A Model for

Fine-Grained Asynchronous Concurrency

Through Parallel Graph Reduction

Barton E. Schaefer
B.S.S., Cornell College, 1984

A dissertation submitted to the faculty
of the Oregon Graduate Institute
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

September, 1990

The dissertation "A Model for Fine-Grained Asynchronous Concurrency Through Parallel Graph Reduction" by Barton Evan Schaefer has been examined and approved by the following Examination Committee:

Richard B. Kieburtz
Professor, Thesis Advisor
Oregon Graduate Institute

Michael Wolfe
Associate Professor
Oregon Graduate Institute

Jonathan Walpole
Assistant Professor
Oregon Graduate Institute

William L. Bain
Adjunct Professor
Oregon Graduate Institute

Dedication

To my wife Maija, for patience in spite of uncertainty,

And to my mom, who convinced me

that graduate school might be a good idea.

Acknowledgements

I wish to acknowledge the faculty, students, and staff of the Computer Science and Engineering Department of the Oregon Graduate Institute for providing an ideal environment for learning and for research. They made my time at OGI more enjoyable and instructive than I would previously have thought possible. I would especially like to thank my advisor, Dr. Richard Kieburtz, for providing me with steady guidance while still allowing me freedom to complete this work in my own way. I would also like to thank my friend and office-mate, Boris Agapiev, for numerous discussions in which he listened to my complaints and helped me to crystallize my ideas. Finally, I would like to thank my committee members for their detailed and helpful commentary, without which this thesis would have been completely incomprehensible.

Table of Contents

Table of Illustrations

Abstract

A Model for

Fine-Grained Asynchronous Concurrency

Through Parallel Graph Reduction

Barton E. Schaefer

Oregon Graduate Institute, 1990

Supervising Professor: Richard B. Kieburtz

This thesis explores techniques for massively parallel computation on MIMD computers executing fine-grained computational tasks asynchronously. It presents a model for evaluating expressions by concurrent graph reduction. The nodes of a computation graph are represented in the memories of a network of identical computing modules. The thesis presents experimental studies of the behavior of a dynamic scheduling algorithm for distributing workload over the modules of a network. Called diffusion scheduling, it uses a measure of workload as the analog of pressure to direct tasks to modules where they are most likely to receive prompt service. A second series of experiments investigates the effectiveness of *speculative* evaluation in stimulating concurrent activity when the more commonly employed approaches of data or control parallelism fail. Parameters of network dimension, message passing characteristics, and data dependencies within programs are considered in development of a heuristic method for creating and distributing speculative work.

CHAPTER 1

Introduction and Motivations

Significant advances have been made in the exploitation of fine-grained, synchronous concurrency, as demonstrated by massively parallel SIMD systems like the Connection Machine [Hil81, Hil85]. Synchronous concurrency has also been called *data parallelism*, and refers to the simultaneous application of a series of identical operations to a large number of data items. Algorithms that repetitively perform a computation can often be reformulated to exploit this type of concurrency, which is well suited to the single instruction stream, multiple data stream (SIMD) model. However, success has been limited in the effort to exploit the fine-grained ***asynchronous*** concurrency that is found in many other types of algorithms.

Asynchronous concurrency arises when two or more different series of operations can be performed independently. The operations need not be completely independent for some parallelism to be achieved. This type of concurrency is better suited to the multiple instruction stream, multiple data stream (MIMD) style of computation. Unfortunately, the overheads of current MIMD machines, especially in communications, and the relatively small numbers of processors available in these machines, make them most appropriate for medium- to large-grained concurrency. New hardware technology such as Dally's ***message-driven processor*** [Dal86] and the MIT ***Monsoon*** processor [PaC90] promise massively parallel, low-overhead MIMD systems in the near future.

As advancing technology provides MIMD systems with increasingly large numbers of processors, new techniques are needed to extract concurrent tasks from programs and to

control the behavior of those tasks. Existing systems rely on program notation and/or compiler analysis for this purpose. However, program notations are not appropriate for expressing parallelism at the level of detail required to effectively utilize thousands of processors. Compiler technology holds more promise, but will always be limited by the inability of static analysis to account for dynamic run-time behaviors. It is therefore essential that at least some identification and control of concurrent tasks be performed without dependence on language notations or compilers. Dataflow and reduction systems have provided many insights and advances towards these goals, but are still plagued by a number of practical problems.

Another significant problem for asynchronous computation is how to mask the relatively long communication latencies of most distributed-memory MIMD machines. Technology is improving in this area as well, but methods for keeping processors busy during communications and other delays must still be considered. Dataflow architectures mask latency by feeding each processor from a pool of very small tasks. The particular task that is communicating must wait for the message cycle to complete, but the processor is kept busy working on other tasks. A similar technique, using fast, fine-grain multiplexing, could be used on general MIMD architectures. However, operations in pure dataflow are sequenced only by the availability of data. Different computations performed in the same program loop may proceed at a different rates, so that some operations may have data available simultaneously from different loop iterations. Without additional synchronization, this may cause the order of accesses to shared data structures to become confused. If structures were distributed in a MIMD environment, problems of this sort would be worse.

In search of solutions to these problems, this thesis explores a technique based on combinator graph reduction [Tur79]. The concepts of combinators and graph reduction will be

described later in more detail. Stated briefly, the mathematical properties of graph reduction permit simple detection of subgraphs that may be evaluated in parallel, to generate large numbers of asynchronous concurrent tasks. Neither program annotations nor "omniscient" compilers are required. These properties also guarantee that the order in which reductions are performed do not affect the result of the computation, which is essential for concurrent execution. In addition, combinator reduction provides granularity similar to that of dataflow, permitting fast multiplexing of tasks. The combinators themselves describe data access, so synchronization is not a problem.

This thesis presents a model and experimental implementation of the Massively Parallel Combinator Reducer (MPCR). The term *massively parallel* refers both to the number of processors the model is designed to support and to the number of concurrent tasks it is intended to generate. Massively parallel computation is attractive not because it promises nearly linear speedups in execution time, but because it allows very large problems to be solved that cannot be solved in reasonable time on less parallel machines. However, the individual processors in a massively parallel system may be of limited power. It is therefore important to keep the individual tasks simple, even if the overhead compared to the size of each task is high.

To help generate these large numbers of tasks, the MPCR employs *speculative* evaluation [Bur85]. If the only tasks executed by a parallel system are those whose results are known to be useful in the future, the system is said to exploit *conservative* parallelism. In some cases, however, a large amount of parallelism is found in tasks whose results may or may not be useful. One example is the problem space search employed in artificial intelligence programs. Several alternative solutions may be tried in parallel and only the first or best one to complete is selected. This type of parallelism is called *speculative* parallelism,

because the system is speculating that the results will be useful.

Speculative evaluation has been employed successfully in the field of distributed simulation, where it is called *optimistic* execution. The message-driven Time Warp system [JBW87] controls optimistic execution by use of a technique called *virtual time* [Jef85]. Tasks in the Time Warp system proceed without regard for synchronization until they receive a message with a time stamp earlier than their current virtual time. At that point, the task is rolled back to a time before the stamp of the message, from which time it proceeds forward once more. Rolling back a task has considerable overheads, and may propagate to other tasks, even causing termination of tasks started during the optimistic execution. The Time Warp experiments are encouraging, because they show that optimistic execution can achieve speedups in spite of high overheads.

Useful tasks are never rolled back in the MPCR speculative evaluation scheme. However, dynamic task control is still important, because some of the alternatives selected for speculative evaluation may be non-terminating. Furthermore, speculative parallelism can produce an overabundance of work even in cases where the results of a terminating computation are useful. The difficulty lies both in preventing computations whose results will never be used from interfering with useful computation, and in preventing other work that is not immediately useful from flooding the system. Finding a means to limit speculation and control non-terminating computations, without sacrificing too much concurrency, is one subject of this research.

From a practical standpoint, MIMD parallel machines with thousands of processors are just beginning to become available. It may be several years before asynchronous concurrency is available on the scale that the 64,000-processor Connection Machine [Hil85] provides for SIMD computation. If such tremendously parallel machines are to be taken advan-

tage of when they finally become available, computation systems that are able to support extremely large numbers of concurrent tasks must be developed. The MPCR is intended to model such a system, using a message-driven computational model to support large numbers of tasks. However, implementing a system to control thousands of tasks on machines with only tens or hundreds of processors requires mapping of tasks to processors in a reasonably efficient manner. Such mappings can be performed either statically before the program begins to execute, or dynamically during the execution of the program. Assignment of tasks to processors is thus analogous to static or dynamic memory allocation in compilation and execution of sequential programs. Each method has advantages and disadvantages for certain classes of computation.

The applications most effectively handled with dynamic scheduling are those where the size and number of tasks is difficult or impossible to determine in advance. These include real-time systems, symbolic computation, some kinds of matrix calculations, quad-trees, and numerous others. Many of these computations will have dependencies referring to other parts of the computation. If references are thought of as arcs of a graph, and data structures as the nodes of the graph, then these computations can be viewed as graph manipulations. Computation modeled in this way is suitable for both shared and distributed memory multiprocessors, provided that the implementation of references in a distributed system models access requests across memory boundaries.

Not surprisingly, this view of computations as graph manipulations is exactly the model for graph reduction. This suggests that dynamic scheduling is most appropriate for the MPCR. Furthermore, the number of tasks alone makes static mapping a daunting prospect. Combined with the dynamic behavior of speculative computation, static mapping becomes impossible. A dynamic algorithm called *diffusion scheduling* was selected because it

is distributed and scalable, has relatively low overhead, and can be implemented in the same message-driven style as the MPCR graph computation. In addition, the load-balancing information employed by the scheduler can also be used in heuristics for speculative task control. This will be discussed more fully in Chapter 5.

To summarize, the contributions of this thesis to the fields of parallel computation and graph reduction are:

1. Detailed development of a fully message-driven model for graph reduction. Although presented as a combinator reduction model, it is in fact extensible to programmed super-combinator reduction, including the *spineless* variation [BPR88, Pey88].
2. Extension of the message-driven model to include creation and control of speculative tasks. This includes heuristics for determining when to create additional tasks as well as a method for assigning priorities to reduce the interference of speculative tasks with conservative work.
3. Development of a task deletion strategy, and integration of that strategy with a storage management algorithm to recover resources from useless speculative tasks.
4. Experimental evaluation of priority scheduling and task deletion as means of controlling speculative evaluation.
5. Algorithm development and experimental evaluation of diffusion scheduling for dynamic assignment of tasks to processors.

The remainder of this chapter covers some of the background that inspired this work. Brief introductions to dataflow processing, graph reduction, and combinator reduction will also introduce the reader to some of the terminology used in later discussions. The last section summarizes the organization of the thesis, and briefly outlines the topics of each chapter.

1.1. Dataflow

Dataflow refers to a computation system in which operations are triggered by the availability of their inputs (arguments). The thesis research borrows only a few ideas from dataflow, so specific dataflow projects will not be discussed. Instead, this section presents an overview of the general concepts of dataflow. More detailed discussions of dataflow architectures, including some specific projects, are presented by Treleaven [TBH82] and Arvind [ArC86].

In most dataflow systems, each operation is equivalent to a single machine instruction. Logically, each instruction is allocated a computing element which waits for the arguments to arrive and then executes that instruction. No other restriction is imposed on the ordering of instruction executions. This computation organization is referred to as *data driven* [TBH82] because operations occur exactly when their associated data is present, and have no explicit temporal relationship to other operations. Dataflow systems thus have a high degree of inherent, fine-grained parallelism.

A dataflow program can be described in terms of a directed graph. The arcs of the graph describe the movement of data from *producer* to *consumer* operations. Each arc corresponds to a reference used by the producer to pass a result to the consumer. Data is transferred via *data tokens*, which may contain tags and a variety of other information in addition to the data. Execution of an operation causes one token to be removed from each of its input arcs and a new set of tokens to be released on its output arcs. The input tokens are "used up" by the operation, and are not available to any other operations.

Dataflow systems are generally implemented by either of two synchronization schemes, both based on packet communications. In the first, called *token storage* [TBH82], data tokens are stored directly into the instructions that will execute them. Programs executed

under this scheme are in a sense self-modifying, and thus cannot make use of reentrant code or recursion. For this reason, the token storage scheme is also referred to as *static* dataflow.

The second scheme, called *token matching* [TBH82], is able to support recursive and reentrant programs. Data tokens are tagged to identify the operation that will consume them and the level of recursion or iteration at which they will be consumed. A special matching mechanism collects the tokens and assembles them into sets. When the complete token set for a particular operation has been assembled, it is made available to the operation, which then executes. This scheme, also known as *dynamic* dataflow, is more versatile than static dataflow but requires larger token packets and has higher overhead.

A number of disadvantages of dataflow have been identified. The most significant for our purposes are:

1. Sequencing of access to shared data structures is difficult, forcing either task synchronization or the use of redundant copies of the structures.
2. Programs can produce non-terminating computations in less-than-obvious ways, because all inputs to any operation must be evaluated even if the result depends only on a subset of the inputs.

Tagged-token dataflow systems solve both problems by using *sequence* [ArI85] or *iteration level* [GKW85] tags to track iterations and recursion depth. These tags allow data accesses to be ordered properly, and can be used to prevent computations from "running ahead" too far. Other techniques also exist to simplify synchronization for data access. However, given the added complexity of a distributed memory environment, the more inherent synchronizations of graph reduction make it more attractive.

In spite of their drawbacks, dataflow systems have demonstrated that a pool of small tasks can be effectively pipelined to limit processor idle times induced by latency

[GKW85,Pap87]. These results were an important factor leading to the decision to use fine-grained tasks in the experimental implementation of the Massively Parallel Combinator Reducer. By supplying each processor with a pool of fine-grained reduction tasks, the MPCR masks communication latency in the same manner as do dataflow machines.

1.2. Graph Reduction

Reduction is the process of evaluating an expression by successive transformations, under a set of rewrite rules, until no further transformations can be applied. The result is said to be the *normal form* of the expression, and is the value attributed to the original expression. An expression not in normal form is referred to as a *reducible expression* or *redex* (plural *redices*).

A reduction system can be used to evaluate functional language programs if it is consistent with the mathematical semantics of applicative expressions, and if it has the Church-Rosser property, *i.e.*, the normal form of any expression is unique regardless of the reduction sequence that produced it. This property is important for parallel evaluation, because it permits subexpressions to be evaluated in any order without affecting the correctness of the result. Two well-known examples of reduction systems that have this property are Church's lambda calculus [Chu41] and Curry's combinatory calculus [CuF58].

Although order of evaluation does not affect the correctness of reduction, it may affect completeness. An important aspect of reduction systems, therefore, is the choice of the *computation rule* [TBH82] that will order expression evaluation. Under the *innermost* reduction rule, also known as *eager* or *applicative-order* reduction, all arguments of an application must be evaluated before the expression can be evaluated. The *outermost* rule, also called *lazy* or *normal-order* reduction, stipulates that arguments shall not be evaluated until they are needed to complete evaluation of the outermost application. The innermost rule

provides more opportunities for parallelism than does the outermost. However, computations that terminate when evaluated lazily may be non-terminating when eager evaluation is used.

Graph reduction refers to a reduction process in which expressions are represented as graphs. Other systems perform reduction by string rewriting [TBH82]. A graph representation has the advantage of allowing subexpressions to be shared, via multiple arcs incident upon the root of a subgraph. String reduction systems, in contrast, represent subexpressions by value, *i.e.*, by textual expressions. Graph reduction systems can thus be more efficient, because copying and redundant re-evaluation can be avoided (at the cost of some limits on parallel evaluation). Shared references are also ideal for use with normal-order reduction, allowing arbitrary objects to be manipulated without being evaluated.

In computational terms, reduction systems can be classified on the basis of their control structure. Dynamic selection of the next reduction step from the form of the expression at each stage has been termed *pure reduction*, and is contrasted with *programmed reduction*, in which control is derived from the original expression by static analysis (compilation). This derived control can be represented by an instruction stream, and is thus easily implemented on conventional von Neumann architectures [Kie85].

In programmed graph reduction, the function symbol in the graph representing an applicative expression can be any defined function, and the number of arguments accepted by the function is not limited. Other graph reduction systems restrict the function symbol to be one of a predefined set, such as the S, K, and I combinators, with fixed numbers of arguments. To reduce the application of a function, a programmed graph reducer executes the program generated when the function definition was compiled. Thus, there may be considerable work involved in a single programmed reduction step.

Reduction systems (and functional language systems in general) have been criticized for their lack of array-like shared data structures. For distributed evaluation, however, we consider this to be an advantage, because there are no synchronization problems such as those found in dataflow. It has also been pointed out (*e.g.*, by Treleaven [TBH82]) that normal-order reduction is wasteful for operators that are *strict* in all arguments. *Strictness* is the property of an operation that requires that an argument expression be in normal form before the operation can be performed; arithmetic operations, for example, have this property. On the other hand, applicative-order evaluation of non-strict arguments can lead to non-termination, as in the case of eagerly evaluating a function that will generate an infinite list. Computations which fail to terminate for this reason are referred to as *divergent*, because the reduction does not converge on a normal form.

Advances in *strictness analysis* [BHA86,HuY86,WaH87] have made it possible to extract additional parallelism without giving up the termination properties of normal-order evaluation. This is called *conservative evaluation*, because it identifies and evaluates subexpressions whose values are certain to be needed. These techniques, however, still fall short of finding all the parallel opportunities exploited by applicative-order evaluation. Furthermore, some language constructs provide for parallel evaluation of a set of alternatives, only some of which are eventually used. The use of controlled *speculative evaluation* offers a solution to both of these difficulties. Techniques for this style of evaluation are a subject of the research presented in this thesis.

1.3. Combinator Reduction

This section provides a brief summary of the concepts of the combinatory calculus and its uses in graph reduction computation. It is intended to be a basis for concepts and terminology used in later discussions, rather than a tutorial. A more complete introduction to

combinators is presented by Stenlund [Ste72].

The combinatory calculus is a calculus of *intensional functions*. This means that the functions are defined in terms of rewrite rules such that, when given an object as an argument, they produce another object as a value. The only primitive operation in such a calculus is *application* of a function to its argument, usually written $f\ x$. Application associates to the left, so $f\ x_1\ x_2$ means to apply f to x_1 and then to apply the result to x_2.

One purpose of the development of the combinatory calculus was to avoid the use of variables when expressing logical properties. All the formulas of the calculus can be defined in terms of two primitive functions, S and K:

$$S\ f\ g\ x = f\ x\ (g\ x)$$

$$K\ x\ y = x$$

S is thus a "composition function" and K is a "constant function." The calculus explicitly permits *self-application*, that is, expressions of the form $f\ f$, so S and K can be applied to each other and to themselves in arbitrary ways to describe other functions. The *identity* combinator I is frequently added to the set of primitive functions; it is most simply defined in terms of S and K by:

$$I = S\ K\ K$$

$$I\ x = S\ K\ K\ x = K\ x\ (K\ x) = x$$

Other combinators can also be defined, including structure constructors and even arithmetic operations. The set used in this research defines eighteen primitive operations, including S, K, and I. Most are defined at a more abstract level for reasons of efficiency and ease of notation, but it is important to remember that all could be defined in terms of combinations of S and K. This set will be presented in a later chapter.

The usual method for evaluating expressions in the combinatory calculus is by term rewriting. Function definitions can be treated as *reduction rules* if read from left to right. For this reason, the definitions are often written with an arrow,

$$S\ f\ g\ x \rightarrow f\ x\ (g\ x)$$

to show the "direction" of rewriting. Any expression that contains a term matching the left side of the definition of a primitive function is a *redex*. Any expression that does not contain such a term is a *normal form*.

Expressions are *reduced*, or evaluated, by repeatedly replacing terms that match the left side of a primitive definition with the right side of that definition, until the expression is in normal form. The normal form of any combinatory expression is provably unique (see Stenlund [Ste72]), and the calculus thus satisfies the requirements for use in evaluation of functional programs.

A combinator-based system for the implementation of functional languages was first proposed by Turner [Tur79]. He first described how variables could be removed by *abstraction* from a program written in a functional language to produce an equivalent combinatory expression. Turner then described a graphical data structure to represent the combinatory expression, and proposed a model of computation based on manipulations of the graph.

In Turner's data structure, every node of the graph represents an *application* in the combinatory calculus. Each node contains two cells, the left cell representing the function and the right cell representing its argument. The contents of a cell may be either a *value* or a *pointer* to another node. Reduction is performed by walking the graph in a left preorder fashion, using a stack to store pointers to the expression currently being evaluated. As long as the top of the stack points to an application, its left subtree is pushed. When a combinator reaches the top of the stack, the reduction rule for the combinator is applied, using the

pointers in the stack to access the arguments. If the stack depth is less than the number of arguments the combinator requires, then the expression graph is in normal form and no reduction takes place. An example of this type of stack-based graph reduction is shown in Figure 1.1. This model can be extended to include basic values other than combinators

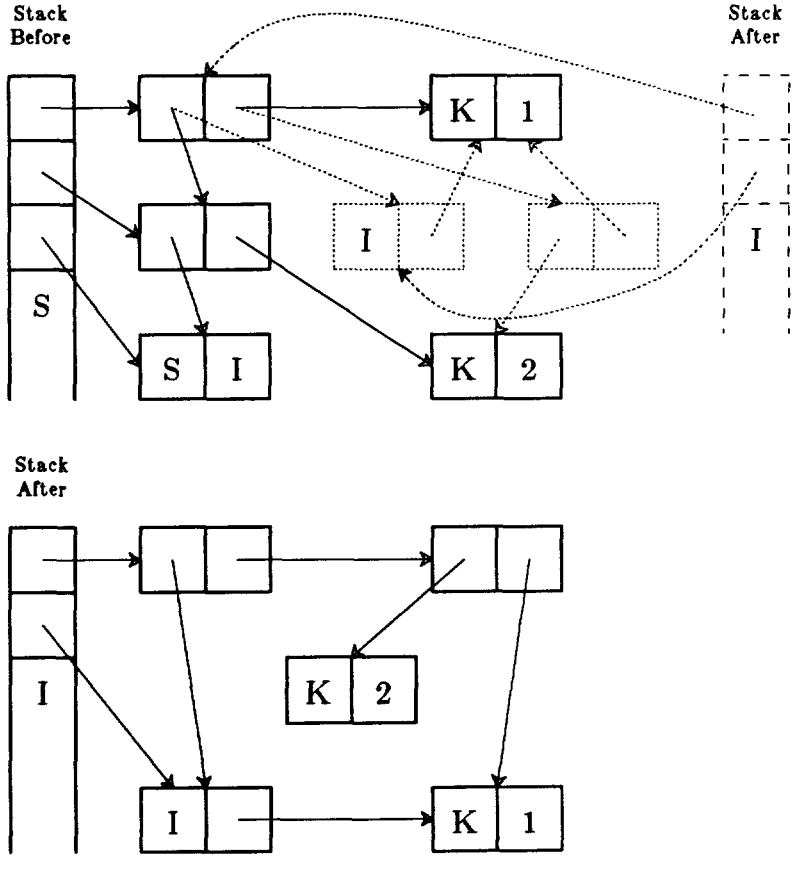

Figure 1.1 — One step of reduction of the graph of $S\ I\ (K\ 2)\ (K\ 1)$, resulting in the graph of $I\ (K\ 1)\ ((K\ 2)\ (K\ 1))$. The upper diagram shows the state before reduction, with dashed lines to indicate new nodes and updated nodes following the reduction. This diagram is adapted from Tur79. The lower diagram shows the state after reduction with nodes rearranged to make relationship to the expression more obvious. The I node after reduction is an indirection.

(e.g., integers in machine representation) and operations on those values (e.g., machine arithmetic).

Whenever a reduction rule is used, the application node to which the rule is applied is *overwritten* with the result. This is referred to in more recent literature as an *update* of the node. This has the desirable effect of preserving *sharing*, which means that all other nodes that contain pointers to the original application will contain pointers to the result. The same application never needs to be evaluated more than once.

Occasionally, it is necessary to update a node with a single pointer or value, as in the K reduction $K\ x\ y \rightarrow x$. In this case, an *indirection* is created by introducing an I combinator as the left cell of the node, and placing the actual value in the right cell. In effect, the reduction becomes $K\ x\ y \rightarrow I\ x$. The term *boxed value* has been used to differentiate an "indirection" node whose right cell is a value from a "real" indirection, whose right cell is a pointer to a subtree.

Combinator graphs provide many opportunities for concurrent evaluation. Many applications will form redices because of the simplicity of the functions. As mentioned in the more general discussion of graph reduction, the properties of the reduction system allow redices to be evaluated in any order, or simultaneously. The simplicity of combinator reduction and its great opportunities for parallelism were primary reasons for its use in the experimental system described in this thesis.

1.4. Plan of Thesis

This chapter has presented an introduction to the goals of this research and has given some background on the techniques employed to meet those goals. Chapter 2 establishes a conceptual framework for this work by describing other systems that have explored similar problems. The relationships among these systems are presented as a taxonomy, and the

position of the Massively Parallel Combinator Reducer in this taxonomy is discussed.

Chapter 3 presents an abstract model for conservative message-driven graph reduction. Several theorems are proven to demonstrate the completeness and correctness of the model. The chapter concludes with a discussion of some implementation ideas and proves that these ideas are faithful to the model. The addition of speculative computation is then discussed in Chapter 4, including proofs that the speculative model remains correct.

Techniques for mapping the abstract model to a physical machine are covered in Chapter 5. Two aspects of the mapping are discussed. One is the assignment of tasks to processors by dynamic diffusion scheduling. Chapter 6 presents preliminary research related to this technique. The second aspect discussed in Chapter 5 is estimation of processor activity to decide whether to create speculative tasks.

Experiments performed to evaluate the techniques for speculative parallelism are presented in Chapter 7. The programs run in simulation are described and results of the runs are discussed. Chapter 8 provides additional details of the simulator used to perform the experiments. Finally, conclusions and some ideas for future research in this area are given in Chapter 9.

CHAPTER 2

Taxonomy of Parallel Reduction Systems

Since the introduction of combinatory graph reduction as a technique for functional language evaluation, a variety of parallel reduction systems have been designed. To provide a context for the discussion of the system described in this thesis, it is useful to examine a taxonomy of other parallel reduction systems. Such systems generally fall into one of two broad categories, although there is some overlap. The categories are *packet-based* reduction and *pure-graph* reduction. *Pure-graph* here refers to the representation of the graph, not the derivation of control, and should be distinguished from *pure reduction*, which was introduced earlier. It is possible for a pure-graph system to use programmed reduction, or for a packet-based system to use pure reduction.

Packet-based systems are mainly derived from standard sequential reduction models. These sequential models include Turner's combinator model [Tur79] and its *super-combinator* variation [Hug82], and also more modern models such as the G-machine [Kie85] and its enhancements [Pey88]. The modern models employ programmed reduction and therefore have a larger average task size. Pure-graph systems are much less numerous and have been derived either from Turner's model or from a data-parallel combinator reduction algorithm [HiS86, HiS87].

In packet-based reduction, a reducible subgraph is collected into a data structure called a *packet* before it is evaluated. Each packet thus represents a *task*, a sequential unit of work which can be executed concurrently with other tasks. Packetization has the advantage that most of the data that will be referenced by the evaluation is immediately

available to the processor to which the task is assigned, but pays a price in overhead for formation and, depending on the execution model, disassembly of the packet. Pure-graph reducers, by contrast, manipulate the graph directly, often by use of a stack as described in Tur79. This has the advantage that there is no delay in making available the result of an evaluated subexpression, but as will be seen, it may involve other overheads. Not surprisingly, pure-graph reducers are designed with abstract models that have a globally-addressable memory space, whereas packet-based reducers usually assume either distributed, locally-addressable memories or a combination of locally and globally addressable spaces.

Variations among the abstract models of the packet-based systems are reflected in their representations of task packets, program graph, and program code. Packets may be either fixed or variable in size, depending on the extent to which nested subgraphs are considered part of a larger expression. Individual nodes of the program graph may also be variable in size. Some models store the packets as nodes of the program graph at least part of the time, but others make a strong distinction between graph nodes and packets. Most of the systems that will be discussed here chose variably-sized packets that can be stored as graph nodes. The program code referenced by a packet may also be stored with the packet, but more commonly is available to all processors either through shared code space or by distributing the code before computation begins. The complexity of the functions represented by the code for each task also varies, and is the primary determinant of task *granularity*.

Granularity refers to the size of each task in terms of its resource requirements. Tasks which represent complete programs or large parts of a program are usually referred to as *large-grained*. *Medium-grained* tasks are those representing a single function on the source language level. Super-combinators are compiled from source functions, and are thus medium-grained tasks. Finally, *fine-grained* tasks are those representing simple components

of functions, down to the level of machine instructions. Obviously, these categories are rather vague, especially the medium-grained classification. However, none of the systems that will be discussed here, whether packet-based or pure-graph, is designed to employ large-grained tasks.

An important characteristic shared by all packet-based reduction systems is their model of communication among tasks. They all employ what could be termed *Demand-Response* communication. Communication among tasks is always initiated by a demand for data (usually the value of a subexpression), and is completed by the response that carries the required data. Sometimes the demand will trigger creation of a new task, rather than requesting information from an existing task, but the effect is the same. Demand-Response communication is common because it is the natural way to express, in parallel terms, the operations of the sequential models from which packet-based systems are derived.

A very different approach is taken in pure-graph reducers derived from Hillis and Steele's data-parallel reduction algorithm. These systems are based on an architectural model similar to the Connection Machine [Hil85], having a large number of small, simple processors. The only variation is that a single instruction stream is not always assumed. There is no conventional memory in this model; processors are the only available resource. For this reason, fixed-sized graph nodes are allocated one per processor, and packets are not used. Communication in this model must therefore include not only queries and responses, but also information about how to manipulate the graph. The complete information needed to perform a graph transformation is never collected in a single processor as it is in a packet-based system. Each processor knows only what it must do with its own graph node.

Pure-graph systems not based on the data-parallel algorithm access a globally addressable memory, so inter-task communication is limited to synchronization of access to

the graph. This fits loosely into the Demand-Response communication model.

The remainder of this chapter will discuss several reduction systems in each of the pure-graph and packet-based categories. Systems that have characteristics of both categories have been classified with those they most closely resemble. Within each category, particular attention should be paid to the approach each system has taken to answering three crucial design questions:

- How are tasks to be identified?
- Which of the possible tasks are useful?
- How are resources to be managed?

Identifying a task can be as simple as recognizing a reducible expression, and in most parallel reduction systems that is the only determination. However, tasks can also be formed from collections of subexpressions, more than one of which may be reducible. Even a non-reducible subexpression can be a task, though such a task doesn't do much work.

Once a potential task has been identified, its usefulness must be determined. At one extreme, a task could be deemed useful only if its result has been *demanded* by some other task. This is the *conservative* evaluation model that has already been discussed. It guarantees that no work will be done that does not contribute to the final result of the whole computation, but it may not take full advantage of opportunities for parallelism. At the other extreme, any task that is able to run could be considered useful and given an equal chance to execute. Such uncontrolled *speculative* evaluation will often result in the consumption of resources by tasks whose results are not required. Most systems employ techniques that aim for a point somewhere between the two extremes. The choice of useful tasks has a significant effect on resource management.

Processor time is the most important resource in any computer system, with memory space a close second. Few of the issues of resource management are specific to reduction systems, but there are special factors to consider. In particular, the recovery of resources from tasks that are no longer useful can be more important in speculative reduction systems than in other computation models.

The chapter concludes by examining how the work described in this thesis fits into this taxonomy, with attention to how these questions are answered.

2.1. Pure-graph Systems

Pure-graph systems are less common than packet-based systems, but a few examples have been designed. The most recent pure-graph systems have been derived from a data-parallel algorithm described by Hillis and Steele [HiS86] and implemented by Kuszmaul [Kus86]. This is a combinator reducer designed for the Connection Machine. As mentioned above, the program graph is distributed one node per processor, and is the "multiple data" on which the instruction stream acts. The instructions cycle through reductions of each of a small set of fixed combinators, first performing all possible S reductions, then all K, all I, and so on.

Processors to which application nodes have been assigned perform the manipulations that reduce the graph. Each queries the processor referenced on the left of its application to determine what function the application represents and the position of the application in the graph spine. This information is used to transform the graph during the appropriate phase of the instruction cycle. Other processors only report the values of their graph nodes, and are inactive during most of the cycle. As graph nodes are created, processors are allocated from a pool, to which they return when the nodes are no longer needed.

This model has the potential for tremendous parallelism because all redices present in the graph at the beginning of an instruction cycle are evaluated during that cycle. This generates new redices that will be evaluated on the next cycle. Every graph node is a task, and no decision is attempted regarding usefulness. This may result in quite a few unnecessary reductions being performed. The rationale is that as long as every node has to be allocated to a processor anyway, that processor might as well be doing something. This only becomes a problem if the graph grows so large that all the available processors are consumed.

Hudak and Mohr [HuM88] note that the set of combinators chosen in such a system limits it in two ways:

1. A small set of combinators leads to a large, inefficient graph. In terms of the design questions, this means that task identification is suboptimal.
2. A large set of combinators requires a long instruction cycle. This is poor resource management, because each processor is idle during the parts of the cycle that do not apply to the node it represents.

Hudak and Mohr propose ***graphinators*** as a solution to both of these problems. Graphinators describe graph transformations at an even lower level than Turner's combinators. Programs can thus be compiled into relatively small graphs using a more extensive combinator set, and the combinators then can be executed via a small set of graphinators to keep the instruction cycle short.

In addition, Hudak and Mohr propose switching from completely eager reduction to a policy called ***prudent evaluation***. This scheme evaluates anything that is not a recursive call in the eager fashion of the combinator system, but evaluates recursion only when it is demanded. This significantly reduces the number of unnecessary reductions, but also limits

parallelism in programs that depend heavily on recursion.

Even at the graphinator level, SIMD parallelism is limited because the phases of the reduction cycle must proceed in sequential order. Truve [Tru89] proposes to avoid this problem by using MIMD evaluation in the MPG-machine. This system uses the same overall model as the SIMD systems, allocating one processor per graph node. However, the processors operate independently, rather than working from a single instruction cycle. This allows evaluation of any compiled super-combinator, rather than a restricted set. The MPG-machine system is still in the design phase, and several problems are unsolved. Most significantly, the system is designed to use controlled speculative parallelism, rather than evaluating all possible redices, but currently lacks any means to delete speculative tasks if resources begin to run out.

The most recent example of a pure-graph system not based on data parallelism is the Distributed Applicative Processing System (DAPS) [HuG84]. Functions in DAPS are simple fixed combinators, but applications of these combinators that do not form reducible expressions (*partial* applications) are noted during compilation and formed into *immutable vertices*. These nonreducible subgraphs represent functions larger than the combinators from which they are composed. The graph is represented by a combination of these immutable vertices, copied into the local store of every processor, and of the remaining *mutable* part of the graph. The mutable graph is placed in a globally addressable memory and manipulated there.

Evaluation in DAPS is demand-driven, so no unnecessary work is done. Tasks are application nodes in the mutable graph, which reference the immutable vertices as functions. There is no notion of a packet, but DAPS represents an interesting halfway point, its immutable vertices reflecting something of the way packets are stored in the graph in other

systems. This scheme also combines features of both programmed and pure control, though primarily the latter.

DAPS shares another characteristic with several packet-based systems, that is, its use of diffusion scheduling to assign tasks to processors. Diffusion scheduling will be discussed in detail in Chapters 5 and 6. One of the interesting results of the DAPS experiments was that a simple diffusion heuristic based only on the length of the task queue at each processor performs nearly as well as a more complex heuristic that takes data locality into account.

2.2. Packet-Based Reduction Systems

Most of the parallel reduction systems designed to date are packet-based. All are derived by some path from Turner's combinator reduction model, with the main difference being how far they followed the path of sequential reduction technology before branching into parallelism. Although the boundaries are not well defined, for convenience this discussion will classify the systems as either *traditional combinator style* or *G-machine style* reducers. "Traditional" here means that the system closely follows Turner's model, except that the combinators used may be derived by compilation. Some control is still determined dynamically by the form of the expression. In contrast, G-machine style systems derive control entirely by compilation, and often use additional optimizations such as avoiding unnecessary updates or condensing the left "spine" of the application tree.

2.2.1. Traditional Combinator Style Systems

The first system designed to employ packet-based reduction was ALICE [DaR81]. In this system, the program graph is represented as a pool of variably-sized packets. ALICE is unique among the packet-based systems in that it stores the entire graph as packets in this pool. Space in the pool is managed by reference-counting the packets, and processor alloca-

tion is managed by having available processors take ready packets from the pool.

A packet consists of a (pointer to a) function, which represents a super-combinator, plus an argument list. A packet is ready for execution when the function and all necessary arguments are present. It is otherwise tagged as suspended. The primary evaluation strategy is thus data-driven and eager, but some decisions are made regarding the usefulness of tasks. In cases in which it is not possible to determine the usefulness of a task, as in selecting the correct branch of a conditional, all of the alternatives are tagged as suspended even if all their arguments are present. When the outcome of the conditional is known, the task representing the conditional removes the suspended tag from the chosen alternative. This is referred to by the ALICE designers as *constrained eager* evaluation.

Another system that employs a data-driven evaluation strategy is Flagship [WWW86, WaW87b], which is a descendant of the ALICE system. The program graph in Flagship is a collection of variably-sized nodes. As in ALICE, no distinction is made between graph nodes and packets in terms of form or content, but in this case the graph is not explicitly stored in a task pool. Packets contain a pointer to code for a super-combinator function, and a list of arguments. Elements of the argument list may be tagged as strict, requiring that they be fully evaluated before the function is applied. A feature unique to Flagship is that it stores the code referenced by each packet directly in the graph, rather than in a separate code store. It is thus the only system that dynamically distributes code as well as data.

Flagship improves on the ALICE model by avoiding the creation of tasks that will be suspended awaiting arguments. A packet that has all arguments available, though not necessarily fully evaluated, is examined and any strict arguments are demanded. (The system is thus not completely data-driven.) When all strict arguments are evaluated, the

packet is activated and its super-combinator code is executed. If, in the course of executing the code, further evaluation of a nonstrict argument is required, that evaluation is demanded and the packet suspends until the result is returned. Rather than feeding processors from a pool, packets in Flagship are distributed by a dynamic load-balancing scheme similar to that devised for the Rediflow system [KeL84,KLT84].

Rediflow was the first system to employ diffusion scheduling for dynamic load balancing†, and is the only system that uses fixed-sized packets. Packets in Rediflow contain a pointer to function code and either a single argument or a pointer to a structure of arguments. They are distributed as they are created, but may not lodge permanently at the first processor to accept them. Each processor has a queue of *migrable* [sic] tasks, which may be moved to another processor, and a queue of local tasks. Migrable tasks may be moved when the load changes, but if the load at all processors exceeds a certain level, no migration occurs. The local tasks represent work that is considered inappropriate for execution elsewhere because of data locality or similar considerations.

Both data-driven and demand-driven computation are supported by Rediflow, but the distinction is much clearer than in Flagship. Subsets of the Rediflow system support dataflow or reduction processing and can be used together or independently.

The three systems discussed so far use super-combinator tasks with code compiled from source-language functions. The Alfalfa system [GoH87,Gol88] is slightly different in that it relies on compilation to identify *serial combinators* [HuG85a,HuG85b]. Serial combinators represent the largest functions within which no opportunities for parallelism exist. This is theoretically advantageous because no parallelism is lost as long as processors are

† Note that diffusion scheduling is not restricted to graph reduction. A general discussion of diffusion scheduling can be found in Chapter 5.

available to execute new serial combinator tasks. The granularity of a serial combinator may range from slightly larger than a source function down to that of a complex machine instruction, and is thus a mixture of fine and medium granularities.

Alfalfa is a direct descendant of DAPS, and continues to employ diffusion scheduling to allocate tasks to processors in a balanced manner. However, new source language notations and compiler techniques have permitted Alfalfa to take better advantage of parallelism. For example, Alfalfa programs can create vectors and select their elements at the language level rather than constructing and traversing lists. Evaluation is still conservative, but better use can be made of strict operations.

2.2.2. G-machine Style Systems

Several parallel reduction systems have been based on the G-machine sequential model or its more efficient derivatives. These systems share the medium-grain parallelism provided by programmed reduction, but differ widely in their approaches to parallelizing the model. All seek to combine the most efficient aspects of sequential evaluation with the benefits of concurrent execution.

The Shared Memory Parallel G-machine [Bur88] and the HDG-machine [KLB89] are shared and distributed memory variations of the same system. In these two systems, based on the spineless G-machine [BPR88], tasks are represented by variably-sized graph nodes which in turn represent the entire left spine of a subgraph†. Packets and graph nodes share this representation, but not every graph node is a packet. The distributed memory HDG-machine adds a special case for non-local pointers, which are accessed through an extra level of indirection. Remote processors are given a pointer to the indirection, which has a

† The term "spineless" comes from the replacement of the spine with these variably-sized nodes.

fixed address in the local store. The indirection then points to the actual node, which can be relocated. This permits memory to be managed through a combination of reference counting for non-local pointers with semi-space allocation and copying garbage collection for local storage.

Although evaluation in these systems is conservative (demand-driven), a model of computation called *evaluation transformers* is used to improve parallelism. In effect, the amount of evaluation required for each argument to a function is encoded at the time of its compilation. This allows some subexpressions whose value will eventually be required to be *sparked* for parallel evaluation. Two types of sparking are employed, one of which demands that evaluation occur immediately. The other type of spark may be ignored if resources for the evaluation are not available.

In the HDG-machine, tasks are maintained in both local and migratable pools. This is similar to the Rediflow organization. Instead of relying on diffusion scheduling, however, processors with no work to do must request tasks from other processors. Tasks are taken first from the local pool, then from the migratable pool, and finally are requested from other processors' migratable pools. The Shared Memory G-machine uses a single shared pool, with synchronization to assure that two processors do not attempt to evaluate the same subexpression.

The GRIP machine [Pey89] is also based on the spineless G-machine but has incorporated ideas from the tagless variant [Pey88]. Packets in GRIP are variably-sized graph nodes, similar to those in the HDG-machine. However, GRIP currently supports only conservative demand-driven evaluation. The program graph is initially loaded into the local store of one processor, where evaluation begins. The local store serves both as a heap and as a local task pool. Tasks are exported from the local pool only when the system load is

low. Whenever a processor decides to offload a task, the entire subgraph accessible from the packet is shifted from the local memory of the processor that created the task into a globally addressable store. GRIP thus maintains both local and global task pools, but does not explicitly classify tasks as migratable or otherwise. Processors keep themselves busy by drawing tasks first from their local pools and then, if the local pool is empty, from the global pool. When a task is taken from the global store to begin executing, the necessary parts of the graph are fetched to the local memory of the processor. GRIP also imposes the requirement that a completed task must return the entire subgraph representing its results to global memory. There, the results can be accessed by any task that needs them. These two restrictions allow garbage collection of local and global stores to be performed independently.

The reduction system of the Parallel Graph Reduction project at the Oregon Graduate Institute shares some aspects of both the HDG-machine and GRIP, although it was developed independently. It is also similar to Alfalfa and Rediflow in that it employs diffusion scheduling to balance processor loads. No other G-machine based system does so. The PGR system has a number of other unique characteristics.

Execution in the PGR system begins in the local store of a single processor, as in GRIP. However, no global store is provided in the PGR model. Instead, when a subexpression is demanded, the complete subgraph below the application is *flattened* into a contiguous space. Such a flattened graph forms a packet, which is passed to the diffusion scheduler to be assigned to a processor. The diffusion scheduling algorithm will be discussed in detail in the next chapter. Once a processor has been selected, the packet is *unflattened* into a *workspace* reserved for that evaluation task in the processor's local memory.

