Chapter 2
Background

This chapter discusses background information pertinent to the research discussed in the remainder of the book. Section 2.1 defines the problem area addressed by the research: combinator graph reduction of lazy functional programs. Section 2.2 discusses previous research on combinator reduction methods of program execution. Section 2.3 outlines the approach used for research presented in later chapters, focusing on the TIGRE abstract machine for combinator graph reduction.

2.1. PROBLEM DEFINITION

The problem area of interest is the efficient execution of lazy functional programs using combinator graph reduction. Since this method of program execution is not well known, Appendix A has been provided as a brief tutorial on the main concepts.

2.1.1. Lazy Functional Programming

Functional programs are built by pairing expressions into *applications*. Each expression may be a function or value, and the result of each pairing may also be a function or a value. Functional programming languages may be contrasted with more conventional, imperative, programming languages by the fact that functional programs preserve referential transparency (*i.e.*, expressions have no side effects and depend only on values returned from subexpressions), and hence lack an assignable program state.

Lazy functional programming languages are further distinguished from imperative languages by the fact that they employ lazy (or, more precisely, nonstrict) evaluation of parameters by default. Lazy evaluation (sometimes called *normal order evaluation*, although this term does not precisely characterize the notion of lazy evaluation) is a call-by-need para-

meter passing mechanism in which only a *thunk*[*] for an argument is passed to a function when that function is called (Henderson & Morris 1976, Friedman & Wise 1976, Vuilleman 1973). Whenever a thunk is evaluated, the result is *memoized* so as to avoid repeated evaluation of the same parameter. Lazy evaluation allows the use of powerful programming techniques such as manipulating functions as so-called first class objects (in other words, using the same manipulation techniques on functions as other data values), infinitely long lists and trees, demand-driven I/O, and implicit coroutining.

A further advantage of lazy functional programming languages is that it is believed they will provide easy-to-use parallelism. This is because the compilation process simply and automatically transforms programs into a format which makes all available parallelism explicit. This is in contrast to the case of imperative languages, where the parallelism can only be partially discovered by a sophisticated compiler, or must be made explicit by the programmer.

A problem with lazy evaluation is that it seems to be costly in practice. Examples of lazy functional programming languages include Miranda (Turner 1985), Lazy ML (Augustsson 1984, Johnsson 1984), SASL (Turner 1976), and Haskell (Hudak et al. 1988).

Recent developments in compilation technology (Hughes 1982, Hudak & Goldberg 1985, Augustsson 1984) and the design of abstract machines (Burn et al. 1988, Fairbairn & Wray 1987, Koopman & Lee 1989) have greatly improved the efficiency of lazy functional programming languages. Whereas they had been two orders of magnitude slower than conventional programming languages, now they are fast enough to rival execution speeds of LISP, Pascal, and C on many programs.

2.1.2. Closure Reduction and Graph Reduction

In *strict* programming languages, such as C or Pascal, all arguments to a function are evaluated before the function is invoked. In languages with lazy evaluation, the arguments are not computed until actually

---

[*] A thunk is a function that, when called, evaluates an argument. This defers the evaluation of the parameter until the thunk is invoked. Lazy (nonstrict) evaluation, call-by-need, call-by-name, and lexical scoping (late binding) are related terms, which all involve deferring the evaluation of a value until it is actually needed during the course of program execution. A thunk is a commonly used mechanism for implementing these parameter evaluation strategies.

needed by the function. In order to accomplish this, the program must create a thunk that computes the argument value.

Passing just a pointer to the code for a thunk as an argument is not sufficient because, in general, the value of an argument depends on the "current" values of other variables in the program. Thus, a computational *suspension*, must be built for each argument. This suspension saves copies of the values upon which the argument's computation depends, as well as a pointer to the code for the thunk. A pointer to this suspension is then sufficient to specify the value of the argument; the suspension can be restarted to compute an argument value when the result is actually needed. Of course, it is possible for the input values required by a suspension to be the results of other suspensions, so values within suspensions can be represented by either actual quantities or pointers to other suspensions.

One important evaluation strategy is *graph reduction*. Graph reduction involves converting the program to a lambda calculus expression (Barendregt 1981), and then to a graph data structure. One method for implementing the graph data structure is to translate the program to combinators (Curry & Feys 1968). A key feature of this method is that all variables are abstracted from the program. The program is represented as a computation graph, with instances of variables replaced by pointers to subgraphs which compute values. Graphs are evaluated by repeatedly applying graph transformations until the graph is irreducible. The irreducible final graph is the result of the computation. In this scheme, graph reduction, also called *combinator graph reduction*, effects the execution of the program.

