COMMUNICATION PATTERNS IN A SYMBOLIC MULTIPROCESSOR

by

PETER ROBERT NUTH

May, 1987

This research was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under contract numbers N00014-83-K-0125 and N00014-84-K-0099.

COMMUNICATION PATTERNS IN A SYMBOLIC MULTIPROCESSOR

by

Peter Robert Nuth

Submitted to the Department of Electrical Engineering
on May 14, 1987 in partial fulfillment of the requirements
for the degrees of [illegible] and Master of
Science in Electrical Engineering.

Abstract

An important design decision for large scale multiprocessors is the balance of processor power to communication network bandwidth. In order to evaluate different design alternatives, it is necessary to [illegible] the communication load imposed on the network by a programming model.

This thesis quantifies that communication load for a model of parallel symbolic computing using the Multilisp language. An organization of a shared memory multiprocessor for Multilisp is proposed. The [illegible] simulator is built to model that organization. Several [illegible] programs are run under Nusim, and the communication [illegible] of each program is measured. The locality of reference of memory accesses by each [illegible] is determined for three proposed multiprocessor topologies. The effect of scheduling decisions in increasing locality of access and in [illegible] is studied. The thesis concludes with implications of [illegible] on the design of parallel computer systems.

Thesis Supervisor:

Robert H. Halstead Jr.
Associate Professor of Computer Science and Electrical Engineering

Key Words and Phrases:

Data reference, Lisp, locality, multiprocessor, symbolic processing.

Acknowledgements

I would like to thank Bert Halstead for all that he has taught me over the last few years. His enthusiasm and support made this project a reality.

I would like to thank all the members of the Parallel Processing Group for creating a friendly and interesting place to work. I would especially like to thank Randy Osborne for using the simulator, offering constructive criticism and being a merciless proof-reader. Thanks as well to Juan Loaiza for being a ready source of information on so many topics.

I thank my officemates, Dan Nussbaum and Liz Bradley, for stimulating conversation, friendship, and occasional unrestrained pandemonium.

I would like to thank Nancy Daly, for making my life so much better these last few years.

Finally, I would like to thank my parents and family, who started me along this path, and never lost faith in me along the way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

The power of present day computer systems is approaching the limits attainable by conventional computer organizations. One solution to this problem lies with architectures for parallel processing, notably multiprocessor systems. However in order to exploit the power of these new organizations, users must have access to languages and programming models that allow algorithms to be executed in parallel. *Multilisp* [31] is one such programming model. It is a language designed for symbolic computation on a parallel computer system.

Multilisp is based upon a dialect of the language Lisp [41,19] with additional constructs for parallelism. A version of Multilisp has been written that runs both on conventional computers [8,6] and on an experimental multiprocessor system [9,33]. The present implementations of Multilisp are efficient enough to run programs of moderate size. However, the real potential of Multilisp is in running on a machine specially designed for the language.

A Multilisp machine would be a large scale, shared memory, MIMD[1] multi-

[1]Multiple instruction, multiple data machine. All of the processors in this machine can operate independently, running different instructions on different data.

processor. The preliminary design of such a machine is a topic of research in the Parallel Processing Group at M.I.T. This machine could contain an estimated 500 to 1000 processors that would be connected together through a fast communications network.

There are a number of different design decisions in the organization of such a Multilisp machine. One of the most important ones is to decide what type of communication network to use to connect the processors. Another is to set the size, speed, and the amount of local storage associated with each processor. These two decisions involve an economic and engineering tradeoff. A designer could invest effort in optimizing the communications network, or could increase the speed of the individual processors at the expense of the network.

The tradeoff between communications network and individual processing node depends on where the system bottlenecks will be. Different programming languages or different application programs might impose a much different load on the system. Some models of computation achieve parallelism by partitioning the program data among the individual processors. For certain classes of applications, this allows each processor in the system to work on a similar sub-task, with little communication between the processors. Another model of computation might have each processor perform a different logical function, and use message passing to communicate the results of one function to another. These two models impose different requirements on the speed of the communications network versus that of the individual processors.