The workspace contains both heap and stack memory for the task. Non-local pointers are managed with extra indirections within workspaces in the same way that the HDG-machine manages non-local pointers. Within its workspace, a task proceeds as a sequential process until another subexpression is demanded. The system is thus a hybrid of packet-based and pure-graph reduction, using packets to distribute work but pure-graph methods for evaluation. This arrangement serves several purposes. First, the tasks can be compiled almost exactly as they would be for a sequential machine, and then embedded in a run-time system that handles the remote data accesses and task distribution. The system can thus be recreated easily on any network of general-purpose processors. Second, the size of any individual task does not affect the efficiency of a context switch when a task must wait for a remote data access or evaluation. Currently, the PGR system uses program notations to control which subexpressions are to be evaluated as parallel tasks, so large tasks with many sequentially-executed subexpressions are possible. Third, improvements of the sequential reduction process can be made easily without affecting the distributed run-time system.

The PGR system model will be discussed in more detail in Chapter 6.

2.3. How the MPCR Fits In

The Massively Parallel Combinator Reducer model developed in this thesis is a packet-based, fine-grained system. Although it is designed to be executed in an asynchronous MIMD fashion, its implementation was inspired by Hillis and Steele's data-parallel reduction algorithm. The MPCR thus places much greater significance on speculative evaluation than do most of the other MIMD systems that have been discussed here. Despite the packet-based evaluation strategy, speculative evaluations are dynamically selected based on the form of the graph. Some pure-graph features are therefore present.

A packet in the MPCR is a fragment of the stack described in the discussion of Turner's combinator graph reduction model. These packets can be stored directly in the program graph, as in the HDG-machine, but the MPCR model is not completely spineless. Only the combinator at the "top" of the stack and the arguments required by that function are placed in the packet. This design is intended to keep packets small, reducing message size and permitting evaluation to occur on simple processors with little local memory. However, the size of a packet is not significant to the abstract model. As long as the combinator correctly handles all arguments stored in the packet, any portion of the spine could be packetized.

A significant feature of the MPCR model is that it can be implemented in a completely message-driven manner. Rather than employing a globally addressable memory space, the MPCR represents a reference by identifying a computing module and a local address at that module. Packets and nodes in the graph are treated as *virtual processors* which respond to messages. The simulations described later in this thesis have used a mixture of message-driven and traditional computation. However, both messages and ready packets are placed in the same task queue, so that each entry in the queue represents one operation of a virtual processor.

Graph nodes that do not form ready packets, or packets representing strict functions that must await evaluation of an argument, are assigned to a processor but are not placed in the task queue. Messages must be directed to a specific graph node and hence must be placed in the queue at the processor where that node is assigned. To maintain relatively balanced loads, ready packets are distributed to task queues by diffusion scheduling. A processor that handles many messages will thus be assigned fewer packets. This will be described further in Chapter 5.

Another feature of the MPCR is that it uses priority scheduling to ensure that conservative tasks will be evaluated before speculative ones. The task queue at each processor is maintained in priority order, with conservative tasks having the highest priority. The assignment of priorities to speculative tasks is detailed in Chapter 4. After the reduction rule for the function part of a packet is applied, the *packet itself* is updated (overwritten) with the result of the reduction. If the updated packet represents a normal form, it is returned to the graph as the value of the application. If the updated packet represents a new application node, it will be repacketized and scheduled as a new task.

Multiplexing among virtual processors in the MPCR therefore consists of returning a previously executed packet to the task queue in priority order and removing the next task from that queue. Maintenance of the priority queue is the only context switch overhead, and efficient algorithms for this purpose are well known [AHU74]. In future implementations, hardware support for message-driven computation [Dal86] could reduce or even eliminate context switches, in addition to reducing other message processing overheads.

CHAPTER 3

A Message-Driven Abstract Model

This chapter presents the basic abstract machine that describes the Massively Parallel Combinator Reducer. As defined here, the model supports only conservative parallelism. This simplifies the presentation of the model, as well as reducing the complexity of some proofs of its properties. The next chapter expands the abstract machine to include speculative computation.

Computation in the MPCR abstract machine is message-driven. The connectivity of the program graph determines how messages are sent. Conceptually, messages are sent by the nodes of the graph, and travel along the arcs. When computation begins, messages can be sent only downwards, from expressions to their subexpressions. However, almost all messages carry information to allow a reply to be sent upwards, thus introducing additional arcs.

When a node receives a message, it *reacts* to the message. The reacting node may transform itself, produce new nodes, and/or produce new messages. The transition is thus described by $Node \times Message \rightarrow Node\ List \times Message\ List$. Once the reaction triggered by receipt of a message is begun, it cannot be interrupted. New messages are therefore queued at the receiving node and delivered one at a time. Computation is initiated by sending a Demand message, as described below, to the node representing the leftmost outermost reducible application of the graph.

The first section of this chapter will describe the nodes and messages that fully define the abstract machine. The second section covers proofs of theorems concerning the

correctness of the machine. Garbage collection is not explicitly considered in these sections; for the base model, it is assumed that the machine "knows" when a node has no arcs incident upon it, and re-uses the resources committed to that node. A garbage collector suitable for use with the abstract model is described in the third section, and the chapter concludes with a discussion of some other implementation considerations.

3.1. Nodes and Messages

A program graph consists of five types of nodes: Application, Marker, Combinator, Packet, and Indirection. Application and Indirection nodes correspond to those in Turner's model. The purposes of the other nodes will be explained when each type is described. Each node in a program graph is assigned to a *virtual processor* which will receive messages directed to the node. Virtual processors implement a logical sharing of the computing resources in a system, in the same way that virtual memory locations implement sharing of the storage resources. Each virtual processor could be mapped onto a single physical processor, but there will often be many virtual processors assigned to each real processor.

There are four message types: Delete, Demand, Combinator, and Packet. Delete and Demand messages are primarily for control, whereas Combinator and Packet messages transmit data. All nodes react in the same way to *Delete* messages. A node that receives a Delete message removes itself from the graph and frees its virtual processor for use by the next new node. *Combinator* and *Packet* messages have exactly the same structure as that of the corresponding nodes, and are always sent in response to a *Demand* message. However, there may be several intervening messages between the demand and the response.

A *Demand* message contains only a *redex address*. The redex address of a Demand message is a reference to which the value of the demanded node should be sent. Every node type also has a redex address field, in addition to the formats described below. The field is

called a *redex* address because by chaining together a series of references through this field, it is possible to trace the paths of successive Demand messages backward to the root of the original reducible expression of the graph. The term is also used to differentiate from *return* address, which has the connotation of an instruction location rather than a data location. The redex address of the initial Demand message is undefined. Unless otherwise specified in the descriptions below, any other Demand message carries a redex address referring to the node that sent the message. The redex address of a node is usually undefined, but may be set during the reaction to a Demand message.

The format of each of the node types is detailed in the following paragraphs. Following these descriptions, the reactions of each node type to the various message types are explained. The reaction of each type of node to each type of message (except Delete messages) is also summarized in Table 3.1. Any message reaction not described in the table is considered a run-time error and should never occur during evaluation, unless the combinator program being evaluated contains an error.

3.1.1. Node Formats

Application Node

Application nodes have two fields, both of which are references to other nodes: a *left function* and a *right argument*. As the name implies, the node represents the application of the function to the argument. The initial graph representing any combinator program consists entirely of Application nodes and Combinator nodes.

Table 3.1 — Reactions of Nodes to Messages

		Message Type		
Node Type	Field	Demand	Combinator	Packet
Application	Redex Address	Copy to notifier list	Unchanged	
	Type	Set to that of message	Unchanged	
	Other	Create new Application as task field	Copy right argument to argument list	Add right argument to argument list
		Send Demand to *left function*		
		Add redex address of message to notifier list		
Marker	Redex Address	No redex address defined		
	Type	Unchanged	Change to Combinator	Change to Packet
	Other	Add redex address of message to notifier list	Send Delete to task field	
			Send Demand to self for each notifier list entry	
Combinator, Partial Packet	Redex Address	Ignored	No reaction defined	
	Type	Unchanged		
	Other	Send self to redex address of demand		
Complete Packet	Redex Address	Set to that of message	No reaction defined	
	Type	Unchanged		
	Other	Evaluate function, re-send Demand to self		

Table 3.1, Reactions of Nodes to Messages. Node types are listed along the left side of the table, message types across the top. The body of the table lists for each type of node the change in redex address, node type, and other fields, upon receipt of each type of message.

Combinator Node

Combinator nodes have a single field which is the name of one of the built-in functions of the machine. Although it is not essential to the model, for purposes of this discussion these built-in functions are assumed to implement the rewrite rules of a simple combinator

set. It would be possible with some minor modifications to employ compiled super-combinators instead. These nodes react only to Demand messages, by sending copies of themselves as messages to the redex address of the Demand.

In Turner's reduction model, combinators are stored directly in Application nodes. However, for clarity in describing the reactions of Application nodes to the various message types, it is helpful if the fields of Application nodes always contain references. Combinator nodes are therefore introduced to serve as the leaf nodes of the tree of Applications that make up the graph. However, there is no reason that an implementation of this model could not make use of Turner's representation to avoid a special Combinator node type.

Marker Node

Marker nodes have two fields, a *task* field and a *notifier list*. The term *task* indicates a node which has been sent a Demand message, but which has not yet sent a response to the redex address of that message. The notifier list may have several subfields, each of which is a reference to a node.

A Marker node serves as a synchronization point between the tasks that need to obtain the value of a subexpression and the task that is producing that value. To the consumer tasks, a Marker indicates that evaluation is in progress so that the consumers must wait. To the producer, it represents a location where the value can be delivered, and from which it will be redistributed to all the consumers.

Packet Node

Packet nodes have two fields, the first of which is the *descriptor*, which must contain the name of a built-in function. As mentioned above, the function is assumed to implement any one of a set of combinators. The other *argument list* field of a Packet node will have

one or more subfields, which are filled in with the arguments to the function. The number of entries in the list can never be greater than the number of arguments accepted by the function. This number is called the ***arity*** of the function.

If the length of the argument list is less than the arity of the function, the Packet is said to be incomplete or *partial*. Such a Packet represents a *weak head normal form* [Pey87]. A combinator expression $f\ x_1\ x_2\ \cdots\ x_n$, where $n \geq 0$, is in weak head normal form (WHNF) if and only if:

Either f is a data object (not a combinator)

or f is a combinator and $f\ x_1\ x_2\ \cdots\ x_m$ is not a redex for any $m \leq n$.

The latter condition can be understood to state that the arity of f is at least $n+1$. Thus, $S\ g$ is a weak head normal form, because the expression has only one argument (g), and S forms a redex only when applied to three arguments.

If the arity is equal to the length of the argument list, the Packet is said to be *complete*. A complete packet represents a redex, and is able to invoke the rewrite rules for its function. When this function is invoked, the packet undergoes an *evaluation transformation*, described below.

Packet nodes may be thought of as variably-sized Applications, representing the application of a function to more than a single argument. In terms of Turner's reduction model, they encapsulate a portion of the stack. The purpose of a Packet node is to collect the essential elements of a reduction so that they may be transmitted as a unit to another processor for evaluation.

Indirection Node

Indirection nodes have a single field which is a reference to another node. Indirection nodes may be introduced during the evaluation transformation, described below.

The following sections will describe the reactions of nodes to messages. Symbols used to diagram the reactions are summarized in Figure 3.1.

3.1.2. Application Node Reactions

Application nodes recognize only Demand, Combinator, and Packet messages. Other messages received by Application nodes generate run-time errors.

Demand message

Demand messages direct an Application node to begin the process of evaluating the reduction represented by the node. The Application is transformed into a Marker, to manage the receipt of additional Demands during the evaluation. However, the original state of the graph must be preserved so that the argument stored in the right field of the

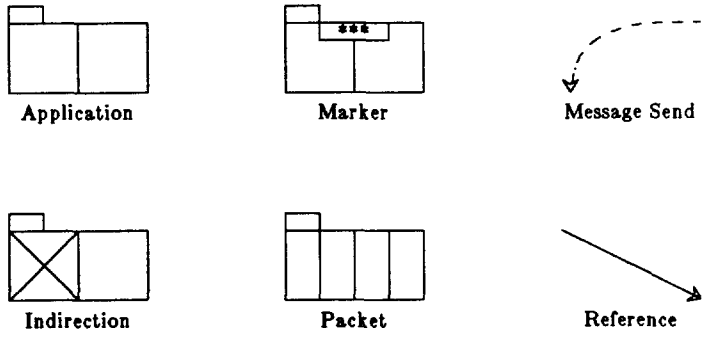

Figure 3.1 — Symbols for Node Types and Reactions.

Application can be accessed. Therefore, the Application is copied before it is transformed (see Figure 3.2).

1. The Application node makes a new node which is a copy of itself.
2. The redex address of the copy is set to a reference to the original Application node.
3. A new Demand message is sent to the left function of the Application node. The redex address of this message is a reference to the *new* Application.
4. The Application node transforms itself into a Marker node. This is referred to as the ***marking transformation***. Its task field is a reference to the new copy of the Application node, and its notifier list contains the redex address from the Demand message. If the new Marker node's redex address is a valid reference (not undefined), it is added to the notifier list, and the Marker's redex address field is erased.

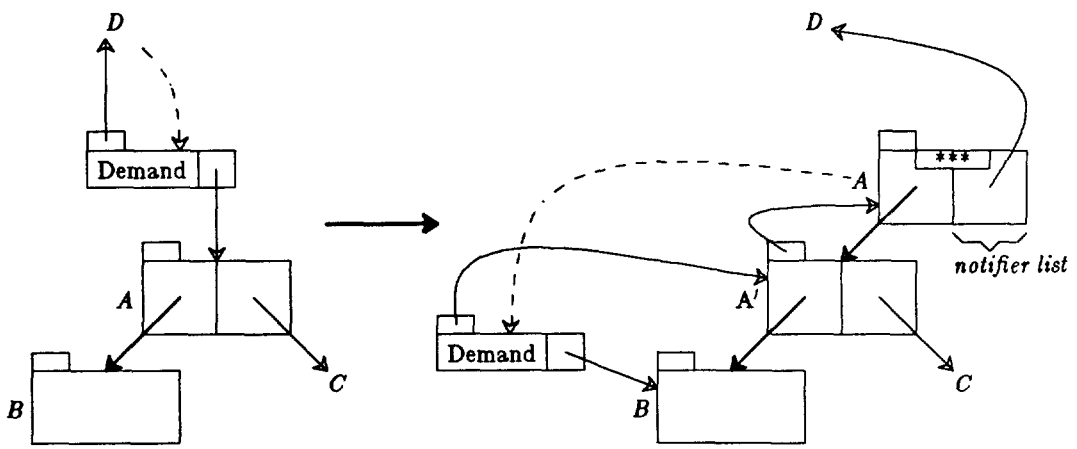

Figure 3.2 — Reaction of Application node to Demand message. This is called the ***marking transformation***. Uppercase letters (*A*, *B*, *C*, . . .) are used as node identifiers. The prime symbol (*e.g.*, *A'*) is used to denote nodes that are created by copying other nodes.

Note that following the transformation from Application to Marker in step (4), the only arc (reference) in the graph which is incident on the new Application node is the task field of the Marker. This guarantees that additional Demand messages will be received only by the Marker, not by the Application.

Combinator or Packet message

Combinator or Packet messages transmit data. Receipt of such a message means that evaluation of the subexpression referenced through the Application's left function field has been completed. A Combinator message is the function itself, and a Packet message is the function encapsulated with a subset of its arguments. The Application node responds by forming new Packet, adding its right argument field to the list of arguments of the function (see Figure 3.3).

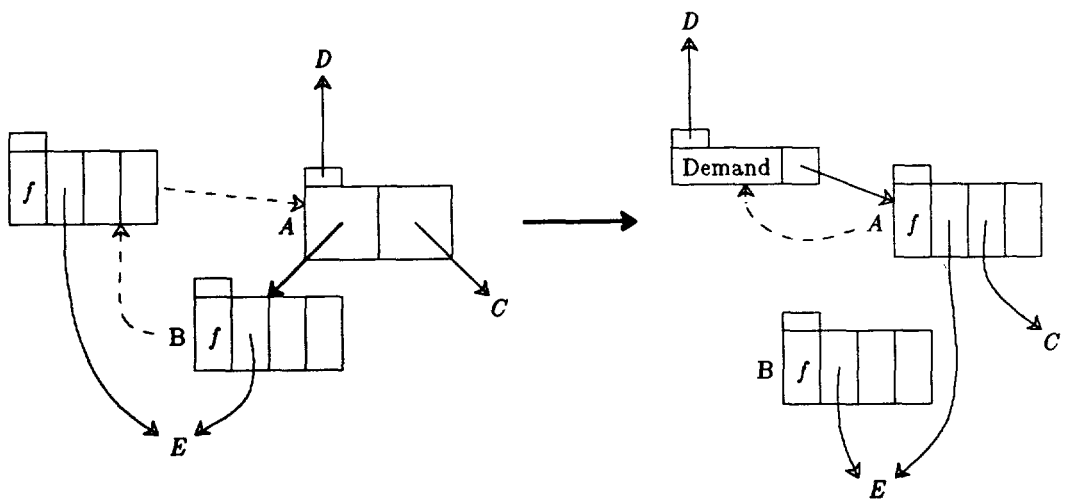

Figure 3.3 — Reaction of Application node to Packet message. The reaction to a Combinator message is the same. This is called the *packetization transformation.*

1. The Application transforms itself into a Packet node. This is called the *packetization transformation*. The redex address of the node is not changed by this transformation, but the left function reference is implicitly released.

2a. If the message is a Combinator message, the descriptor field of the new Packet is the name of the function specified by the Combinator message. The new Packet has one subfield in its argument list field, containing the right argument field of the Application.

2b. If the message is a Packet message, the descriptor and argument list of the new Packet are identical to those of the message, with the addition of one new argument list subfield. This additional subfield contains the right argument field of the Application.

3. The redex address of the newly transformed Packet node is copied to the redex address of a new Demand message. The redex address of the new Packet is then erased, and the Packet sends the Demand message to itself.

3.1.3. Marker Node Reactions

Demand message

The Marker node adds the redex address of the Demand message to its notifier list (see Figure 3.4). Recall that several expressions may share references to the same subexpression. This operation synchronizes multiple demands for the subexpression, and arranges for later distribution of its value to all expressions that require it.

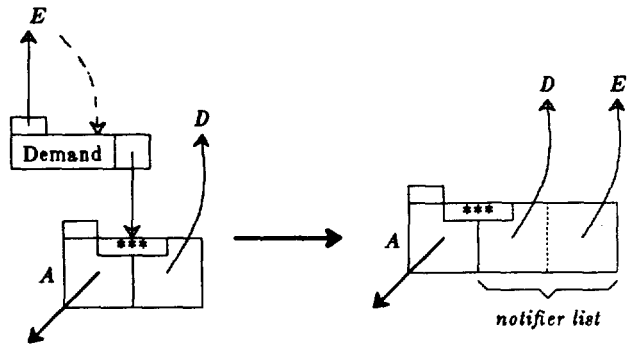

Figure 3.4 — Reaction of Marker node to Demand message.

Combinator and Packet messages

These messages represent normal forms (data), and cause the Marker to undergo an *update transformation* (see Figure 3.5). Receipt of such a message means that evaluation of the expression referenced through the Marker's task field has been completed. The corresponding operation in Turner's model is update of the redex with the value computed by applying the function at the top of the stack.

1. A Delete message is sent to the node referenced by the Marker's task field.
2. For each reference in the Marker's notifier list, a Demand message is generated whose redex address is that reference. These messages are not immediately sent.
3. The Marker node transforms itself into a node whose type is the same as the type of the message. The redex address of the transformed Marker remains unchanged. All other fields of the message are transferred from the message to the node.
4. The Demand messages generated in step (2) are sent by the transformed node, to itself.

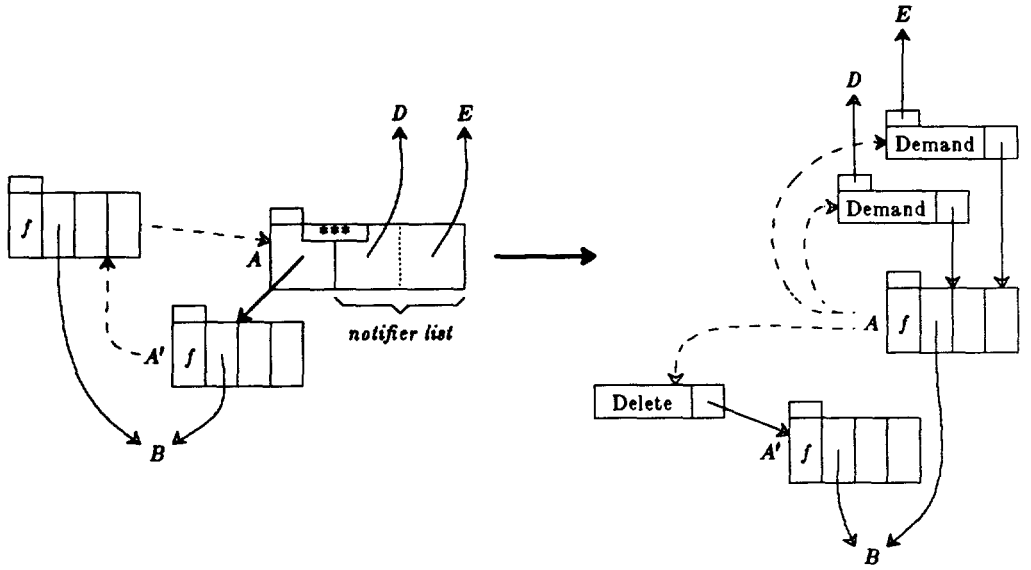

Figure 3.5 — Reaction of Marker node to Packet message. The reaction to a Combinator message is the same. This is called the *update transformation*.

3.1.4. Packet Node Reactions

Demand message

As always, receipt of a Demand message is a request for a value. The Packet node must either evaluate its function, when all arguments are present, or return as data the function and arguments that have been encapsulated so far, so that additional arguments can be added to the list.

1a. The Packet node checks the number of entries in its argument list against the *arity* of the function in its descriptor field. If the Packet is *partial*, the Packet sends a copy of itself as a message to the redex address of the Demand message. Steps (2) and (3) are skipped in this case.

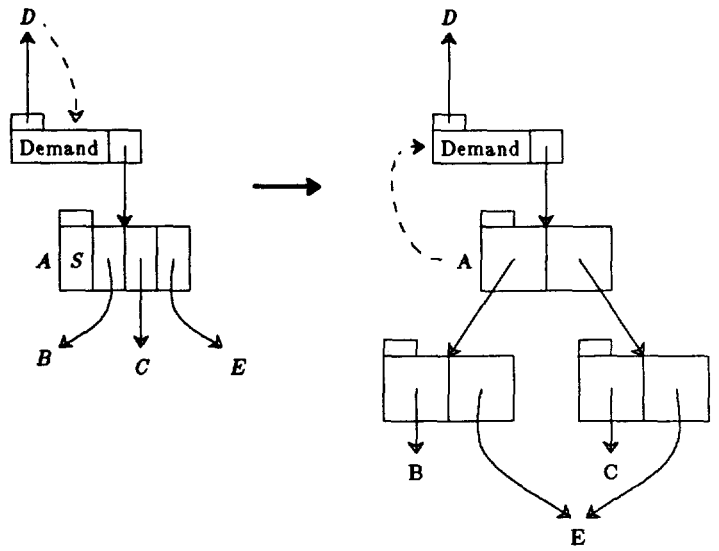

Figure 3.6 — Reaction of complete Packet node to Demand message. This is called the *evaluation transformation*. The evaluation of the *S* combinator is shown as an example; the actual reaction depends on the specific combinator in the Packet's descriptor field. During evaluation of the combinator, the redex address of the Demand message is stored in the redex address of the Packet, then used to generate a new Demand when the transformation is complete.

1b. If the Packet is *complete*, the redex address of the Demand message is transferred to the redex address of the Packet. This is done to save the redex address until evaluation has completed.

2. The node invokes the function named in its descriptor field, providing it the arguments in the subfields of its argument list. The result of this function overwrites (updates) the Packet node. This is called an *evaluation transformation* (see Figure 3.6). The function may transform the node into either a new Application node or a Combinator node, as determined by the rewrite rule for the combinator the function represents. The function may remove arcs (references) or create new arcs and nodes, also as deter-

mined by its definition.

3. The redex address field is copied into the redex address of a newly generated Demand message. The redex address of the evaluated Packet is then erased. This Demand message is sent by the transformed node, to itself.

3.1.5. Combinator Node Reactions

Combinator nodes react only to Demand messages. The reaction is that the combinator sends itself as a message to the redex address of the Demand.

3.1.6. Indirection Node Reactions

Indirection nodes react to all messages by forwarding the same message to their reference field. In the case of a Delete message, the indirection removes itself from the graph after forwarding the message.

3.2. Proofs for the Abstract Model

This section presents proofs of several theorems that assert the completeness and correctness of the abstract model.

The following theorem guarantees that nontermination occurs only as a result of the combinator program represented by the graph, not as a result of the message-driven abstract model.

Theorem 3.1

In the absence of evaluation transformations, a Marker node will receive at least one message representing either a Combinator or a partial Packet.

Proof

Recall that when an Application node A receives a Demand, it creates a new Applica-

tion A' as a copy of itself, and sets the redex address of A' to refer back to A. A then sends a Demand to its left function F, where the redex address of the Demand refers to A'. Finally, A transforms to a Marker node whose task field refers to A'. This is the only way in which a Marker node is created. Recall also that redex addresses are initialized only by this process or by transfer of the redex address of a Demand message.

Proceed by induction on the structure of the message-driven computation. Let A be an Application node that has received a Demand message. A copies itself and transforms to a Marker. Let A' be the copy of Application A, referred to by the task field of Marker A.

It is helpful to first state the following Lemma:

Lemma 3.2

Upon receiving a Combinator or Packet message, A' will either:

1. send a Combinator or a partial Packet to A, or
2. send a Demand message to a complete Packet, resulting in an evaluation transformation.

Proof of Lemma 3.2 is trivial from the definition of the abstract machine.

Proof (Theorem 3.1)

Consider cases for each of the node types for the left function F of A.

Combinator node:

Combinator nodes are already in normal form. When sent a Demand message by A, F will send a Combinator message to A'. By Lemma 3.2, this either sends the required message to A or results in an evaluation transformation.

Packet node:

If F is a partial Packet, it will send a Packet message to A'. Again, the theorem holds by Lemma 3.2. If F is a complete packet, an evaluation transformation immediately occurs.

Application node:

Upon receiving the Demand, F will transform to a Marker. By the inductive hypothesis, either F is replaced by a normal form, or an evaluation transformation occurs. If F is replaced by a normal form, the theorem holds by Lemma 3.2.

□

Theorem 3.1 asserts the completeness of the model under the assumption that the evaluation transformations correctly implement the rewrite rules of the combinators. The following theorems assert the correctness of the model by showing that it fulfills these requirements:

1. Marker nodes are updated only by the normal forms of the subgraph they represent.
2. Marker nodes are always updated if the program graph represents a combinator expression having a normal form.

Theorem 3.3

A Marker is updated ***only*** by a message representing the node referenced by that Marker's task field.

Proof

Let A be a Marker node. A is therefore a transformed Application. Let F_A be the node referenced by the left function of that Application, before the marking transformation. Let A' be the node referenced by the task field of A.

A can be updated only by a Combinator or Packet message. Let N be the node which sends the Combinator or Packet message to A. By the definition of the abstract model, N must be of the same type as the message, and must have received a Demand message whose redex address refers to A. Consider the cases in which such a Demand could be sent.

1. A sent the Demand to N. By definition, A sends Demands only to A' and to F_A. If $N{=}F_A$, the Combinator or Packet message will be sent to A', not A. If $N{=}A'$, the theorem holds trivially.

2. Another node B sent a Demand with a redex address that refers to A. There are four cases in which the redex address of a Demand does not refer to the node that sent the message.

   a. B is undergoing the marking transformation. In this case, the Demand is sent to F_B, the left function of B. The redex address of the Demand is defined to refer to B', the task field of B following the transformation. B' must therefore be an Application node. A is a Marker, so $B' \neq A$, and therefore $F_B \neq N$.

   b. B sent the demand to itself, transferring its own redex address to the Demand. By definition of the initialization of redex addresses, one of the following must hold:

      1. $N{=}B{=}A'$, so the theorem holds trivially.

      2. Node A originally sent the demand to B, so the theorem holds by induction.

   c. B is undergoing the update transformation. In this case, B sent the demand to itself, transferring a redex address from its notifier list to the

Demand. By definition of the marking transformation and Theorem 3.1, if B is referenced by the task field of any Marker M, then one of the entries in the notifier list of B must refer to M. Furthermore, by definition such a B can receive Demand messages only from M or from itself. There must therefore be ***exactly*** one entry M in its notifier list. If $M{=}A$, then $B{=}A'$ and Theorem 3.3 holds.

If B is not referred to by the task field of any M, then B must be the left function of one or more other Application nodes. This is true by definition of the cases in which Demand messages are sent. If B is a left function, then the entries in its notifier list must refer to Application nodes, not Markers. Therefore $B \neq N$, so A is not updated.

□

Theorem 3.4

Under the assumption that the evaluation transformation correctly implements the rewrite rules for the combinator set, the execution of the abstract model will reduce a program graph to normal form if the combinator expression represented by the graph has a normal form.

Proof

By Theorem 3.1 and Theorem 3.3, Marker nodes are correctly updated if no evaluation transformations occur. If an evaluation transformation does occur, it overwrites a node with either an Application or a normal form, which then Demands itself. Proceed by induction on the two cases:

1. If the transformation results in a normal form, Theorem 3.1 and the definition of the evaluation transformation guarantees that the Marker created by the initial

Demand will be updated.

2. If the transformation results in an Application, the Demand will transform it to a Marker. By the inductive hypothesis, this Marker must also be updated by a normal form if the combinator expression it represents has one.

□

3.3. Garbage Collection

The Delete message handles explicit deletion of graph nodes during update transformations. However, arcs can also be removed during the evaluation and packetization transformations. For example, recall the rewrite rule for the K combinator:

$$K\ x\ y \rightarrow x$$

An arc (reference) to subexpression y is removed in this evaluation. In the packetization transformation, the arc to the left function is implicitly removed. Garbage collection must be done to recover subgraphs when the last arc incident upon that graph is removed.

Collection in a distributed-memory MIMD environment is a difficult problem. Although the specific garbage collection algorithm used is not significant to the abstract model, it is desirable to select a strategy that can be implemented in a message-driven manner. Furthermore, for purposes of simulating the machine, a straightforward technique that could be quickly implemented was sought. A reference counting scheme devised by Watson†, which has been used in other research at OGI, was the most appropriate choice. The count associated with each pointer in this scheme is referred to as the *reference rights* held by the reference.

---

† Probable reference is WaW87a. The technique used in the MPCR is based on an informal discussion which preceded publication of that paper, and may not be identical.

When a graph node is allocated, a predefined count of rights is assigned to the pointer. This value is also stored in the new node as the node's reference count. If at any time such a pointer is to be duplicated, the original rights are reduced by some amount, and that amount is assigned to the new reference as its initial rights value. This is called *sharing* the rights. For purposes of this discussion, it will be assumed that rights are shared by dividing the amount evenly between the new reference and the original. In the abstract model, whenever a node makes a copy of itself, rights to all references that it holds are *shared* between the original and the copy.

The initial reference count of a node—and hence the total rights held by all pointers to it—never increases. Instead, if a pointer that holds the minimum allowable number of

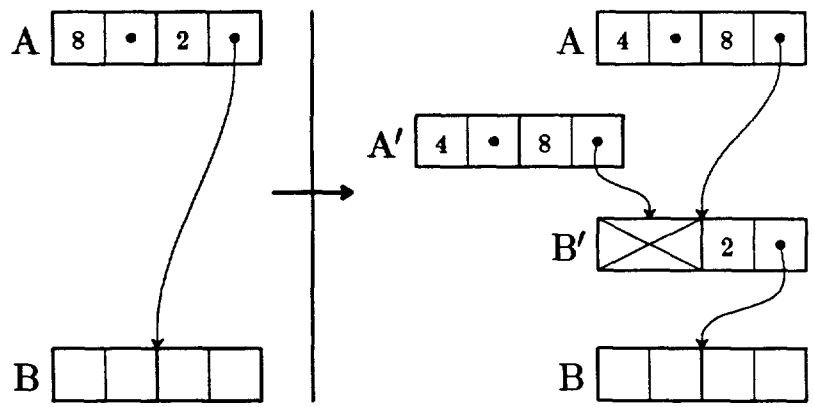

Figure 3.7 — Sharing of Reference Rights. Assume the original count of rights is 16, and the minimum is 2. Application A wishes to make a copy of itself. The right argument of A contains a reference with the minimum number of rights, too few to share. Therefore A creates an indirection with the original 2 rights, and divides the 16 rights to the Indirection when creating the copy. The other reference held by A (arrow not shown) has enough rights to be divided without creating an indirection.

rights must be shared, a new Indirection node is allocated, and the original pointer is copied into the Indirection. The rights to the Indirection are then divided between the original task and the new reference it wishes to create. This avoids race conditions among increments and decrements of reference counts, which is important in a distributed concurrent system.

Whenever a pointer is deleted, the reference rights it holds are deducted from the node's reference count. When a node's count reaches zero, it is deallocated. This is added to the MPCR abstract model by extending the *Delete* message to include a count of the rights released. Receipt of a Delete message thus need not result in the immediate removal of the node from the graph. Several Deletes may be required.

Other message types also carry references. Rights are *transferred* from a message to a node when the entire message is copied or when a reference carried in a message is copied. For example, rights are transferred in the *update* transformation of a Marker. Reference rights sent in a message are therefore given up by the sending node. Creating a message must involve either a transfer or a division (sharing) of rights. Furthermore, every message must carry a reference and rights to its target node, so that the node cannot be collected before the message is delivered.

Rights carried by any message are *released* when that message is delivered, except for Demand messages received by Marker nodes. Rights carried by these messages are released only when the response is sent, following an update transformation. This means that the notifier list of a Marker must actually store the entire Demand message, and these messages therefore need not be recreated during the update transformation. If a Demand message is not entered in the notifier list because its redex address already appears, rights are immediately released because no response is needed.

Forwarding of messages by Indirection nodes is a special case, because the rights held by an Indirection can be divided only a finite number of times. Delete messages are not a problem, because they are not forwarded until the last rights to the Indirection itself are released. In that event, *all* of the Indirection's rights are sent in a Delete message, so no division is necessary. Combinator and Packet messages will never be sent through Indirections, because they are always sent directly to the demanding node through the redex address of the Demand message. Forwarding of Demand messages still poses a problem, but this can be remedied by a *remote reference* scheme. This is discussed in the next section.

Other specific cases where rights are *shared* are:

- Initialization of the redex address of an Application node, in step (2) of the marking transformation.
- Creation of the Demand message in step (3) of the marking transformation.
- Initialization of the task field of the Marker node in step (4) of marking transformation.
- Creation of the Demand message in step (1) of a Marker node's reaction to a Demand message.

Specific cases where rights are *transferred* are:

- Inclusion of a redex address in a notifier list, in step (4) of the marking transformation, and in step (2) of the reaction of a Marker to a Demand.
- Initialization of a subfield of an argument list, in the packetization transformation.
- Creation of a Delete message sent to the task field reference during the update transformation.

- Creation of a Demand message, as the final step of the evaluation transformation.

Like any reference counting collector, the reference rights algorithm is sufficient provided that no cyclic structures are introduced. The rewrite rules implemented by the evaluation transformation are assumed to comply with this restriction. However, the references supplied as redex addresses are by definition cyclic. They point to a node which must have a reference to the node it has demanded. Thus a true cycle is introduced in the marking transformation, from the task field of the Marker to the copy of the Application and back through the redex address of the copy. A similar cycle is created whenever a Demand message is stored in the notifier list of a Marker. These cycles are not a problem because they are always broken by the update transformation.

3.4. Implementation Considerations

3.5. Remote References

An important consideration when using recursion to descend the left spine is the need to differentiate *local* references from *remote* references. A *local* reference is one that can be followed within memory that is directly addressable by the local processor. A *remote* reference is one that can be accessed only indirectly, i.e. one that is local to some other processor. Remote references force the machine to revert in part to the message-driven model.

Remote references can be treated as Application nodes whose left function knows how to retrieve the node referenced by its right argument. In the pure message-driven model, if such a node received a Demand message, the series of messages triggered by reaction to the Demand would transform the Application into a Marker and evaluate the function to retrieve the reference. In the recursive variant, the processor can be made to recognize the

reference as remote, create the Marker, and send a request for the referenced node (see Figure 3.8). It then continues as if a complete Packet had been formed, leaving a Demand message in the notifier list of the new Marker. This is discussed in detail in the chapter that describes the simulator.

Another application of this technique is in dealing with Indirection nodes whose rights have been reduced to the minimum allowable value. If such an Indirection receives a Demand message, it can be treated as a remote reference as follows:

1. The Indirection is transformed to a Marker, with an empty task field, and with the Demand that it received stored in its notifier list.
2. The new Marker sends a Demand to the reference it held as an Indirection. All remaining rights of the reference are carried by this message.

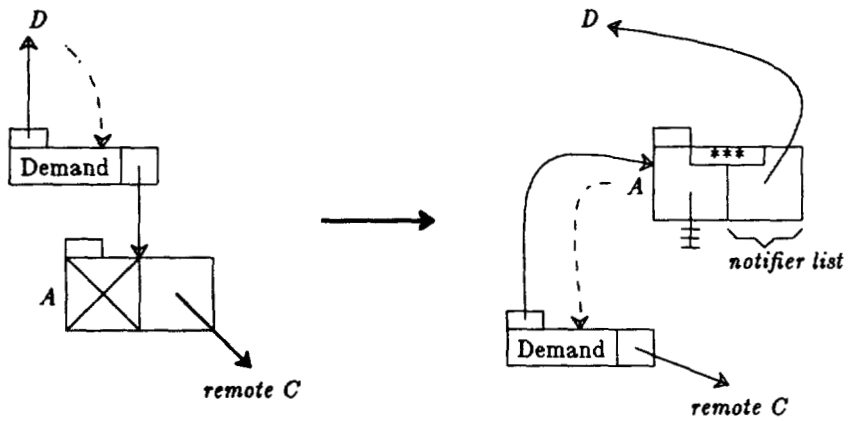

Figure 3.8 — Demanding a Remote Reference. The indirection representing the remote reference is transformed into a Marker. This is necessary only when the indirection has run out of reference rights to share or when the remote reference is found in the argument list of a strict function (see below).

When the response to the "forwarded" Demand is returned, it updates the Marker. The original Demand is then re-sent.

3.6. Pre-Formed Packets

One additional simplification that does not directly affect the model is the formation of all possible complete or partial Packets at compile time. This reduces the number of Demand messages that must be sent early in the computation. In a system that is simulating many virtual processors on each real processor, this pre-Packetization also makes it easier to distribute the graph and encourages locality of reference. This simplification has been adopted for the MPCR simulator, described in Chapter 8.

3.7. Constructor Nodes and Basic Values

Most reduction systems support several primitive data types, such as integers, and constructed data types, such as lists. In the combinatory calculus, such data types are represented by application expressions that cannot be reduced, i.e. by the application of a combinator to too few arguments. This corresponds precisely to partial Packet nodes in the MPCR abstract model. It is therefore possible to implement data structures directly as *Constructor* nodes. A constructor node is a WHNF.

If Constructor nodes are included, they must either be introduced before computation begins or be introduced during evaluation transformations. Constructors are treated as partial Packets for purposes of the message-driven computation. However, a Constructor can never be the left function of an Application, in a correct program. Instead, other functions must be provided that implement the interface to each type of Constructor. These functions employ a set of *Selector* messages that can be sent to the Constructor.

For example, the implementation of a *List* node might include the functions (combinators) *head* and *tail*. When the evaluation transformation invokes these functions, a corresponding Selector message is sent to the List node. The List node reacts by returning the appropriate part of its structure. Of course, it is also possible to allow the evaluation transformations for head and tail to access the List node directly, if the nodes reside on the same physical processor.

A complication of the inclusion of functions to manipulate Constructor nodes is that those functions must be strict. That is, the argument of such a function must be evaluated to Constructor form before the function can be applied. Selector messages must therefore include an implicit Demand, so that, if they are received by an Application or Marker node rather than by a Constructor, appropriate action can be taken. It is thus possible for the evaluation transformation to become blocked until the response to a Selector message arrives.

When a Selector message is received by an Application or Marker, it is treated as a Demand and placed in the notifier list. The type of the Selector is included in the notifier entry, so that when the message is re-sent at the end of the update transformation, the correct response is made. An evaluating Packet that has sent a Selector message becomes suspended until it receives a response. It will receive no other messages during this time, because it is "protected" from Demands by its Marker.

3.8. Shortening Marker Chains

It is clear that the Application node to Marker node transformation that follows receipt of a Demand message could result in chains of Marker nodes. As each Marker in the chain is updated by a Combinator or Packet message, it re-sends at least one Demand to itself. This Demand then results in the update in the next Marker in the chain, and so on.

The topmost Marker of any such chain represents the demanded subexpression, as seen by any tasks evaluating the surrounding expression. The lowest Marker in the chain represents the currently-evaluating redex. At any time, then, these two Markers suffice to represent the chain from the viewpoint of the rest of the graph, and the intermediate Markers need not be created. It would be sufficient to introduce Markers only for the topmost Application, and for any Application that will be transformed into a complete Packet.

To permit two Markers to represent the chain, a few modifications are made to the abstract model. First, the reaction of Application and Marker nodes to Demand messages is altered, and a *Query* message is introduced. When an Application node receives a Demand message, it sends a Query message to its left function, rather than another Demand. It then transforms to a Marker as usual. Similarly, when in the original model a Marker node would send a Demand to the node referenced in its task field, in this altered model it instead sends a Query. The redex address of a Query is always a reference to the node that sends it.

A Query message is treated identically to a Demand message by all nodes except Application nodes. An Application node reacts by copying the redex address of the Query message into its own redex address. A run-time error occurs if the Application's redex is already defined and is not the same as the redex address of the Query. (For garbage collection purposes, if the redex addresses are duplicates, their reference rights are combined in the redex address of the Application.) The Application then sends a Query message to its left function, and transforms itself into a new node type called a *Marked Application* node.

Marked Application nodes are identical to Application nodes in all respects but one. Upon receiving a Packet message which requires only one more argument to complete the argument list field, the Marked Application node reacts as follows:

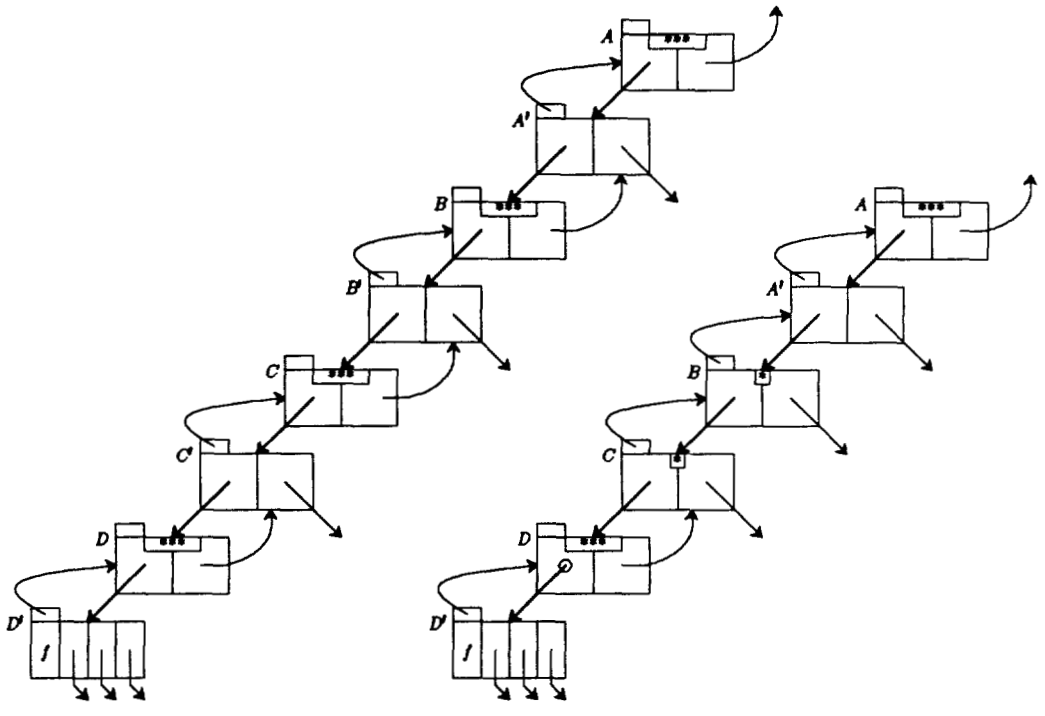

Figure 3.9 — Substituting Marked Applications (right graph) for Application-Marker chains (left graph). Instead of a chain that looks like

$$Packet \rightarrow Marker_0 \rightarrow Application_1 \rightarrow Marker_1 \rightarrow Application_2 \rightarrow \cdots \rightarrow Marker_n$$

the chain is

$$Packet \rightarrow Marker_0 \rightarrow Marked\ Application_1 \rightarrow \cdots \rightarrow Application_n \rightarrow Marker_n$$

1. The Marked Application node makes a copy of itself. This copy is transformed into a new Application node whose redex address is a reference to the Marked Application.