The *SK-combinators* [*] (Turner 1979a, 1979b) are a small collection of combinators implemented as graph rewriting rules. A major advantage of the SK-combinator scheme is that creation and evaluation of suspensions is inherent in the operation of the graph networks, and so happens automatically. No suspensions are explicitly constructed; they are implicit in the organization of the graph itself. All the usually "tricky" details of managing the manipulation of suspensions are handled automatically.

The mechanics of SK-combinator graph compilation and reduction are extraordinarily simple, raising the possibility of very simple and efficient special-purpose hardware support (Clarke *et al.* 1980). Because of the simplicity of combinator graph reduction and the solid theoretical foundation in the lambda calculus, many early lazy function-

---

[*] See Appendix A for a tutorial on SK-combinators and the Turner Set of SK-combinators.

al programming language implementations have been based on SK-combinators.

Another means of creating suspensions is to build *closures*. A closure contains the information required to compute the value of an argument. The contents of the closure must include a pointer to code, as well as copies of the values (which may be constants or pointers to other closures) of all the data needed to compute the argument. Passing arguments to a function then involves simply passing a pointer to the closure. This method of normal order evaluation is also known as *closure reduction* (Wray & Fairbairn 1988).

As the implementation technology for lazy functional programming languages has matured, research attention has focused on closure reduction approaches. Closure reduction is more subtle in its operation than graph reduction. However, its proponents claim that it has a reduced bookkeeping load for program execution, and is inherently more efficient since it does not always need to perform actual graph manipulations (Wray & Fairbairn 1988). When combined with sophisticated compiler technology, closure reducers seem to be more efficient than previously implemented graph reducers. Closure reducers also have the advantage that they seem to map more readily than graph reduction implementations onto conventional hardware, especially RISC hardware with register windows, since they use linear data structure records for their suspensions instead of tree data structures.

The trend in research seems to be away from graph reduction and toward closure reduction. However, comparisons of the relative merits of these two methods, especially from a computer architecture point of view, are essentially nonexistent. Comparisons thus far have been based on implementations that do not necessarily represent the best mapping of graph reduction onto conventional hardware.

2.1.3. Performance Inefficiencies

A major problem with lazy functional languages is that they are notoriously slow, often as much as two orders of magnitude slower than "eager" functional languages and imperative languages. Some of this speed problem is inherent in the inefficiencies introduced by building suspensions to lazily evaluate arguments. One obvious inefficiency is that suspension creation requires dynamic memory allocation from a heap data structure. When a suspension is no longer needed, it becomes an unreferenced data structure known as garbage. Recovering the storage (a process known as *garbage collection*) can be a computationally expensive operation. Other parts of the speed problem are simply due

to inefficient implementation techniques. With this much of a performance degradation, it is difficult to write meaningful programs to exercise and evaluate the capabilities of this class of languages.

A large part of the development efforts by other researchers are focused on increasing execution speed. Software techniques such as supercombinators (Hughes 1982) and strictness analysis (Hudak & Goldberg 1985) have resulted in substantial speedups. Also, several custom hardware designs have been built and shown impressive results. These efforts have made good progress toward closing the performance gap between lazy functional programs and imperative programs. But, further speedups will be required before programming environments based on functional languages will be considered viable.

2.2. PREVIOUS RESEARCH

There have been many implementations of graph reducers on both conventional and special-purpose hardware. They vary in both their software approaches and the hardware used. Software approaches include simple SK-combinator reduction, Turner Set combinator reduction, supercombinator reduction, and closure reduction. Hardware approaches include stock uniprocessor hardware, special-purpose graph reduction hardware, and stock parallel processing hardware (both SIMD and MIMD). The work reviewed here represents many implementations for important combinations of the known hardware and software techniques.

2.2.1. Miranda

The Miranda (Turner 1985) system is a straightforward commercial implementation of a lazy functional programming language that uses combinator graph reduction. Miranda has a reputation in the research community of being somewhat slow and unsophisticated, but we surmise that there is no more than a factor of two speed increase possible without a major redesign effort*. Miranda apparently does not make use of supercombinator compilation or strictness analysis techniques. For these reasons, Miranda makes a good baseline for comparisons among graph reducers, since it forms a widely available lower bound on expected performance.

---

* The author understands that such a redesign effort is, in fact, in progress as this is being written

2.2.2. Hyperlazy Evaluation

One approach to increasing the speed of graph reduction is to concentrate on only the three basic combinators **S**, **K**, and **I** in hopes of better understanding the underlying principles of operation. Hyperlazy evaluation (Norman 1988) uses this idea to implement combinator graph reduction that is lazy at two levels. It provides for lazy function evaluation, and it provides for 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 executed. This finite state machine enforces a discipline of maintaining outputs of a combinator 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. Problems with this finite state machine approach include 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.