In order to evaluate different design decisions about the organization of a Multilisp multiprocessor, we need to know the communication requirements of Multilisp programs. This includes determining the types of data that Multilisp programs access, and how that data might be distributed across a real multiprocessor.

1.1.1 Goals

The primary goal of this thesis is to determine the type of accesses made by Multilisp programs, and the spatial distribution of those accesses in a proposed machine organization. This goal has two parts: one is to determine the real communication requirements of Multilisp programs, and the second is to see how much the cost of that communication can be reduced by appropriate scheduling decisions.

Multilisp programs touch many different types of data. Some of this data is shared between parallel *tasks* running on different processors, some could potentially be accessed by several processors, and some is completely private to a processor. In some Multilisp machine organizations, private data could be kept in the local storage of a processor. Accesses to data that is shared between different processors is the 'real' communication needed by an application program. Each of these accesses requires a transaction across the communications network. A goal of this thesis is to determine this maximum real communication cost for a variety of Multilisp programs. This will form a basis for decisions about system organization.

A Multilisp implementation has some flexibility in how to dynamically schedule tasks and allocate data in a multiprocessor system. While the same program will always touch the same data objects, it may be possible to reduce the distance of those accesses. An allocation strategy could cluster data objects near the processors that refer to them most often. Task scheduling algorithms could reduce the distance that tasks move across the system. A second goal of this thesis is to see how these different scheduling decisions affect the cost of communication for Multilisp programs.

There are several ways of quantifying the cost of communication on a Multilisp machine. One is the number of data accesses that a program makes. The second is the *locality* of those accesses. The locality of data references indicates how closely data is clustered to the processor that accessed it. This thesis tries to quantify this locality of reference as a way of measuring the effect of scheduling decisions on communication cost.

1.1.2 Justification

Future multiprocessors will attempt to harness the power of hundreds to thousands of individual processors. As machines get bigger, communication costs increase. To some degree, it is possible to increase the speed of communication networks by using more complex connection strategies. The speed of the network is fundamentally limited by the speed of light. However, most practical networks are limited first by economics.

In a large network, distant accesses may take a long time. Accesses may contend for communication paths, and 'hot spots' could develop in the network. These points of contention are unpredictable, and can slow down the network considerably. On the other hand, communication between a processor and its local memory is much more predictable. Computer architects have much more experience in building fast processor – memory pairs than in designing large networks. If the processors in a multiprocessor system all run at full speed, it likely that communication networks will become a bottleneck.

There have been a number of different approaches to avoiding this bottleneck. The literature contains many discussions of network topologies intended to be fast, non-blocking and cheap to build at the same time [24,49,5]. Other networks reduce contention for resources by combining accesses to the same data object [26,47,14]. Some memory systems are able to perform complex operations, to synchronize accesses to shared data, or to enforce mutual exclusion [25,10]. Many systems shuffle memory addresses in an effort to distribute data as evenly as possible through the system [15,14]. This is a case where designers work very hard to **avoid** data locality, by forcing a random distribution of data throughout the system. Finally, some systems treat the communications network as the single most important shared resource, and reduce the speed of all other system components to match the network

[25].

All of these ideas are ways of coping with a high rate of global accesses across a network. However, if global communication is expensive, it makes sense to try to reduce the load on the network as much as possible. If a multiprocessor has local memory associated with a processor as well as global shared memory, there is a benefit to allocating data in local memory. Not only can that memory run faster than distant accesses, but using local memory will reduce the contention for global resources.

In some kinds of networks, the time necessary for a particular access is proportional to the distance that that reference must travel. In this case, the communication load can be reduced even further by increasing the locality of reference to data objects. Though using more sophisticated data allocation and scheduling algorithms may take more time than simple random scheduling, it may be worth the complexity to reduce the total communication requirements of a program.

In order to evaluate the effect of these scheduling strategies, we need to quantify the real communication load of different benchmarks. Then the difference in communication load due to sophisticated scheduling gives us a measure of the performance gain.

1.2 Experimental Method

I started this thesis by proposing a model of a multiprocessor system. There are two components to this model: the first is the language that will run on the machine, and the second is the physical organization of the multiprocessor.