2. The Marked Application forwards the Packet message to the new Application.

3. The Marked Application transforms itself into a Marker node. The task field of the new Marker is set to a reference to the new Application created in step (1). The redex address of the Marked Application is transferred to the notifier list and the redex

address is erased.

The effect of this is to remove one level of indirection for every Application in the chain (see Figure 3.9).

An obvious drawback of this scheme is that it introduces a number of cyclic references. These references will all be removed by successive update and packetization transformations, but it would be preferable to avoid them entirely. If a purely message-driven implementation is not required, a recursive packet formation algorithm can produce the same effect without introducing as many cycles.

3.9. Recursive Formation of Packet Nodes

Combinator and Packet messages and their handling by Application (or Marked Application) nodes are essential to the fully message-driven abstract model. However, in an implementation where the number of processors is too few to allocate a real processor to each node, the entire left spine of a subexpression may be available to a single processor. In this case it is simpler to let the processor descend the spine recursively until it encounters a Combinator node, and then build a Packet as it returns towards the upper application. This is essentially the same as performing a stack-based evaluation.

Such a recursive technique eliminates the need for a number of node type transformations, and also replaces some Evaluate and Demand messages sent as the result of Packet messages. Aside from efficiency issues, elimination of these messages is useful for garbage collection purposes. Reference rights would normally need to be shared in order to send the messages. The drawback is that the recursive examination must be suspended whenever there are more Application nodes in the spine than are needed to fill the argument list of one Packet.

Fortunately, in such a case the processor has references both to the topmost Application and to the Marker representing the completed packet. Rather than construct the chain of Marked Application nodes described above, the topmost Marker can be formed and a Demand message for that Marker can be placed in the notifier list of the Marker representing the Packet. Then, when the lower Marker is updated, the Demand message is sent and the recursive Packet formation will be restarted from the top. The following recursive algorithm describes this process more formally.

Algorithm 3.5

Input:

References to the demanded node T, the root node R, and the demanding node N. For the first call to the algorithm, T=R.

Output:

One of:

MESSAGE(x)

The node x in message form, where x must be a Combinator or partial Packet.

SUSPENDED(x)

A reference to node x, tagged so it can be identified as a Marker node where Packet formation suspended.

DEMANDED(x)

A reference to node x, tagged so it can be identified as a Packet node that has been demanded.

Algorithm:

Apply the function ***packetize*** as defined in Figure 3.10. If the return is MESSAGE, deliver the message to N. Otherwise, computation is in progress and R will send a

message to N at some later time. The following theorem asserts the correctness of this algorithm.

Theorem 3.6: Extension of Theorem 3.4

The recursive Packet formation algorithm demands the subexpression rooted at R in a manner equivalent to sending a Demand message to R in the pure message-driven abstract model.

Proof

The proof will make use of the following lemma:

Lemma 3.7

If *packetize* returns SUSPENDED(x), then either x must be a Marker whose notifier list contains a Demand for R, or x=R.

Proof (3.7)

By induction on the recursive definition of *packetize*. The returns on lines (7), (14) and (19) satisfy the condition trivially. On lines (8) and (16) the value of a recursive call to *packetize* is being returned, so if that value is SUSPENDED it must refer to such a Marker.

□

Proof (Theorem 3.6)

Assume that the combinator program represented by the graph has a normal form. Let D, R, and N be defined as in Figure 3.10.

Base case:

Lines (17), (21) and (23) are equivalent to direct delivery of a Demand message.

Case 1:

On line (13), a Demand for the root R is placed in the notifier list of D. By

definition of the DEMANDED return, on line (14), and the inductive hypothesis, D will be updated. This will send the Demand from the notifier list to R.

Case 2:

Again, on line (18) a Demand for the root R is placed in the notifier list of D. By the inductive hypothesis and Theorem 3.4, the task field of D has been demanded, so it will update D. R will therefore be sent the Demand from the notifier list.

Case 3:

By the inductive hypothesis, if the recursive calls on line (1) returns MESSAGE, then delivery of that messages on line (15) will correctly update D. The recursive calls on line (16), whose value is being returned, must therefore produce either MESSAGE or DEMANDED, correct by another appeal to induction.

Case 4:

By 3.7, if the call on line (1) returned SUSPENDED(x), then x must have a Demand for R in its notifier list. By the inductive hypothesis and Theorem 3.4, x will be updated, so R will receive a Demand.

□

```
packetize(D, R, N)
        CASE D.type IN
        Application:
(1)             CASE packetize(D.function, R, N) IN
                SUSPENDED(z):
                        IF D=R THEN
(2)                             copy D to D'
(3)                             set D'.redex=D
(4)                             change D.type to Marker
(5)                             set D.task=D'
(6)                             set D.count=1
(7)                             return SUSPENDED(D)
                        ELSE
(8)                             return SUSPENDED(z)
                        FI
                DEMANDED(z):
(9)                     let f=D.function
(10)                    change type of D to Marker
(11)                    set D.task=f
(12)                    set D.count=1
(13)                    place Demand(R for N) in D.notifier_list
(14)                    return SUSPENDED(D)
                MESSAGE(z):
(15)                    deliver MESSAGE(z) to D
(16)                    return packetize(D, R, N)
                ESAC
        Combinator:
(17)            return MESSAGE(D)
        Marker:
(18)            place Demand(R for N) in D.notifier_list
(19)            return SUSPENDED(D)
        Packet:
(20)            IF D.argument_list is complete THEN
(21)                    deliver Demand(D for N) to D
(22)                    return DEMANDED(D)
                ELSE
(23)                    return MESSAGE(D)
                FI
        ESAC
END(packetize)
```

Figure 3.10 — Pseudo-code detailing the recursive packet formation algorithm. D is the demanded node, R is the root of the expression, and N is the node that demanded the expression. The initial call is `packetize`(D, D, N).

CHAPTER 4

Speculative Computation and Priorities

An important goal of this research is to explore ways to automatically discover parallelism in programs, without introducing explicit parallel constructs into the source language. Parallel constructs are useful, but are difficult to apply to programs that will run in a massively parallel environment because the number of tasks is very large and the execution dynamics may be uncertain. To discover parallelism, the MPCR execution mechanism performs *speculative* evaluation, that is, attempts evaluation of subexpressions without knowing whether the values are needed.

Other techniques to detect parallelism, such as strictness analysis [BHA86], depend on static analysis of programs. These techniques are improving, but are still unable to detect all available concurrency in the general case. Speculative evaluation is able to uncover this parallelism automatically and dynamically, allowing the system to adapt to run-time variations in program behavior. A drawback to this approach, however, is that a speculatively evaluated subexpression may represent a nonterminating, or *divergent*, computation. Some mechanism is therefore required to control these computations.

4.1. Creating and Controlling Speculative Tasks

The MPCR controls divergence by a combination of two methods. The first is an adaptation of the dataflow iteration level tagging scheme, and the second is based on known properties of combinators. For purposes of the discussion which follows, we will consider only the S, K, and I combinators, but these techniques can be generalized to any fixed

combinator. †

The problem with beginning computations that may diverge is that they steal CPU cycles and memory from more useful work. It is the goal of the first method to control *computational divergence* by limiting the CPU resources applied to tasks whose results are not immediately required. An effect of this limitation is to control *memory divergence*, that is, divergent computations that consume memory needed for useful work. Computational and memory divergence are closely linked in combinator reduction, because the only source of nonterminating computation is recursion.

The first strategy is to tag redices with an *evaluation priority*, and to schedule reductions according to this priority. Sub-expressions whose values are known to be required are given higher priority than those whose value may not be needed. Using priorities to control speculative evaluation was first proposed by Burton [Bur85]. Burton's scheme assigns priorities explicitly, by program annotation. However, it is possible to automatically derive priorities for speculative evaluations from the forms of combinator expressions.

Priorities are assigned to subexpressions by comparison with the priority of the outer expression. For example, the usual reduction rule for the S combinator is

$$S\ f\ g\ x \rightarrow f\ x\ (g\ x)$$

Representing priorities by superscripts, reduction of an S combinator expression with priority i follows the transformation

$$(S\ f^j\ g^k\ x^l)^i \rightarrow (f^{max(i,j)}\ x^l\ (g^{max(i-1,k)}\ x^l)^{i-1})^i$$

Note that this transformation does not change the priority of subexpression x, and assigns new priority $i-1$ to the new subexpression $g\ x$. Such priority assignments are ap-

† Compiled super-combinators *ala* the G-machine [Kie85] are another matter. It is possible that a compiler might be able to classify them, in which case this technique could be extended to programmed graph reduction.

propriate for normal-order reduction of the expression. To approximate applicative-order (eager) reduction, the transformation might become

$$(S\ f^{j}\ g^{k}\ x^{l})^{i} \longrightarrow (f^{max(i,j)}\ x^{max(i,l)}\ (g^{max(i,k)}\ x^{max(i,l)})^{i})^{i}$$

Here, the priorities of f and x would be increased if necessary, and the new subexpression $g\ x$ given a priority at least equal to that of the application in which it is used.

Eager evaluation and lazy evaluation are equivalent for K and I, so in either case their transformations would be

$$(K\ x^{j}\ y^{k})^{i} \longrightarrow x^{max(i,j)}$$

$$(I\ x^{j})^{i} \longrightarrow x^{max(i,j)}$$

It should be noted that the priority of subexpression f in the S reduction and x in the K and I reductions may increase even in the normal-order priority formulation. When f is a weak head normal form this priority change has no effect. The subexpressions of WHNFs are given a priority only when they are accessed, for example by supplying additional arguments to make the expression reducible or by using a special combinator to select an element of a Constructor node. If f is a redex and has not yet been evaluated, the new priority is assigned before evaluation begins†. However, if f has already begun evaluating speculatively and the result of that evaluation has not yet been returned, the priority of the task representing f must be increased. If priority is not increased, the task that originally demanded evaluation of the subexpression will be delayed, perhaps indefinitely. The technique used to increase priorities will be described completely in the next section.

The second technique also addresses the problem of memory-divergent computations. We classify all combinators as either *expansive* or *contractive* depending upon the effects of

† A remote reference may be considered an unevaluated redex for purposes of assigning new priorities.

their application. *Expansive* combinators cause additional nodes to be added to the program graph, and an *expansive redex* is an expression whose reduction involves application of an expansive combinator. *Contractive* combinators remove nodes, and are applied when evaluating a *contractive redex*. *Neutral* combinators, which do not change the number of nodes in the graph, may be considered contractive for purposes of this discussion. When eagerly evaluated, expansive redices provide new opportunities for parallelism, whereas contractive redices normally do not. (However, *strict* combinators, which require their arguments to be fully evaluated, can be contractive and still provide opportunities for parallelism.)

The reduction of too many expansive combinators leads to memory divergence. Evaluation of contractive redices, however, at worst maintains the current memory usage and generally will decrease memory occupancy. When high memory occupancy is detected, the run-time system attempts to reduce only contractive combinators until a sufficient amount of memory has been made available. Even if some expansive combinators must be reduced, new speculative evaluations can be avoided, thus greatly slowing the expansion.

Another possible use of the expansive/contractive classification strategy is to speed the creation of new work. When too little work is available, the system could eagerly reduce expansive combinators to increase the number of tasks. This is impractical for simple, fine-grained combinators, because contractive combinators appear too frequently. This strategy has therefore not been used in the MPCR simulations. However, it may be useful for super-combinator reduction.

When the resources at any node begin to near saturation, speculative evaluations must be sacrificed. The details of task deletion in the abstract model are discussed below. The technique is simply to select a low-priority speculative task and terminate it. A mes-

sage is sent to the task's redex address so that other tasks awaiting the deleted task can be dealt with. Other reference rights held by the task are then released, and the task is deallocated.

4.2. Speculation in the Abstract Model

The abstract model as described in the previous chapter supports only *conservative* evaluation. Speculative evaluation can be added to the model with a few changes. One important modification is to record the priorities of each evaluation. In addition to the formats already described, every node type is given a *priority* field. Demand messages are also supplied with a priority field. The priority of any node is initially undefined, but will record the priority at which it was demanded. The priority field of a Demand message gives the priority at which the demand is to be satisfied. The initial Demand message therefore carries the highest possible priority. Unless otherwise noted, any other Demand message carries the priority of the node which sent it. No node may ever send a Demand with higher priority than its own.

Priorities assigned to subexpressions by the method described above can never increase above that of the outermost expression. For simplicity, then, zero (0) will be used as the highest priority, and all lower priorities will be represented by negative integers.

4.2.1. Creation of Speculative Tasks

Speculative tasks are created only during the evaluation transformation. When a function evaluating a complete Packet wishes to create a speculative task, it sends to the selected subexpression a Demand message with priority one less than than that of the Packet it is evaluating. The redex address of this Demand message is undefined, because it is not yet known which node is interested in the value of the subexpression. In fact, it is inherent

in the definition of speculative evaluation that *no* node may be interested in the value. These reduced-priority Demand messages, having no redex address, will be referred to as *Speculate* messages. If a Demand message has either a redex address or a priority of zero, it is not a Speculate message.

Speculate messages are normally sent only to Application nodes, but as they are for all other purposes Demand messages, they can be sent to any node type. Combinator nodes and partial Packet nodes ignore Speculate messages, because they are already in normal form. An Application, Marker, or complete Packet node either may react to a Speculate message as if it were a regular Demand message, or may ignore the message. The decision to ignore the message is implementation specific, but in principle, it is made when no virtual processors are available to handle the new nodes that the marking or evaluation transformations may create.

The lack of a redex address in a Speculate message means that at least one "real" Demand message must be sent to obtain the value of a speculative evaluation. Furthermore, the priority of a speculative subexpression may need to be increased. This is handled by sending either a Demand or another Speculate message with higher priority than the original message. Any speculatively evaluated subexpression will thus receive two or more Demand messages.

One final change is necessary to support multiple Demands. Combinator and partial Packet nodes must react to messages representing normal forms. In the conservative model, such messages would be a run-time error. However, the speculative model may evaluate any subexpression several times. Combinator and partial Packet nodes therefore ignore messages that have identical type and content to the node that receives them.

4.2.1.1. Changing the Priority of a Task

Fortunately, the abstract model already supports delivery of multiple Demand messages through the introduction of Marker nodes. New nodes created during the marking transformation are referenced only through the task field of the Marker. The Marker thus receives each Demand or Speculate message, and may determine whether it is necessary to increase the priority of the task node. In addition, to support task deletion, a *task count* is added to each Marker to record the number of evaluations represented by that Marker node. This will be explained more fully in the next sections. The Marker's task count is initially one (1).

When a Marker node receives a Demand message, it checks its task count and compares its priority to that of the message. If the Demand priority is lower than the Marker's priority *and* the task count is greater than zero, nothing needs to be done. Otherwise, the priority of both the Marker node and the task node must be increased. The Marker node also checks the references in its notifier list. If none of them is the same as the redex address of the Demand message, that redex address is added to the list. If any one of them is the same, the one with higher priority is retained in the notifier list, and the other is returned to its redex address as an Exited message. This new message type will be described later. With the exception of this new first step, the reaction of a Marker to a Demand message remains unchanged.

The most straightforward way to increase the priority of the task node would be to send it a message informing it of its new priority. This introduces a few complications:

1. If the task node is an Application, it must propagate the increased priority to its left function node.

2. If the task node is a Marker, the increase in priority must be treated as a new Demand.

3. If the task node is a complete Packet and an evaluation transformation is in progress, its priority must be increased immediately.

4. Any subexpression demanded during an evaluation in progress should also have its priority increased. To accomplish this, the evaluation might have to be restarted, at least from the point where it issued the Demands to its subexpression(s).

The fourth complication is quite serious. For a strict combinator, it may not be possible to complete the evaluation without obtaining the value of a subexpression. If the priorities of subexpressions are not increased, an evaluation which should have high priority may instead be delayed indefinitely.

Direct increase of priorities requires that a Marker node must be able to communicate directly with the task node. In a system with no global address space, such a reference can be provided in any of several ways. The task node may be allocated to the same physical processor as the Marker, but this permits no concurrent evaluation, defeating the purpose in creating a new task. It could be allocated to a specific processor (as in Alfalfa), but selection of the processor may be limited by the global scheduling algorithm, causing a poor choice to be made. Or the task could be tracked by use of messages, either by leaving forwarding pointers or by returning an extra message when a processor is selected. The latter requires additional space or message-passing overhead and may introduce still other delays for high-priority tasks. For these reasons, a different tactic was adopted.

The task node created during the marking transformation is required to be allocated on the same physical processor as the Marker, but that node is never evaluated directly. Instead, the reaction of a Packet to a Demand message is altered, and a new message type is

introduced. The new message is the *Evaluate* message, which carries no other information than its type. These messages are understood only by Packet nodes. The evaluation transformation, that is, steps (2) and (3) of the reaction of a Packet to a Demand, is now assigned as the reaction to the Evaluate message.

The reaction of a Packet to a Demand is unchanged if the Packet is *partial*, lacking one or more argument list subfields. In this case, the Packet returns itself as usual. Every complete Packet that receives a Demand message reacts by creating a new copy of itself. The redex address of the Demand is transferred to the new copy, *not* to the original Packet. The new copy of the Packet is then sent an Evaluate message, to complete the evaluation transformation. This revised transformation is shown in Figure 4.1.

If a Marker needs to increase the priority of its task, a new Demand message is sent to the Marker's task field reference. The redex address of the new Demand message is a reference to the Marker. This may result in nodes other than Markers receiving multiple

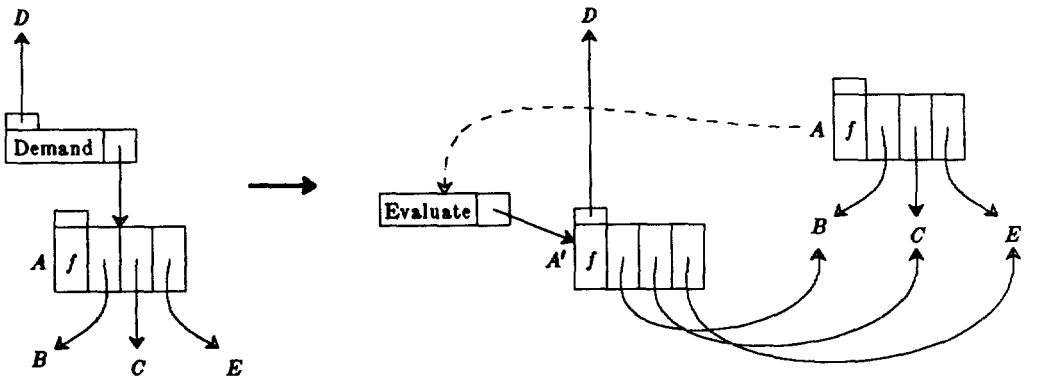

Figure 4.1 — Reaction of complete Packet node to Demand message. This is the initialization step of the *evaluation transformation*. In the conservative model, the Packet is not copied before evaluation.

Demands, but, with the exception of Markers, all Demands to the same node will have the same redex address. If the task field refers to a complete Packet, each such Demand will create a new task with a new priority. In order to count the number of tasks its Demands have created, a Marker node must "know" whether its task field refers to a complete Packet. In a purely message-driven model, this information is not available. However, with the restriction that the task field must refer to a node on the same physical processor, a complete Packet may use its redex address reference to increment the task count of its Marker whenever a new evaluation is begun.

From this point forward, the term *task* will be redefined to refer to a complete Packet which has received an Evaluate message. Evaluate messages are a special case for garbage collection, because they carry *all* rights to the node to which they are sent. The original Packet does not maintain a copy of the reference (in fact, it cannot, having no field in which to store it). For this reason, rights carried by an Evaluate message are not released immediately when it is received. Instead, they are transferred to the Demand message in the final step of the evaluation transformation. The independence of these new tasks from the rest of the graph has two important side-effects:

1. A new task need not retain any specific spatial relationship to the node which created it. This is important for processor mappings. If the global load distribution in the system has changed, the new task has a chance to migrate to a less-loaded processor.
2. Speculative computation can be controlled simply by deleting Packets with priority less than zero. Another copy of the task will be created if it is ever demanded again.

The price paid for restarting work in this way is some duplication of effort. However, restarting the task does not cause a loss of sharing. Any work that has already been completed by a previously created task will be accessible to the new task. In the worst case, the en-

tire subexpression could be recomputed once for each priority at which it is (re)scheduled.

The worst case occurs when values are demanded more quickly than they can be computed. In this case, a higher-priority task will be created before a previously created, lower-priority task is able to return its results. If there are sufficient processor cycles available, two or more tasks may be created for every component of the expression, and every set of these tasks may successfully compute and return a value. This multiplication of effort can only occur when there is not enough highest-priority work to keep all processors busy. However, if processors would have been idle without the lower-priority work, the evaluation would have completed no more quickly even if the duplicate effort had been avoided. This technique trades the overhead of performing duplicate work for the overhead of tracing each task in order to be able to increase its priority.

A consequence of starting duplicate tasks to increase priorities is that multiple copies may be simultaneously active. This does not pose a problem for updates, because at most one Combinator or Packet message can update a given Marker. Once the update transformation has occurred, the node simply ignores any Combinator or Packet messages that the extra copies may send. However, for purposes of cleaning up these multiple copies should the speculative evaluations prove unnecessary, it is useful to maintain a count of the duplicate tasks. The count is also necessary for making the decision to start another new copy.

4.2.1.2. Proofs

This section presents theorems to show that the addition of speculative computation, by the changes described here, does not affect the correctness and completeness results of the previous chapter.

Lemma 4.8

Copying a Packet node and evaluating the copy is equivalent to evaluating the origi-

nal Packet directly.

Proof

Let A be a Packet node. When A receives a Demand message, it creates new node A' as a copy of itself. A then transfers the redex address of the message to the redex address of A', and sends an Evaluate message to A'. This Evaluate message represents the only reference to A', so A' will not receive any Demand messages (except those sent to itself). As the final step of the evaluation transformation, A' sends itself a Demand message, using its own redex address as that of the Demand message. This redex address is the same as that of the original Demand message, sent to A. Therefore any response made by A' to this Demand will be returned to the node that sent the original Demand.

QED.

□

Lemma 4.9

Multiple Demand messages sent to the task field of a Marker will not cause the Marker to be incorrectly updated.

Proof

Let A be the Marker and A' be the node referenced by the task field of A. Consider cases on the type of A' when A receives a Demand:

Combinator or partial Packet:

The first Demand will result in the update of A. By definition, normal forms ignore subsequent, equivalent messages. By Theorem 3.3, only the node referenced in the task field or an equivalent node can send such a message, so the computation is unaffected.

Application:

The first Demand will transform A' to a Marker. By the inductive hypothesis, A' will be correctly updated. Therefore A will be correctly updated.

Marker:

Demands of increasing priority will cause new Demands to be sent to the task field of A'. By the inductive hypothesis, this has no effect on correctness. Demands of equal or lesser priority will be ignored.

Complete Packet:

By Lemma 4.8, all copies will send equivalent Combinator or Packet messages to the Marker. By definition, only the first of these is recognized, and will update the Marker. Correctness is not affected.

□

Theorem 4.10

A Demand message sent to any node always creates a node which has priority equal to the Demand and which is not also a Marker, or a node that has equal or greater priority and is not a Marker must already exist.

Proof

The only case in which a Demand message does not immediately create such a node is when that Demand is received by a Marker. Let the Demand have priority P, and the Marker have priority P_M.

1. If M is a Marker with priority $P_M \geq P$, then by definition the task field M' of M also refers to a node with priority $P_M \geq M$. If M' is not a Marker, the theorem holds trivially. If M' is a Marker, the theorem holds by induction.

2. If M is a Marker with priority $P_M < P$, M must by definition send a Demand to its task field M'. By Lemma 4.9, this does not affect the correctness of the computation. Again, if M' is not a Marker, the priorities of M and M' will be set to P and the theorem holds. If M' is a Marker, the theorem holds by induction.

□

4.2.2. Deletion of Speculative Tasks

Certain tasks (demanded subexpressions) that have priority less than zero may be deleted to make a virtual processor available to a higher-priority task. For purposes of this discussion, it is assumed that a task whose priority is equal to the lowest of any task in the machine can be selected. As long as the priority of the deleted task is less than zero, however, the absolute priority is not significant. The selected task is terminated, in effect by obtaining the reference rights from its Evaluate message and sending them in a Delete message instead.

Only complete Packet nodes that have a valid redex address field and that have received an Evaluate message are candidates for this type of deletion. This may seem to be a rather small subset of all nodes, but remember that every reducible expression in the graph must at some time have this form. When the correspondence of real to virtual processors is not one-to-one, low priority evaluations may spend considerable time in this state.

When a Packet of this type is terminated, it first sends an *Exited* message to its redex address, and erases the redex address field. If the node receiving this Exited message is an Application, the message is forwarded through the redex address of that node. Combinator and Packet nodes ignore Exited messages. When a Marker receives an Exited message, it decrements its task count by one. If the task count is zero, Exited messages are also sent to every reference in the Marker's notifier list, and the list is emptied. In this way, every

Marker affected by the deletion of the task is notified.

The behavior already described for Marker nodes that receive Demand messages guarantees that priority zero computations will continue in spite of such deletions.

Theorem 4.11

Deletion of Packets whose priority is less than zero (the highest) does not affect correctness of a computation.

Proof

Recall that only Packets which have received an Evaluate message can be deleted. Such Packets by definition must be copies of another Packet.

Speculate messages are sent only during the evaluation transformation. Demands sent in all other cases carry both a redex address and a priority. By definition, the priority of the Demand is the same as that of the node that sent the demand. Also by definition, an evaluation transformation can occur only after a complete Packet has been formed. Since the initial Demand has priority 0, either no complete Packet is ever formed or the first complete Packet formed must be a node with priority 0. Call this node T. For purposes of this proof, the equivalence from Lemma 4.8 is used to ignore the fact that T is copied before evaluation.

By definition of the evaluation transformation, a Demand is sent by T to itself. If T is an Application, another priority 0 Demand is sent to its left function F_T. By Theorem 4.10, this must result in another non-Marker with priority 0. By consequence of Theorem 3.1, and the definition of Demand messages, this non-Marker with priority 0 must update a Marker. By Lemma 4.9 and consequence of Theorem 3.4, the computation must be correct.

□

4.2.3. Summary of Changes to the Abstract Model

Priority Fields

Nodes and messages have an additional field for priorities. Priorities are initially undefined, and are propagated by Demand messages. The initial Demand has highest priority. All other Demands have priority equal to or less than that of the node which sent the Demand. In particular, note that remote reference requests are also Demand messages, and are given the same priority as the node which originated the request.

Application and Marker nodes reset their priority to the highest received via Demand messages so far. Complete Packets set their priority to that of the most recently received Demand. Note that this will never result in a decrease in priority unless all other copies of the Packet have been terminated, because by definition a Marker will never send a new Demand to its task field unless the priority has increased or the Marker's task count is zero. Normal forms (Combinators and partial Packets) do not change their priority when they receive a Demand, because they respond immediately regardless of priority.

Speculate Messages

Demand messages with reduced priority and undefined redex address are called Speculate messages. They are sent during the evaluation transformation to start a speculative evaluation. They are treated as Demand messages, except that they may be ignored if the receiving node so chooses.

Task Counts

Marker nodes cooperate with the nodes referenced through their task field to maintain task counts. If the task field refers to a complete Packet, the task count is incremented each time a new copy of the Packet is created. The task count is decremented each time an

Exited message is received by the Marker.

Evaluate Messages

To support multiple Demands for the same complete Packet at different priorities, the evaluation transformation has been split into two parts. Upon receiving a Demand, a complete Packet makes a copy of itself and sends the copy an Evaluate message. The copy then undergoes the evaluation transformation.

Exited Messages

The creation of tasks that may or may not be needed, or of multiple copies of the same task, may eventually strain the limits of various resources. To free these resources for more important tasks, low-priority tasks may be deleted. When a task is deleted, it sends an Exited message to its redex address.

Exited messages are also sent to every redex address in the notifier list of a Marker, when that Marker's task count reaches zero. This propagates the information that a task has been terminated to all the other tasks awaiting it. It also permits garbage collection of Marker nodes that represent useless work.

CHAPTER 5

Mapping Virtual Processors to Physical Processors

Up to this point, the discussion of massively parallel reduction has remained on a mostly abstract level. Implementation considerations have been mentioned only in passing. One important part of the implementation of an abstract model that allocates large numbers of virtual processors is the mapping of those virtual processors to the physical processors that are available in a real machine.

For the case of the MPCR, there are two decisions to be made in making the virtual-to-physical mapping. The first, common to all multitasking systems, is the choice of the best processor on which to execute a given task. A wide variety of techniques exist for making this choice. Rather than attempt to detail the possibilities, the first section of this chapter will introduce the technique that was chosen for the simulations described in later chapters. The reasons for this choice will also be discussed.

The second decision is related to the model of speculative computations, and is the choice of whether to start a given speculative task at all. The most naive answer is to continue creating speculative tasks until some resource is exhausted. This leaves the question of which resource to monitor and how to determine that it is used up. Full memory occupancy can easily be determined, but in a system employing extremely fine-grained tasks, processor cycles are an equally important resource.

One measure of the availability of processor cycles is to keep track of the processor idle time. However, this measure may be inaccurate in the presence of speculative tasks. The processor may be busy, but it may be performing useless speculative work. The priority

scheme already described will limit this effect, but there may not be enough high-priority work to keep the processor active. The goal is to create enough speculative work to occupy otherwise idle cycles, without allowing the overheads of maintaining the priority queue to become excessive. The second section of this chapter presents a technique to estimate the number of tasks that will acheive this goal.

5.1. Distributing Workload

The goal of any scheduling algorithm is to distribute tasks among processors in a manner that will produce the best performance, though the notion of what is "best" depends on the nature of the tasks and the processors. In heterogeneous systems, wherein different processors may have different characteristics, some tasks may need to be assigned to a particular processor. If tasks have known time limits within which they must complete, as in real-time systems, a task may be assigned to any processor that is able to guarantee its completion before the deadline, even if the task might finish slightly sooner elsewhere. For this research, however, a homogeneous system with no deadlines is assumed, so an algorithm which produces balanced workloads among the processors is sufficient. Remember that in the fine-grained model, all operations including memory accesses are treated as part of the workload. Load-balancing cannot entirely compensate for nonuniform accesses, but should perform as well as any other scheme in terms of directing other work away from processors that must service many data requests.

Diffusion scheduling is an heuristic method for dynamic distribution of workload in a multiprocessor system, with the goal of achieving nearly equal loads at all processor nodes. It differs from other distributed scheduling algorithms in that communication takes place only between directly connected processing nodes in a network with less than complete connectivity. Diffusion scheduling thus scales well as the number of processors increases, which

is important in a system with potentially thousands of processors.

The name *diffusion* scheduling is drawn from an analogy to gas diffusion physics, which describes the tendency of molecules to migrate from areas of greater density or *pressure* to areas of lesser density. This flow across a *pressure gradient* is modeled by computing workloads at each processor and sharing this information with neighboring processors. Tasks are then transferred from processors with high loads to those with lower loads. This method was introduced in the Rediflow system [KeL84] and DAPS [HuG84]. A summary of other dynamic scheduling techniques can be found in WaM85.

Control of scheduling is distributed among all the participating nodes, rather than residing at some central location. This is advantageous because the overheads of the scheduling process are divided among the processors. In any load-balancing scheme with distributed control, each node periodically examines its workload and decides whether some portion of the work should be offloaded to another processor. How often this examination is undertaken depends on the particular scheduling algorithm. The decision of when to offload work is called the *transfer policy* [ELZ86]. It is usually based on some measure of the current workload at a subset of the processors [GoH87, Gol88, HuG84, LiK86, LiK87, Sta84]. However, other system state information may be used. Systems with several different processor types or nonuniform connectivity [LiK87] may scale loads for more powerful processors or use a cost function for communications. Systems executing several different types of tasks [NiH85] may vary the perceived pressure depending on the task type.

If the decision to shift workload is to be based on information about other processors, then this information must somehow be communicated to each node. The *information policy* [BaS85, LiM82] determines which processors will share information, how often, and what information they will share. In a completely connected system, each node may receive infor-

mation about the load from every other node, and may send work to any other node. However, in a system with lesser connectivity, such as a hypercube or a mesh, it is often far more efficient to have each node communicate only with its directly-connected (*nearest*) neighbors. Distribution decisions are thus based on regional, or *neighborhood*, load information. This avoids the potential bottleneck of maintaining all load information in a central location. To overcome local load maxima or minima, the information provided by each node to its nearest neighbors can be modified in some way to reflect its knowledge of the neighborhood load [LiK86,LiK87]. The experiments described in this thesis have used an average of the local load plus the load values received from all neighbors to compute the load value that is exchanged.

Once a decision to offload work has been made, the *location policy* [ELZ86] determines where the work will be sent. This determination may employ much of the same information as the transfer policy, but may also use additional information such as locality of references [GoH87,Gol88,HuG84]. Some algorithms use predetermined criteria such as round-robin selection [GoH87,Gol88] or a set of probabilities [ChA82,HsL86]. The latter are usually used when no state information is exchanged among nodes. In diffusion scheduling, tasks are always sent to nearest neighbors first. Depending on the particular algorithm, those tasks may or may not later be allowed to move to a more distant node. The algorithm used in this thesis research permits tasks to migrate until they are accepted by a processor, but once accepted, tasks do not migrate further.

One difficulty with diffusion scheduling is the possibility of *processor thrashing* [BrF81,ELZ86,NXG85]. This refers to a state in which all nodes spend all their time transferring tasks. Conditions under which this occurs are algorithm dependent, but it is usually associated with uniformly high loads. The research described in Chapter 6 has given

some insights on controlling processor thrashing, and has also provided a great deal of understanding about the dynamic interactions among the information, location, and transfer policies.

There are two other advantages of diffusion scheduling that have particular significance for implementation of the MPCR model. First, the amount of information that is carried as a single load message is quite small. In a message-driven system such as the MPCR, processors exchange messages frequently in the course of a computation. It is therefore possible to "piggyback" the load information on other messages, without greatly increasing the size of any individual message. This avoids the need for neighborhood-wide broadcasts and reduces scheduling overheads. Second, the neighborhood load information that is exchanged can be used in determining whether speculative evaluation is appropriate. This is described in the next two sections.

5.2. Deciding to Speculate

The decision to create speculative work should depend on the state of both the memory and processor resources of the machine. Free memory space can be measured directly, but measuring processor availability is a more difficult question. In a system with fine-grained tasks, a measure of current processor utilization may almost immediately become inaccurate, because each task completes very quickly. An estimate of current and *future* processor utilization is needed.

Three factors contribute to the time required to perform a computation: task execution, overhead, and latency. Task execution represents work that contributes directly to completing the computation, and overhead includes all other work that utilizes the same processor. Conversely, latency refers to the time required to perform some operation that does not directly involve the processor, such as accessing a disk drive or exchanging mes-

sages with another processor. Thus, the processor is active when dealing with tasks and overhead, but is idle during latency. The goal is to minimize the time spent in overheads and to eliminate idle cycles caused by latency.

Overhead has the effect of "slowing down" each processor. Each can spend only a portion of its time executing tasks, so the result is the same as if the time required to execute a task were increased. In a parallel system, overhead includes context switching, local and global task scheduling, and possibly some message pre- and post-processing. The reason that the diffusion scheduler is designed to make decisions at each processor, using a subset of the complete system load information, is that we desire to distribute this overhead equitably among all processors, as well as distributing the computation tasks. Use of fine-grained tasks also minimizes the overhead of context switches.

Latency, on the other hand, has the effect of "slowing down" *tasks*, rather than processors. For example, a task which requires data from another processor must wait for messages to be exchanged, and possibly for the time to compute the data as well. Multiprocessing, in the sense of executing one task on each of many processors, cannot directly compensate for latency. However, the basis for speedup in multiprogramming is interleaving execution of tasks which do not depend on one another in this way. When dependencies exist, the processor executing the dependent task should not be allowed to become idle. If the processor is quickly provided with another task that can execute while the dependent task is waiting, the latency of the dependent task has been *masked*, and has not slowed down the system. The combination of multiprocessing and multiprogramming permits some overheads, which cannot be masked, to be replaced with latency that can be. It is therefore important to generate enough work to mask latency, without generating so much work that the overhead of multiprogramming itself becomes unmanageable.

In addition to reducing the overhead of individual context switches, one goal in choosing a fine task granularity is to use multiplexing to mask latency. This idea is borrowed from dataflow, in which many small operations keep the processor(s) busy during memory accesses and other delays. Local task scheduling is viewed as an instruction pipeline. Some tasks may be waiting for communications to complete, but these tasks are not allowed to block the pipeline. Instead, they are placed in a pool, returning to the queue of ready tasks only when they have received the awaited communication. In a system where the same processor responsible for task execution is also responsible for some message processing, it is not possible to absorb the entire communication latency. However, as long as a significant part of the communication can occur concurrently with task execution, and there are sufficiently many ready tasks to keep the pipeline full, performance will not suffer as a result of communication delays.

In larger-grained parallel systems, the size and complexity of a task is used to mask latency. The complexity of the task contributes to masking because any task usually needs to wait for only *part* of the complete result computed by another task. That is, two larger tasks may need to synchronize at certain points in the course of their computations, but otherwise can continue independently. The latency of such tasks is reduced as compared to their execution time. The size of the task also contributes to masking because the run time of any task is a significant portion of the message transmission time. Only one or a few tasks need to be ready to run to keep the processor busy throughout a communication.

In fine-grained systems, two tasks with a direct dependency are rarely ready simultaneously, because the operations are so simple that the dependent task cannot even begin to execute before receiving data from the depended-upon task. Furthermore, the run time of a fine-grained task is a smaller fraction of the communication time, and context switches

are more frequent, so more tasks must be available to mask latency. Estimating this number of tasks and comparing the result to the current number of ready tasks provides the information on future processor utilization. If the current number of ready tasks appears too low, speculative work can be created to supply additional tasks.

If T_e is the time required to reduce a single combinator expression, and L is the latency, a given processor needs to perform $\frac{L}{T_e}$ reductions to absorb that latency. Computing accurate values of T_e for various combinators is not difficult, because each reduction step is very simple. The value of L is more difficult to determine, because it must account for delays occurring because data requested from another node may not be available, as well as for two-way transfer time, but a reasonable approximation can be made. Let

T_m be the average message transfer time between any two nodes;

T_e be the execution time for a task when it has all its data, as adjusted for overhead;

$N_r(x)$ be the length of the ready queue at processor x;

P_{ij} be the probability that i requires data produced by another task j—that is, that i will demand an unresolved reference (either a remote reference or a reference to an unevaluated subexpression).

We want to compute L, the *average latency* of any task from the time that it is demanded to the time it completes execution. For simplicity, we assume that no task i depends on more than one other task, though in reality this is determined by the strictness of each combinator. We also assume that task i does not execute on the same processor where it was demanded, because there is no maskable latency in the latter case. This also represents a worst-case scenario, in that both communication and evaluation time are included. First,

define $T_r(x)$, the time for a task to reach the front of the ready queue at processor x, as

$$T_r(x) = N_r(x) \cdot T_e \cdot (1-P_{ij}) \quad (1)$$

This represents the time for all tasks in the queue at x that do not depend on other tasks to complete, and assumes that each task enters the queue at the rear. Equivalently, this is the time for a task i scheduled at processor x to reach the front of the ready queue.

Upon reaching the front of the queue, task i may (with probability P_{ij}) demand evaluation of another task j. If some j is demanded, i will wait for j to run on processor y, then return to the ready queue and eventually execute. Otherwise it will execute immediately. To account for this, the latency equation for a task at processor x can be written as

$$L(x) = P_{ij} \cdot (T_r(x) + L(y) + P_{ij} \cdot T_r(x) + T_e + 2T_m) \quad (2)$$

T_e is of course the time for i to execute. The addition of $2T_m$ accounts for the message transfer time for task i to reach processor x and for the result to be returned. All of this is multiplied by P_{ij}, the probability that the task will be demanded at all.

To get a perfect picture of the latency, it would be necessary to consider fluctuations of T_e with changes in overhead and fluctuations of $N_r(x)$ for all x over changes in time. However, all that is needed is an approximation of the latency, so several simplifying assumptions can be made.

The first assumption is that the global scheduling process is in equilibrium. This should be the case except in the very early or very late stages of the entire computation. Under this assumption, the overhead at all processors will be nearly equal, so (given a homogeneous processor network) T_e can be taken as constant. This assumption also allows N_r to be treated as a constant independent of x and of time, because fluctuations in the lengths of the ready queues will be evenly distributed about the mean length, and the mean lengths of

the queues will be nearly the same at all processors. It is then reasonable to take L to be the same for any two processors, and solve for L:

$$L = \frac{P_{ij} \cdot ((1+P_{ij}) \cdot T_r + T_e + 2T_m)}{1-P_{ij}} \tag{3}$$

The subexpression $(1+P_{ij}) \cdot T_r + T_e$ can be viewed as the time to complete this task, including the time to bring i through the queue a second time if it had to wait while j was demanded. $1-P_{ij}$ in the denominator is the fraction of tasks that do not depend on any other task. The latency thus increases as the proportion of dependencies increases. The message transfer time also increases as the proportion of dependencies increases, to account for the possibility that j depends on another task, which may depend on yet another, and so on. Thus, as expected, the total time from the demand for an expression to the return of its value depends on the time to complete tasks for it and for all its subexpressions. Note that as the proportion of dependencies goes to zero, the time to complete a task becomes only the time to schedule and run that single task; and as the proportion of dependencies goes to one, the time to complete becomes infinite. This corresponds precisely to intuition about terminating and divergent computations.

Recalling that the number of reductions needed to mask this latency is given by $\frac{L}{T_e}$, we want to solve

$$N_r = \frac{\left(\frac{P_{ij} \cdot ((1+P_{ij}) \cdot N_r \cdot T_e \cdot (1-P_{ij}) + T_e + 2T_m)}{1-P_{ij}} \right)}{T_e} \tag{4}$$

Which gives

$$N_r = \frac{P_{ij}}{1-2P_{ij}+P_{ij}^3} \cdot \frac{T_e + 2T_m}{T_e} \tag{5}$$

The intuition about this equation is less clear. $T_e + 2T_m$ represents the average time for a single task with no dependencies to execute on a remote processor. If every task on the local processor were guaranteed to have no dependencies, $(T_e + 2T_m)/T_e$ tasks would mask that time. However, this guarantee cannot be made, so the estimate is adjusted by a factor of P_{ij}. Equation (5) still has the expected behaviors as P_{ij} varies from zero to one, i.e., when $P_{ij} = 0$ no tasks are required (there is no latency to mask), and when $P_{ij} = 1$ infinitely many tasks are required (no task ever completes).

