, S., and Robson, J. The spineless G-Machine. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming* (Snowbird, Utah, July 1988), pp. 25-37.

9. Digital Equipment Corporation. *DECstation 3100 Technical Overview (EZ-J4052-28).* Maynard, Mass., 1989.
10. Fairbairn, J., and Wray, S. TIM: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of the Conference on Functional Programming and Computer Architecture* (Portland, Oregon, 1987), G. Kahn, Ed., Springer-Verlag, pp. 34-45.
11. Fenichel, R. R., and Yochelson, J. C. A LISP garbage-collector for virtual-memory systems. *Commun. ACM. 12*, 11 (Nov. 1969), 611-612.
12. Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, Calif., 1990.
13. Hill, M. D. Experimental evaluation of on-chip microprocessor cache memories. In *Proceedings of the Eleventh International Symposium on Computer Architecture* (Ann Arbor, Mich., June 1984), pp. 158-166.
14. Hudak, P., and Wadler, P. *Report on the Programming Language Haskell, Version 1.0*, Res. Rep. YALEU/DCS/RR-777, Apr. 1990.
15. Hughes, R. J. Supercombinators: A new implementation method for applicative languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming* (Pittsburgh, Penn., Aug. 1982), pp. 1-10.
16. Johnsson, T. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction* (Montreal, Canada, June 1984), pp. 58-69.
17. Kabakibo, A., Milutinovic, V., Silbey, A., and Furht B. A survey of cache memory in modern microcomputer and minicomputer systems. In *Tutorial: Computer Architecture.* D. Gajski, V. Milutinovic, H. Siegel, and B. Furht, Eds. IEEE Computer Society Press, 1987, pp. 210-227.
18. Koopman, P. *An Architecture for Combinator Graph Reduction.* Academic Press, Boston, Mass., 1990.
19. Koopman, P., and Lee, P. A Fresh Look at Combinator Graph Reduction. In *Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation* (Portland, Oregon, July 1989), *SIGPLAN Notices. 24*, 7 110-119.
20. Koopman, P., Lee, P., and Siewiorek. Cache Performance of Combinator Graph Reduction. In *Proceedings of the 1990 International Conference on Computer Languages* (New Orleans, Mar. 1990), IEEE, pp. 39-48.
21. Norman, A. C. Faster combinator reduction using stock hardware. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming* (Snowbird, Utah, July 1988), ACM, pp. 235-243.
22. Peyton Jones, S. L. FLIC—A functional language intermediate code. *SIGPLAN Notices. 23*, 8 (Feb. 1988), 30-48.
23. Peyton Jones, S. L., Clack, C., Salkild, J., and Hardie, M. GRIP—A high-performance architecture for parallel graph reduction. In *Functional Programming Languages and Computer Architecture* G. Kahn, Ed. (Portland, Oregon, 1988), Springer-Verlag, pp. 98-112.
24. Peyton Jones, S. L., and Salkild, J. The spineless tagless G-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (London, Sept. 1989), pp. 184-201.
25. Peyton Jones, S. L. *The Implementation of Functional Programming Languages.* Prentice-Hall, London, 1987.
26. Pittsburgh Supercomputer Center. *Facilities and Services Guide.* Pittsburgh, Penn., 1989.
27. Przybylski, S., Horowitz, M., and Hennessy, J. Performance tradeoffs in cache design. In *Proceedings of the 15th Annual International Symposium on Computer Architecture* (Honolulu, Hawaii, June 1988), IEEE Computer Society Press, pp. 290-298.
28. Scheevel, M. NORMA: A graph reduction processor. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (Cambridge, Mass., Aug. 1986), pp. 212-219.
29. Siewiorek, D., and Koopman, P. *A Case Study of a Parallel, Vector Workstation: The Titan Architecture.* Academic Press, Boston. In press.
30. Smetsers, J. E. W. Compiling Clean to abstract ABC-machine code. Tech. Rep. 89-20, Univ. of Nijmegen, 1989.

31. Smith, A. J. Cache memories, *ACM Comput. Surv. 14*, 3 (Sept. 1982), 473-530.
32. Stallman, R. GNU Project C Compiler. In *UNIX Programmer's Manual.* On-line system documentation, Unix version 4.3, 1988.
33. Stoye, W. *The Implementation of Functional Languages using Custom Hardware*, Tech. Rep. No. 81, Univ. of Cambridge Computer Laboratory, 1984.
34. Turner, D. A. *SASL Reference Manual.* Univ. of St. Andrews, 1976.
35. Turner, D. A. A new implementation technique for applicative languages. *Software—Practice and Experience* 9, 1 (Jan. 1979), 31-49.
36. Turner, D. A. Another algorithm for bracket abstraction. *J. Symbolic Logic. 44*, 2 (1979), 67-270.
37. Vuillemin, J. Exact real computer arithmetic with continued fractions. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming* (Snowbird, Utah, Aug. 1988), pp. 14-27.
38. Wray, S. C., and Fairbairn, J. Non-strict languages—Programming and implementation. Draft, Oct. 16, 1988.
39. Wray, S. C. Private communication, Oct. 24, 1988.

Received July 1990; revised February 1991; accepted March 1991