2.2.3. The G-Machine

The G-Machine (Augustsson 1984, Johnsson 1984, Peyton Jones 1987) is a graph reducer that uses supercombinators to increase execution speed. The idea is that in most combinator reduction schemes traversing the graph tree, performing case analysis on node *tags* (values which identify the data type of each node), and performing case analysis to decide which combinator to execute are all quite expensive. Therefore, using supercombinators speeds up the system, since supercombinators reduce the number of nodes traversed and the number of combinators executed. The G-Machine is representative of the most sophisticated graph reducers developed.

A novel idea introduced by the G-Machine is the concept of using macro instructions to synthesize sequences of machine instructions for executing combinators. Each supercombinator is built using a sequence

of G-code instructions, which are then expanded by a macro assembler into the assembly language of the target system.

The way the G-machine implements the case analysis for tag values of nodes in the graph is a good example of its sophistication. Each node has not only a pair of 32-bit data fields, but also a 32-bit tag field. This 32-bit tag field is actually the value of a base pointer to a jump table that then contains pointers to different code for each mode of the G-Machine. The case analysis performed when touching any node is a double-indirect fetch with an offset computation. The expense of deciphering a tag is significantly reduced compared to previously used strategies, but still quite expensive because of additional overhead instructions required in addition to the case analysis. A newer version of the G-machine has been developed that is tagless (Peyton Jones & Salkild 1989), but this machine is a closure reducer, more of the nature of TIM, described in the next section.

2.2.4. TIM

The Three Instruction Machine (TIM) (Fairbairn & Wray 1987, Wray & Fairbairn 1988) is an evolution beyond the G-Machine graph reducer into the realm of closure reducers. An important realization is that graph reducers must produce suspensions to accomplish lazy evaluation. Pointers to these suspensions are stored in the ancestor nodes to a combinator in the tree. As the *left spine* (the leftmost path down the program graph, which is the path taken by normal order reduction) is traversed, the stack contains pointers to the ancestor nodes, forming a list of pointers to the suspension elements. TIM goes a step further, and copies the top stack elements to a memory location so that they form a closure. This closure is simply a tuple of elements forming a vector of data in the memory heap.

The driving force behind TIM is to make closures inexpensive to create and manipulate. But, since the cost of traversing the spine is not free, and since the cost of manipulating graphs is not free, TIM also uses supercombinators to reduce the number of closures that must be created and manipulated. Costs are greatly reduced by executing code that pushes pointers directly onto a stack instead of traversing a graph that incurs overhead for each node to accomplish this same building of a pointer list on the stack. An important cost of TIM is that memory bandwidth is expended to copy the top stack elements into a closure created from heap memory. This is roughly equivalent in cost to a context switch (where a set of registers are copied out to memory when switching tasks) for each invocation of a combinator. Furthermore, the

closures are of various sizes, complicating the garbage collection process.

The closure building on top of a stack is roughly analogous to a machine using a set of register windows. This is not an accident. TIM is the result of an evolution of software techniques that have transformed the representation of the combinator graph reduction problem from one of interpreting a combinator graph to one of executing sequences of inline code using register windows to contain groups of arguments. In other words, TIM shows how the graph reduction problem can be made to fit conventional hardware and software techniques. Since TIM is optimized for the use of conventional software and hardware techniques, it is unlikely that TIM performance can be significantly improved by the use of any special-purpose hardware, beyond that available in a well-designed general-purpose Reduced Instruction Set Computer (RISC).

2.2.5. NORMA

The Normal Order Reduction MAchine (NORMA) (Scheevel 1986) at one time was widely acknowledged to be the highest performance combinator graph reducer hardware built. It is special-purpose graph reduction hardware optimized for the fastest possible operation. Among NORMA's features are a 370-bit wide microinstruction, five cooperating processors, a 64-bit wide memory bus, and extensive use of semicustom chips to optimize performance. NORMA uses a highly structured node representation 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. NORMA uses the Turner Set of combinators to accomplish graph reduction.

2.2.6. The Combinatorgraph Reducer

The concept of self-reducing combinator graphs for implementing graph reduction was first reported by Augusteijn and van der Hoeven (Augusteijn & van der Hoeven 1984, van der Hoeven 1985). Their approach was to add `jsb` instructions to the nodes of a combinator graph, then directly execute the graph as instructions instead of data.