Another interesting thing to note about Equation (5) is that it has reasonable values when P_{ij} ranges as high as approximately 50% to 58%, *independent* of T_m and T_e (see Figure 5.1). From 58% to 61%, N_r rapidly becomes unmanageably large, and if P_{ij} reaches 62%, the result is *negative*—effectively infinite, since negative values are meaningless in this case. This does not mean that it is impossible to get parallel speedup when the proportion of dependencies exceeds 61%, but it does indicate that this level of dependencies represents a threshold of diminishing returns, beyond which latency cannot be entirely masked.

All of these equations have explicitly ignored the possibility that a task may depend on more than one other task. However, this corresponds to the situation in a large-grain system wherein two tasks with a dependency relation are able to be ready at the same time, because the dependent task incrementally consumes a stream of results produced by the depended-on task. For example, suppose that task i depends on two tasks j and k. If j and k can run concurrently, i will be delayed only by the longer of $T_e(j)$ and $T_e(k)$. Thus, i actually depends on only *one* of j and k, the one that takes longer to complete. Furthermore, unless there is a dependency relation between j and k, the effect is as if i depended on only *part of* the longer-running task, because the shorter-running task will "stand in" for i during part of the delay. Therefore, multiple dependencies can be modeled by *reducing* P_{ij}

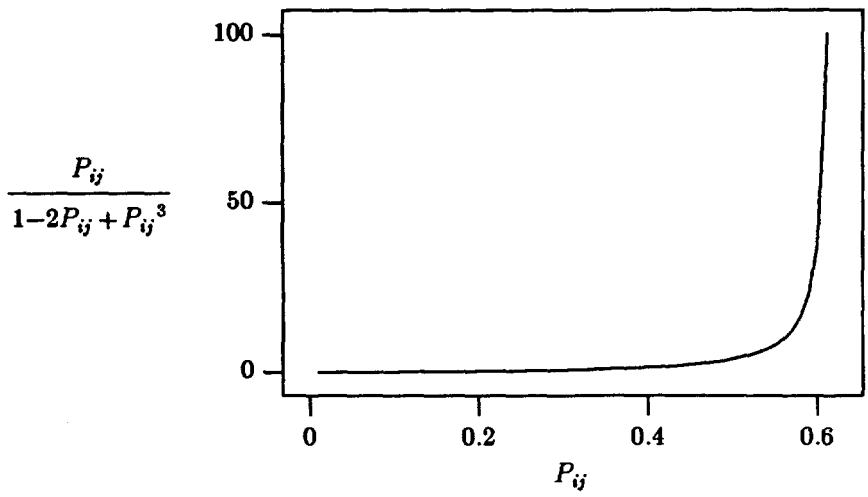

Figure 5.1 — The effect of dependencies on attempts to mask latency. As the proportion of dependencies increases, it becomes impossible to mask latency completely, regardless of the speed of processors or of communications.

as the proportion of tasks with multiple dependencies increases.

Other factors not considered here are speculative computation and priority; it has always been assumed that a task is unconditionally demanded and enters the ready queue at the rear. The most direct way to model these factors would be to compute $L(p)$ for each priority p, where $N_r(p)$ depends on the proportion of all tasks that have priority p or higher, and also on the proportion of tasks at each priority that are *useless* (never demanded). N_r is then the sum of all the $N_r(p)$. It is easy to see that these equations quickly become too complex to be readily usable. As with multiple dependencies, however, speculative computation can be modeled indirectly. Note that each useful speculative task that completes has the effect of eliminating a dependency. Also, each useful speculative task in the queue reduces the number of nonspeculative tasks needed to mask latency. Both of these

effects can be simulated by reducing P_{ij} in Equations (1) and (2). Priorities serve only to ensure that the latency of nonspeculative tasks will be small when compared to speculative tasks, and can be ignored in computing the overall average latency.

Unfortunately, this still leaves the problem of determining the value of P_{ij} for any given program. This could probably be accomplished by applying standard complexity analysis to each operation within a program, combined with analysis of the behavior of the combinators in the expression compiled from the operation. Such analysis is a possible topic of future research, but for purposes of this research a more general estimate, likely to be applicable to a range of programs, is desired. Estimating P_{ij} for the experiments in this thesis is discussed in Chapter 7.

5.3. Combining the Strategies

Although the heuristic described in the previous section can give clues about when it is desirable to speculate, a given processor may find itself unable to create speculative tasks. This can occur because only a subset of the combinators used in the system will have definitions suitable for speculative evaluation of subexpressions. A processor in this situation must depend on other processors to supply it with work.

Fortunately, in a system that employs diffusion scheduling, a processor need not rely only on its own load to decide whether speculation is worthwhile. The processor can examine the *neighborhood* load average compiled from the reports of its neighboring processors. If the neighborhood load is less than N_r, then some processors nearby must be in need of additional work, even if no more tasks are needed locally. On the other hand, if the neighborhood load is greater than N_r, then nearby processors will be attempting to offload work. In the former case, the processor can create speculative tasks to send to its needy neighbors. In the latter, new speculative work can be avoided because additional tasks are expected to

arrive from other processors.

It is also possible for the diffusion scheduler to make use of the estimated optimal queue length in its decision to accept or reject a task. If T_m is considered to be long relative to T_e, high priority tasks will suffer greater delay when rejected by a processor than they would in executing at a slightly more loaded processor. Furthermore, since a processor may become idle if its queue has fewer than N_r tasks, it is advantageous to accept tasks until that optimum is reached.

A consideration in using this strategy is that the equations for N_r include only non-speculative tasks. More accurately, the best case is for every one of the N_r tasks in a given ready queue to be *useful*, whether speculative or not. The approach needed in addressing this problem is to increase the *proportion* of useful tasks in each ready queue. The greater this proportion, the better the processor is able to mask latency. Increasing the estimate of N_r would increase the number of useful tasks, but not necessarily in proportion to the increase in queue length. Lengthening the queue would only overload each processor with no clear gain in terms of the amount of useful work performed.

To increase the proportion of highest-priority tasks at a given processor, the diffusion scheduler might be told to accept highest-priority packets preferentially. Only highest-priority tasks are known to be useful, so the best way to increase the proportion of useful tasks is to increase the proportion of highest-priority tasks. However, building up a queue of entirely highest-priority tasks may reduce parallelism. Several tasks of the same priority queued at the same processor have an artificially-introduced dependency on the tasks ahead of them in the queue, in the sense that tasks later in the queue do not execute until those earlier in the queue have completed.

The experiments described in Chapter 7 use a technique which attempts to balance these concerns. Optimal queue length is considered by the diffusion scheduler only if the priority of the task for which the decision is being made is greater than that of the highest priority task in the queue. This gives preference to highest-priority tasks without needlessly scheduling same-priority tasks on the same processor.

The combination of speculation heuristics and shared load information makes task creation, distribution and control very flexible. The techniques can be adapted to a variety of hardware architectures because they account for both message transfer and processing time in determining the number of tasks needed to mask latency. Furthermore, the amount of speculative work attempted will scale with the number of processors available. This permits a system employing these techniques to be used with little change as the size of MIMD computers continues to increase.

CHAPTER 6

Preliminary Research in Diffusion Scheduling

This chapter discusses simulated diffusion scheduling experiments performed prior to the development of the MPCR simulation. These experiments provided experience in designing and evaluating diffusion scheduling algorithms. Also, to make practical use of the results, the experiments evaluated a diffusion scheduler that is designed to be used in the Parallel Graph Reduction system, another research project at OGI. The PGR system is a parallel implementation of the G-machine [Kie85] that will run on the Intel iPSC/2, a hypercube-connected multiprocessor.

The goals of these experiments were:

1. To explore general considerations for the location policy for assigning tasks to processors.
2. To determine the effects of a specific architecture on the information policy for exchanging pressure data, and to tune the policy for that architecture.
3. To estimate the performance of a real architecture using a diffusion scheduling algorithm derived from the location and information policies.

The experiments were performed in simulation rather than on the real architecture because of the additional control that could be maintained. Coarse tuning of the scheduling algorithm proved as expected to be easier under simulation, because the behavior of the system could be observed and controlled while "computations" were in progress.

6.1. System Modeled

The run-time system for each processor node, shown in Figure 6.1, consists of two processes: the *Task Scheduler*, which provides the environment for the execution of tasks, and the *Task Distributor*, which is responsible for implementation of the diffusion scheduling algorithm. This division is intended to allow a high volume of inter-processor communication for dynamic scheduling, without requiring the task execution system to support inter-

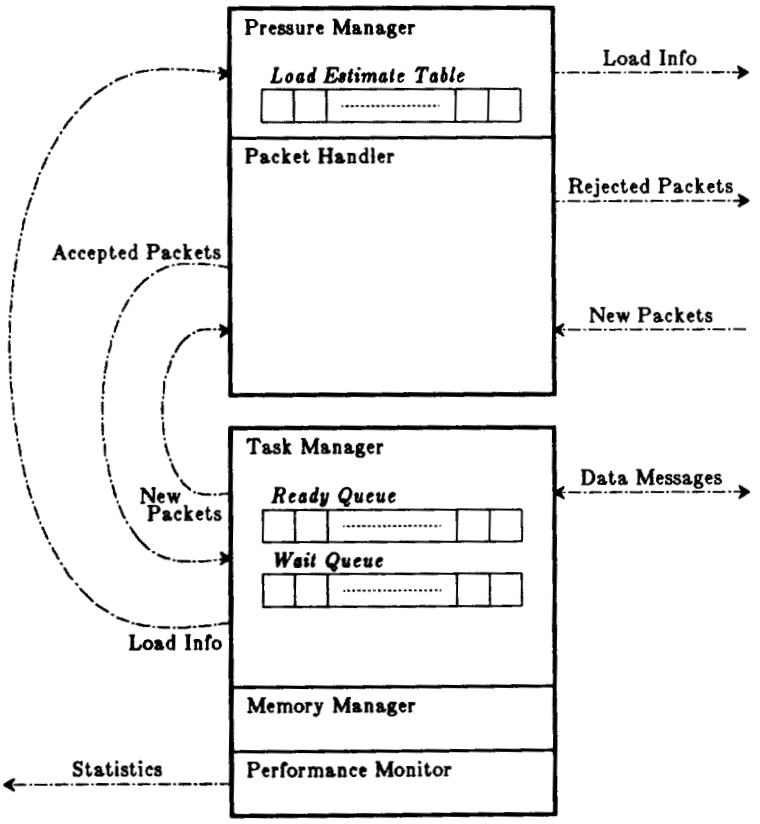

Figure 6.1 — Run-time System Design. The upper block is the Task Distributor, the lower block is the Task Scheduler.

ruptions for messages. These processes run in alternation via the processor operating system's time-sharing mechanism, and each is conceptually divided into a number of subprocesses.

6.1.1. Task Distributor

The Task Distributor is divided into the *Pressure Manager* and the *Packet Handler*. A *packet* is the encapsulated form of a task, constructed by a Task Scheduler and sent first to the local Task Distributor; from there, a packet may be sent to the Task Distributor of any neighboring node. The Packet Handler receives incoming packets and decides whether their tasks should be executed locally or on another node. If the decision is for remote execution, the Packet Handler forwards the packet to the Task Distributor on that node for further distribution, otherwise it sends it to the local Task Scheduler.

The Pressure Manager maintains estimates of load for the local node and its immediate neighbors by processing load messages from the local Task Scheduler and from neighbor Task Distributors. This load information is used by the Packet Handler to make distribution decisions. The Pressure Manager is also responsible for periodically sending local load information to neighbor Task Distributors, so that neighborhood load information is kept reasonably current. Load information consists of two values, the *pressure*, based on the number of ready and waiting tasks, and the *memory occupancy*, the percentage of total available memory that is currently in use.

6.1.2. Task Scheduler

Components of the Task Scheduler are the *Task Manager*, the *Memory Manager*, and the *Performance Monitor*. The Task Manager controls the execution of tasks, including management of READY and WAIT queues; sends and receives data messages on behalf of

tasks; constructs new packets and sends them to the local Task Distributor; receives packets from the local Task Distributor; and sends local load information to the Task Distributor. Packets are unpacked as they are received and the tasks are added to the READY queue. Tasks are run in a non-preemptive manner, so no other Task Scheduler functions are performed while a task is running.

Although task workspaces are allocated, initialized, and reference-counted from within the Task Manager, they are considered part of the Memory Manager. Similarly, much of the collection of various statistics is incorporated in the Task Manager and in the system calls available to tasks, but is considered part of the Performance Monitor, which periodically summarizes and reports this data. Collection of statistics can be controlled for each task by a *profiling* flag. If no statistics have been collected for any task, no performance report is made.

6.2. Diffusion Scheduling Task Simulator

The simulator is structured to match the assumed architecture as nearly as possible. All components of the architecture are present in the simulation, although the "memory manager" consists only of a counter. A simulation run consists of the simulated execution of a number of packets, starting with a single "seed" packet and expanding into a tree structure. Each simulated packet may cause a number of new packets to be created. In order to simulate completion of a computation, packet creation is bounded by limiting the depth of the tree, but this limitation is not inherent in the computation model.

Each packet is assigned a *size*, an *evaluation time*, and a *branching factor*. When a packet is accepted at a simulated processor, the memory manager increments its counter by the size of the packet, and the packet is placed in the READY queue as a task. Upon reaching the front of the queue, a task first creates as many *child* packets as indicated by

its branching factor. It then "executes" for its evaluation time. At the end of that time, if the task has no children, a dummy data message is sent to its parent. If the simulated task has children and has not received data messages from all of them, it will become inactive (move to the WAIT queue). No simulated task sends its own data message until it has received messages from all its children. After a task sends its data massage, the memory counter is decremented by its size and the task is discarded.

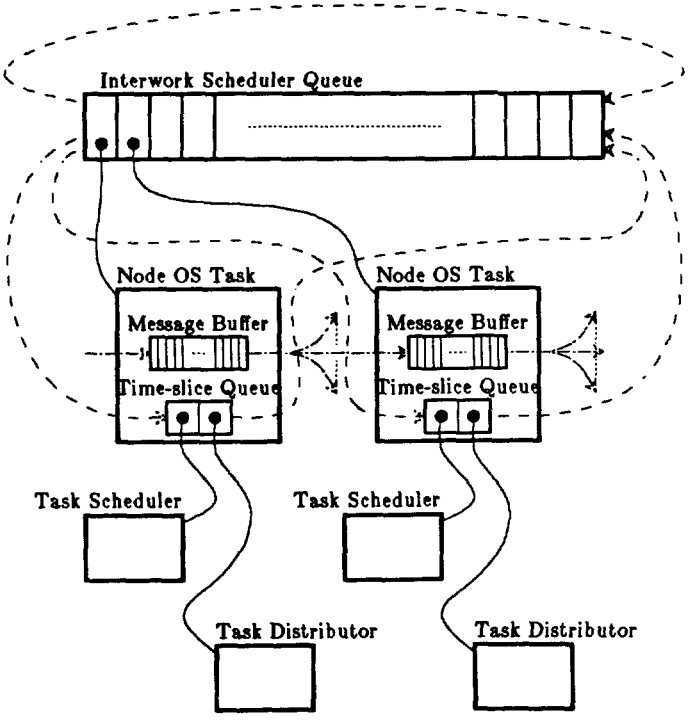

Figure 6.2 — Simulator Design. Node OS routines enter context-switch delay when both the Task Scheduler and the Task Distributor for that node have entered the local time-slice queue. When awakened by the Interwork scheduler, the NOS routine unblocks the next routine from its time-slice queue.

The simulator is written in C using Block Island Technologies' *Interwork*™ [Bai86] Concurrent Programming Toolkit. Interwork routines are created to represent the node operating system (NOS) on each processor node, the Task Scheduler on each node, and the Task Distributor on each node (see Figure 6.2). These routines are lightweight tasks, implemented as coroutines. An additional Interwork routine can be scheduled to run at regular intervals, to extract and report general performance statistics.

Time in the simulation is kept in internal units which do not correspond directly to real time. The Interwork scheduler maintains a global "clock," which is used to maintain local clocks at each NOS routine in approximate synchronization. Interwork routines are scheduled in order of increasing "wake-up" time, and the global clock is advanced only when all routines are "sleeping." Time required for various Scheduler and Distributor operations, context switching, and execution of tasks is simulated by causing the Interwork routines to sleep for an appropriate interval. The local clock is updated at this point, so that local and global clocks synchronize at the wake-up time, but the local clocks are rarely in sync with one another.

The experiments performed assumed a hypercube connectivity among processors. Parameters of the simulation were taken from the Intel iPSC/1 hypercube multiprocessor. The iPSC communication network is duplicated as nearly as possible in the simulator, including the routing algorithm used to determine the path that a message traverses from one node to another. Times to perform various functions such as message passing and context switching are also proportional to estimates of actual times for the same operations on the iPSC/1.

6.2.1. Decision Algorithm

Before discussing results, a description of the decision algorithm used at each node to accept or reject packets may be helpful. The information policy used is to exchange pressure information among directly connected neighbors only. Two pieces of information are sent to each neighbor by a Task Distributor: an average of the pressures it receives from all its neighbors and from the local Task Scheduler, and a memory usage indication expressed as a percentage of its total memory size. Pressure is a measure of workload, and in these experiments is defined to be the number of ready tasks at a given Task Scheduler plus a fraction of the number of waiting tasks at that Scheduler. Waiting tasks are included to account for the memory they occupy, as explained below. The location and transfer policies are implemented by the decision algorithm; the discussion which follows describes the development and refinement of this algorithm.

Each time a new packet is received by the Packet Handler, a decision must be made to *accept* that packet, that is, to execute it locally, or to *reject* that packet and send it to another processor node. The decision to accept or reject is based on a *weight* computed for the local node and each of its neighbor. The weight is computed from the perceived pressure for the node and the distance (in *hops*) between the node and the origin of the packet. Distance is included because each packet is assumed to represent a large task, which will transmit a significant amount of data back to its origin processor.

Also added for the neighbor nodes is a constant *hop weight* multiplied by the number of Task Distributors the packet has visited, including the current one. If the local node is the same as the origin of the packet, the hop weight is balanced by the addition of a constant *launch weight* to the weight of the local node. The launch weight is a tunable parameter designed to encourage parallel execution. If one of the neighbor nodes is the same as the

origin of the packet, a constant *home weight* is added to the weight for that node to discourage packets from returning to a node that has already rejected them. For the same reason, the neighbor from which the packet was received is never considered.

Memory usage information is considered indirectly in two ways. The first is the inclusion in the pressure value of a fraction of the number of waiting tasks at a node. That fraction was chosen as 0.5 based on early observations. This is sufficient to prevent nodes with many waiting tasks but few ready ones from sending work to nodes with few waiting tasks and many ready ones. However, a node with many waiting tasks cannot continue to accept work if its memory is full. Therefore, a very high weight is assigned to any node at which the memory usage exceeds a threshold fraction. This weight is computed by multiplying a large *overflow pressure* value by the fraction of memory used. If memory occupancy is above the threshold at all nodes, this will cause the node with the least memory occupancy to have the lowest weight.

Once all computations have been done, the node with the lowest weight is selected. If this is the local node, the packet is accepted; otherwise, it is rejected and sent on to the indicated node. The weight comparison cycles through the list of neighbors, always beginning with the neighbor *after* the last neighbor to have been sent a packet; in case of equivalent weights, the first node encountered having that weight is selected. The reason for this cyclic search will be explained in the discussion of the simulation results. The complete algorithm is given in Figure 6.3.

6.3. Evaluation and Tuning of Scheduler Policies

The goals of these first experiments were to study the interactions of the diffusion scheduling policies and the simulated architecture, and to adjust the algorithm as necessary to distribute tasks in a uniform manner. Delay (latency) in exchanging load information

```
IF memory percentage < maximum memory percentage THEN
        local weight = local pressure + distance(local node, packet origin)
        IF local node == packet origin THEN
                local weight = local weight + home weight + launch weight
        FI
ELSE
        local weight = overflow pressure * fraction of memory used
FI

FOR each neighbor node DO
        IF neighbor memory percentage < maximum memory percentage THEN
                neighbor weight = neighbor pressure +
                                  distance(neighbor node, packet origin) +
                                  hop weight
                IF neighbor node == packet origin THEN
                        neighbor weight = neighbor weight + home weight
                FI
        ELSE
                neighbor weight = overflow pressure * neighbor memory percentage
        FI
DONE
```

Figure 6.3 — Pseudo-code showing weight computations of the decision algorithm for local and neighbor nodes.

among different processor nodes was expected to produce some imbalance, though how that imbalance would appear was a subject of study. However, what had not been anticipated was the significant effect of latency in exchanging information between the components of the architecture within the same processor node. This proved serious enough that the architecture was modified slightly to relieve the problem.

The experiments simulated the architecture components as loaded on each node of a four-dimensional hypercube (16 nodes). Simulations were performed using packets with a range of sizes, evaluation times and branching factors. Certain simulations were also examined via the monitor task to observe distribution of load and performance of individual nodes over the course of the simulation.

6.3.1. Initial Behavioral Anomalies

The first simulations were run with a very simple decision algorithm that considered only the numbers of ready and waiting tasks, and which did not use the cyclic search of neighbors' weights. Load messages were sent by the Task Scheduler once per time slice, and were broadcast by the Task Distributor at most once per slice and at least whenever a change was detected. Furthermore, only the Task Distributor would surrender the processor if it ran out of work before the end of its time slice. The Task Scheduler would busy-wait, checking for messages or for tasks to become ready as the result of receiving a data message. These simulations used a small number of packets (tasks) with fixed evaluation times and fixed numbers of child tasks, and analysis of the data concentrated on studying the load in the early stages of distribution.

The results showed a tendency for a few nodes three to four hops from the root (node zero) to accept a disproportionate number of packets. Closer examination of the pattern in which packets were sent out and accepted revealed two unexpected behaviors:

1) each node sent several packets in succession to a particular neighbor, and
2) when these "bursts" of packets arrived at the neighbor, all packets were accepted.

The expanding tree structure of the computation graph and the connectivity of the hypercube combined with these behaviors to "focus" packets from many nodes on a few others, which then accepted most of those packets. An example of focusing in shown in Figure 6.4. Both behaviors are related to latency in transmission of load information; the first is due to latency in communicating load among nodes, and the second is due to latency in communicating load between the Task Scheduler and the Task Distributor on the same node.

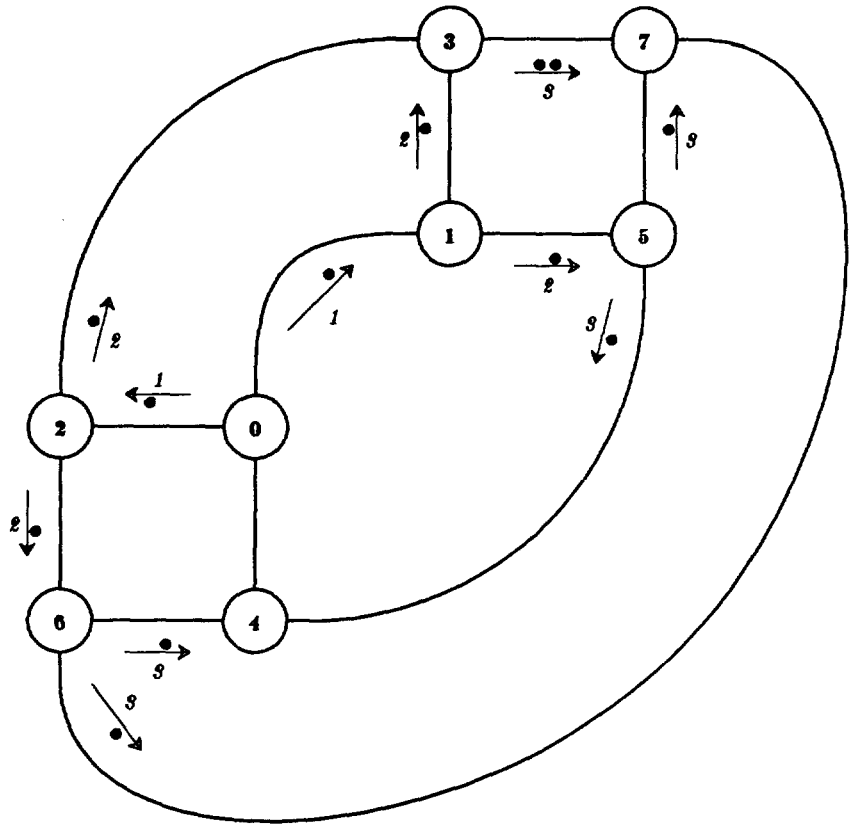

Figure 6.4 — Focusing of packets after introduction of cyclic selection, for a binary process tree on an 8-node cube. Each dot represents a packet, and arrows are numbered with the depth of the process tree at the time the packet is created. Without cyclic selection, this effect is magnified, because each processor directs both branches at each level of the process tree to the *same* neighbor processor.

Three changes were made to correct this problem. The first was to add the cyclic search of neighbors to the decision algorithm (see Figure 6.5). This causes packets to be distributed more evenly in the beginning of the computation, when little or no load information is available and the rate of change in load is greatest.

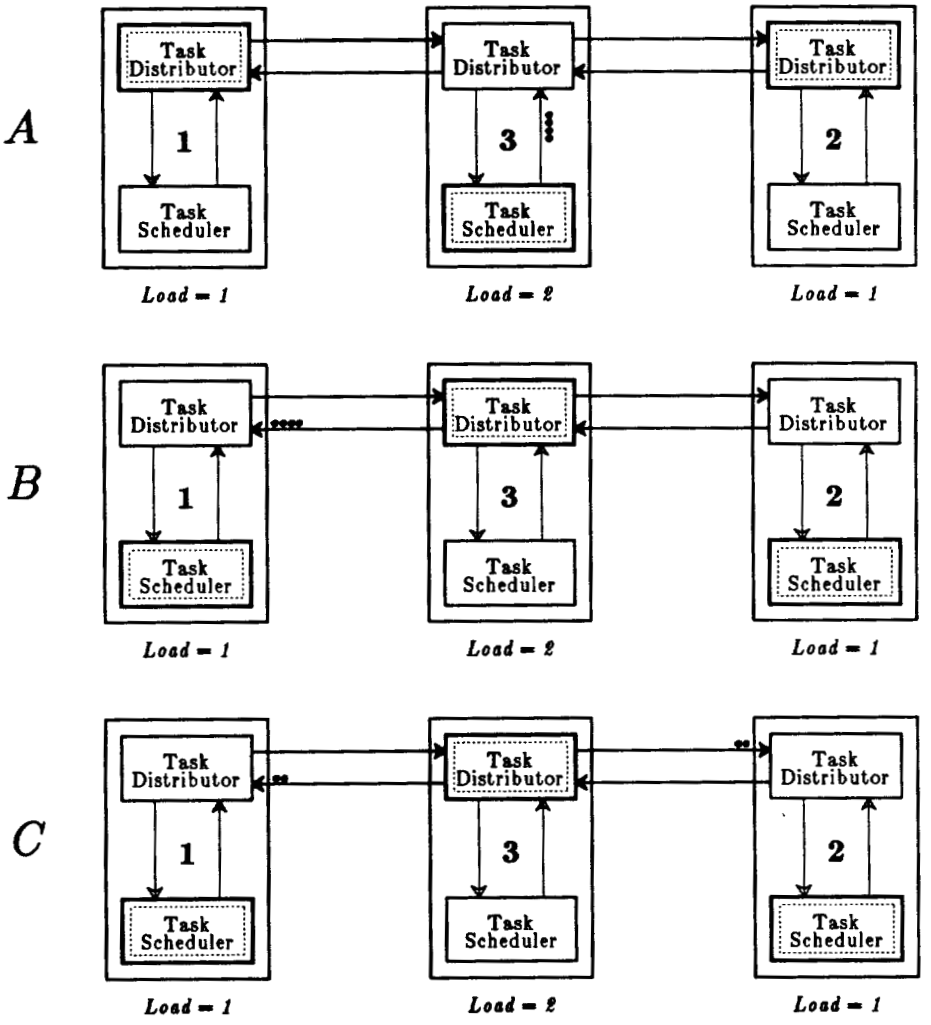

Figure 6.5 — Distribution before and after cyclic selection. Diagram *A* shows four packets being passed from the Scheduler to the Distributor at processor 3, with neighboring processors 1 and 2 having equal loads. Diagram *B* shows the distribution of those packets without cyclic selection. Diagram *C* shows the more balanced distribution with cyclic selection.

The second and third changes were an attempt to reduce latency in packet processing and in load communications. Communication between processes on the same node is limited mainly by the time that each process is allocated as its share of the processor, whereas com-

munication among nodes is limited primarily by message transfer time. Before enough packets have been generated to provide work for all nodes, the Task Schedulers have nothing to do, and are essentially busy-waiting for their entire time slice. This contributes to mass acceptance by increasing the delay between runs of the Task Distributor, allowing more unprocessed packets to accumulate. To reduce local communication latency (in this case, latency in processing and delivering packets), the Task Scheduler was caused to surrender the processor when it had nothing to do, rather than busy-waiting. The Task Distributor also was modified to make a load broadcast whenever a significant number of packets had been accepted, thus reducing communication latency among Task Distributors on different nodes.

These changes did not entirely solve the "burst acceptance" problem, but reduced it so much that we at first felt them to be sufficient. Later, however, simulations using packets with variable run times and variable numbers of children revealed that certain tree structures could still produce large jumps in acceptances (and correspondingly, in load) at some nodes. Again, the problem was local communication latency. Several packets could arrive during the Task Distributor time slice, when the Task Scheduler was unable to send up-to-date load information. The Task Distributor used the same information for each acceptance decision, even though that information became increasingly inaccurate with each acceptance. Therefore, the Task Distributor was modified to estimate the increase in local pressure when a packet was accepted, anticipating the consequent increase in load that would be reported by the Task Scheduler. Figure 6.6 shows the effects of this change. Each ready task is represented as one unit of pressure, so the Task Distributor increments its record of the local pressure each time it accepts a task.

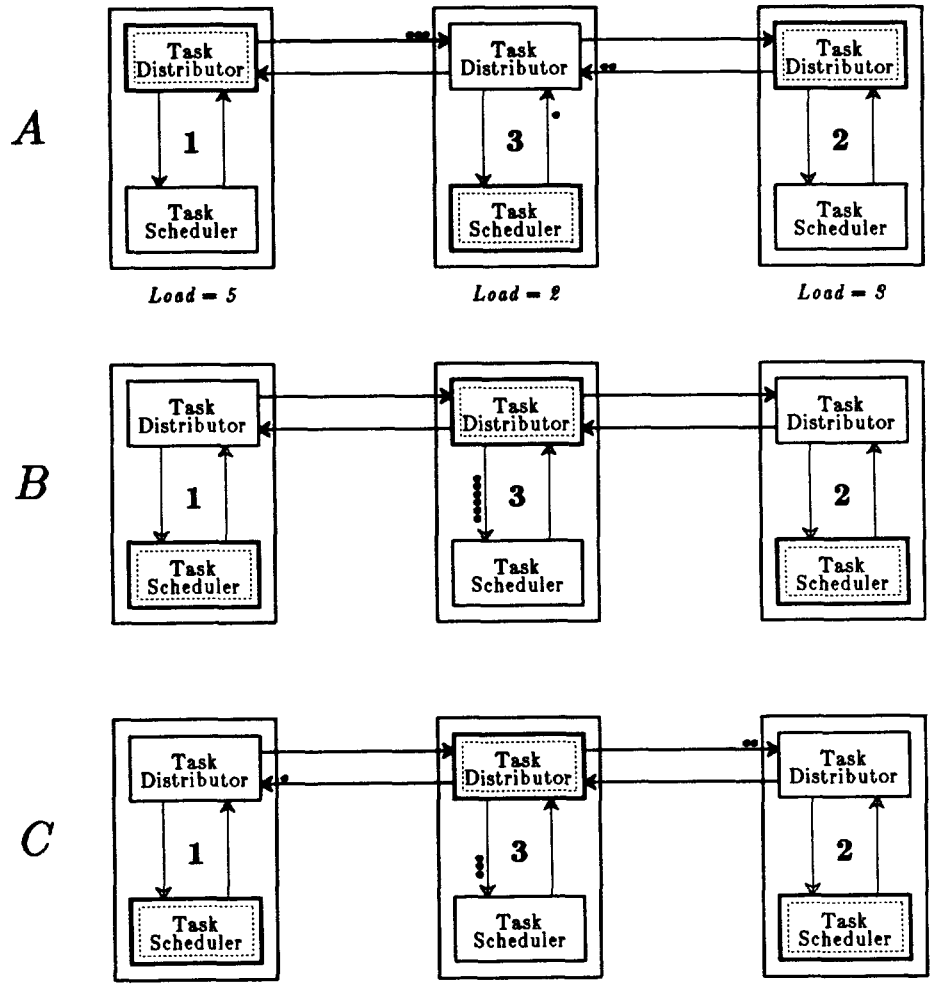

Figure 6.6 — Mass Acceptance. Diagram *A* shows the Distributor at lightly loaded processor 3 receiving six packets. Diagram B shows the mass acceptance problem, as Distributor 3 accepts all the packets. Diagram C shows a possible distribution if Distributor 3 estimates the load change from each acceptance (actual distribution depends on the order in which packets from each source are received).

6.3.2. Evolution of the Decision Algorithm

With initial distribution problems at least partially solved, simulations were run with a significantly greater number of packets. Additional packets are generated by increasing

either of the depth of the simulated process tree, the number of child packets created by each task, or both. Data from experiments with short packet evaluation times relative to the time slice of the run-time system processes showed that, although the load was reasonably well distributed among the nodes, packets were being rejected by an average of more than three Task Distributors before finally being accepted. At least one rejection per packet was expected, because the Task Distributors were given the launch and home weight biases against local packets, but three seemed excessive. Also, the packets were lodging at nodes only one or two hops away from the nodes which created them.

This seemed to indicate that the packets were "orbiting" their origin node, failing to move to more distant, possibly less loaded nodes because of the "pull" exerted by the distance factor in the decision algorithm (see Figure 6.7). However, experiments using packets with longer evaluation times did not show multiple rejections of packets. This leads us to believe that in computations where tasks have short individual run times, the repeated rejections are a result of the high rate of new packet creation, which greatly exceeds the rate at which load broadcasts can be made.

Interestingly, multiple rejections are related to the mass acceptance problem discussed in the last section. At the beginning of the computation, a Task Distributor tends to accept too many packets because the local Task Scheduler cannot send load updates during the Task Distributor's time slice. Later in the computation, when local load messages *have* been transmitted, communication latency *among* nodes becomes dominant. Every Task Distributor perceives the local load to be higher than that at neighbor nodes, so most packets are rejected. The orbiting effect is secondary, though the distance factor can have an effect when loads at different nodes are nearly identical.

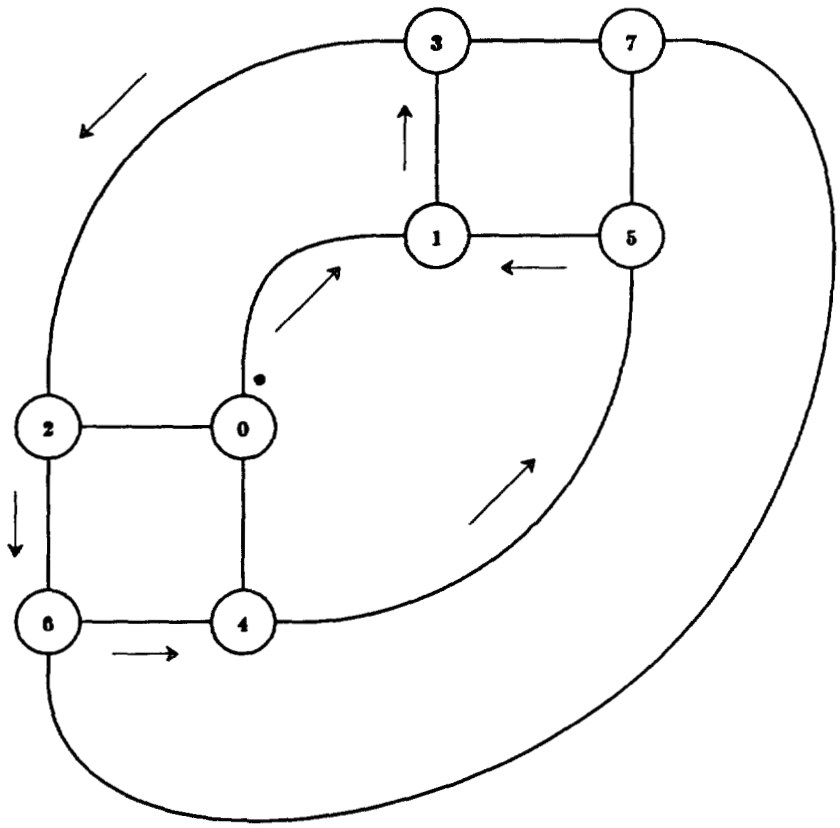

Figure 6.7 — Orbit of radius ≤ 2 around Node 0 in an 8-node cube. Distributors at processors 3 and 5 may never send the packet to processor 7, even if it has a load lighter than processors 1 and 2, because the distance from origin processor 0 is greatest at processor 7. The *home weight* bias will discourage the return of the packet to processor 0, so it orbits until the *hop weight* becomes excessive.

At present, there seems to be no way to avoid multiple rejections except to ensure that the packet creation rate is kept low relative to the rate of load broadcasts. In an actual computation, the creation rate is dependent on the structure of the compiled program, so it may be necessary to adjust the run-time system to improve the performance of some programs. Nevertheless, to ensure that packets were not being rejected solely on the basis

of distance from their origin node, the ***hop weight*** penalty was increased considerably, and the simulations rerun. The hop weight represents the time and effort required to send a packet to another node and to make the acceptance decision there, which is several times as great as the effort required to send a data message one additional hop. For the packets with short run times, the average number of rejections fell to just over two when the hop weight was increased, but the number of rejections was essentially unchanged for packets with long evaluation times. This is consistent with our hypothesis that a high rate of new packet creation is the primary cause of multiple rejections.

6.3.3. Final Refinements

The difficulties encountered in balancing the load in the early and middle stages of the computation stem from delays in the exchange of load information. As the computation nears completion, new tasks are no longer being created. At this stage, it is less important to balance the load than it is to complete the existing tasks. Recall that in our early simulations, only the Task Distributor was programmed to surrender the processor if there was no work to be done in its time slice; the Task Scheduler would use its entire slice whether or not it had any ready tasks to run. The better load distribution that resulted from correction of the mass acceptance and multiple rejection problems, which included the elimination of busy-waiting from the Task Scheduler, was expected to improve the overall performance of the system; instead, we saw a slight degradation.

Examination of the data for individual nodes indicated that the performance loss was due to a few nodes that took significantly longer to complete their work than did the rest of the system. A comparison of the change in load over time for one such "slow" node with a "normal" node showed that, as the computation neared completion, the loads fell quickly and at very nearly the same rate for both *except* that the normal node finished its last

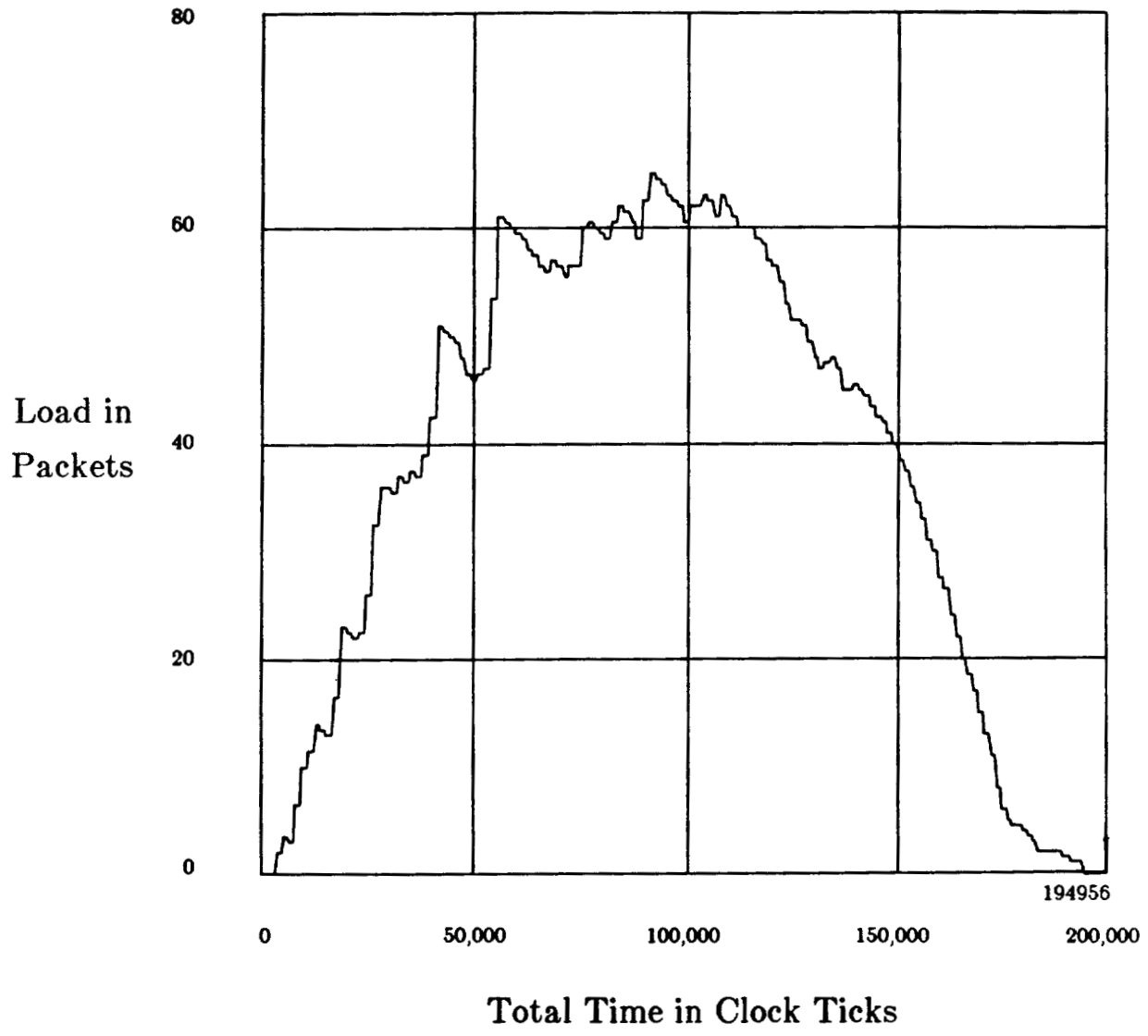

Figure 6.8 — Load *vs.* Time for a "slow" Node (Node 14)

ready task when the slow node still had a few tasks to run. Both nodes still had a number of tasks waiting for data. Those last few ready tasks on the slow node took considerably longer to complete than an average task, *i.e.*, at that point the rate of change in load for the slow node dropped sharply (Figure 6.8).

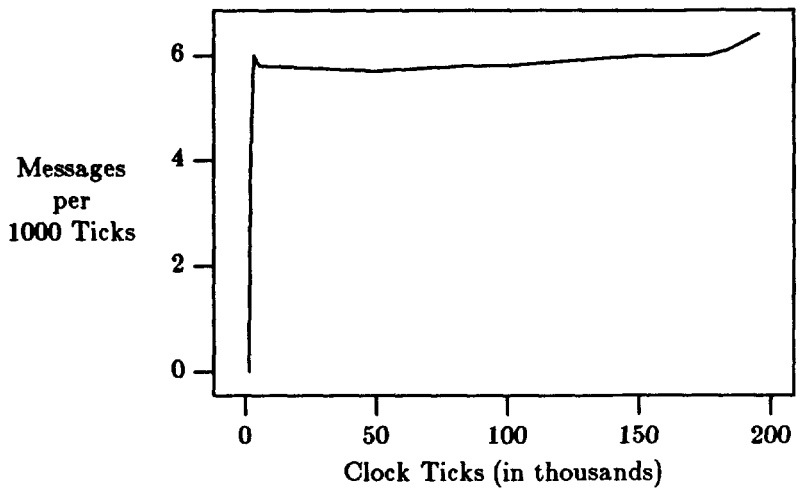

Figure 6.9 — Messages Received *vs.* Time at "slow" Node 14, with communications expensive in comparison to task evaluation time. Communication time for this figure is comparable to iPSC/1 timings.