I decided to use Multilisp as the programming language, and to concentrate on applications in symbolic computing. The implementation of Multilisp that I used is not much different from existing versions of the language.

I described a possible organization for a multiprocessor in enough detail to predict how its memory system would respond to different types of accesses. Using this model as a base, I built *Nusim*, an architectural simulator for this machine.

The Nusim simulator directly executes Multilisp programs. It simulates one processor of a multiprocessor at the level of functional blocks. By running one copy of the simulator on each processor of an existing multiprocessor system, I simulated the proposed multiprocessor organization.

Nusim allows a user to vary a number of parameters in the implementation of Multilisp. It can use several different strategies for scheduling tasks. Nusim can also simulate different topologies of processing nodes. Finally, a user can vary some of the internal characteristics of a Multilisp processor in Nusim.

I used Nusim to simulate several different processor interconnection topologies. I ran a set of Multilisp application programs on these simulated topologies. I counted the types of data references that the programs made, the locality of that data, the amount of parallelism and the task handling behavior of the programs. I then varied a number of machine parameters within Nusim and saw what effect these variables had on the locality of reference of the benchmarks. These experiments were used to predict the influence of these parameters on a future Multilisp machine.

1.3 Chapter Outline

Chapter 2 discusses some of the problems involved in building a parallel Lisp machine. It then presents a brief description of the programming model used for this thesis, namely Multilisp.

Chapter 3 discusses some of different design decisions in the organization of a symbolic multiprocessor. It then presents the model of a multiprocessor that was used in this thesis. It concludes by justifying that this style of multiprocessor can be built using conventional technology.

Chapter 4 describes the Nusim simulator and some of the details of its operation.

It discusses different parameters that affect the operation of Nusim.

Chapter 5 describes the test programs that were run under Nusim. It describes the variables to Nusim that were modified in different runs of the test programs. It discusses what kinds of data were collected from these different runs, and how that data was presented. Chapter 5 then presents the results of the experiments with Nusim. For all the data presented, I have tried to point out any trends, and to discuss the reasons for that behavior. Chapter 5 concludes with a summary of what we have learned about Multilisp programs and Multilisp machines from these experiments.

Chapter 6 concludes, discussing the relevance of this data. It also discusses what questions remain unanswered, and what experiments would be useful to build upon these studies.

Appendices follow, giving details of the implementation of Nusim, how the data presented in Chapter 5 was collected, and full descriptions of the benchmark programs.

Chapter 2

Multilisp Architecture

2.1 Discussion of Scope

In order to simulate the operation of a Multilisp multiprocessor, we must first start with an idea of how that machine might be built. The design of a machine architecture for Multilisp is tailored both to the requirements of the language and to the inefficiencies in a large multiprocessor. We begin by presenting the Multilisp language and the special features it requires. Chapter 3 discusses the organization of a Multilisp machine in more detail.

We justify a design by simulating its performance in running its intended workload. Engineers often have a good understanding of the code that runs on conventional computers. Many machines are designed to be compatible with an existing body of code [8]. In such cases, an architect can study an existing design, find the bottlenecks, and propose incremental changes to speed up the system.

However, in the case of Multilisp, there is no existing processor design to use as a standard. There are a small number of applications written in Multilisp, but no one has studied the code to find the most common functions that need to be speeded up. The lack of performance measurements is particularly bothersome since Multilisp is unlike conventional numeric programming languages,[1] and is not intended to run on conventional architectures. Multilisp also has features that distinguish it from other dialects of Lisp. It is unclear how many of the lessons of conventional computer architecture apply to Multilisp.

A processor could be designed at several levels, from a general description of its organization to the level of circuit diagrams. I have specified the processor itself only in enough detail to model its behavior in running Multilisp programs. I have outlined the structure of the processor, and the functional units of which it is built. I have simulated the processor as a black box that executes assembly language programs. By making assumptions about how the processor handles each such instruction, it is possible to predict the performance of the system in running large applications.

2.2 Building a Parallel Lisp Machine

There are two classes of problems in building a parallel Lisp machine. The first deals simply with the problem of executing Lisp efficiently. Traditionally, Lisp has been a difficult language to implement on standard computer architectures. [48] The second is one common to many multiprocessor organizations: how to manage parallel tasks and communicate with other processors.