This use of self-reducing graphs is identical to the threaded interpretive tree traversal mechanism described in this book (although they were independent discoveries). Complete details of the Combinatorgraph Reducer are not available, but it appears that there are

differences in the operation of some of the combinators and the runtime environment between that implementation and the TIGRE reducer presented here.

2.2.7. Analysis and Summary

There are now two basic strategies for the lazy functional programs: combinator graph reduction and closure reduction. Figure 2-1 shows

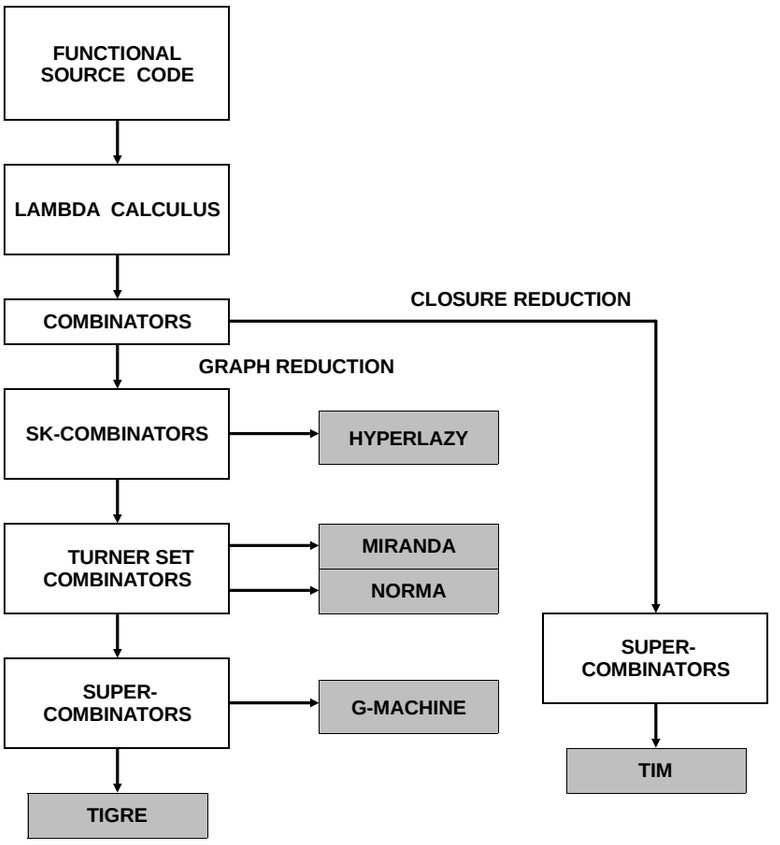

Figure 2-1. Evolution of lazy functional program implementation tech-

the paths of development from functional program source code through the lambda calculus and combinators. To improve performance beyond the G-Machine, one can either shift to closure reduction (along the lines of TIM), or try to minimize the costs involved with graph reduction. The technique proposed in the next chapter (based on an abstract machine called TIGRE) retains the combinator graph reduction approach, but uses an unconventional software technique to reduce the cost of the spine traversal, and from there significantly reduces the cost of executing combinators as well.

It is important to note that this review of previous work is by no means exhaustive. Many researchers have been working on the problem of generating more efficient means of evaluating lazy functional languages. There have been several special-purpose hardware projects as well as software projects. Furthermore, this activity is not likely to diminish soon.

2.3. APPROACH OF THIS RESEARCH

This book reports the results of a study of an abstract machine for graph reduction called the Threaded Interpretive Graph Reduction Engine (TIGRE). The goal of TIGRE is to achieve significant speedups over other existing evaluation techniques for lazy functional programming. A constraint on achieving this goal is to do so while remaining in a pure graph-reduction paradigm, in order to preserve the simplest and most obvious program structure for exploitation using parallel processors in future endeavors.

TIGRE achieves its speedups by adopting a simplified model of combinator graph reduction based on viewing the graph as an executable program structure instead of an interpretable data structure. This shift of viewpoint causes the program graph to be viewed as a self-modifying threaded interpretive program. The TIGRE technique achieves significant speedups over previous combinator graph reduction and closure reduction methods on identical standard hardware. Furthermore, TIGRE on a stock workstation platform is substantially faster than existing special-purpose graph reduction hardware.

Since TIGRE uses some unconventional software techniques, it exhibits unusual behavior on conventional architectures. In order to understand observed performance variations, TIGRE behavior was instrumented and simulated. This has led to better understanding of required hardware support for combinator graph reduction.

The next chapter describes the development of the TIGRE implementation method. Chapter 4 describes actual implementation of the TIGRE abstract machine.