The only reasonable explanation for this behavior was an unexpected increase in overhead at the slow nodes. Message processing is the primary source of overhead in the system, so we looked for an increase in message traffic. As was to be expected, there was a slight increase in the number of data messages from the completing tasks to their parents, but the timing of this increase did not correspond to that of the change in task run times. Instead, we found an increase in the number of load messages from neighboring nodes being received at the slow nodes, which corresponded precisely to the slowdown in task completions (Figure 6.9). An increase from 6 to 6.4 messages per 1000 clock ticks does not seem significant, but a single node represents only one-sixteenth of the total message traffic in the system. Each load message received by a Distributor could result in that Distributor making a broadcast of its own. Furthermore, the increase in load messages appeared to be bounded only by the

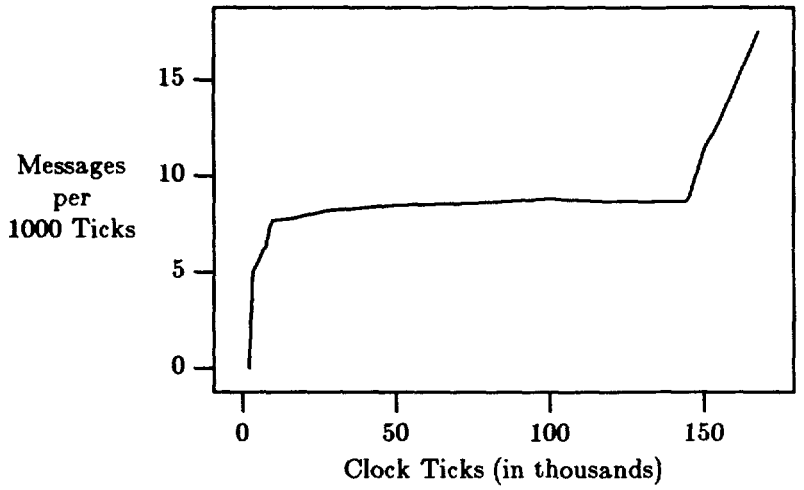

Figure 6.10 — Messages Received *vs.* Time at "slow" Node 14, with communications less costly in comparison to task evaluation time. Communication time in this figure is approximately half that of Figure 6.9, with average task evaluation time unchanged.

relatively slow communication network. A much greater increase was seen in tests where the communication time was less (Figure 6.10).

The increase in load messages was a result of the policy, implemented in the Task Distributor, of making a load broadcast whenever a change in the local load was perceived. At the normal nodes, the Task Schedulers received data messages which allowed them to complete some waiting tasks; this resulted in a small change in load, which was immediately communicated to the Task Distributors. Then, finding no ready tasks, the Task Schedulers surrendered the processor; the Task Distributors, taking over, would detect the change in load, make a broadcast, and immediately switch back to the Task Schedulers. This cycle repeated so rapidly that the slow nodes, where work was still being done by the Task Schedulers, were flooded with load messages. Their overhead soared, and the performance

of the whole system was affected.

This situation presented an interesting problem. It is obviously necessary to restrict the rate of load broadcasts in the later portion of the computation to avoid flooding nodes that still have work to do. However, as the mass acceptance problem demonstrated, it is also necessary to make broadcasts fairly frequently in the early stages. Furthermore, since there is no way to determine, at the individual nodes, how far the computation has progressed, any scheme used to regulate broadcasts must be independent of the general state of the computation, though it can be modified to respond to short-term and local variations.

To find an answer, we considered the maximum rate of load broadcasts in that part of the computation when all nodes still have ready tasks to run, but the overall load is decreasing and very few new packets are being created. There, load broadcasts are limited because the Task Schedulers are using their full time slices; only when some Schedulers ran out of ready tasks did the number of load broadcasts become excessive. The length of the time slice thus seems to be a natural bound on the frequency of load broadcasts when the overall load is decreasing, although more frequent load broadcasts are needed when the load is increasing.

The solution we adopted is to record the time that a load broadcast is made, and not to send another until a full time slice has expired (regardless of how many context switches occur in that period). However, if a Task Distributor receives several packets in quick succession, it can override the time limit and make load broadcasts more often. This provides the higher frequency of broadcasts needed as the computation starts up, but controls the rate of broadcasts when quick completion of existing tasks becomes more important than scheduling new work. Once these changes had been made, message traffic became uniform (Figure 6.11) and the "tailing off" at slow nodes disappeared (Figure 6.12).

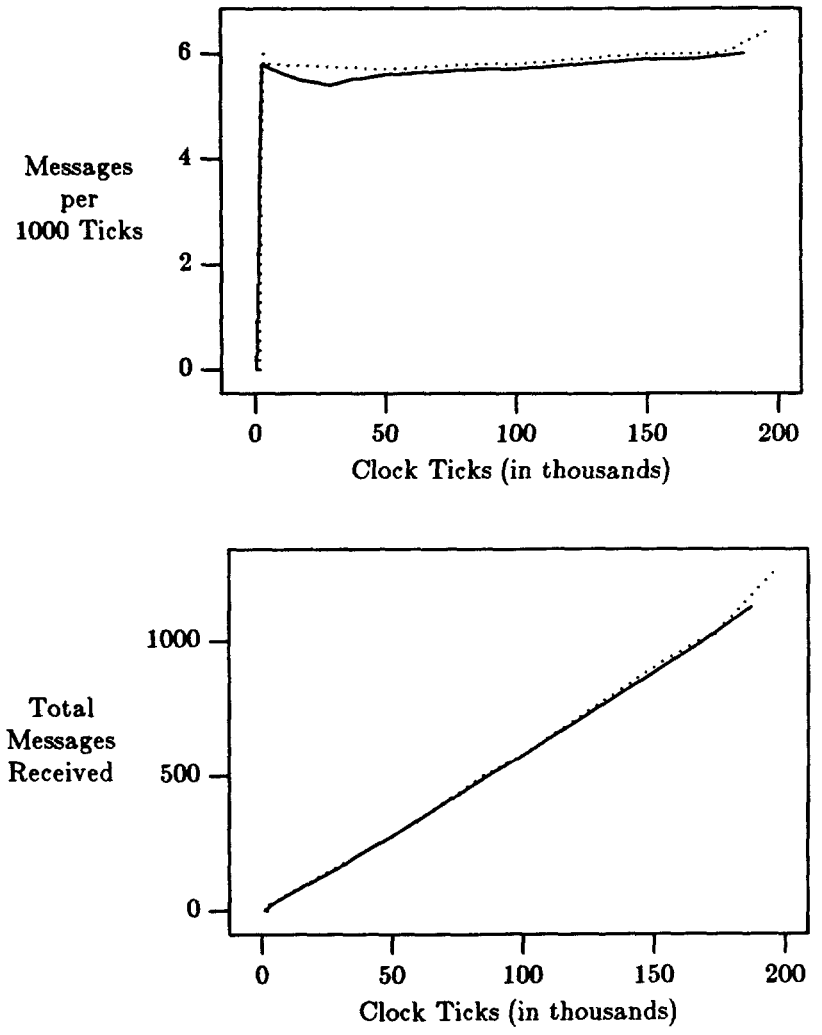

Figure 6.11 — Messages Received *vs.* Time at formerly "slow" Node 14 after broadcast limit. The dotted lines show the previous behavior.

Also note that if the Task Distributor is to estimate load changes based on accepted packets, as described earlier, some form of broadcast limitation must be in effect. Otherwise, the same explosion of load messages will occur at the beginning of the computation

and continue until enough packets have been generated for nearly all nodes to have some work.

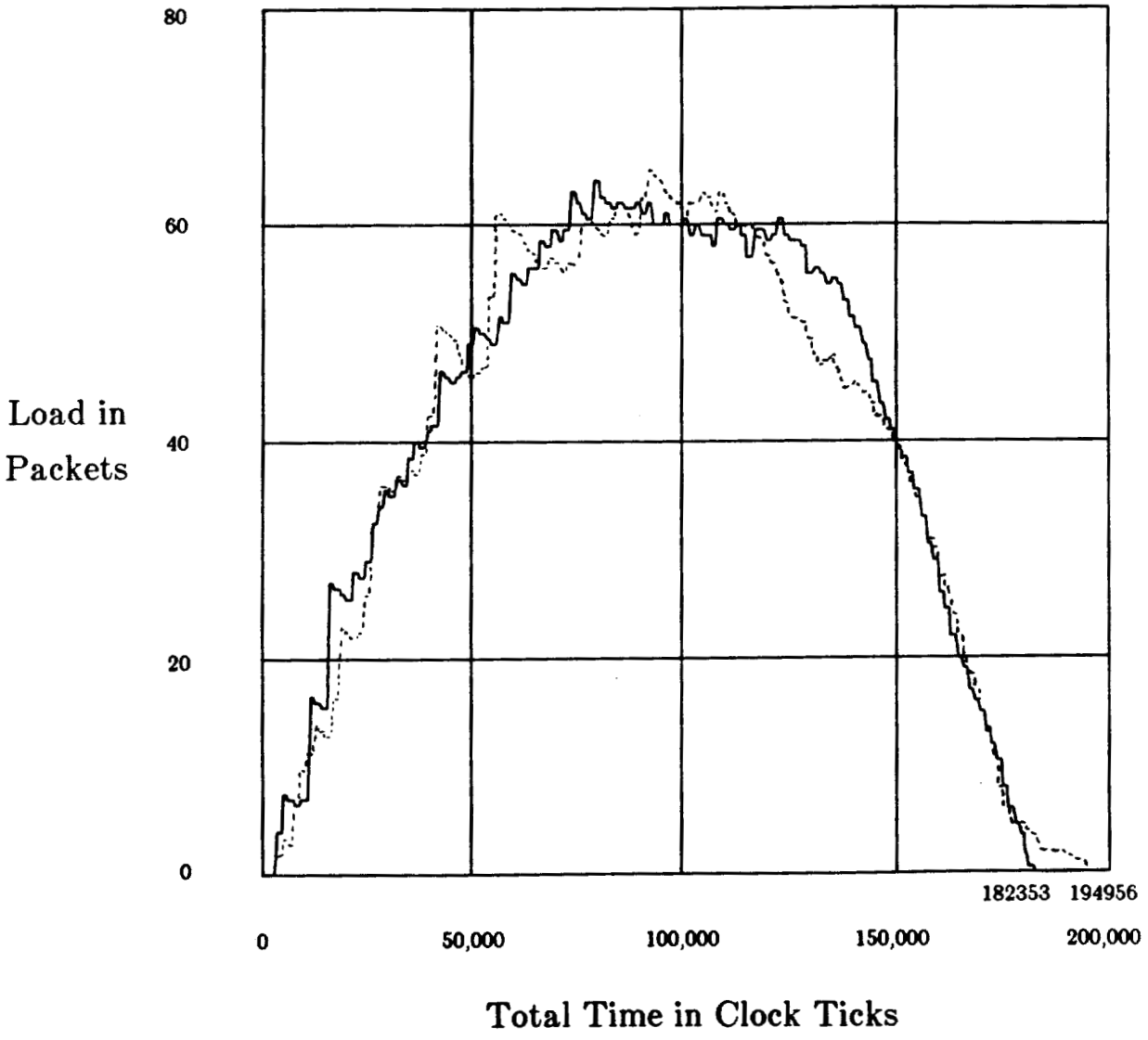

Figure 6.12 — Load *vs.* Time at formerly "slow" Node 14 after broadcast limit. The dotted line shows previous behavior. The last ready task is completed 6.5% faster.

6.4. Results of Performance Evaluations

Simulations discussed in this section were run with fixed packet memory usage, and with the number of subtasks per packet determined by its level in the computation tree. The only variable was the evaluation time of a packet, which was fixed for a given simulation but varied among simulations. The length of the time-slice for the Task Scheduler and Task Distributor on each node was set at 500 internal clock ticks, corresponding to a 50 millisecond time-slice on the iPSC, so one tick is approximately equivalent to 100 microseconds.

Performance of the system as a whole was evaluated in terms of speedup as compared to a uniprocessor system (both with and without considering overheads for communication and task scheduling). Figure 6.13 shows speedups from a series of simulations using a four-dimensional hypercube, with per-packet evaluation time varying from 25 to 2000 clock ticks. Evaluation time here refers to the minimum time to complete a task, not including overhead. Speedup is computed by

$$\frac{\textit{Sequential execution time}}{\textit{Parallel execution time}}$$

which is represented here in two ways. The first is

$$\frac{\textit{Sum of task evaluation times}}{\textit{Time to complete seed task}}$$

where the time to complete the seed task is the time from the start of the simulation until the final "result" data message is received by the root task. The second representation defines the sequential time as the time to complete the seed task on a one-node (zero-dimension) network. The latter includes all overheads of the run-time system and diffusion scheduling.

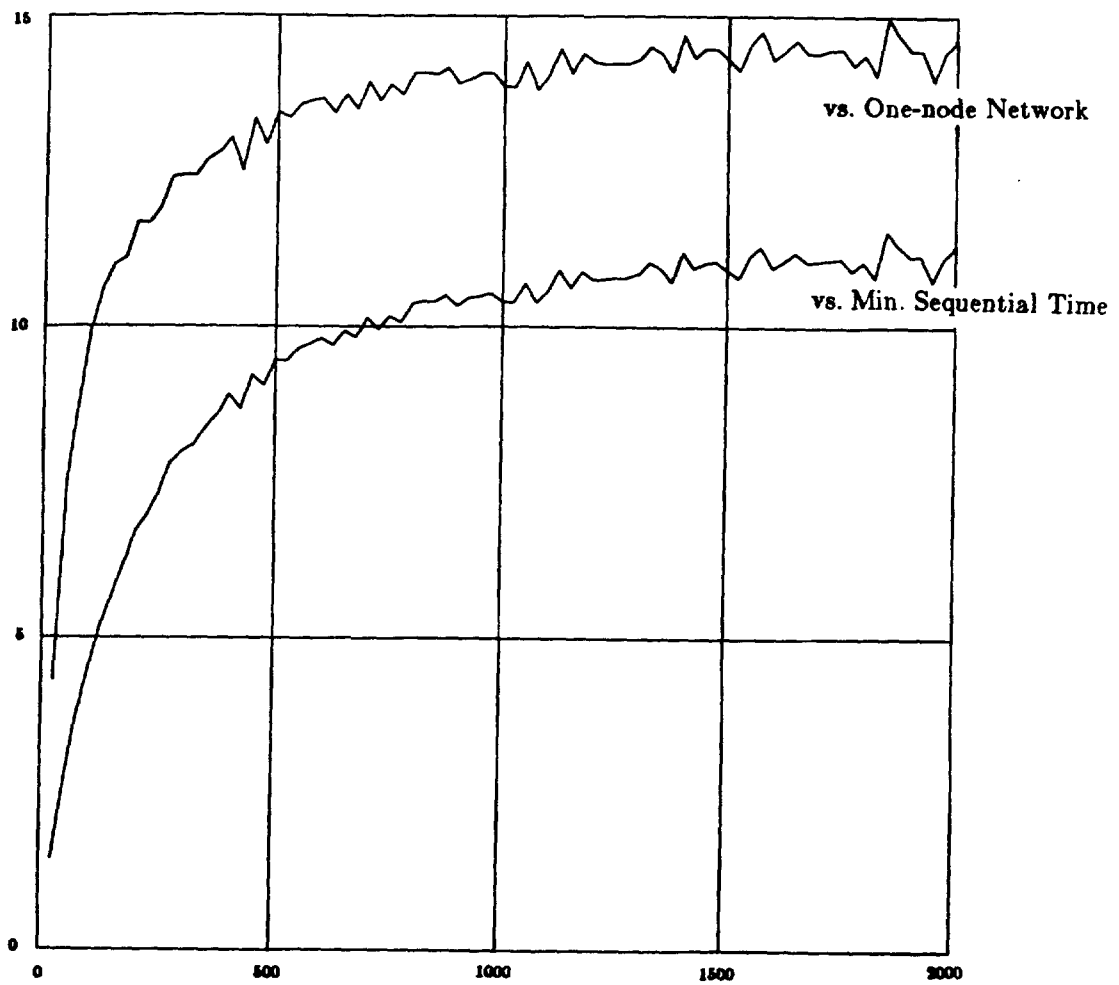

Figure 6.13 — Speedup *vs.* Per-Packet Evaluation Time for a 4-dimensional (16 node) hypercube.

For computations in which individual tasks have evaluation times of 1500 ticks or more, speedup averages 11 times the minimum sequential evaluation time, which translates to an efficiency of 68 percent of the maximum possible speedup. Efficiency is computed by

$$\frac{\textit{Observed Speedup}}{\textit{Number of Processors}}$$

Speedup averages just over 14 when compared to the time to complete on a one-node network, which thus includes wait time and Scheduler overheads in the sequential time, for 88 percent efficiency. This suggests that the system is spending about 12 percent of its time in the Task Distributor, and about 20 percent in other overheads or waiting.

Figure 6.14 shows fractional overhead as a function of per-packet evaluation time. Fractional overhead is computed by

$$\frac{\textit{Observed time} - \textit{Optimal time}}{\textit{Observed time}}$$

where the optimal time is given by

$$\frac{\textit{Sum of task evaluation times}}{\textit{Number of processors}}$$

and the observed time is the time to complete the seed task, as before. The significantly greater speedups with longer per-packet evaluation times are a result of reduced overhead. Message processing is the only major source of overhead, and several factors contribute to a larger message workload (relative to the packet evaluation time) when evaluation times are short. The most obvious of these is that the message transfer time is a larger fraction of the evaluation time, but the number of messages that must be handled per unit time is also greater.

Short evaluation times allow more tasks to be started and completed per time slice of each Task Scheduler, which in turn produces more new packets and more data messages per time slice. A high rate of packet creation, as noted earlier, results in multiple rejections of packets; this increases the number of messages that must be handled to get each task started. Furthermore, in response to the rapid arrival of new work, the Task Distributors make load broadcasts more frequently to keep load information as current as possible, and the Task Schedulers send more local load messages as well.

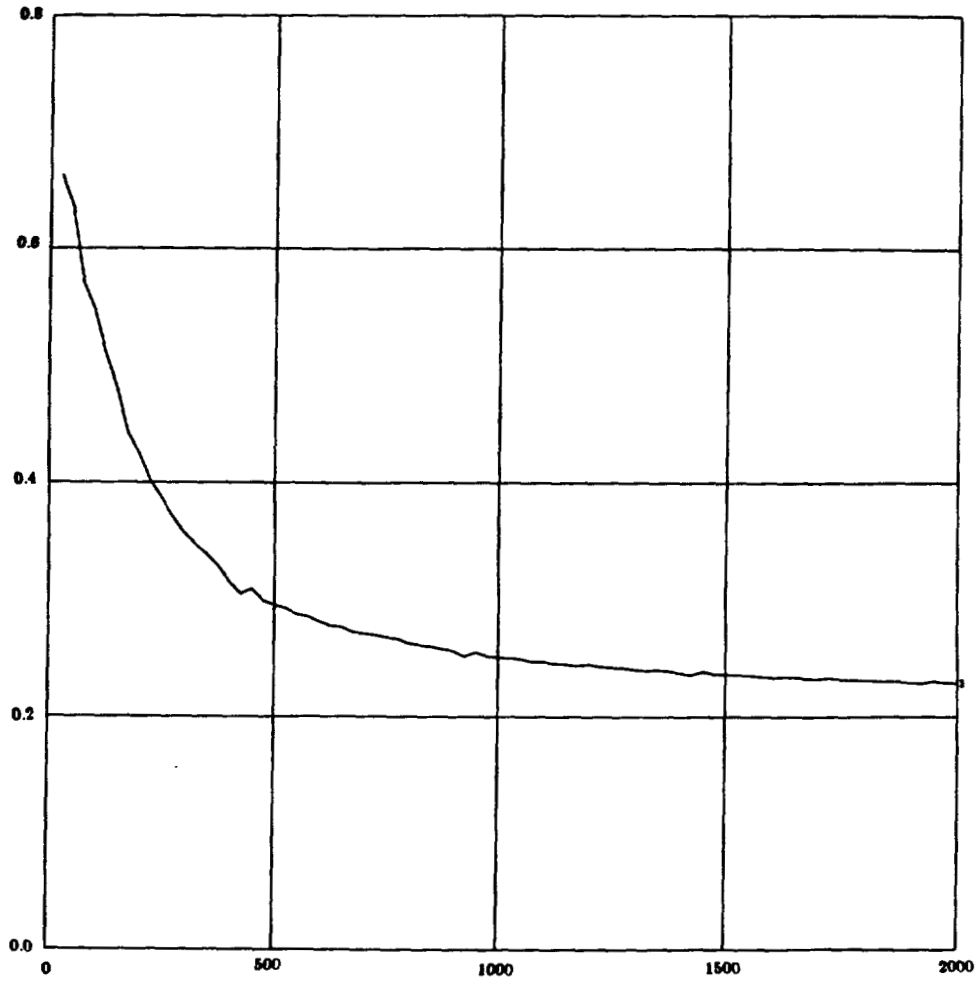

Figure 6.14 — Overhead as Fraction of Per-Packet Evaluation Time for 4-dimensional (16 node) Hypercube

As evaluation times lengthen, all of these effects begin to disappear. One packet is enough to keep a Task Scheduler busy for one or more time slices, so new packets, data, and local load messages are sent at a more leisurely rate. The Task Distributors respond by making fewer broadcasts, and multiple rejections are no longer a problem. However, overhead for context switches (from the Task Scheduler to the Task Distributor and back) may

actually increase slightly, because there are more Scheduler/Distributor alternations during the execution of each task.

6.4.1. Comparison with Alfalfa Results

Diffusion scheduling experiments performed by Goldberg [GoH87, Gol88] for the Alfalfa system also show encouraging results for this type of parallel evaluation. A direct comparison to Goldberg's results is impossible, due both to differences in the form of the experiments and to differences in the diffusion models used, but a number of similarities can be pointed out.

The Alfalfa system was tested on an iPSC/1 hypercube, the same type of machine as that from which our simulation parameters were drawn. Goldberg studied several information and transfer policies, including non-communicating algorithms. We studied only a communicating policy, so this comparison will focus on the communicating algorithms.

One significant difference between the diffusion model in this research and that of the Alfalfa system is in restrictions placed on the location policy. In the Alfalfa diffusion model, the processor at which a task will execute is determined solely by application of the algorithm at the processor where the task originates. Once a processor has been selected, that processor is required to accept the task. By contrast, the model used in our research applies the decision algorithm at each node visited by a task. The Alfalfa model avoids the problems of multiple rejections we have described, but it severely limits the system's ability to react to rapid creation of tasks.

Goldberg studied two communicating algorithms, referred to as *Simple Communicating Diffusion* and *Dependent Communicating Diffusion*. Both use measures of the change in load to determine frequency of load broadcasts, and both select a neighbor processor to receive work if its load is less than the local load by some constant amount. They differ primarily

in that the *Dependent* algorithm factors the dimension of the network into the decisions.

The information policy used in our experiments is most similar to Goldberg's *Simple* algorithm. However, our practice of exchanging average load values for the neighborhood accounts for network dimension in a way that neither of Goldberg's policies can. Goldberg's policies do not include information from neighbors in the pressure reported by each processor node, so local maxima or minima can have a greater effect on the distribution of tasks. The *Simple* algorithm limits load broadcasts by establishing a threshold M which the load must exceed before broadcasts are considered. This appears to have an effect similar to our time-based broadcast limits. The threshold has the advantage of being independent of implementation considerations such as the Distributor/Scheduler dichotomy that led to our choice. However, it would appear to have the effect of sequentializing the final stages of any computation, by concealing load changes.

Both the *Simple* and *Dependent* algorithms avoid load broadcasts unless the local load changes by more than a factor of two. This understandably dominates the M threshold most of the time, because the total load is much higher than M for most of the run time of a computation. Our experiments also require a minimum percentage change before a broadcast is made, but percentages higher than 10% were not considered. This is because the possibility of multiple rejections makes our system much more sensitive to inaccurate pressure information.

The transfer policies in all of the Alfalfa experiments differ from ours in one very important respect. The Alfalfa algorithms tend to prefer to retain tasks locally until the local pressure is fairly high. Our algorithm prefers to move work to other processors as quickly as possible. This difference may mean that Alfalfa will perform better for small programs because the expenses of communication will not be incurred until the number of concurrent

tasks exceeds the M threshold. Our experiments with this model have assumed that the complexity of each task is always sufficient to justify the communication required for parallel execution, so we have not simulated the small process trees where this would become noticeable.

Five sample programs were used in the Alfalfa experiments: *Parallel Factorial*, *Eight Queens* (actually seven queens due to memory limitations), *Adaptive Quadrature*, *Matrix Multiplication*, and *Quicksort*. Of these, only the first three decompose in a tree structure comparable to our simulated process trees. Evaluation times for tasks in these programs are unavailable, but *Parallel Factorial* and *Adaptive Quadrature* have relatively small tasks as compared to *Queens*. *Factorial* generated 3,998 tasks, *Quadrature* 5,567 tasks, and *Queens* 29,682 tasks. Our final experiments used randomly branching trees of on the order of 2,000 tasks, so a comparison with these programs is reasonable.

Although Goldberg's results varied widely for suboptimal parameter choices of each of the communicating algorithms, the best-case results for each sample program were very similar. The communicating algorithms also performed at least as well as the non-communicating algorithms in the best cases for all the programs. For 16 processors, Alfalfa achieved speedups of approximately 4 to 4.5 for *Parallel Factorial*, 3.5 to 4 for *Adaptive Quadrature*, and 7 to 7.5 for *Queens*. These figures are based on analysis of graphs presented in Goldberg's thesis [Gol88]; exact data are unavailable. The sequential time used to compute speedup is the completion time on a one-node network.

The observed speedups for the Alfalfa system correspond to our results for tasks with less than 50 milliseconds (500 ticks) execution time. This seems reasonable for *Parallel Factorial* and *Adaptive Quadrature*, because those are tree-structured problems in which each task performs only a few arithmetic operations. However, the granularity of *Queens* is more

difficult to determine. The parallel formulation of the *Queens* algorithm uses a number of Alfalfa operations not used in either of the other programs, so an estimate based on code comparisons is not sufficient. It may be the case that tasks in *Queens* are not large enough to reach the 100 or more millisecond times that showed the best speedups in our experiments. It is also possible that the very large number of tasks generated by *Queens* introduces overhead that our simulations did not duplicate.

6.5. Summary

These experiments have provided insight into the problems encountered in developing an effective diffusion scheduling algorithm. Related but subtly different scheduling problems were found in different stages of a computation, and in each case reasons were suggested for the problem behavior and solutions were explored. Finally, results in terms of speedups were presented for a set of simulations with varying task evaluation times, and sources of overhead were identified to explain the increasing efficiency of the system as task evaluation times lengthen.

The problems identified include "burst acceptances" of packets (tasks) in the early stages of computation, "multiple rejections" in the middle stages, and "slow nodes" at the end. The first two are related to communication delays, and the third is caused by too much communication. Solutions include cyclic search of neighbors, to more evenly distribute load; early context switches, or *flicking*, in the Task Scheduler and Task Distributor, to reduce communication delay; estimation of load changes, to compensate for remaining delays; and limitations on load broadcast frequency, to prevent excessive communications when they are not needed.

The performance results confirm our expectations that a system using fine-grained tasks will not produce linear speedups. However, some speedup was obtained even for tasks

with short evaluation times. For the architecture studied in these simulations, it is important to keep the unit of work large enough that its run time exceeds the interval between load communications. Small units of work do not produce significant load imbalances, but do result in greater per-task overhead and, therefore, reduced performance. With longer-running tasks these overheads are amortized sufficiently that considerable speedup can be attained, even in a system with relatively slow message processing.

CHAPTER 7

Experiments in Speculative Evaluation with Priority Scheduling

7.1. The MPCR Simulator

The Massively Parallel Combinator Reducer simulator emulates the mapping onto a parallel machine of the abstract model of Chapters 3 and 4. Most of the implementation techniques discussed in those chapters were used, including reference rights garbage collection and recursive packet formation. Diffusion scheduling and the speculation heuristics described in Chapter 5 were also used.

The simulator differs in a number of ways from the diffusion scheduling simulation prototype, most of them related to the design differences in the simulated machine models. Both are designed to support multiple simulated processors for each real processor of the machine on which they are run. One difference is that the MPCR simulator implements a complete combinator reduction engine, capable of running real programs. The diffusion simulator used a randomly generated, simulated program. In addition, although the organization on a per-processor basis is very similar to the diffusion simulator, the MPCR simulator utilizes a true multiprocessor machine. Each real processor runs independently, synchronized only by the constraints of message-passing among the simulated processors. This introduces nondeterminacy not present in any of the earlier diffusion scheduling simulations, and therefore provides a more accurate picture of the true behavior of the simulated machine.

Mapping of simulated processors to real processors is done statically, because the number of simulated processors is known in advance. The overhead of supporting these simulated processors is considerable, but is intended to allow simulation of massively parallel machines on the somewhat less parallel architectures that are currently available. Unfortunately, memory limitations and the time required for each run made it impossible to perform simulations on the scale that was originally anticipated. Experiments using 1024 or 2048 processors were planned, but the largest network that it was possible to simulate was 256 processors.

Each simulated processor is represented by a data structure and a set of coroutines. The data structure for each simulated processor contains:

- A local copy of a system-wide clock used to synchronize operations among the simulated processors.
- A counter to track the memory usage of the simulated processor.
- A table of *channel identifiers* describing the connections of the simulated processor to its neighbors.
- The message buffers used to pass messages between the primary CPU and the communication channels.
- The statically allocated structures used by the run-time system, such as the ready queue and pressure tables.
- The operation counts and timings collected in the course of the simulation.

One coroutine simulates the primary CPU, which runs the MPCR run-time system. This run-time system is described in more detail in Chapter 8. The other coroutines act as message transmitters to simulate the communication channels linking the CPU with its neigh-

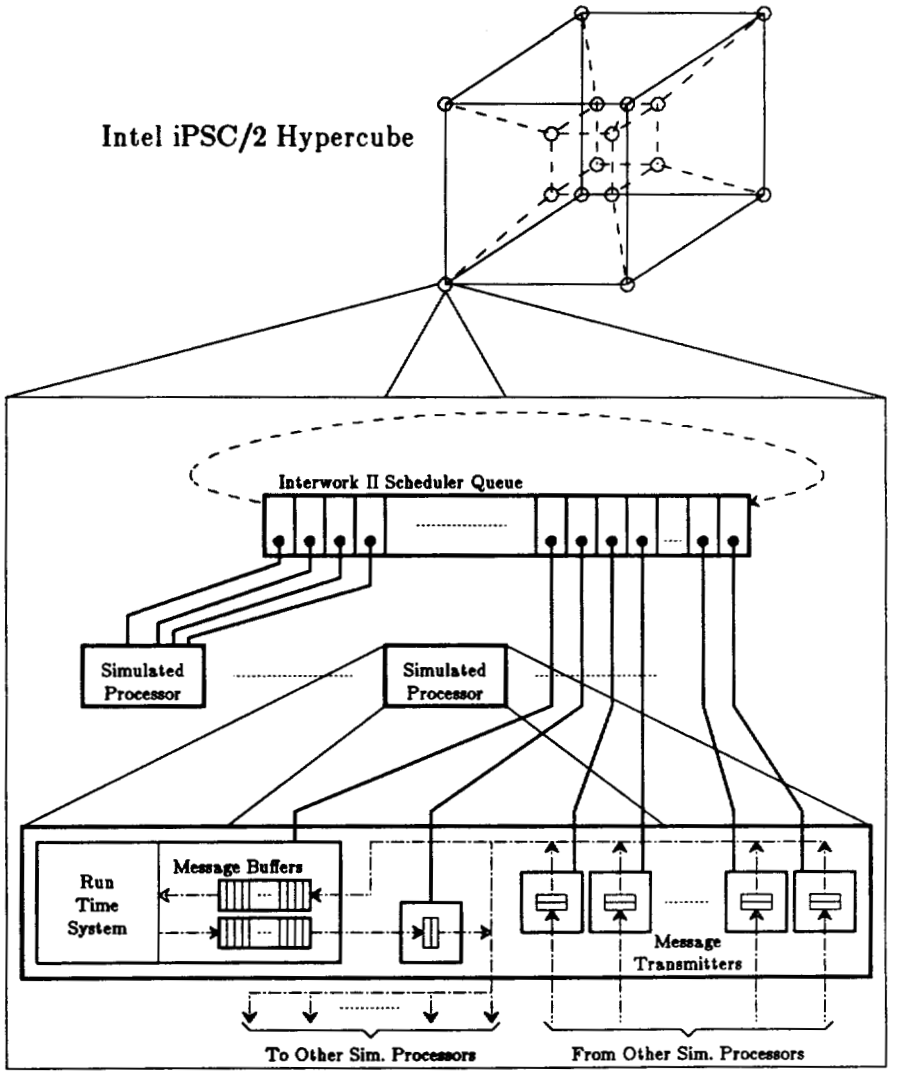

Figure 7.1 — MPCR Simulator Design. Each simulated processor consists of several Interwork II coroutines, representing the primary CPU and message passing coprocessors.

boring processors. These channels are assumed to operate independently of the primary CPU when handling messages that neither originate from the CPU nor are destined for it. Figure 7.1 summarizes this design.

Messages transmitted from (or through) a simulated processor A to a specific neighbor B always use the same receiving channel at B. However, any of the channels at A may connect to that channel. The channel used is determined by a deadlock-free routing algorithm [DaS87]. The simulator is designed to accommodate different communication network configurations by replacing the module that implements the routing functions. However, due to time constraints, all experiments described in this chapter were run using binary hypercubes of varying sizes. The routing module therefore implements e-cube routing [SuB77], as shown in Figure 7.2.

Contention in passing messages through the transmitter coroutines is mediated by *cut-through routing* [KeK79]. That is, the entire message may be buffered at each simulated processor before transmission to the next is begun. This is reasonable because all messages in the MPCR model are very small. A channel is active from the time it makes a connection with a sending transmitter until transmission of the message to the next simulated processor in the e-cube route is completed. Until its channel becomes inactive, a transmitter will not accept a connection from a second sending transmitter. If two different transmitters at some simulated processor A simultaneously attempt to transmit messages to neighbor B, one of the A transmitters will be blocked until the other finishes communicating with B.

The MPCR simulator is implemented in C, using Block Island Technologies' *Interwork II* parallel programming toolkit [Bai88]. *Interwork II* provides coroutines, queuing mechanisms, automated distribution of data structures among physical processors, and a global namespace. This makes the MPCR simulation portable to uniprocessor machines and to both shared and distributed memory multiprocessors. Initial implementation work was done on a DEC VAX 11/780 and later on a Sequent Symmetry S81. At the time the simulation

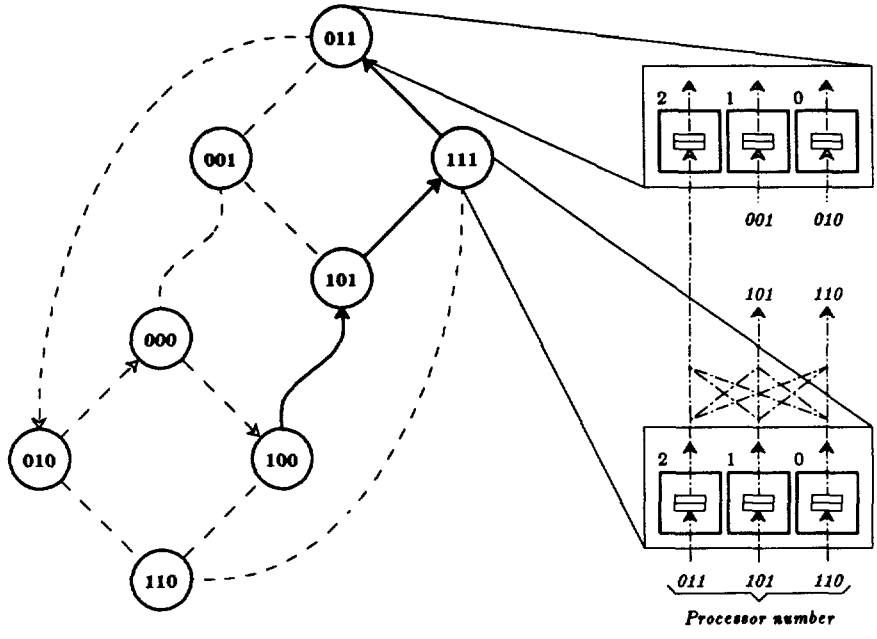

Figure 7.2 — e-cube Routing. The route is generated by toggling the bits in the binary representation of the processor number, one by one, from right to left. Only the bits that differ between the source and destination numbers are toggled. The solid lines show the route from processor 4 (100) through processors 5 (101) and 7 (111) to processor 3 (011). Note that the "return" route from 3 to 4 is not the same; it passes through 2 (010) and 0.

In the simulator, each of the neighbors of a processor uses a different message transmitter (channel) to pass messages to or through that processor. The transmitter used is always the same and is determined by the bit position of the sender's number that must be toggled to generate the receiver's number. For example, processors 1, 2, and 7 can each connect to a transmitter at processor 3. Processor 1 uses transmitter 1, processor 2 uses transmitter 0, and processor 7 uses transmitter 2 (shown in the diagram at right).

Any transmitter at the sending processor can connect to the transmitter at the receiving processor, but only one such connection can be made at a time. For example, if processors 5 and 6 simultaneously send messages to 3, one of the two messages will be temporarily blocked, because the e-cube routes to 3 for both of these senders pass through 7.

Each simulated processor has one additional transmitter (shown in Figure 7.1) to handle messages originating from that processor.

was designed, Interwork II did not yet support use of more than one processor on the Sequent. That version was therefore used for development, because the determinacy of

sequential execution simplifies debugging (errors are repeatable). The program was then ported unchanged (except for removal of some debugging code via the C preprocessor) to the Intel iPSC/2 hypercube multiprocessor to collect data.

7.2. Diffusion Scheduling Policies

7.2.1. Information Policy

As in the diffusion scheduling experiments, information is exchanged only among directly connected processors. The pressures received from neighbors are averaged with the pressure computed locally to obtain the value broadcast to the neighborhood. The pressure value for each processor is the length of its ready queue. Most tasks pass through the ready queue very quickly, but speculative tasks executing at very low priority may remain either in the queue or in a suspended (waiting) state for a long time. In contrast to our assumptions in the diffusion scheduling simulation, MPCR tasks occupy very little memory when suspended because of their extremely small size, and speculative tasks can be garbage-collected whenever memory usage becomes excessive. For these reasons, no direct accounting for the number of suspended tasks is included in the pressure value. Including suspended tasks in the base pressure computation would misrepresent the resources available for higher-priority work.

To simplify weight computations and reduce load message size, memory usage information is not treated as a separate component of the pressure. A fixed, very high pressure is still used to represent memory approaching its maximum capacity, but processors do not store memory statistics for their neighbors. When memory at a processor is nearly full, it broadcasts the "memory full" pressure value and begins deleting speculative tasks from its ready queue. If it is unable to delete a sufficient number of tasks to reduce its memory

usage, it will execute higher-priority tasks normally until the memory usage declines. No new load broadcast is made until memory occupancy has been reduced.

The most recently computed pressure is "piggybacked" on a reduction packet when one is sent to a neighbor processor. The load value most recently sent to a neighbor by this method is stored. When a new pressure is computed, the percentage change between the new pressure and the last pressure sent to each node is compared to a *threshold* value. Additional load messages are sent only to those neighbors for which the difference exceeds the threshold. A load broadcast therefore need not include all neighbors. Pressure is not piggybacked on data messages. Although a data message must pass through the message coprocessors on one neighbor node, such a message is not examined by the run-time system.

Unless otherwise noted, the load broadcast threshold was fixed at a 15% change in load for all the experiments described in this chapter. This value was determined empirically from several early tests of the simulator and may not be optimal for all combinations of programs and numbers of processors used. In particular, programs which generate fewer tasks compared to the number of processors make more load broadcasts, because the average loads from which the change is computed are small.

7.2.2. Transfer Policy

The decision to transfer is made for each complete reduction packet as it is formed. If any neighboring processor has a lower pressure than the local processor, the packet is transferred. Otherwise the packet is placed in the local ready queue. The determination of whether a neighbor has lower pressure is made by the same method as is used in the location policy, discussed below. Once a packet has entered the ready queue of a processor, it remains at that processor until it has completed its evaluation transformation. That is, a processor that accepts a packet commits itself to performing one combinator step of the

reduction of the subexpression represented by that packet. If that reduction step introduces a new redex, the subgraph is repacketized and the new packet may at that time be transferred to another processor.

7.2.3. Location Policy

The fine granularity of MPCR tasks also makes a significant difference with respect to the location policy of the diffusion scheduler. Small tasks require that diffusion scheduling decisions be made as quickly as possible. The weight computation described in Chapter 6 is complex enough that the execution time of the average MPCR task would be much less than that to decide where to schedule it. The decision is therefore reduced to three comparisons. A task may be accepted if any of the following conditions hold:

1. The local pressure is less than the lowest neighbor pressure.
2. The task has been rejected too often by other processors. This controls "thrashing," that is, tasks are never delayed indefinitely by the diffusion scheduler.
3. The current number of ready tasks s less than the optimal queue length.

Condition (3) is tested only for tasks that have higher priority than the highest currently ready at the local processor, and have been rejected at least once by other processors. In the simulations described in Chapter 6, it is assumed that task size is sufficient to always make parallel evaluation worthwhile. With very small tasks, the expenses of parallel evaluation are worthwhile only if there is already other work ready to execute locally. The *optimal queue length* for each processor is therefore estimated as described in Chapter 5. As long as the ready queue length remains below this value, high-priority tasks arriving from other nodes are automatically accepted. However, high-priority tasks that have not yet been sent to at least one other processor are always allowed to migrate, to prevent the com-

putation from becoming sequential.

If none of the criteria require that the packet be accepted, the task is rejected and sent to the neighbor with the lowest pressure. The *home* and *launch* weights described in Chapter 6 are not used when determining the least-loaded neighbor. In addition to speeding up decision-making by simplifying the weight computation, the goal of achieving optimal ready queue length makes use of a launch weight inappropriate. Transferring a task is acceptable if a neighbor has a lower load, but there is no need to "encourage" parallel evaluation when the local processor has available resources.

The distance from the origin processor is not considered when placing a packet. Each packet returns only a single value to its origin node, so repeated communication between specific parent and child tasks does not occur. Furthermore, packets that represent subexpressions requiring several reductions to reach normal form may migrate repeatedly. The notion of an "origin" of the computation becomes unclear in such an environment. Overcoming a distance bias is the purpose of the home weight, so it is not useful if distances are not computed. Eliminating distance from the computation will also prevent the orbiting effect described in Chapter 6.

For similar reasons, the number of rejections of (hops traveled by) the packet is not included in the computation. Instead, the hop count is given an upper bound, and if that bound is exceeded then the processor currently considering the packet must accept it. This controls thrashing by forcing tasks to eventually stop their migration.

7.3. Optimal Queue Length

Computation of an optimal ready queue length requires an estimate of the number of tasks which have dependencies on other tasks. This estimate is expressed as a probability P_{ij}, the probability that a task i depends on some other task j. Considering only reduction

packets as tasks, strict combinators form packets with dependencies, and nonstrict combinators form packets with no dependencies. Test runs of several different programs, with speculative evaluation disabled, show that strict combinators are the functors in about 15% of the total useful reductions performed. However, the true situation is more complex, because the formation of each packet from its corresponding application graph must also be considered.

In the test cases, approximately 50% to 75% of the useful reductions involve combinators that introduce a new application (expansive combinators). Forming a reduction packet from an application node involves an implicit dependency on the left function of the application. Considering packet formation to be a task, this indicates that P_{ij} for many problems may be over 50%. This is discouraging, because it means that dependencies are approaching the range in which it is impossible to mask latency. However, preliminary runs on small numbers of processors showed reduced performance when P_{ij} was less than 50%, so it was set at 50% for the experiments described here.

Recall that estimating the optimal queue length requires an estimate of the average message-passing time between any two processors, called T_m. In the experiments described here, the time for a message to travel between adjacent nodes, assuming that no collisions delay the message, is approximately the same as the time to form and execute a reduction packet. This means that remote execution takes at least three times as long as local execution, because messages must be sent in each direction. The estimate of average message-passing time is made by multiplying the one-hop transfer time by the average diameter of the communication network. This is based on the characteristics of the MPCR simulator's message-passing system, with the assumption that every node exchanges messages with every other node with equal probability. Although processors are in direct communication

only with their nearest neighbors, outgoing reduction packets may be considered and rejected several times. Also, the initial graph is distributed to as many nodes as possible, so remote dereferences frequently must travel several hops. For a hypercube configuration, the average diameter is $\frac{d}{2}$ where d is the dimension of the hypercube [Dal86].

This scaling model is pessimistic; real systems such as the iPSC/2 can exchange messages between distant processors nearly as quickly as between adjacent ones [Arl88]. Furthermore, the simulator imposes delays on the primary CPU to transfer messages to the (simulated) transmitter coprocessors. Providing the coprocessors with direct memory access, as in the Transputer [Whi85], would eliminate this overhead.

The estimate of T_e, the average execution time, is based on the implementations of the combinators. Re-packetizing the result of an evaluation transformation when an appli-

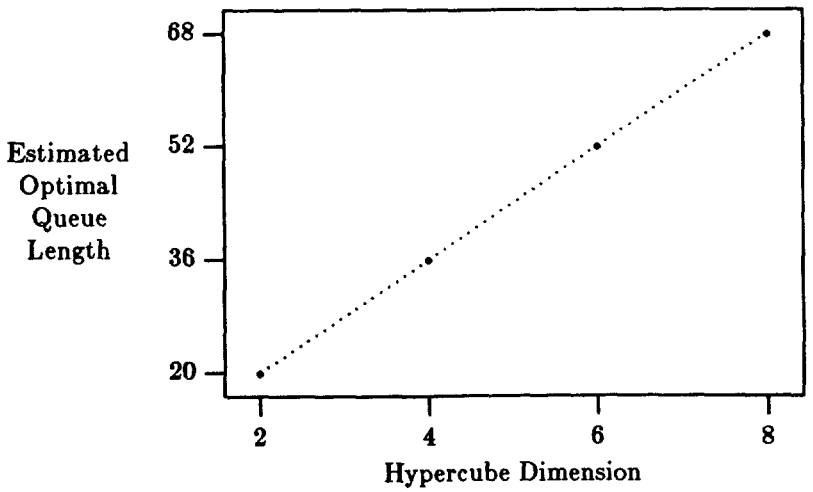

Figure 7.3 — Estimated Optimal Queue Lengths.

cation node is created is treated as a separate task for purposes of making this estimate. Fortunately, the average time to assemble a packet is approximately the same as the average time to evaluate one. The time spent in suspending tasks and returning them to the ready queue is not included in the estimate, because suspension is inexpensive (it requires only placing an entry in a notifier list) and affects only a subset of all tasks. This means that the estimate used for T_e is slightly less than the actual execution time for those tasks that suspend.

Estimated queue lengths for the network sizes tested are shown in Figure 7.3.

7.4. Combinators

This section describes the set of combinators implemented by the simulation. The code for the graph manipulations corresponding to these combinators is built into the run-time system, so any expression can be executed at any processor. Turner's basic set of combinators [Tur79], extended by some arithmetic and comparisons, was chosen primarily because of the fine granularity of its operations. However, this set also has the advantage of being very straightforward to implement, which allowed simulator design and implementation to concentrate on the mechanisms to support distributed speculative evaluation. Finally, Turner's set has a well-defined abstraction algorithm which simplified development of the *Lambda* compiler described in Appendix B. This compiler was used to produce combinator expressions from programs written in a simple lambda-calculus language.

The reduction system implemented in the simulator uses the outermost-first reduction rule to avoid nonterminating computations. However, certain combinators were selected to be sources of speculative evaluation. The speculative subexpressions created by these combinators are scheduled at reduced priority as described in Chapter 5. Like all reduction tasks, speculative evaluations are distributed by the diffusion scheduler to the heuristically

most appropriate processor.

A listing of the set of combinators follows. The classification of each as expansive, contractive, neutral or strict is noted, and the reduction rule for each is described. If a combinator causes a subexpression to be evaluated speculatively, this is also described. Figure 7.4 shows the compilation of a simple program into this combinator set, and the corresponding program graph.

C (C (C I 3) 4) 5 (B (B Add) Add)

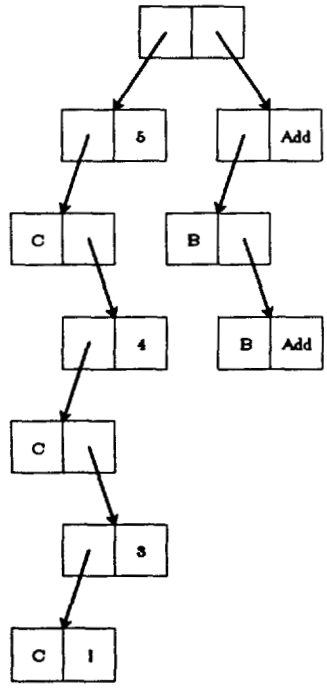

Figure 7.4 — Program graph for the compilation of the program
`sum 3 4 5 where sum = λx.λy.λz. x + y + z`

B (expansive)

$B\ f\ g\ x \rightarrow f\ (g\ x)$. The evaluation of $g\ x$ is begun speculatively.

C (neutral, considered expansive)

$C\ f\ g\ x \rightarrow f\ x\ g$. This combinator is treated as expansive because it does not reduce the number of applications in the program graph and may initiate evaluation of the subexpression f.

I (contractive)

Identity: $I\ x \rightarrow x$.

K (contractive)

$K\ x\ y \rightarrow x$.

P (neutral, considered contractive)

Construct cons/pair: $P\ x\ y \rightarrow (x,y)$. This is implemented as actual formation of a pair-tagged node, for ease of manipulation. Lists and pairs are not distinguished from one another, except that the special element *nil* (or `[]`) is provided as a canonical for lists. Note that pair formation is fully lazy, i.e. the components of a pair are not evaluated unless demanded.

S (expansive)

$S\ f\ g\ x \rightarrow f\ x\ (g\ x)$. As with B, the subexpression $g\ x$ is evaluated speculatively.

U (expansive)

Disassemble cons/pair: $U\ f\ (x,y) \rightarrow f\ x\ y$

Y (expansive)

The standard fixpoint combinator for compiling recursive functions. Implemented as $Y\ h \rightarrow h\ (Y\ h)$ because of difficulties with reference counting cyclic structures. Turner presents an optimized Y reduction which introduces a cycle, but the tradition-

al equation introduces none, at the cost of some loss of sharing. Also, Turner's Y optimization cannot be directly applied because of the way subexpressions are evaluated as new, independent tasks.

Cond

(strict contractive)

A conditional test:

Cond b x y $\rightarrow$ x when b is *true*;
Cond b x y $\rightarrow$ y when b is *false*.

The strict evaluation required of the first argument of *Cond* makes it difficult to classify as contractive, because additional evaluation may be necessary before the contractive effect occurs. It would be possible to speculatively evaluate x and y while awaiting the result of b, but in practice the boolean test b is usually a very simple operation, so little is gained by speculating on both of the cases.

Add, Sub, Mult, Div, Mod

(strict contractive)

Perform (integer) arithmetic operations. Again, strictness prevents these from being considered truly contractive. These operations, along with the boolean comparisons, are the only source of nonspeculative parallelism in the combinator set. Both arguments of an arithmetic combinator can be evaluated simultaneously at the same priority as the application of the combinator itself.

Eq, Gt, Lt

(strict contractive)

Boolean comparisons (on integers). Both arguments of a boolean combinator can be evaluated simultaneously at the same priority as the application of the combinator it-

self.

Nil (strict contractive)

Test whether a cons/pair structure is the nil list. This combinator is *strict only* up to WHNF, that is, it does not force evaluation of the head or tail of the list.

7.5. Description of Experiments

The initial graph for each program is distributed across the simulated processor network before computation begins. For simplicity, this distribution is done in a round-robin fashion by increasing processor number, with no consideration of referential locality. The structure of each program graph is different, and implementing a generalized static mapper to match graphs to networks is beyond the scope of this research. However, the initial graph is pre-processed to form all possible reduction packets, complete or partial, before the program is loaded. This preserves some locality, and assures that at least a few steps of computation can be performed without waiting for remote requests. Distributing the graph reduces the severity of "hot spots" [PfN85] but can not entirely eliminate them. As used here, the term "hot spots" refers to processors which receive a large number of requests for data and must therefore process many more messages than other processors in the network.

Although the graph is distributed, execution is defined to begin at simulated processor number 0. To detect completion of the program, a finished normal form of the result is collected at that processor. This means that there may be a slight delay between actually finishing the computation (printing the last part of the result) and detecting that completion. When processor 0 has received the entire normal form, it signals all other simulated processors to send their final statistics and shut down. The collected statistics are then output. Similar statistics are collected by a checkpointing process (an Interwork II task) that runs at regular intervals throughout the course of the simulation.

Values of Pij, T_m, and T_e are held constant across all programs. As has been noted, this probably leads to less than ideal behavior, but time constraints limited the number of trials that could be made for each program. The available memory at each simulated processor and the ready queue length considered to constitute overflow is also fixed. The only variables are the input programs themselves and the dimension of the hypercube network of simulated processors.

Simulated time does not have a direct relationship to real time, so the experimental runs can only be considered in comparison to one another. As base cases for the speculative, parallel runs, each program was also run with no speculative evaluation on a single simulated processor (a zero-dimensional network). These base runs were used to determine the minimum number of reductions required to complete each program, and the sequential time required to perform those reductions.

It should be noted that this testing method fails to exploit speedups predicted by *Gustafson's Law* [Gus88a, Gus88b] (also called *Moler's Law*). This law states that large numbers of processors achieve greater speedups when they are used to perform larger computations. For purposes of comparing the same programs across different network sizes, the experiments described here do not scale problem size as the number of processors increases. Speedups are therefore likely to be less than could be expected for larger problems.

7.6. Programs Tested

This section presents the results of running three simple programs on various sizes of simulated processor networks. Before the individual programs are described, however, some general comments should be made. First, although the simulator is designed to switch to evaluation of contractive combinators when memory nears capacity, none of the programs tested generated sufficient work to evaluate the usefulness of this strategy. Experiments us-

ing larger programs were planned to evaluate this, but were dropped due to time constraints.

Second, the selection of the S and B combinators as the source of speculative evaluations makes the potential for speculation dependent on the form of the source program. This is because these two combinators perform the function of rearranging the graph so that function arguments are supplied to the correct subexpressions. It is therefore possible to tell which types of programs have a chance of speedup, but difficult to tell exactly how well they will do.

7.6.1. Map-Squares

This program was the simplest one tested. It generates a list of integers and squares each of the integers to generate an output list. Although the mapping operation has much potential parallelism if implemented eagerly, list construction is fully lazy as defined in our test combinator set. Furthermore, the computation is limited by generation of the list. Each element is created by adding one to the preceding element. Therefore, even with an eager mapping (that is, processing of the tail of the list begun concurrently with processing of the head), maximum speedup from function-level parallelism is approximately a factor of 2. This would represent the squaring of one element happening concurrently with generation of the next element.

Throughout this chapter, test programs will be expressed in a simple lambda-calculus language, called *Lambda*. The syntax of this language is summarized in Appendix B.

```
mapsq = λn. map (λx.x * x) (to n)
where rec
      map = λf.λx.
            if null x
            then nil
            else f (fst x), map f (snd x)
where
      to = λx.
           (range 1 x
            where rec
                  range = λb.λe.
                          if b > e
                          then nil
                          else b, range (b+1) e
           )
```

The results summarized here show representative runs of *mapsq 100* (generating and mapping over a 100-element list), on hypercubes of dimensions 4, 6, and 8. There were two reasons that this experiment was performed. First, to determine whether the speculative evaluation mechanism could compensate for a sequential algorithm. Second, it was hoped that the fine-grained evaluation strategy might uncover parallelism that was not obvious from the definition of the program. However, *mapsq* does not show parallel speedup for any of the network sizes tested. In fact, there is a slight slowdown, as shown in Figure 7.5. The sequentiality imposed by lazy list construction is probably the major factor here.

None of the network sizes is able to achieve the loads approaching the estimated optimal level, so speculative work is never curtailed. In the case of the 8-dimensional network, the computation is 60% complete before all processors in the network are simultaneously active. Before that point, some processor is always idle. However, by the time the computation completes, the total work performed by the busiest processor is not more than double that done by the least-utilized one.

These results show that the heuristics for creating speculative work are working properly, although for *mapsq* much of the effort is misguided. The extra work done does not appear to interfere significantly with useful computation. Note that the number of remote

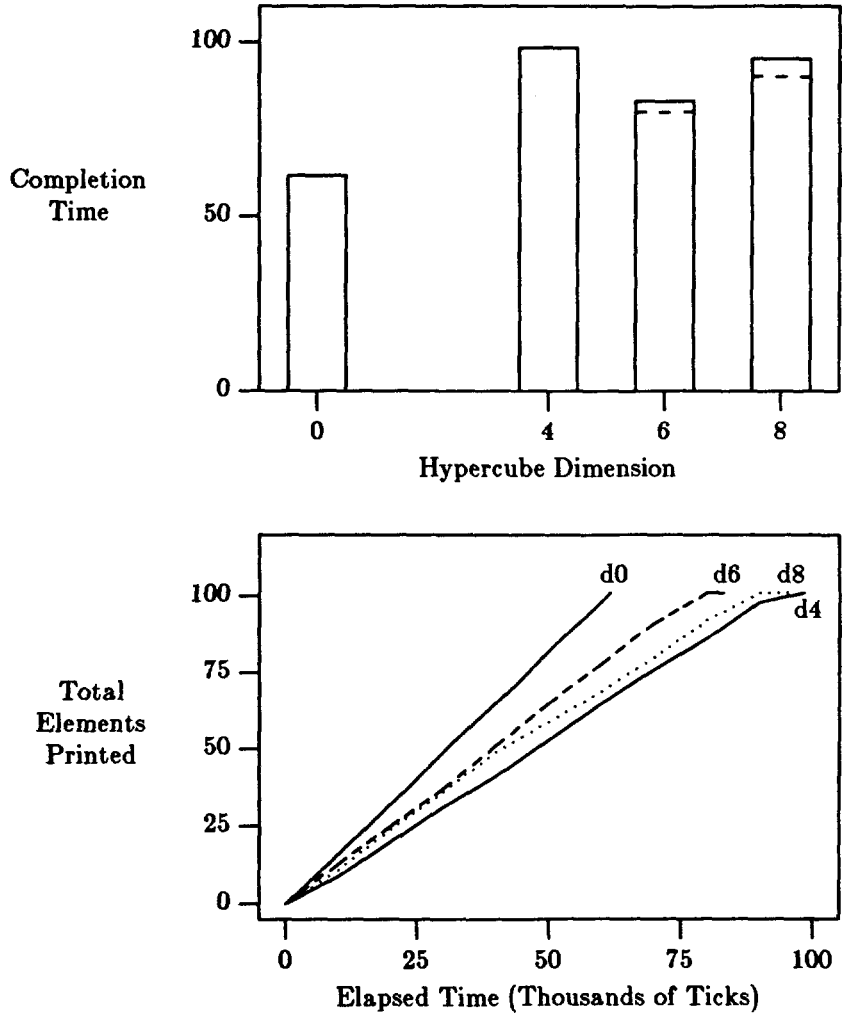

Figure 7.5 — Completion Times for Map-Squares. The dashed lines in the bar graph at dimensions 6 and 8 show the earliest checkpoint at which all elements of the list had been printed. The apparently sharp "tailing off" in the rate of printing the final few elements is an artifact of the checkpointing process and the method used to detect program completion.

dereferences is approximately equal to the total number of reductions performed. This is shown in the second graph of Figure 7.6, and in more detail in Figure 7.7. This indicates that some useful speculative work is successfully being used to mask latency. Recall that

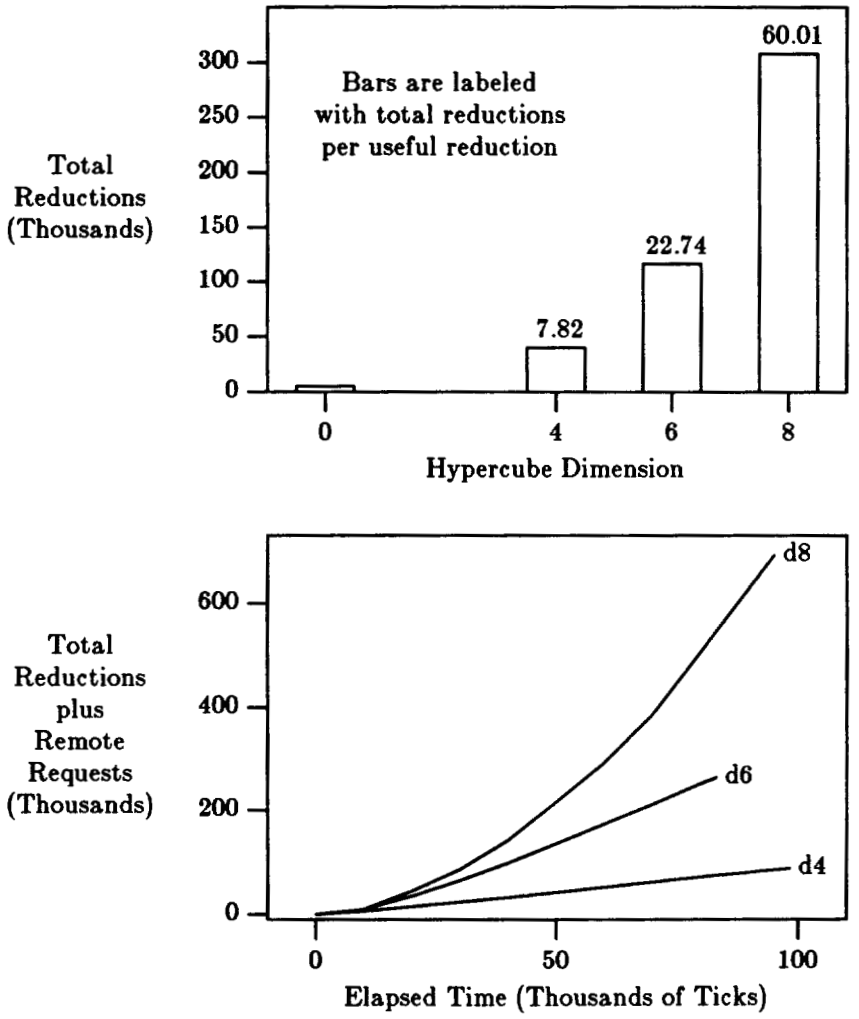

Figure 7.6 — Reductions Performed for Map-Squares. Total reductions counts only those reduction packets that completed their evaluation transformation.

the introduction of a remote request at least triples the expected evaluation time of a reduction packet. If speculation were not masking the extra latency of exchanging messages, the slowdowns would be much more pronounced. Further evidence of this is discussed in relation to the *psum* program.

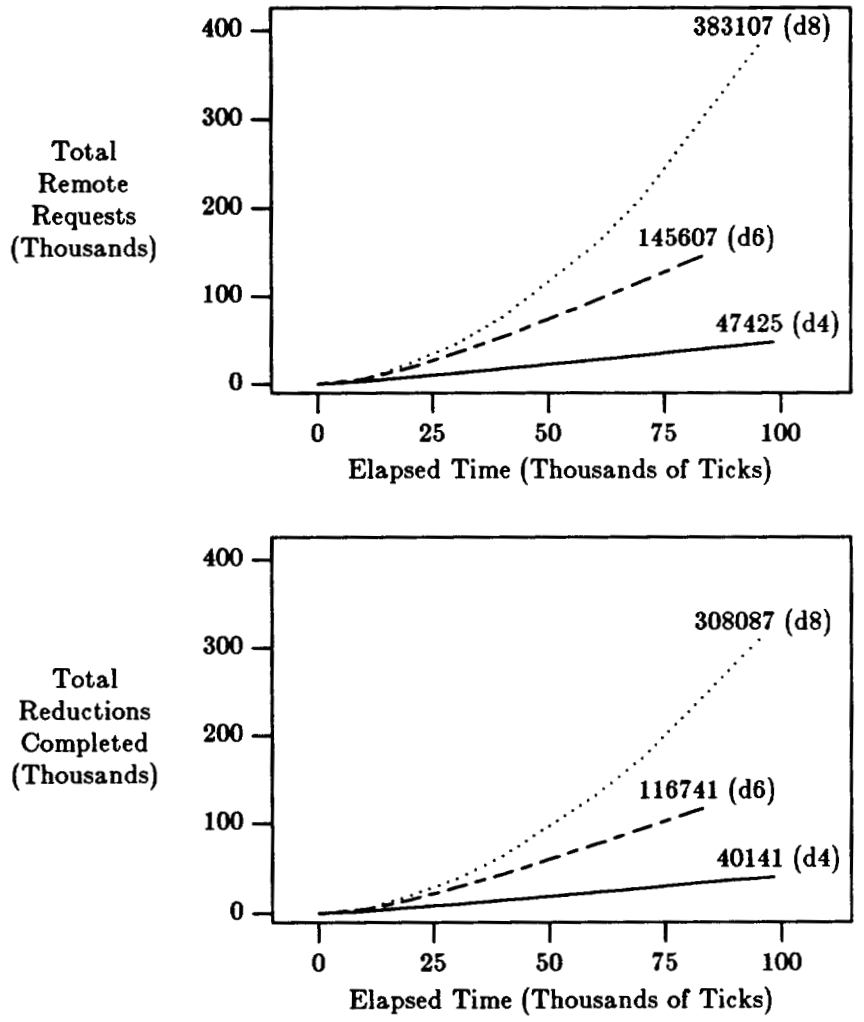

Figure 7.7 — Reductions Compared to Remote Dereferences for Map-Squares. The count of remote requests includes automatic elimination of indirections before returning the normal form of a reduction packet to a remote processor. Indirections are introduced by the I and K reductions. This pattern of remote requests approximately equaling all other reductions combined is consistent for all three test programs and all network sizes.

7.6.2. Parallel Sum

This is the well known divide-and-conquer algorithm for generating the sum of n consecutive integers in a given range. Sequentially, this requires $O(n)$ time, but in parallel it can potentially be done in $O(\log_2 n)$ time. This computation is actually better suited to a data-parallel SIMD decomposition, where the n integers are multiple data. The algorithm used here generates the integers as it adds them, which introduces additional work proportional to the number of integers summed.

```
psum = λlow.λhigh. dsum low high
where rec
      dsum = λlow.λhigh.
             (if (low = high)
              then high
              else dsum low mid + dsum (mid+1) high
              where
                    mid = (low+high)/2
             )
```

This algorithm has the best potential for nonspeculative parallelism of the three programs tested. It also has the least potential for useless speculative computation. Not surprisingly, it shows the best speedups. The computation is tree-structured, very similar to the simulated process trees used in the diffusion scheduling experiments described in Chapter 6. It is encouraging to note that the speedup of *psum* on the 4-dimensional (16-processor) hypercube corresponds to the results of Chapter 6 for test cases with a similar ratio of execution time to message passing time.

Figure 7.8 summarizes completion times and speedups for runs of *psum 1 1000* (sum integers from 1 to 1000) on 4-, 6-, and 8-dimensional hypercubes. The 4-dimensional network quickly reached optimal queue lengths at all nodes, and spent the majority of the computation time with loads considerably above optimal at all nodes. This is reflected in the relatively small number of excess reductions performed, as shown in Figure 7.9 and Table 7.1.

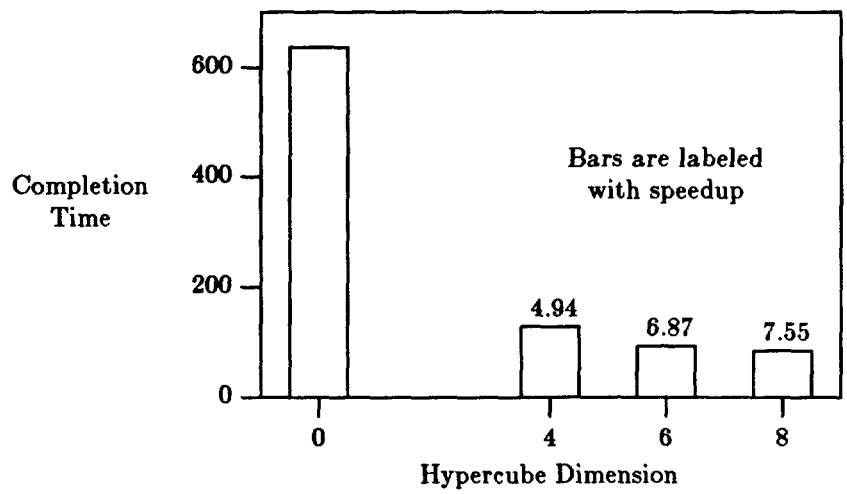

Figure 7.8 — Completion Times for Parallel Sum.

In contrast, the two larger networks never achieve a balanced, near-optimal load. Concentration of the work in a few nodes is probably the result of a combination of factors. In the 4-dimensional network, all nodes have a portion of the initial graph, so remote reference requests help to distribute the work. Only a fraction of the nodes in the 8-dimensional network have parts of the initial graph. In addition, the diffusion criterion of accepting higher-priority tasks until the local queue is at optimal length may be concentrating work at a few nodes early in the computation. The latter is unlikely, however, because those queues that exceed the optimal length grow very rapidly to the maximum, as shown in Figure 7.10.

A more likely explanation is that the queues are filling with remote reference requests. Recall that these requests are not handled by the diffusion scheduler, but instead must be scheduled at the node which holds the reference. As speculative work is created to try to fill

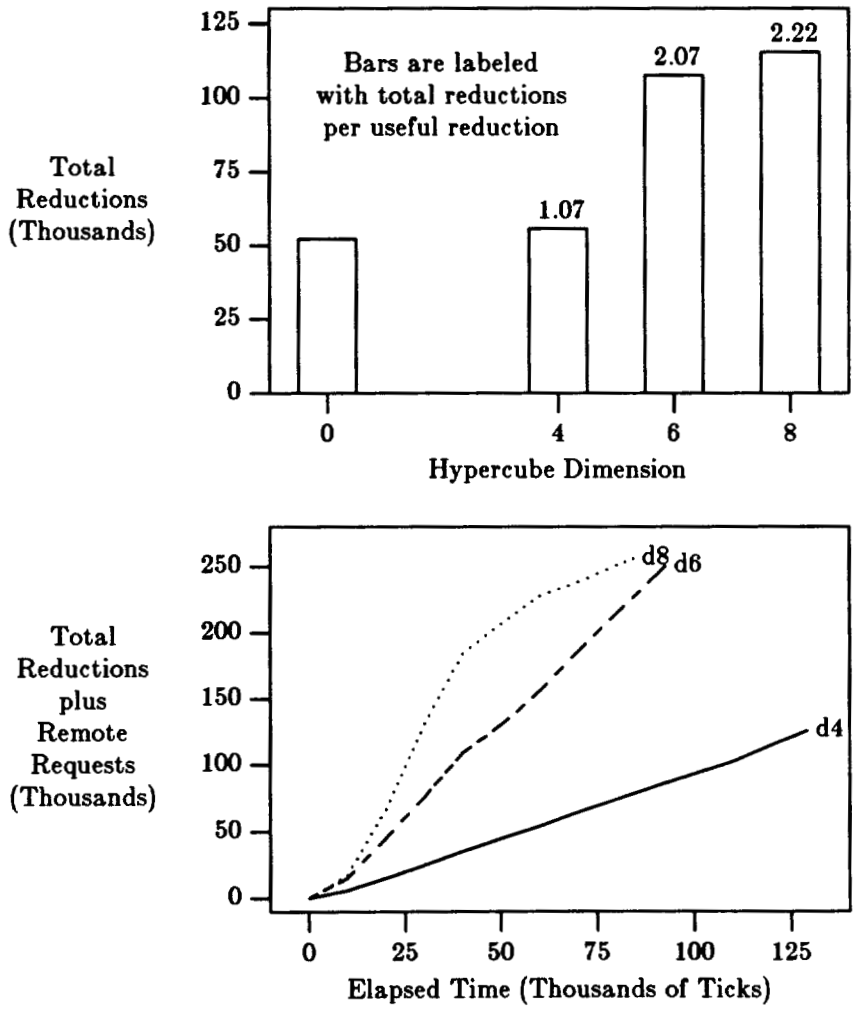

Figure 7.9 — Reductions Performed for Parallel Sum. See also Table 7.1.

the large processor network, important subexpressions or "hot spots" in the initial graph become the target of large numbers of low-priority remote requests. The scheduling of higher-priority tasks prevents many of these requests from ever reaching the front of the ready queue, and so the queue fills with them. This in turn causes the local node to send any oth-

Table 7.1 — Summary of Other Excess Work
Parallel Sum

Dim	Deleted	Left	Duplicate	% Wasted
4	665	517	942	1.7
6	71895	972	14437	34.9
8	85959	9	15834	39.8

The summary lists the number of packets that were deleted, that were left incomplete at program termination, or that returned a duplicate result. Deleted and unfinished tasks are not included in the bar graph in Figure 7.9, but duplicate work is included in that graph. The percentage wasted reflects the sum of the first three columns as compared to the total reductions completed.

er reduction packets it generates to neighboring nodes. Those packets may in turn send remote requests, forcing still more work into the already overloaded queue.

To make certain that speculative computation was not unreasonably limiting speedups for the larger hypercubes, an additional test was run. Using an 8-dimensional network, and with speculative computation disabled, *psum 1 1000* was run again. If nonspeculative parallelism in the arithmetic operations was entirely responsible for the speedups, the nonspeculative test should perform at least as well as the previous tests. Instead, it performs much worse, running about 3 times *slower* than single-processor nonspeculative execution. Not surprisingly, this is exactly the slowdown anticipated due to the speed (or rather, the lack of speed) of the message-passing system. The system's reliance on the outermost-first (lazy) reduction rule is the most likely reason that more parallelism is not discovered. Recall that this reduction rule avoids following non-terminating paths, by stipulating that subexpressions (*i.e.*, the arguments of a function application) are not evaluated until after the application itself has been evaluated. Thus, even though nonspeculative parallelism is present in the tree of additions, the evaluation order prevents the system from discovering it. This

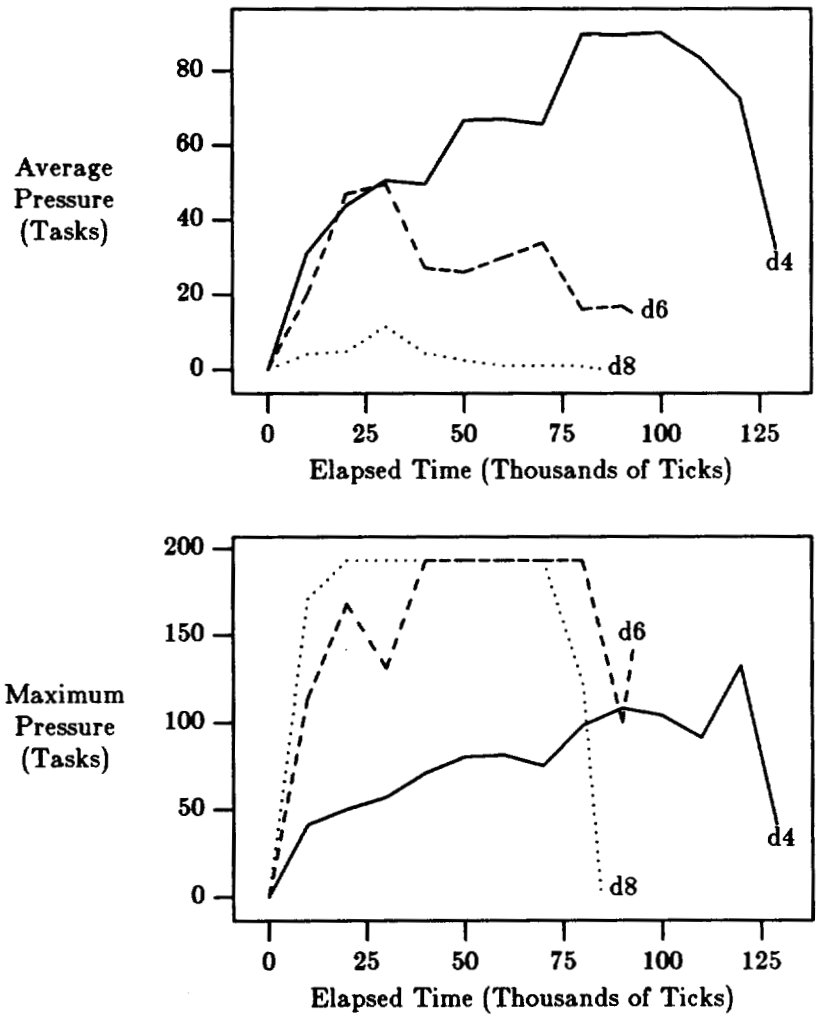

Figure 7.10 — Average and Maximum Per-Processor Pressure for Parallel Sum. The maximum pressure is that for ***any*** processor, not necessarily the same one each time. However, note the apparent "hot-spot" behavior for the 6- and 8-dimensional networks. The flat area at the top of the graphs is the maximum queue length, at which a processor begins to delete speculative tasks.

In the 8-dimensional network, ***minimum*** pressure is never found to be above zero, but every processor performs at least a few reductions. The total work done at each processor in the 6-dimensional hypercube is much more even, although the load at any given checkpoint is not balanced.

suggests that speculative computation is essential in a concurrent evaluator that wishes to take advantage outermost-first semantics, at least to stimulate parallelism, if not to mask message latency.

7.6.3. Towers of Hanoi

This algorithm produces a list of two-digit integers. The digits represent one move of a disk from the tower numbered by the first digit to the tower numbered by the second. This is quite similar to parallel sum, except that each branch of the recursion performs $O(n)$ operations where n is the height of the towers. Note that the operations in the branches of the tree are completely independent, so the only limitation on parallelism in this algorithm is the construction of the process tree. However, the process tree is joined by list construction, so this parallelism is uncovered only by speculative evaluation originating in other parts of the program.

```
rec
    hanoi = λf.λs.λt.λn.λr.
            if n < 2
            then move f s, r
            else hanoi f t s (n-1) (move f s, hanoi t s f (n-1) r)
    where
          move = λx.λy. 10 * x + y
```

Although this program shows only modest speedups, as seen in Figure 7.11, it provides the most insights of any of the tests. The load balancing behavior of the 4-dimensional network is particularly satisfying (see Figure 7.12). The average pressure at each processor is maintained consistently near the optimal queue length throughout the computation. In contrast, the average load varies widely when tested on a 6-dimensional hypercube, and the 8-dimensional case never manages to utilize all processors.

As seen in Figure 7.13, at the 8-dimensional network size the system begins to complete reductions at a slightly higher rate than for the 6-dimensional hypercube, then experi-

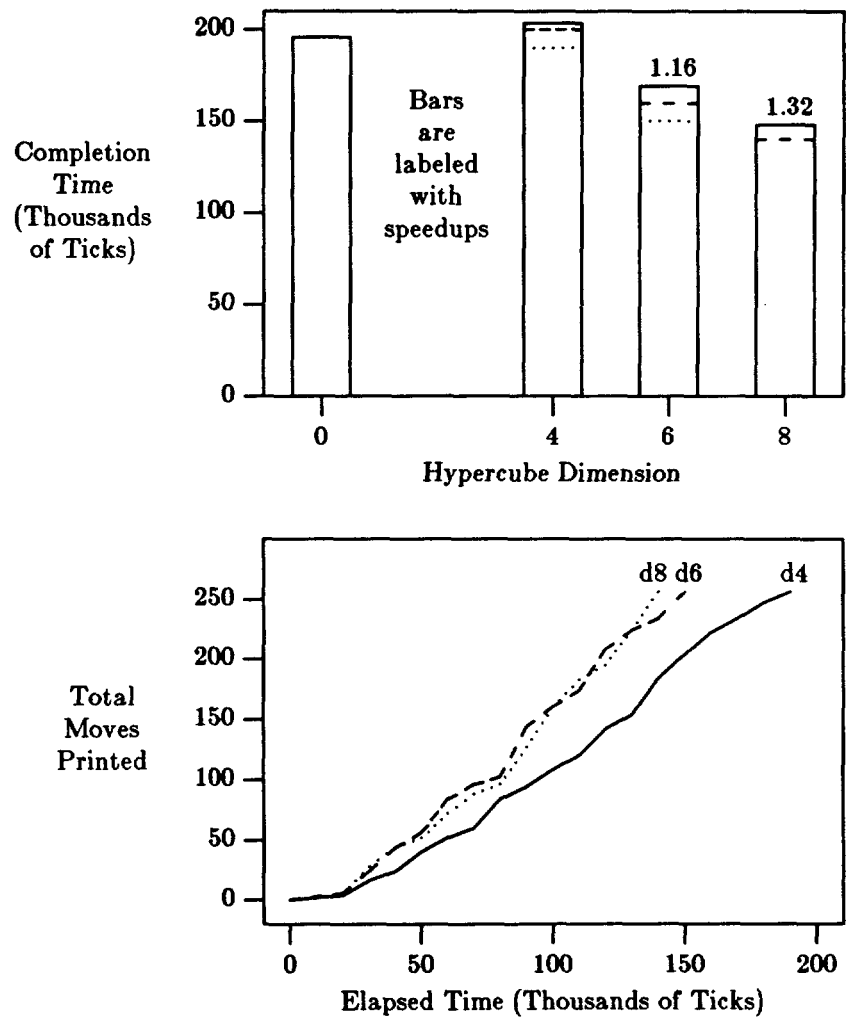

Figure 7.11 — Completion Times for Towers of Hanoi. Dashed lines in bars show the earliest checkpoint at which the full list of moves had been printed. Dotted lines show the earliest checkpoint at which all moves had been computed. The "tail" artifacts mentioned in Figure 7.5 were so pronounced in this case that they have been cut from the graph of total moves printed.

ences a sharp drop in the rate of completions. This is so different from the linear behavior of the other two cases that the experiment was repeated to be sure that no transient error

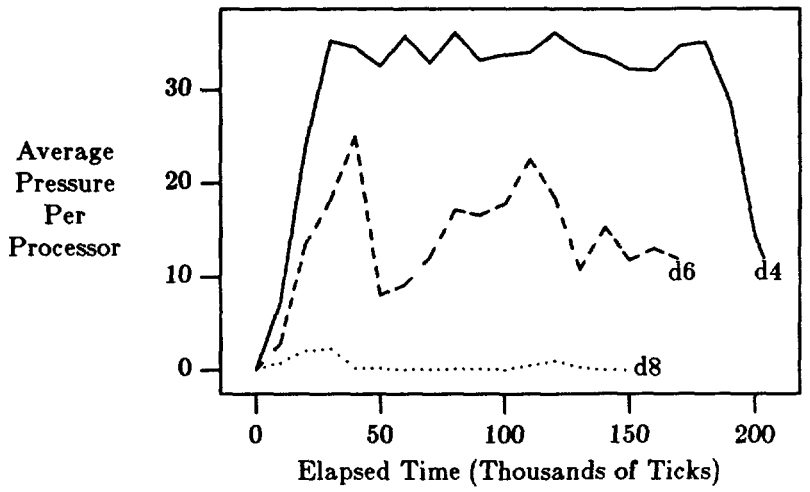

Figure 7.12 — Average Per-Processor Pressure for Towers of Hanoi.

was responsible. A comparison of two runs is shown in Figure 7.14. Although the second run shows considerably different behavior during the course of the computation, the completion times of the two differ by only a few clocks. Also interesting was the series of jumps in the packet completion rate for the second test, corresponding to large variations in the total pressure.

This is most likely explained as an indirect effect of the "hot spot" behavior observed earlier for parallel sum. As the system attempts to generate enough work to load all processors, a large number of speculative tasks are created. Most of these tasks are completed very quickly, but eventually the queues at a few processors fill up with remote requests. At this point those processors begin deleting speculative tasks, including some of those requests, to reduce their queue lengths to a manageable level. In the case of *psum* (Figure 7.9 and Figure 7.10), there is plenty of nonspeculative parallelism to take up the slack as speculative

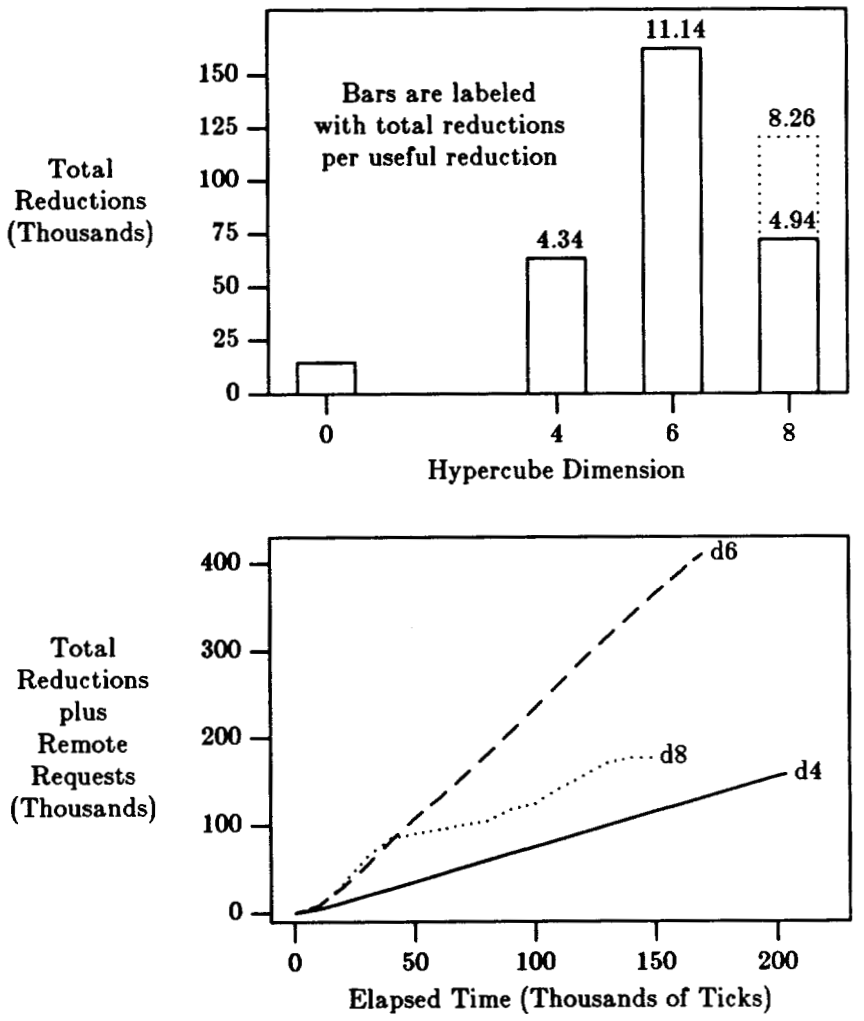

Figure 7.13 — Reductions Performed for Towers of Hanoi. Note the sharp drop-off in rate of completions for the 8-dimensional network. A similar effect is seen in Figure 7.9, but it is much clearer here. The behavior was considered odd enough that a second run was made for comparison. The dotted portion of the bar graph shows results of that test.

tasks disappear, so no drop in the pressure is ever detected by the checkpointing process. However, a drop in the rate of completions appears as processors begin to spend more time cleaning excess tasks out of their queues.

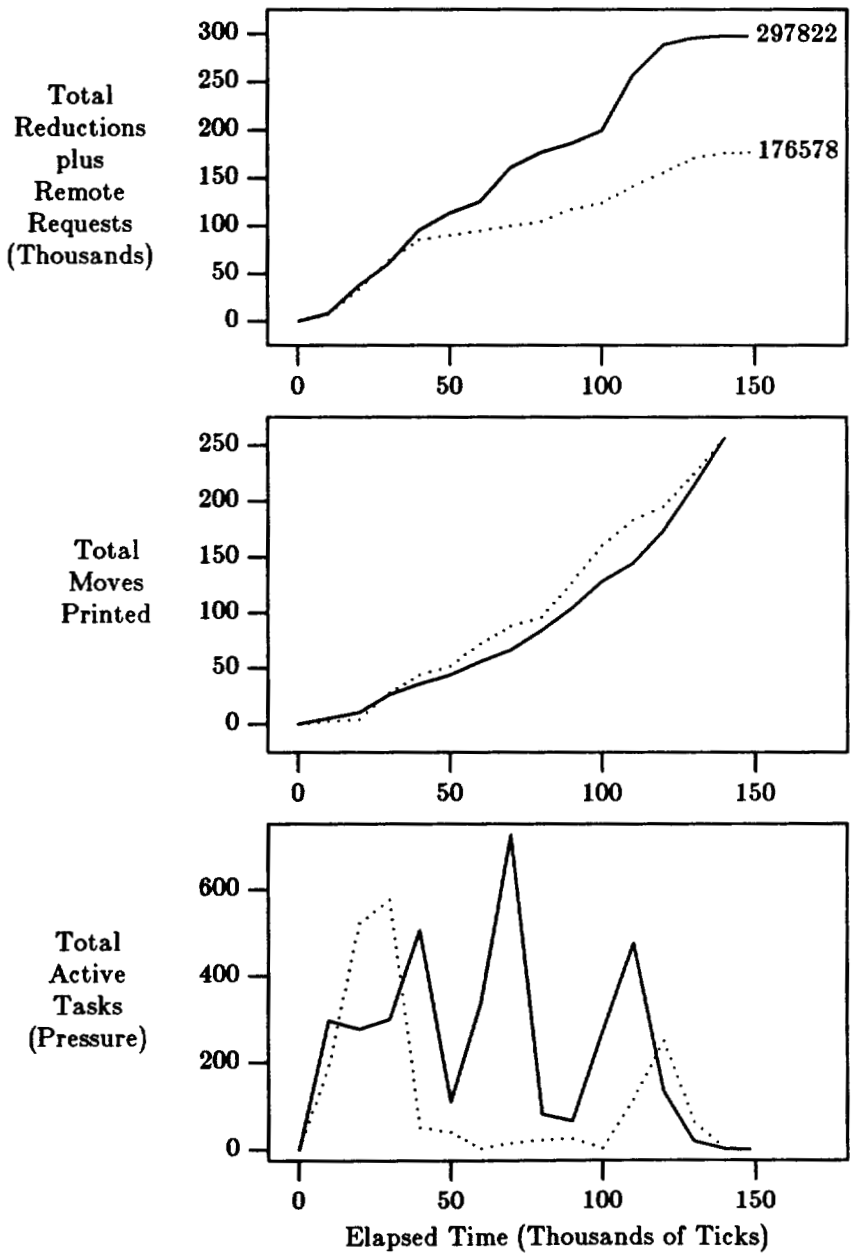

Figure 7.14 — Comparison of Two Runs, Towers of Hanoi, 8-Dimensional Hypercube.

By contrast, *hanoi* has almost exclusively speculative parallelism. When a ready queue at a "hot spot" processor fills up and tasks are deleted, parallel evaluation is sharply reduced. The system then resumes creating speculative tasks, attempting to load all processors. The result is a repeated rise and fall in the number of active tasks, as seen in Figure 7.14.

If the nondeterministic effects of dynamic scheduling are considered, this hypothesis can also explain the behavior of the first test. In order for speculative computation to build to its former level after the first drop in pressure, high-priority tasks must take long enough to complete that some useful speculative tasks return their results first. The first test has a slightly higher rate of completion of nonspeculative work, as seen by the rate at which moves are printed. This may be enough to prevent a resurgence of speculative work until late in the computation.

Despite the presence of some differences, the general similarity in the rate of printing moves for both cases and their nearly identical completion times are encouraging. The steady increase in the printing rate in the second test, in spite of wide fluctuations in the pressure, shows that the priority system is successful in reducing or preventing interference from speculative tasks. Furthermore, this suggests that elimination of "hot spots" may allow much greater speedups even without improving the speed of message transmission.

7.7. Summary

These experiments indicate that, in general, reduced-priority speculative parallelism cannot completely replace strict parallelism. Slowing of computation in the *mapsq* test and only moderate speedups in *hanoi* demonstrate that speculation is insufficient to overcome a strongly sequentializing factor (*e.g.*, lazy constructors). The most successful application of speculative computation appears to be in combination with a few operators that exhibit

strict parallelism.

However, even when a reasonable degree of strict parallelism is present, speculative computation was important to overcome latencies. In the absence of strictness analysis, speculation appears to be necessary in order for the outermost-first reduction rule to be applied successfully in a parallel environment. Speculating within the S and B reductions, which implement function composition, causes speculation on the arguments of functions. This can to some extent take the place of pre-evaluating arguments that are known to be required.

The most significant drawback of speculative evaluation is that known problems of parallel computation are magnified. In particular, nondeterministic effects can significantly alter the behavior of the program, and "hot spots" tend to become much hotter. Fortunately, the priority scheme is generally successful in preventing this from interfering with important work, at least until resources begin to overflow.

Although techniques such as copying heavily-referenced parts of the initial graph to all processors could reduce or eliminate the effects of hot spots, further investigation is needed to discover ways to control nondeterminism. It is probable that the strategy of restarting tasks to increase priorities contributes to nondeterminism, as well as to hot spots. A strategy that allows deleted tasks to be restarted but also allows task priorities to be increased directly, might be worth the added complexity involved.

CHAPTER 8

MPCR Simulator Reduction System

This chapter describes those mechanics of the parallel reduction system that are not a direct implementation of the abstract model of Chapters 3 and 4. The graph manipulations needed to perform individual combinator reductions are not included, because they are obvious from the evaluation rules for each combinator, and follow very closely the diagrams in Tur79. As implied by the re-sending of Demand messages in the abstract model, when a reduction is performed the reduction packet describing the reduction is updated with the result. The updated packet is then re-evaluated until it is found to be in weak-head normal form (WHNF). These forms are then sent back to the origin processor of the packet, which is determined from the redex address. When the WHNF reaches the origin processor, the local access pointer in the redex address is consulted, and the indicated marker is updated, completing evaluation of the subexpression.

Figure 8.1 summarizes the symbols used in the diagrams in the remainder of this chapter. References are represented as either solid or dotted arrows, depending on whether the object pointed to is significant to the diagram. Solid arrows that do not point to graph node symbols indicate references that are significant but that may be non-local (remote). The *current pointer* indicates a reference which was taken either from the local ready queue or from a message, and which therefore is not contained within the expression graph itself. All operations of the reduction system begin by dereferencing such a pointer; the initial current pointer is supplied by the run-time system of the root processor node, and is obtained by loading a program graph from external storage. All applications in this initial

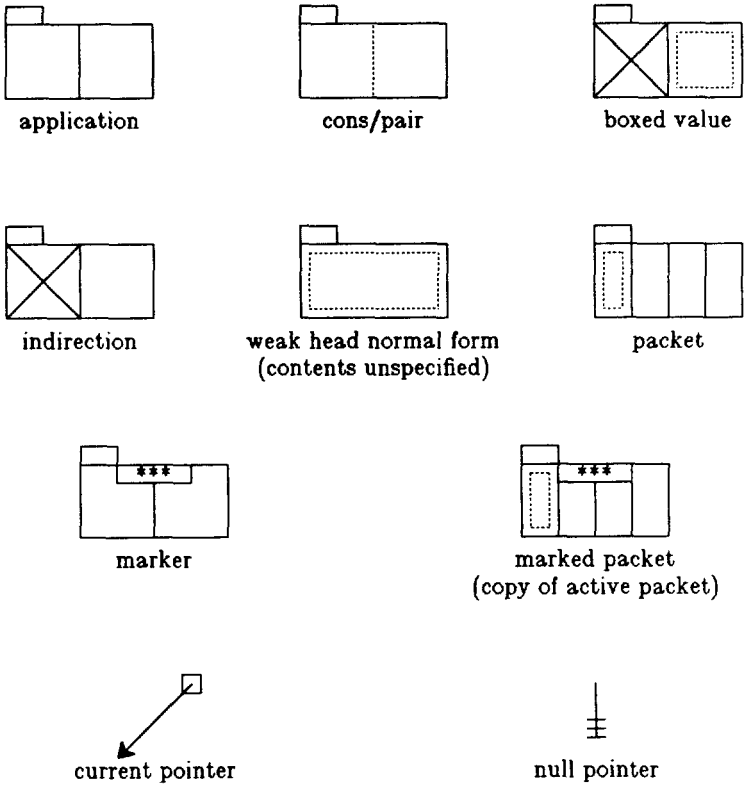

Figure 8.1 — Symbols

program graph are assigned the ***minimum possible*** priority; higher priorities are assigned as the graph is evaluated (see ***Packet Formation***). Evaluation of the root of the graph then begins at the ***maximum*** possible priority, and priorities propagate downward.

8.1. Packet Formation

Before any evaluation can occur, a reducible subexpression must be collapsed into a task description called a ***reduction packet***. Packet formation is summarized in Figure 8.2. This corresponds to the recursive ***packetize*** algorithm given in Figure 3.10. The right cell of the marker is initially empty, but will be used to keep track of the ***notifier list*** of tasks that

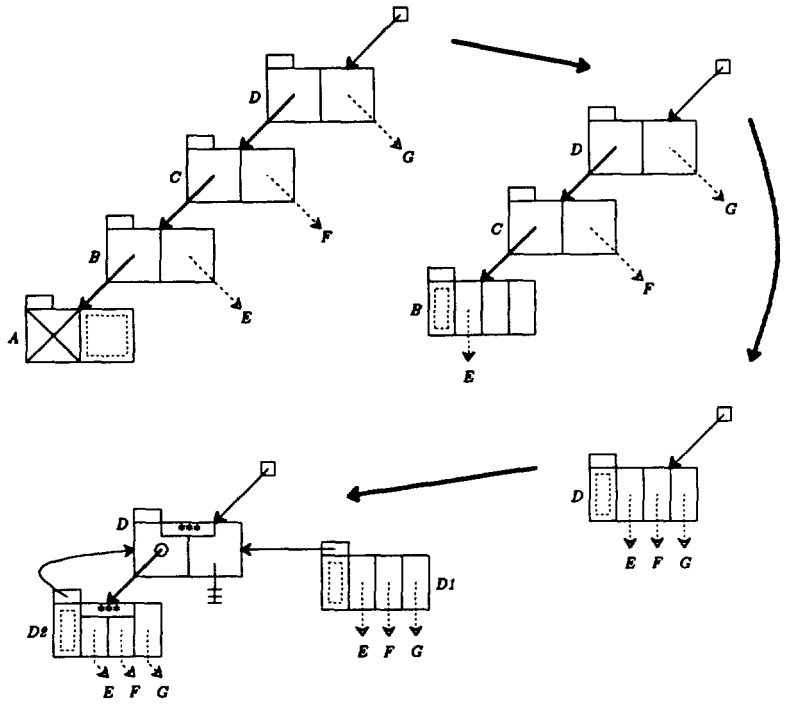

Figure 8.2 — Formation of a reduction packet. The large dark arrows show progression of the subexpression graph through each of the four stages; two repetitions of step 3 are omitted.

1. The left spine of application nodes in the subexpression is recursively descended until a combinator is found (node *A*). The combinator may be in the left cell of an application node, as a boxed value, or in a partially filled *template*; see step 2.
2. The application node immediately above the combinator (node *B*) is overwritten by a reduction packet *template*. This template contains the combinator from the left cell of the application, the reference (or canonical value) from the right cell, and empty slots for the remaining arguments of the combinator (if any). Templates represent *partial Packets* from the abstract model.
3. The application node immediately above the template (node *C*) is overwritten by a new copy of the template, with the next argument slot filled in by the right cell of the application. This step is repeated as the recursion unwinds, until a template has been constructed with all argument slots filled (node *D*).
4. A completed template is copied into a finished reduction packet (node *D1*), and the template node is replaced by a *marker*. A specially tagged copy of the packet, called a *marked packet*, is made (node *D2*).

should be resumed (in terms of the abstract model, sent a Demand) when the marker is updated. The marker also keeps track of the *task count* for the redex, which is initially set to one. Formation of a packet from a sample program graph is shown in Figure 8.3.

Although it is not necessary for implementation of the abstract model, both the task field of the marker and the unevaluated copy of the packet are tagged for error-checking. The task field tag differentiates an updatable marker from those used to implement the

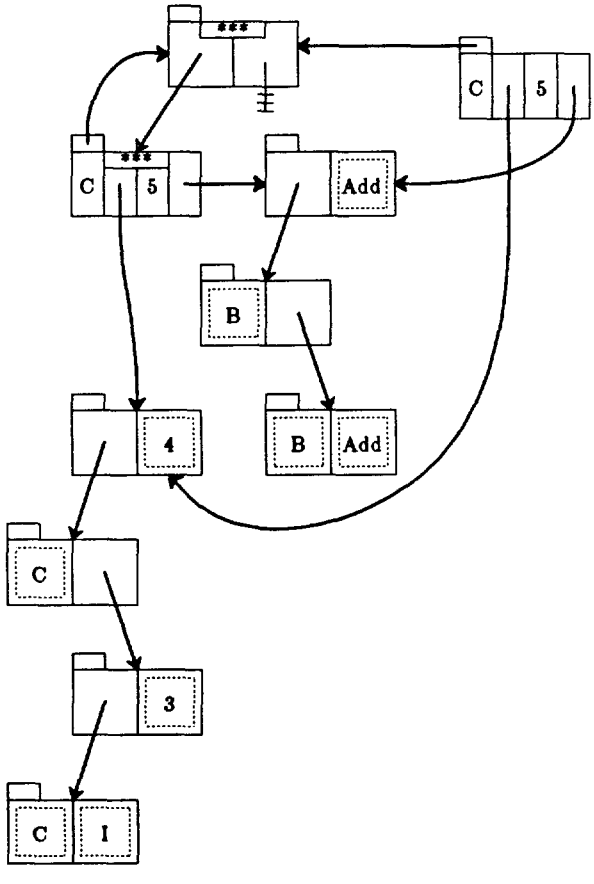

Figure 8.3 — First reduction packet of the sample sum program.

notifier list, as described in the next section. The *marked packet* tag, or more accurately, its absence, is used to check that only correctly copied packets are scheduled for evaluation.

After the reduction packet has been formed, it is passed to the diffusion scheduler for assignment to a processor. This corresponds to sending an Evaluate message to the packet. The marker node remains in the expression graph until one of its associated reduction packets has been fully evaluated, at which time the marker is updated by a weak-head normal form.

Packets can be formed either at the priority of the task demanding the value or at speculative priority. As explained in Chapter 4, speculative evaluation occurs during the evaluation transformation of certain expansive combinators. Application nodes along the left spine which are folded into templates (partial packets) are not increased in priority until the template is filled. When the packet is complete, it is assigned the greater of the priority at which it was requested or the priority already recorded at that application node. This is the *max* computation used in the prioritized reduction rules (Chapter 4). If packet formation does not consume the entire left spine of a redex, this *max* priority assignment is also applied to all application nodes in the spine above the completed packet. This assures that any future evaluations of the redex will take place at the highest priority ever assigned to the subexpression.

8.2. Suspension

The template created in the packet formation process has in its argument list one slot for each argument of the combinator in its descriptor field. If the left spine of the expression graph contains more than this number of arguments, packet formation will not consume the entire spine. When this occurs, a recursive call to *packetize* returns DEMANDED. In this event, the current evaluation must *suspend* to await the return of the evaluated subexpres-

sion. It is also possible for another evaluation, going on concurrently, to encounter a marker when descending the left spine of a subgraph.

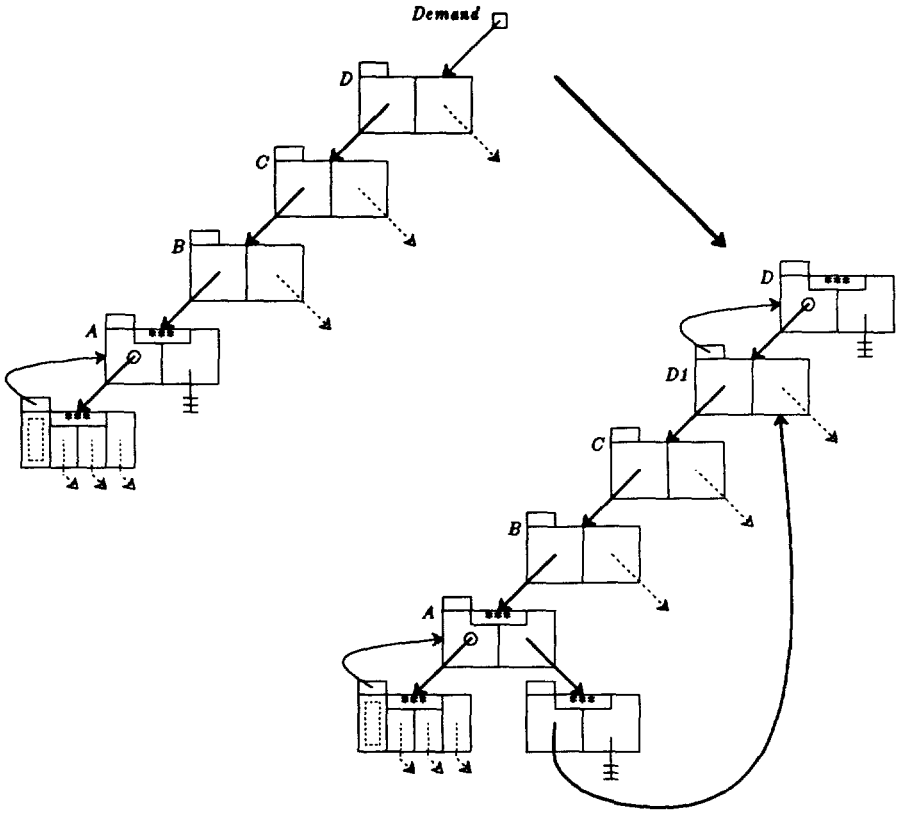

Figure 8.4 — Suspension of an Application Graph (*Marking Transformation*). To suspend an evaluation, a new graph node *D1* is first created, into which the application *D* is copied. *D1* represents the continuation of the packet formation process and will be resumed when the marker *A* is updated (that is, when the demanded value is returned). A reference to *D1* is placed in the task field of *D*. Finally, a reference to *D1* is added to the ***notifier list*** of *A*. This reference represents the Demand message used in the *packetize* algorithm. Although the algorithm of Figure 3.10 would indicate that the Demand message be sent to node *D*, the reaction of *D* would be to propagate the Demand to *D1*. Since the redex address of *D1* already refers to *D*, an additional Demand message from *D* would carry no useful information. It is therefore slightly more efficient to resume packetization at *D1*.

The notifier list is implemented in the simulation as a linked list of marker nodes. The right cell of each marker in the list holds a reference to the next marker in the list; no such

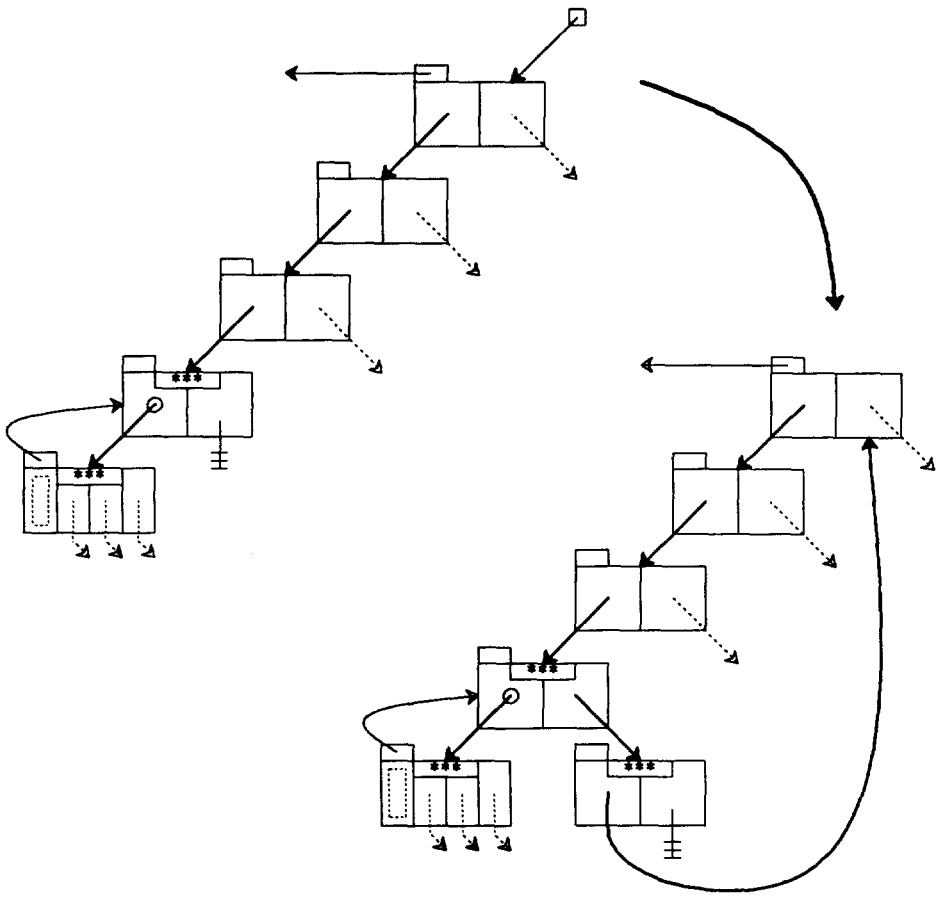

Figure 8.5 — Optimization: Suspension of a Partially Evaluated Task. An opportunity for optimization arises when the root application of the expression to be suspended is the result of evaluating a reduction packet. (Recall that when a reduction packet executes, it is transformed into a graph and then re-demanded until the expression is in weak-head normal form.) In this case, there is known to be exactly one reference to the root application, so it is not necessary to replace that application with a marker. Instead, the nonshared reference is placed directly in the notifier list.

marker can ever hold a remote reference. Instead of placing a Demand message in the list, the left cell of each marker in the list holds the reference to the task to be notified. This is because all Demands are handled through *packetize*. It is sufficient to know the node at

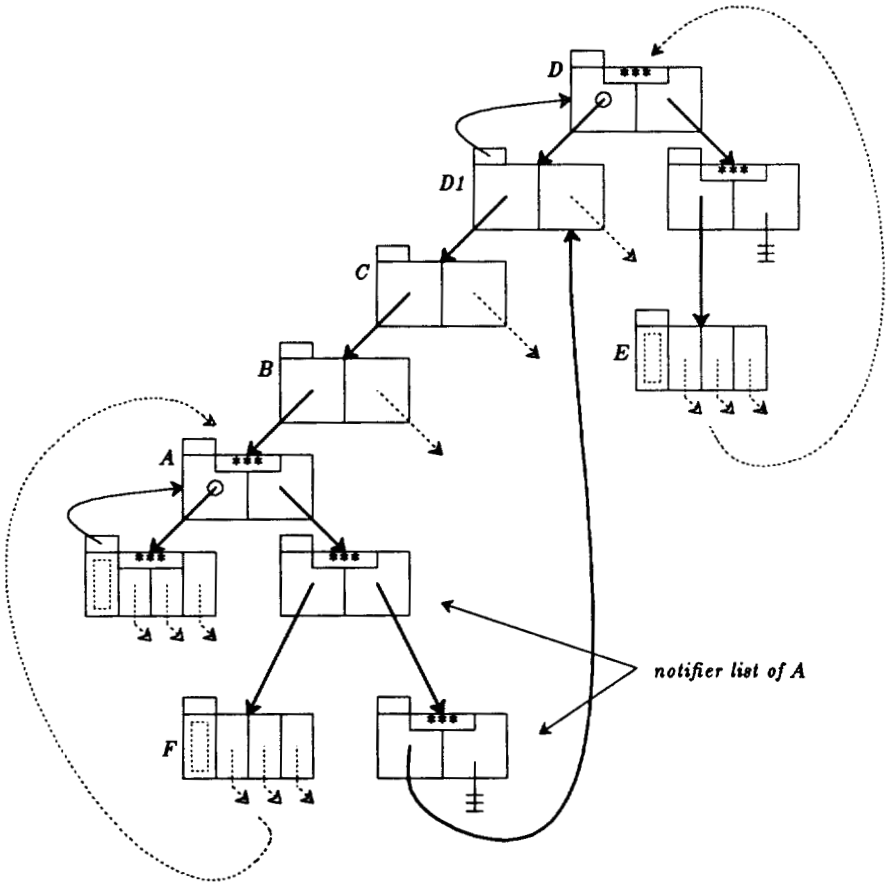

Figure 8.6 — Suspension of Additional Demands for a Subexpression. Task E is the first to suspend after demanding redex D. $D1$ is suspended on A by the *packetize* algorithm, as explained in Figure 8.4. Later, task F suspends upon demanding redex A, and is inserted into the notifier list of A.

The dashed arrows from E to D and from F to A indicate that such a path must exist in the graph, although it may pass through the left cells of one or more intervening application nodes.

which the next call to that algorithm should begin. Suspension of a task and several optimized special cases are shown in Figures 8.4 through 8.8.

As a detail, it should be mentioned how additional tasks are added to the notifier lists of markers. Concurrently executing evaluations which attempt to access existing markers are inserted into the linked list of notifiers stackwise, at the front. Figure 8.6 shows such an insertion. The rightmost reference in the notifier list is always null.

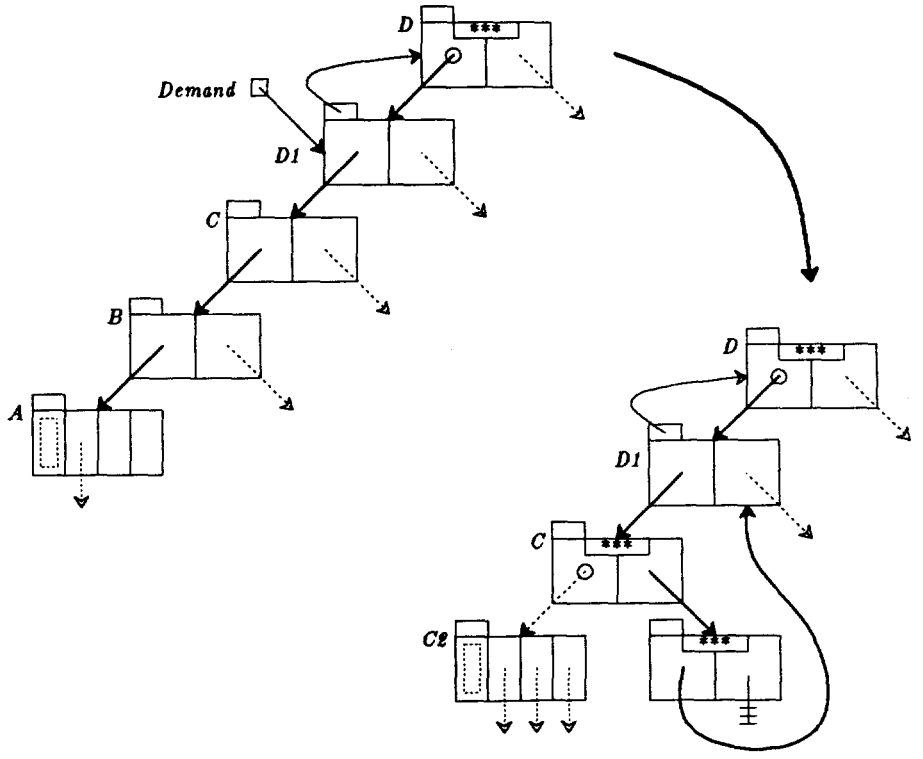

Figure 8.7 — Repeated Suspension of Application Graph. Another case of some interest occurs when two or more repeats of packet formation, evaluation, update, and re-packetizing are necessary in order to consume the entire left spine of a reducible graph. Here, the marking transformation has already been performed for node *D*, and it is therefore unnecessary to repeat it for *D1*. Note that the end result is equivalent to that of Figure 8.4.

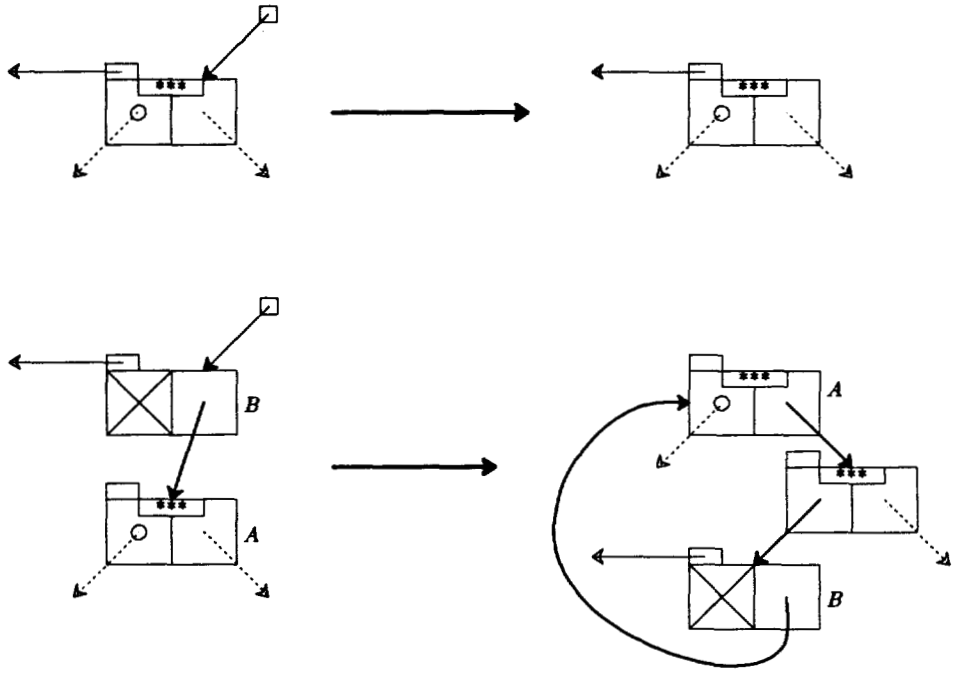

Figure 8.8 — Suspensions of Markers. The abstract model specifies that the marking transformation should move the redex address of an application into its notifier list. By definition, however, an application must never have more than one redex address. The marking transformation can thus be optimized by leaving the redex address unchanged. The update transformation is then responsible for reactivating the evaluation (see below).

If instead the redex address is found at an indirection to the marker, as in node *B*, this optimization is not possible. In this case a reference to the indirection *B* must be placed in the notifier list of the marker *A*. If this were not done, the update transformation would not have enough information to reactivate the task.

8.3. Update and Awakening

When an evaluation task has reduced to a weak-head normal form, it is returned to its processor of origin, where the marker node left as a place-holder for the value is updated. The several possible alternatives for performing this operation are shown in Figures 8.9 through 8.12.

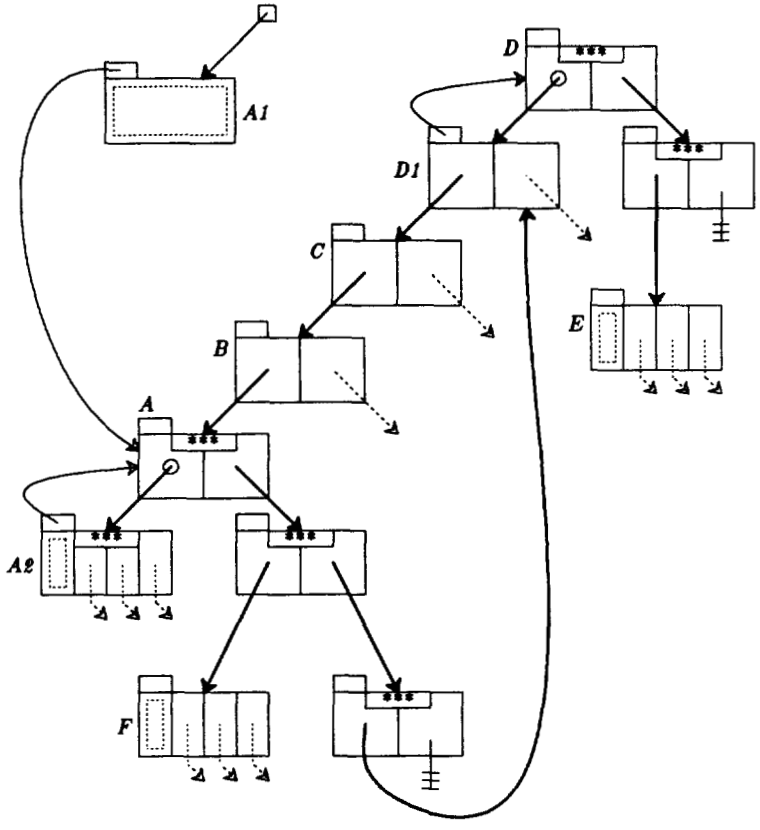

Figure 8.9 — Update Transformation, General Case. The most common update situation, task *A1* returning a value to update marker *A*. Five steps are followed to update a marker:

1. The task field reference to marked packet *A2* is released. The reference to the notifier list, in *A*'s right cell, is saved for future use.
2. The weak-head normal form in *A1* is transferred into *A*. This transfer erases the contents of *A1*, except for the redex address, which is not transferred because the redex address of *A* must not be overwritten.
3. The updated node *A* is checked for the presence of a redex address. If no redex address is found, the reference to *A* (that is, the redex address of *A1*) is released. The case when a redex address is present is described below.
4. All useful information has now been removed from *A1*, so the current reference to it is released.

The final step is shown in Figure 8.10.

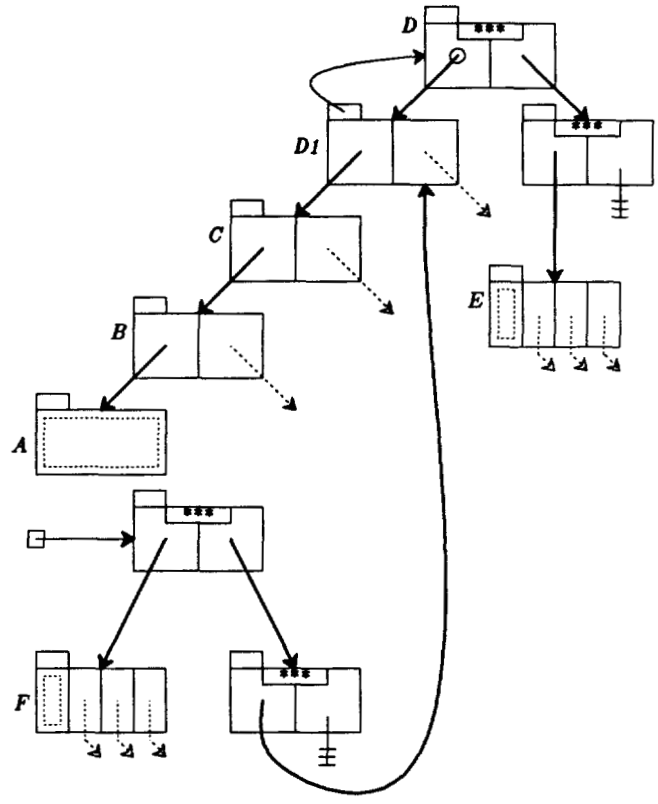

Figure 8.10 — Awakening Tasks from Notifier List (update transformation, continued).

5. The notifier list saved in step 2 is traversed. References found in the left cells of the linked markers in the list are placed in the ready queue at the local node, in priority order. This is referred to as ***awakening*** the suspended evaluations. References to the markers in the notifier list are released as the tasks they point to are awakened. This frees the linked markers, because they can have no references from outside the list.

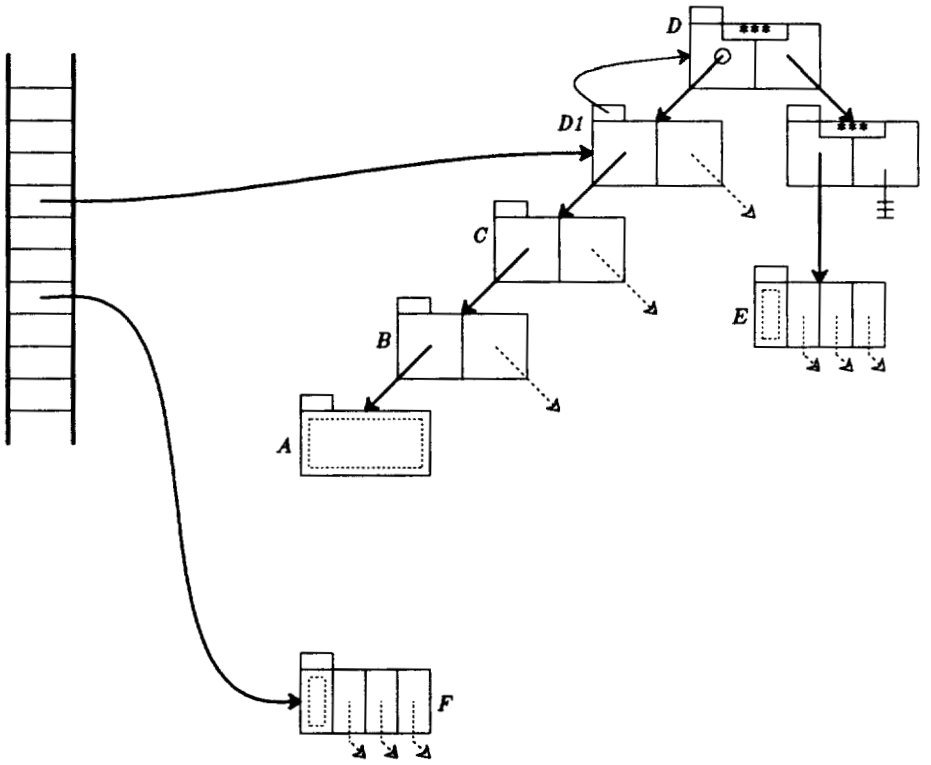

Figure 8.11 — Awakened Tasks Enqueued (update transformation). The completion of the update transformation when updated node *A* had no redex address is shown.

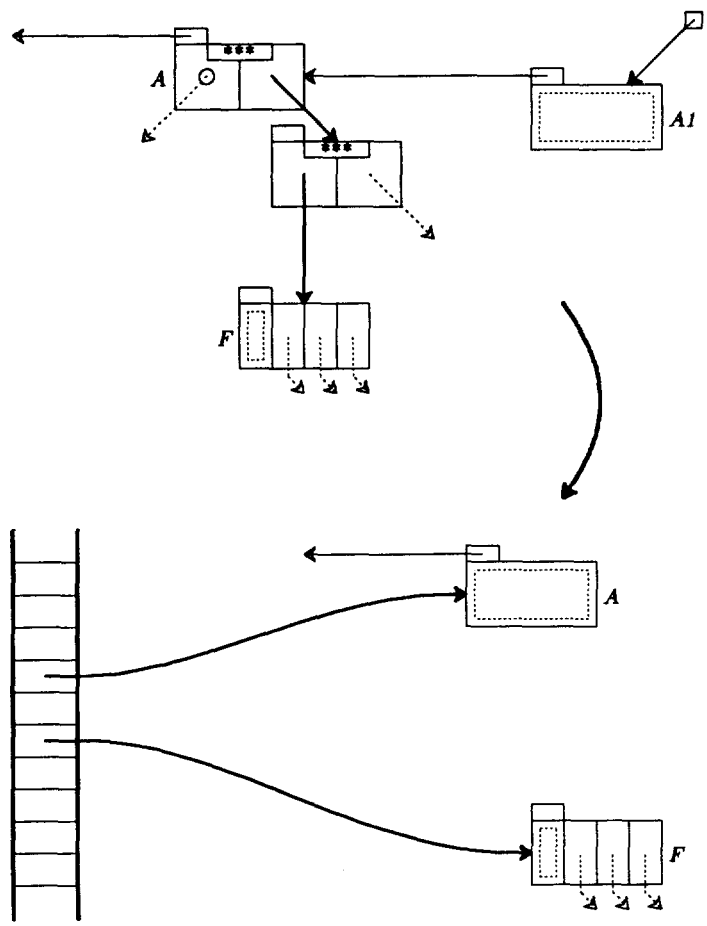

Figure 8.12 — Awakening the Updated Node (update transformation). When the updated marker A contains a redex address, A represents a suspended task which must be awakened. Recall that the redex address of A represents a Demand message, which as an optimization was not moved to the notifier list. The abstract model specifies that A should send this Demand to itself. To accomplish this, instead of releasing $A1$'s redex address reference in step 3 of Figure 8.9, that reference is placed in the ready queue. Other tasks (such as E) in the notifier list of A are awakened as usual. If A does not contain a redex address, it must represent either a speculative evaluation or a strict parallel evaluation (such as the second argument to an arithmetic combinator).

8.4. Rescheduling Tasks

As has been described, when a reducible graph is converted to a reduction packet for scheduling, a copy of the packet is left behind, referenced by the task field of the marker node. Each time a task is started to evaluate a particular redex, the priority of that task is recorded and the task count is incremented. The priority information is stored in the copy of the packet. The counter is kept in the marker node where the value will return. New demands on the marker do not start new tasks unless either (a) their priority is higher than that stored in the copy, or (b) the count of tasks associated with the marker is zero. The counter value may be zero because it is decremented whenever one of the speculative tasks evaluating that redex is deleted.

However, the most common case of rescheduling occurs when the task field of the marker refers to an application node. In this case, rescheduling must use the *packetize* function to demand that the subgraph be evaluated. This traverses the left spine of the subgraph until it encounters a marker whose task field refers to a complete packet. A new copy of that packet is then scheduled. Figure 8.13 details this operation.

8.5. Forced Task Exit

Another situation in which the notifier list must be traversed is when a speculative task has been terminated without being fully evaluated, as described in Chapter 4. When the run-time system at some processor detects that its resources are nearing their limits, it examines the queue of ready tasks to find the lowest-priority speculative task not already in normal form. When a suitable task has been selected, the processor forces the task to exit by first sending an *Exited* message, containing the redex address from the killed task, to the processor where the reduction packet originated. This message carries the reference rights held by the deleted task to its marker. Once the *Exited* message has been sent, the rest of

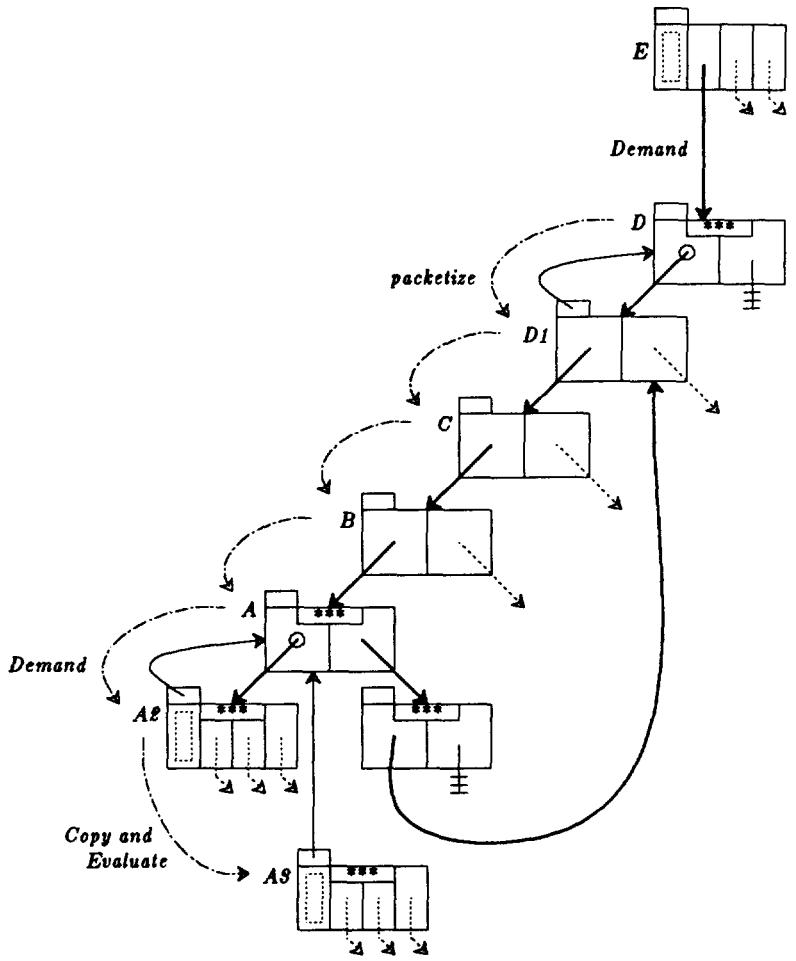

Figure 8.13 — Rescheduling a Speculative Task. Packet E, which is strict in its first argument D, demands the evaluation of D. D was previously begun as a speculative computation, resulting in reduced-priority packet *A1* (not shown) and its copy *A2*. When demanded again at higher priority, D invokes *packetize* on its task field reference. This recursively descends the graph to marker A, which schedules a new copy *A3* of the packet *A2*. The same procedure is followed if *A1* has *exited* (see below) and the priority of E is equal to or less than that previously assigned to A and D.

Note that packet E will suspend on node D, as shown in Figure 8.6. The task count of marker A is incremented. The task count of D is incremented only if it was previously zero, because there is only a single application node whose redex address refers to D. The task count of each marker thus reflects the number of other nodes that have the potential to update that marker.

the task's references are released, and the task is deleted. The rescheduling mechanism already described guarantees that if the task is ever again demanded, a higher-priority task can be scheduled to perform the computation.

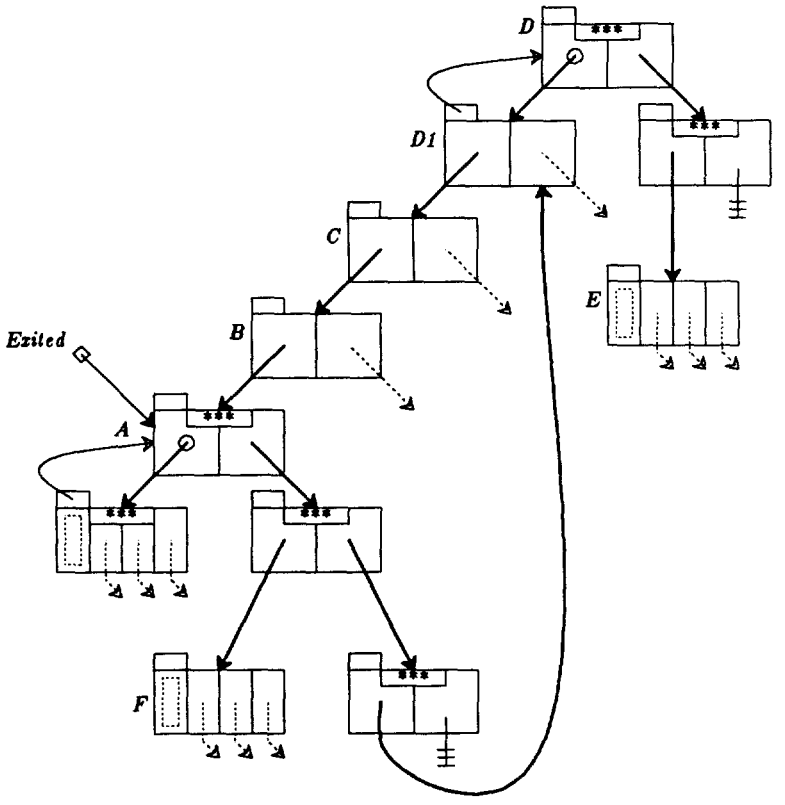

Figure 8.14 — Notification of Task Exit. The task count stored in the marker *A* is decremented, and if it has reached zero, the notifier list is traversed. All tasks in the list are forced to exit, because there is no way to guarantee that the data for which they are waiting will ever become available. Note that the task count cannot reach zero if there is any nonspeculative (highest priority) task associated with the marker. As with any deleted speculative task, the rescheduling mechanism ensures that the results of any of these tasks can still be obtained.

Figure 8.14 shows a graph at the time an *Exited* message is received at the origin processor, and Figures 8.15 and 8.16 show propagation of the exit notification.

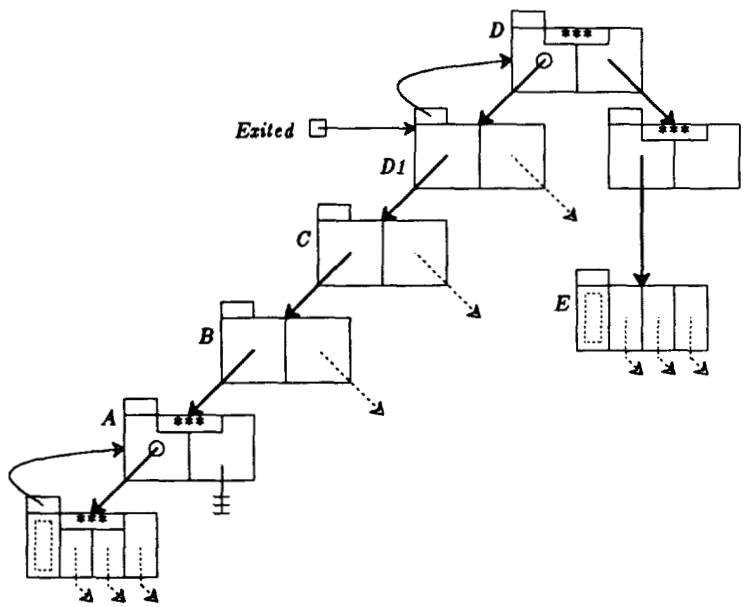

Figure 8.15 — Notification of Exit: Suspended Tasks Discarded. To discard a task from the notifier list, after an *Exited* message is sent to its redex address, the reference to the node is released and the linked marker which previously held that reference is retagged as an indirection. Task *E* of Figure 8.14 has been discarded, and application *D1* is receiving an *Exited* message. Once the notifier list has been emptied, the marker *A* is examined to see if it contains a redex address. If it does, it is also forced to exit.

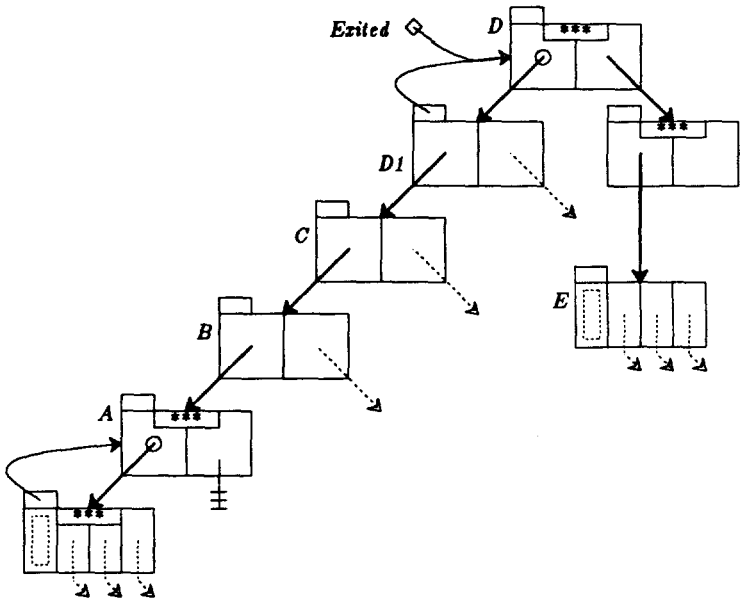

Figure 8.16 — Propagating Exit Through an Application. Applications referenced by a marker's task field require some special handling. The redex address of the application must be shared (its reference rights divided) to create the *Exited* message. This is due to the optimizations described in Figure 8.4 and Figure 8.8, which require that the redex address of such an application is not released or transferred until the task field of the associated marker is released. This does not introduce difficulties, because implementation of task counting requires the redex address of such an application to be a local reference.

The termination of a speculative evaluation may be propagated all the way back to the original node that received the *Speculate* message. If no other references to the speculative task's marker remain, the entire speculative subexpression is deleted. Useless speculative computations can thus be completely removed. They are not eliminated as soon as they become useless, but their reduced priority prevents them from delaying essential work in the meantime.

8.6. Remote Reference Requests

When the value of a remote reference is demanded, a *remote reference request* is generated and sent to the processor where the referenced node is to be found. Generation of a remote request is shown in Figure 8.17. At the origin processor, the request packet is treated as a strict *I* combinator reduction. If the argument reference has not been evaluated, it is demanded, and the request task suspends to await it. In the case that the argument reference points to yet another remote reference, the evaluation is simplified whenever possible by forwarding the original request packet without changing its redex address. The final value is thus returned directly to the marker at the processor where the reference was originally demanded.

Unlike other evaluation tasks, request packets cannot always be rescheduled. When the number of rights held by the argument reference reaches the minimum, no new copies of the request packet can be generated. This is because any attempt to share the argument reference would result in the introduction of an indirection, which is a local reference. Only when the demanding task has the highest priority is no new copy needed. This is therefore the only case in which remote requests are guaranteed to be sent. If it is not possible to send a request, the demanding task is forced to exit, and notification is sent to its redex address, as has been described. This does not affect the completeness of the computation, because normal-order tasks always have highest priority, but it may limit some speculative computations.

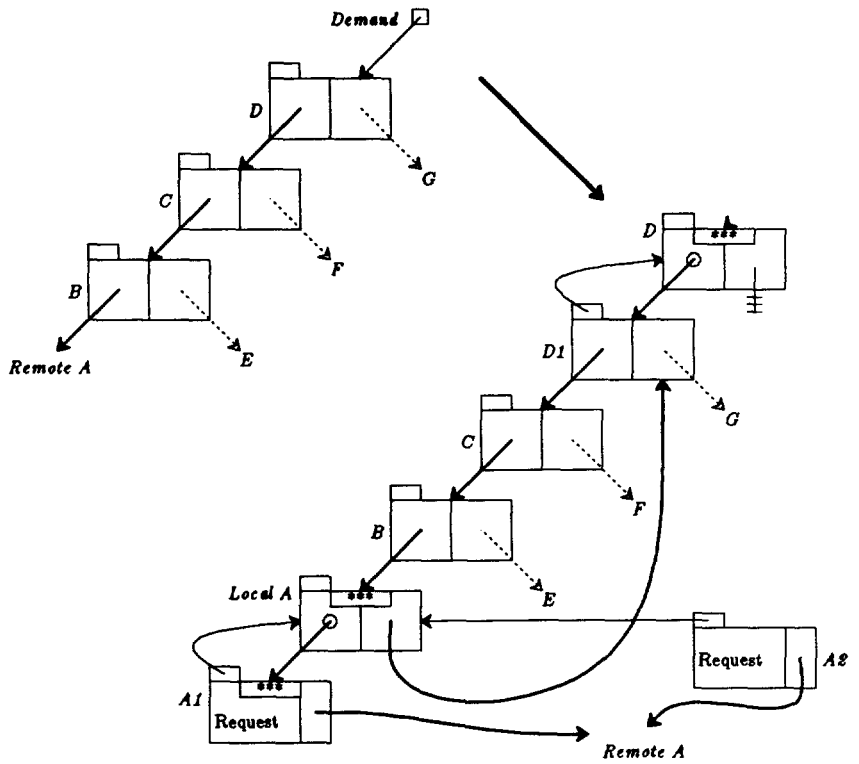

Figure 8.17 — Remote Reference Request. This involves five steps:

1. A *request packet* is created (*A2*), and the remote reference is transferred into its argument vector. Request packets are scheduled like any other packet except that the diffusion scheduler is not used to choose a processor. Instead, the request packet will be sent to the processor indicated by the remote reference, and that processor is required to accept and schedule it.
2. A marker node to reserve space for *Local A* is created, and the memory cell originally containing the remote reference is updated with a reference to *Local A*.
3. The reference to the marker is shared (copied with sharing of reference rights) into the redex address of the request packet.
4. As with other evaluation tasks, a copy of the request packet is made and stored in the task field of the marker.
5. The task count of the marker is initialized to one, and the request packet is sent.

Once the remote reference request has been sent, the originally demanded application *D* is suspended in the usual way.

CHAPTER 9

Conclusions and Directions for Future Research

9.1. Summary and Conclusions

This thesis has explored some specific techniques for scheduling location and processor allocation to tasks in order to achieve massively parallel, asynchronous computation on MIMD computers. It emphasizes automatic, dynamic decomposition of a program into concurrent tasks, in order to reduce the complexity of programming and to find parallelism that static techniques overlook. Techniques for control of dynamic task generation were also investigated. Some control is necessary in order to assure that important work is completed first and that resources are not swamped. Throughout these investigations it was crucial to understand the design and implementation considerations of a model that would scale well with increasing machine size.

Evaluation of expressions by concurrent graph reduction was chosen as the basic model of computation. Graph reduction has several characteristics that make it attractive for parallel evaluation.

- Opportunities for parallel evaluation can be discovered by direct examination of the form of the graph representing an expression.
- The results of an evaluation are deterministic, regardless of the order in which subexpressions are evaluated.
- Synchronization is simplified, both because of the deterministic property and because the functions which operate on the graph make manifest their accesses to data.

- The granularity of reduction operations can be chosen to match the requirements of an evaluation model and architecture.

The experiments described in this thesis have focused on fine-grained combinator operations. Combinator graphs contain numerous, potentially concurrent subexpressions that can provide work for the large number of processors available in a massively parallel machine. In addition, the individual tasks are small; that is, they can be compactly represented as data objects. This means that their images are easily transmitted and collected into pools of work that will keep processors busy while other tasks are suspended, awaiting data.

The thesis makes contributions on three distinct topics.

1. *A detailed mechanism for distributed graph reduction.* A message-driven protocol for task distribution and scheduling on a non-shared memory multiprocessor has been described and correctness proofs given for its principal functions. This includes a task deletion and garbage collection subsystem. The model treats the nodes of a program graph as virtual processors that exchange messages. This defines units of work that may be mapped onto a physical machine of any size.

2. *Experimental evaluation of a diffusion scheduling algorithm.* To map a program onto a network of processor modules, a dynamic scheduling algorithm called diffusion scheduling has been suggested. It uses a measure of workload as the analog of pressure to direct tasks to modules where they are most likely to receive prompt service. Collection of workload information and control of task distribution are both managed in a decentralized manner, by exchange of messages. This makes diffusion scheduling a good match for the message-driven evaluation model, in terms of both operation and scalability. Experimental studies of this

algorithm extend the existing body of work on distributed scheduling, and expose some of the pitfalls and dynamic behavior of these techniques. The simulation results indicate that a diffusion scheduling algorithm, properly tuned to the architectural parameters of a system, can find placements for dynamically created tasks that will effectively balance workload among multiple processing nodes.

3. *Analysis and experimental evaluation of speculative evaluation.* Speculative evaluation attempts the reduction of subexpressions whose values cannot be guaranteed to be useful, in order to stimulate concurrent activity. To provide control, a priority scheme was used to schedule speculative tasks and to aid in garbage collection of excess tasks. Performance of a multiprocessor using speculative evaluation was measured by simulation, which allowed a significantly wider range of variation of the multiprocessor network size than was available in a physical system.

This work has been concerned with studying the dynamics of the parallel evaluation model. This aspect of parallel systems has frequently been overlooked by other researchers. The experiments performed have provided insights into the behaviors of diffusion scheduling and speculative evaluation, and into the ways in which these two techniques can interact. By presenting a detailed model for massively parallel reduction, and by exploring the behaviors of this model and its attendant problems, this thesis has provided a basis on which future investigation in this area can be built.

Although graph reduction provides a conceptual framework for the model, its implementation required considerably more mechanism than was expected at the outset. New node types had to be added to the graph structures in order to account for state transitions induced by the receipt of messages. Priority scheduling required methods to modify priori-

ties, so that a high-priority task would not be forced to wait should it be found to depend on the result of a lower-priority task. Furthermore, the use of priority as a criterion for deleting excess tasks required a means of propagating the deletion notice to dependent tasks. It also required the system to be made robust under uncertainty that a previously requested task would ever complete. Success in meeting these challenges has been demonstrated by informal but rigorous proofs that the model behaves correctly.

A new result of this research is an analytic estimate of the number of tasks required to successfully mask communication latency. This analysis reveals that latency cannot be masked completely when the proportion of tasks that are themselves dependent on communicated parameters is greater than 60 percent. This is an important result, because it provides a measure that can be used to determine whether a given program or even a given computation model is appropriate for massively parallel asynchronous computation. Clearly, asynchronous parallelism thrives on independent tasks. Program analysis should seek to discover independent tasks and should report an estimate of the dependency parameter for the program. If a program shows a high degree of intertask dependency, it may require synchronous or systolic parallel evaluation to achieve speedup.

Experiments investigating the effectiveness of the speculative evaluation strategies indicate that progress has been made towards the goal of automatically discovering parallelism. Speculation is essential to achieve speedups by parallel evaluation of lazy functional programs if strictness analysis is not performed. This is the case even when the function-level parallelism of a program is high. However, the anticipated discovery of fine-grained parallelism within a program having little coarser-grained concurrency was not realized. The system also failed to produce significant speedups in programs whose parallelism lay primarily in speculative operations.

Much of the failure to see better speedups is probably due to sequentiality imposed by the lazy list construction operation. Furthermore, use of Turner's combinator set imposes artificial dependencies in accessing the arguments of multi-argument functions. Architectural aspects of the simulated system, such as a relatively inefficient message-passing system, may also be a factor in limiting speedups, but the data do not clearly indicate whether this is the case.

Efforts to control speculative computation have had mixed success with respect to performance. The priority scheme accomplishes the goal of completing useful work quickly, even when a large amount of useless speculation has been attempted. Throttling of speculation also prevents the overuse of system resources, but has a number of disadvantages. "Hot spot" behavior seems to be exaggerated by speculative computation, especially when the number of processors begins to exceed the number of nodes in the initial program graph. Task deletion to eliminate excess work causes eccentric performance in some cases. How rapidly tasks should be allowed to spread across the network has also not been determined. The optimal solution depends upon characteristics specific to each program, and it is difficult to find a policy that behaves equally well in all cases.

9.2. Directions for Future Work

Although the results of speculative evaluation presented in Chapter 7 show some promise, it is clear that the system is still greatly at the mercy of dynamic variations. Excess speculative work, multiple evaluations, and especially task deletion, all add complexity and uncertainty of performance to the distributed evaluation model. Further work is necessary to explore ways of reducing the occurrence of "hot spots" and to control nondeterministic performance. The few test cases that it was possible to study do not provide a complete picture of the range of behaviors. More experimental data is needed, especially for larger

problem sizes.

The drastic reduction in parallelism seen in the *hanoi* example suggests that task deletion may not be the best way to control flooding of resources. Although the capability is useful in extreme situations, some programs perform badly under the deletion of speculative tasks. Alternatives that avoid task deletion should also be explored. One possibility is to allow low-priority reduction packets to migrate again if high-priority work pushes them to the back of the ready queue. However, this may only flood the communication network instead of flooding processor resources.

9.3. Other Applications

This research has considered the application of speculative evaluation with priority scheduling to parallel execution of a single program. However, the use of global diffusion scheduling and local priority scheduling are not dependent on the evaluation of one graph at a time. Multiprogramming is used in the MPCR at the combinator level to supply each processor with a pool of tasks. The global system could be multiprogrammed as well, evaluating several disjoint graphs simultaneously. This would be best utilized when none of the programs being evaluated were of sufficient size to occupy the entire processor network. Associating a tag with the priority field of each node or message would also provide for a different "highest" priority within each graph. This would allow different programs to be assigned different starting priorities, as is common in multiprogramming operating systems. The tag would be used to determine which "low priority" tasks are eligible for deletion, because the speculative tasks of one graph might have the same priority as nonspeculative tasks of another.

Finally, it should be noted that these techniques are not restricted to graph reduction computations. Although graph reduction provides a convenient framework for identifying

concurrent tasks and assigning priorities to them, another method could easily be applied. The property of uniqueness of normal forms is also helpful when using speculative evaluation, because it means that all copies of a task will produce the same results. However, this property is not required if tasks that produce side-effects, such as performing I/O or updating a database, are never allowed to execute speculatively. That is, such a task must not be started by another task unless that task already has highest priority. Executing a task at highest priority is then equivalent to *committing* to the operation. Any synchronization required to order the side-effects can be built into the functions that implement them. The MPCR simulator, in fact, makes use of this technique to guarantee output ordering when printing lists.

References

[AHU74] Aho, A. V., Hopcroft, J. E. and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Menlo Park, California, 1974.

[Arl88] Arlauskas, R., "iPSC/2 System: A Second Generation Hypercube," *3rd Conference on Hypercube Concurrent Computers and Applications*, vol. I, Architecture, Software, Computer Systems and General Issues, January 1988, pp. 38-42, ACM.

[ArI85] Arvind and Iannuci, R. A., "Two Fundamental Issues in Multiprocessing: The Dataflow Solution," Computation Structures Group Memo 226-3, Massachusettes Institute of Technology, Cambridge, Massachusettes, August 7, 1985.

[ArC86] Arvind and Culler, D. E., "Dataflow Architectures," in *Annual Reviews of Computer Science I*, 1986, pp. 225-253.

[AuJ88] Augustsson, L. and Johnsson, T., *Lazy ML user's manual*, Chalmers University of Technology, Gothenburg, Sweden, May 1988.

[Bai86] Bain, W. L., *The Interwork Programmer's Reference Manual*, Block Island Technologies, Beaverton, Oregon, 1986.

[Bai88] Bain, W. L., *The Interwork II Programmer's Reference Manual*, Block Island Technologies, Beaverton, Oregon, 1988.

[BaS85] Barak, A. and Shiloh, A., "A Distributed Load-balancing Policy for a Multicomputer," *Software—Practice and Experience*, vol. 15, 9, September 1985, pp. 902-913.

[BrF81] Bryant, R. M. and Finkel, R. A., "A stable distributed scheduling algorithm," *Proceedings of the 2nd International Conference on Distributed Computing Systems*, 1981, pp. 314-323. Discussed in NXG85.

[BHA86] Burn, G. L., Hankin, C. L. and Abramsky, S., "Strictness analysis for higher order functions," *Science of Computer Programming*, vol. 7, 1986, pp. 249-278.

[Bur88] Burn, G. L., "A shared memory parallel G-machine based on the evaluation transformer model of computation," in *Proceedings of the Workshop on the Implementation of Lazy Functional Languages*, Aspenas, Goteborg, Sweden, September 1988.

[BPR88] Burn, G. L., Peyton-Jones, S. L. and Robson, J. D., "The spineless G-machine," in *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, Snowbird, Utah, 1988.

[Bur85] Burton, F. W., "Controlling Speculative Computation in a Parallel Functional Programming Language," *Proceedings of the 5th International Conference on Distributed Computing Systems*, Denver, Colorado, May 1985, pp. 453-458.

[ChA82] Chou, T. C. K. and Abraham, J. A., "Load Balancing in Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-8, 4, July 1982, pp. 401-412.

[Chu41] Church, A., *The calculi of lambda-conversion*, Princeton University Press, Princeton, New Jersey, 1941.

[CuF58] Curry, H. B. and Feys, R., *Combinatory Logic*, vol. I, North-Holland, 1958.

[Dal86] Dally, W. J., "A VLSI Architecture for Concurrent Data Structures," Ph.D. Thesis, California Institute of Technology, 1986.

[DaS87] Dally, W. J. and Seitz, C. L., "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, vol. C-36, 5, May 1987, pp. 547-553, IEEE.

[DaR81] Darlington, J. and Reeve, M., "ALICE—A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages," *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1981, pp. 65-75.

[ELZ86] Eager, D. L., Lazowska, E. D. and Zahorjan, J., "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, 5, May 1986, pp. 662-675.

[GoH87] Goldberg, B. and Hudak, P., "Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor," in *Proceedings of the Los Alamos/MCC Graph Reduction Workshop*, Lecture Notes in Computer Science, Springer-Verlag, 1987. From a draft dated November 1986.

[Gol88] Goldberg, B. F., "Multiprocessor execution of functional programs," YALEU/DCS/RR-618, Department of Computer Science, Yale University, April 1988. Pre-publication version of chapters 7 and 8.

[GKW85] Gurd, J. R., Kirkham, C. C. and Watson, I., "The Manchester Prototype Dataflow Computer," *Communications of the ACM*, vol. 28, 1, January 1985, pp. 34-52.

[Gus88a] Gustafson, J. L., "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, 5, May 1988, pp. 532-533.

[Gus88b] Gustafson, J. L., "The Scaled-Sized Model: A Revision of Amdahl's Law," in *Proceedings of the Third International Conference on Supercomputing 1988:*

Technology Assessment, Industrial Supercomputer Outlooks, European Supercomputing Accomplishments, and Performance & Computations, vol. 2, ICS 88, St. Petersberg, FL, 1988, pp. 130-133.

[Hil81] Hillis, W. D., "The Connection Machine," Technical Report 646, Massachusettes Institute of Technology AI Laboratory, September 1981.

[Hil85] Hillis, W. D., *The Connection Machine*, MIT Press, Cambridge, MA, 1985.

[HiS86] Hillis, W. D. and Steele, G. L., "Data Parallel Algorithms," *Communications of the ACM*, vol. 29, 12, December 1986, pp. 1170-1183.

[HiS87] Hillis, W. D. and Steele, G. L., "Update to 'Data Parallel Algorithms'," *Comm. ACM*, vol. 30, 1, January 1987, pp. 78. Half page letter..

[HsL86] Hsu, C. H. and Liu, J. W., "Dynamic Load Balancing Algorithms in Homogeneous Distributed Systems," *6th International Conference on Distributed Computing Systems*, Cambridge, Massachusetts, May 1986, pp. 216-223.

[HuG84] Hudak, P. and Goldberg, B., "Experiments in Diffused Combinator Reduction," *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, August 1984, pp. 167-176.

[HuG85a] Hudak, P. and Goldberg, B., "Distributed Execution of Functional Programs Using Serial Combinators," *IEEE Transactions on Computers*, vol. C-34, 10, October 1985, pp. 881-891.

[HuG85b] Hudak, P. and Goldberg, B., "Serial Combinators: 'Optimal' Grains of Parallelism," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Nancy, France, September 1985), J. Jouannaud (ed.), Lecture Notes in Computer Science, vol. 201, Springer-Verlag,

New York, 1985, pp. 382-399.

[HuY86] Hudak, P. and Young, J., "Higher-order strictness analysis for untyped lambda calculus," in *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, ACM, 1986, pp. 97-109.

[HuM88] Hudak, P. and Mohr, E., "Graphinators and the Duality of SIMD and MIMD," in *Proceedings of the ACM Conference on Lisp and Functional Programming*, ACM, August 1988, pp. 224-234.

[Hug82] Hughes, R. J. M., "Super-combinators: A new implementation method for applicative languages," *Proceedings of the ACM Symposium on LISP and Functional Programming*, Pittsburgh, August 1982, pp. 1-10.

[Jef85] Jefferson, D. R., "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, 3, July 1985, pp. 404-425.

[JBW87] Jefferson, D. R., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H. and Bellenot, S., "Distributed Simulation and the Time Warp Operating System," *Operating System Reviews*, vol. 21, 5, 1987. Proceedings of the Eleventh ACM Symposium on Operating Systems Principles.

[KeL84] Keller, R. M. and Lin, F. C. H., "Simulated Performance of a Reduction-Based Multiprocessor," *IEEE Computer*, vol. 17, 7, July 1984, pp. 70-82.

[KLT84] Keller, R., Lin, F. C. H. and Tanaka, J., "Rediflow Multiprocessing," *Proceedings of Compcon Spring 84*, February 1984, pp. 410-417.

[KeK79] Kermani, P. and Kleinrock, L., "Virtial Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, vol. 3, 1979, pp.

267-286.

[Kie85] Kieburtz, R. B., "The G-machine: A fast, graph-reduction evaluator," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Nancy, France, September 1985), J. Jouannaud (ed.), Lecture Notes in Computer Science, vol. 201, Springer-Verlag, New York, 1985, pp. 400-413.

[KLB89] Kingdon, H., Lester, D. R. and Burn, G. L., "The HDG-Machine: A Highly Distributed Graph Reducer for a Transputer Network," *Draft submitted to FPCA 89*, Middlesex, UK, March 7, 1989.

[Kus86] Kuszmaul, B. C., "Simulating Applicative Architectures on the Connection Machine," Masters Thesis, Massachusetts Institute of Techonology, May 1986.

[LiK86] Lin, F. C. H. and Keller, R. M., "Gradient Model: A Demand-Driven Load Balancing Scheme," *6th International Conference on Distributed Computing Systems*, Cambridge, Massachusetts, May 1986, pp. 329-336.

[LiK87] Lin, F. C. H. and Keller, R. M., "The Gradient Model Load Balancing Method," *IEEE Transactions on Software Engineering*, vol. SE-13, 1, January 1987, pp. 32-38.

[LiM82] Livny, M. and Melman, M., "Load balancing in homogenous broadcast distributed systems," *Proceedings of the ACM Computer Network Performance Symposium*, 1982, pp. 47-55. Discussed in Gol88.

[NXG85] Ni, L. M., Xu, C. and Gendreau, T. B., "Distributed Drafting Algorithm for Load Balancing," *IEEE Transactions on Software Engineering*, vol. SE-11, 10, October 1985, pp. 1153-1161.

[NiH85] Ni, L. M. and Hwang, K., "Optimal Load Balancing in a Multiple Processor System with Many Job Classes," *IEEE Transactions on Software Engineering*, vol. SE-11, 5, May 1985, pp. 491-496.

[Pap87] Papadopoulos, G. M., "Implementation of a General Purpose Dataflow Multiprocessor," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, 1987.

[PaC90] Papadopoulos, G. M. and Culler, D. E., "Monsoon: an Explicit Token-Store Architecture," in *Proceedings of the 1990 Conference on Computer Architecture*, IEEE, May 1990, pp. 82-91.

[Pey87] Peyton-Jones, S. L., *The Implementation of Functional Programming Languages*, Prentice-Hall International, Englewood Cliffs, NJ, 1987.

[Pey88] Peyton-Jones, S. L., "The Spineless Tagless G-Machine," *Proceedings of the Workshop on Graph Reduction, Aspenas, Switzerland*, Sweden, September 1988.

[Pey89] Peyton-Jones, S. L., "Parallel Implementation of Functional Programming Langauges," *The Computer Journal*, vol. 32, 2, October, 1989, pp. 175-186.

[PfN85] Pfister, G. F. and Norton, V. A., "'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers*, vol. C-34, 10, October 1985.

[Sta84] Stankovic, J. A., "Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms," *Computer Networks*, vol. 8, 3, June 1984, pp. 199-217, North-Holland.

[Ste72] Stenlund, S., *Combinators, λ-Terms, and Proof Theory*, D. Reidel Publishing Company, Dordrecht, Holland, 1972.

[SuB77] Sullivan, H. and Bradshaw, T. R., "A large scale homogenous machine," in *Proceedings of the 4th Annual Symposium on Computer Architecture*, 1977, pp. 105-124.

[TBH82] Treleaven, P. C., Brownbridge, D. R. and Hopkins, R. P., "Data-Driven and Demand-Driven Computer Architecture," *Computing Surveys*, vol. 14, 1, March 1982, pp. 94-143.

[Tru89] Truve, S., "The Massively Parallel G-Machine," *Draft submitted to FPCA 89*, Gotebog, Sweden, March 10, 1989.

[Tur79] Turner, D. A., "New implementation techniques for applicative languages," *Software—Practice and Experience*, vol. 9, 1, January 1979, pp. 31-49.

[WaH87] Wadler, P. and Hughes, J., "Projections for strictness analysis," in *Functional Programming Languages and Computer Architecture*, G. Kahn (ed.), Lecture Notes in Computer Science, vol. 274, Springer-Verlag, New York, 1987, pp. 385-407.

[WaM85] Wang, Y. and Morris, R. J. T., "Load Sharing in Distributed Systems," *IEEE Transactions on Computers*, vol. C-34, 3, March 1985, pp. 204-217.

[WWW86] Watson, I., Watson, P. and Woods, V., "Parallel Data-Driven Graph Reduction," in *Fifth Generation Computer Architectures*, J. V. Woods (ed.), North-Holland, 1986.

[WaW87a] Watson, P. and Watson, I., "An Efficient Garbage Collection Scheme for Parallel Computer Architectures," *Proceedings of the European Conference on Parallel Architectures and Languages*, Eindhoven, The Netherlands, June 1987.

[WaW87b] Watson, P. and Watson, I., "Evaluating Functional Programs on the FLAGSHIP Machine," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Portland, Oregon, September 1987), G. Kahn (ed.), Lecture Notes in Computer Science, vol. 274, Springer-Verlag, New York, 1987, pp. 80-97.

[Whi85] Whitby-Strevens, C., "The Transputer," *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, MA, June 1985, pp. 292-300. IEEE, 1986, pp. 320-328."" Reproduced in Dharma P. Agrawal's (ed.) "Advanced Computer Architecture," IEEE, 1986, pp. 320-328..

APPENDIX A

Simulation Parameters

Table A.1 summarizes the important parameters used in the MPCR simulator to perform the experiments described in Chapter 7. Times were assigned based on a correspondence of one time unit (one Interwork II clock tick) to one reduction operation, where a reduction operation is a simple calculation plus an update of one graph node. The simplest combinators thus execute in unit time. More complex combinators were assigned times based on the number of computations and updates they required. Time for other operations was then set by comparing the complexity of their implementations to those of the various combinators.

Table A.1 — Simulation Parameters

Operation	Time	Notes
Perform reductions:		1
B	3	
C	3	
I	1	
K	1	
P	1	
S	5	
U	3	
Y	2	
Cond	1	
Nil	1	
Arithmetic	1	2
Comparison	1	3
Remote Request	2	4
Process messages:		
Pressure	1	
Data (Update)	2	5
Evaluate (Packet)	1	6
Delete	1	
Pass message between CPU and transmitter	1	
Pass message between adjacent transmitters	2	7

Notes:

1. Time for reduction of each combinator is based on the number of allocations and updates required to perform the graph manipulation.
2. Arithmetic combinators are Add, Sub, Div, and Mult.
3. Comparison combinators are Eq, Gt, and Lt.
4. Time shown is for a simple remote request, where the requested graph node is already evaluated. Time varies if evaluation must be demanded (in which case the request suspends until it completes) or if the request must be forwarded to another processor.
5. Base time for processing a data message does not include awakening tasks from the notifier list.
6. Evaluate messages are complete packets being distributed for execution. The time shown represents only the time to accept or reject the packet, and to add it to the ready queue if it is accepted. Additional overhead is required if the packet is rejected, because it must be reprocessed as an outgoing message.
7. This does not include block time caused by message routing collisions.

APPENDIX B

The Lambda Compiler

Programs used in the experiments described in Chapter 7 were written in a simple untyped lambda calculus language. The syntax of this language is summarized in Figure B.1. A yacc-generated parser, called *lamb*, reads the Lambda program and translates it into an intermediate expression language having only abstractions and applications. Each subexpression in this form is explicitly tagged as either application or abstraction. The transformations given by Turner [Tur79] are used to perform this translation.

The intermediate form can be defined directly in *LML* [AuJ88] as a recursive data structure. An LML program, called *cbc*, creates such a structure from the yacc output, then translates the data structure into combinator expressions, again following Turner. The resulting expressions are loaded into *C* data structures by another yacc parser, and finally distributed to the simulated processors for execution.

As a check on the correctness of compilation and of the program output produced by the simulator, a different LML program, *lbev*, can replace *cbc*. This program performs the same translation as *cbc*, but instead of printing the finished combinator expression, it transforms it into an LML application graph using functions defined for each of the combinators. The application graph is then executed directly as LML code to produce the output specified by the original Lambda program.

---

where-expr → *where-expr* `where` *id* = *application-expr*
| *where-expr* `where rec` *id* = *application-expr*
| *application-expr*

application-expr → *function-application*
| − *application-expr*
| *application-expr arithmetic-op application-expr*
| *application-expr comparison-op application-expr*
| *application-expr* , *application-expr*

arithmetic-op → * | / | % | + | −

comparison-op → = | < | >

function-application → *function-application simple-expr*
| *simple-expr*

simple-expr → *lambda-abstraction*
| *conditional-expr*
| (*where-expr*)
| *identifier*
| *integer*
| `nil` | `[]`
| `null`
| `fst`
| `snd`

conditional-expr → `if` *application-expr* `then` *application-expr* `else` *application-expr*

lambda-abstraction → λ *identifier*. *application-expr*

Figure B.1 — Syntax of the *Lambda* Language. Where the grammar is ambiguous, alternatives are given in order of decreasing precedence. Function application by juxtaposition of expressions has highest precedence. Arithmetic operators have the usual algebraic precedence.

The canonical empty list is named `nil` (also `[]`). The function `null` tests whether its argument is an empty list. Lambda programs are not type-checked, so the operator "," (comma) is used for both list and pair construction.

APPENDIX C

Simulator Data Structures

The first operation performed by the simulation is to construct an initial graph from the input combinator expression. This initial graph consists entirely of application nodes. Some cons/pair nodes could have been constructed at compile time, but our objective is to study the behavior of the simulation, not to produce an optimally compiled graph. The basic structure of application nodes, which is shared by cons/pair nodes, is shown in Figure C.1. The node types are differentiated by their *Tags*.

Each *Memory Cell* contains either a canonical, a combinator name (*descriptor*), or a reference to a graph node. Additional tags associated with each cell are used to identify the contents. The format of a memory cell is shown in Figure C.2.

The *Redex Address* field of the application node is initially unused, and is in fact required only for a subset of node types. For simulation purposes, however, all nodes are given a redex address field so they may be handled orthogonally. The redex address is a

---

Application, Cons, or Pair Node

Priority	Redex Address	Reference Count	Tags
Memory Cell		Memory Cell	

Figure C.1 — Format of a Basic Graph Node

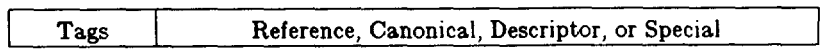

Figure C.2 — Format of a Memory Cell

reference to a place-holder node which is to be updated when the node containing the address has been evaluated to weak-head normal form. Normally, this place-holder (or *marker*) represents the root of a reducible expression; hence the term *redex address*. Due to the nature of reduction task distribution, redex addresses are always *remote* references.

The *Reference Count* field records the total number of *reference rights* held for the graph node, and is used in garbage collection.

The *Priority* of each node is also recorded. Priority is meaningless for nodes in weak-head normal form (boxed values or cons/pair nodes) but is required for application nodes.

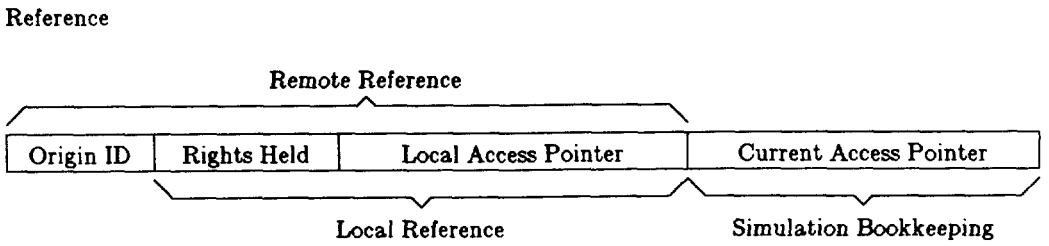

Figure C.3 — Format of a Reference

The format of a reference is shown in Figure C.3. References contain at minimum a count of the *reference rights* they hold and a pointer for local access. *Remote* references must also contain a field that identifies the processor node where the value of the referred-to graph node can be obtained. The *Local Access Pointer* of a reference is valid only at the origin processor node of the reference. For purposes of the simulation, a *Current Access Pointer* is also maintained; this is actually an address descriptor in the global namespace provided by *Interwork II*, and must be used to obtain the local pointer. Also for simulation purposes, all references are treated as remote, and the *Origin ID* field is checked on each reference to the current access pointer to be sure that the global namespace is not misused.

Two additional node types can be formed during expression evaluation. These are *boxed values* and *indirections*. Indirections can also be formed to provide additional reference rights, as described in the discussion of the reference rights garbage collector. The formats of boxed value and indirection nodes are similar to the basic graph node format, and are summarized in Figure C.4 and Figure C.5.

One additional type of graph node is created by the reduction system. This node type is called a *reduction packet* and represents the basic unit of work in the reduction engine. A reduction packet consists of a combinator name (descriptor), an argument count, and a

---

Boxed Value

Priority	Redex Address	Reference Count	Tags
Unused		Canonical or Descriptor	

Figure C.4 — Format of a Boxed Value

Indirection

Priority	Redex Address	Reference Count	Tags
Unused		Reference	

Figure C.5 — Format of an Indirection Node

variable-sized argument list. The argument list is made up of memory cells and may be of any size, but for ease of manipulation and update the simulation limits it to at most three cells, the maximum number of arguments required by any of the chosen set of combinators. This limitation is also desirable because it keeps messages small. A packet thus contains exactly the information needed to perform a single reduction step.

A reduction packet *must* contain a redex address and a priority, but its reference rights may be omitted†, since there is guaranteed to be only a single reference to any reduction packet (see the discussion of packet formation, below). One bit in the reference rights field of the reference to a reduction packet is used to record the information that this is an exclusive, nonshared reference; it cannot be shared until the packet has executed and thereby been transformed back into an expression graph, at which time the reference count information can be extracted from the reference itself and stored in the expression graph. Instead of a reference count, then, a reduction packet carries information that is examined and modified by the diffusion scheduler at each processor visited by the packet. This ***diffusion data*** is used by the heuristic algorithm that decides whether the packet will be

† The simulation actually does record the reference rights of reduction packets, and examines them for error detection.

accepted at any given node (see Chapter 5). The format of a reduction packet is shown in Figure C.6.

Reduction Packet

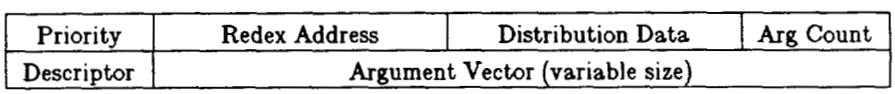

Figure C.6 — Format of a Reduction Packet

Biographical Note

The author was born and raised in Iowa, making his entrance in Ames on January 3, 1962, and spending most of his life in Des Moines and Urbandale. The author has never lived on a farm, and potatoes are not a major crop in his home state. Immediately after his graduation from Valley High School of West Des Moines, he was awarded the William Fletcher King Scholarship by Cornell College of Mt. Vernon, Iowa. He began studies at "the one that's *not* in Ithaca" in 1980.

During his time at Cornell, the author received the Chandler Award, the Norton Award, and the Barton Award. The latter has nothing to do with the author's first name. In 1984, he received the Degree of Bachelor of Special Studies in Computer Science.

The author began his studies at the Oregon Graduate Center in the fall of 1984, and saw the Pacific Ocean for the first time shortly thereafter. He was a Clark Foundation Fellow in 1985. The highlight of 1988 was the author's June marriage to the former Maija Kuhlman, at the Living History Farms in Des Moines. So perhaps he has lived on a farm after all. The Graduate Center renamed itself as the Oregon Graduate Institute in 1989, and the author completed the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering at OGI in 1990.

Interests of the author include, in no particular order, distributed operating systems, science fantasy novels, parallel algorithms and languages, long walks on the beach, functional programming, fantasy role-playing games, and computer graphics and graphical interface systems. He is leaving the Graduate Institute to try his hand at entrepreneurial enterprise, as a drastic change from ten years of higher education.