2.2.1 Running Lisp Efficiently

Most dialects of Lisp assume the existence of run-time type checking and garbage collected heap memory. On machines without special purpose hardware, these tasks require a large amount of processing time. Computers specifically designed to execute Lisp often use a typed data architecture, in which each word of memory is tagged with the type of data that it contains [43]. While the processor is operating

[1]See [32] for an explanation of what distinguishes symbolic computing from numeric processing.

on the data part of words, it can simultaneously check the tags to ensure that the objects are of the correct types. This added complexity in the processor hardware thus removes most of the overhead of run-time type checking.

Symbolic computing emphasizes sorting and selecting of data objects, rather than numeric functions of that data. Therefore, operations such as pointer following, procedure calls, and creation of lists are much more common in Lisp-like languages than the tight loops and arithmetic functions of other languages. Furthermore, the flexibility of Lisp, its emphasis on late-binding of procedures, and its use of untyped, generic operators reduces many of the optimizations possible at compile time in typical programming languages. Most data operations in a Lisp program must be able to deal with exceptional conditions and multiple data types.

There are three ways that Lisp implementations handle this need for flexibility at run time and frequent use of complex operations. Some implementations of Lisp are interpreted, rather than compiled, hiding the complexity of the basic operations within the interpreter itself [38,52]. Others compile down to relatively complex assembly language instructions [40,43,22]. though there has been at least one attempt to compile Lisp to run on a very simple Load/Store architecture [54].

2.2.2 Parallelism Issues

A processor for a parallel machine must be able to deal with a number of problems created by the organization of the system. The model of computer that we have been using in our research is that of a homogeneous, shared memory multiprocessor. The goal is a system containing several hundred processors. In a large system of this type, it is not feasible to have all processors fully connected by high-bandwidth paths. Thus, the average interprocessor latency must increase with the number of processors in the system. One would hope that there will be enough locality in data references that the communications network will not be swamped with traffic. But even so, an individual processor must be able to operate efficiently even if there is a high latency in accessing some areas of memory.

Finally, any programming model for a multiprocessor system must support multiple, concurrent tasks. In Multilisp, tasks are created and destroyed at run-time. They are also dynamically moved from one processor to another during the course of computation. Typically, each processor in the system may have access to a pool of tasks that it can run. Thus, a processor must be able to manipulate tasks efficiently, to select one from amongst a set of tasks, and to transfer tasks to other processors. This may require hardware support for manipulating a processor's tasks. It also requires scheduling mechanisms for distributing tasks around the system. In this thesis, I investigated a number different approaches to scheduling tasks.

2.3 Design Components

There are two components to the design of any computer architecture: the hardware itself, and the programming model that it is intended to support. Neither can be proposed independently of the other, since there must be a continual tradeoff between complexity in hardware and software.

In a computer system that must support many diverse programming languages, it is difficult for any one application to determine the hardware design. Features that might speed execution of one programming model might impede another. This is also a common argument for complex instruction set architectures.[2] However, in computers that are only intended to run one application language, the hardware should be tailored as much as possible to the specifics of the language. This is expressed most forcefully in such innovative architectures as the *Connection Machine*, where constructs and operators in the **Lisp* language are closely coupled to the internal structure of the machine [36].

While most Lisp machines use this software-specific approach to hardware design

[2]See [18] for a statement of this argument.

[43,40], a number of Lisp implementations have recently been developed that run on more conventional architectures [22,16]. These designs invest much more effort in software technology, such as optimizing compilers and explicit type declarations for data objects. The benefit is that they can run on a much simpler, and presumably faster, basic processor.

The limit of this hardware-driven approach is probably best personified by RISC processors [54]. These designs expect that type mismatches and error conditions will be relatively rare. This architecture promotes a style in which the user declares the type of most Lisp objects in the program. By assuming that the type of these objects will not vary, the compiler can generate code without many expensive run-time type checks. Here is an example of where the programming model that will run on a computer is constrained by the limits of the underlying hardware.

2.4 Programming Model

2.4.1 Priorities

The overriding goal of research in the Parallel Processing Group at M.I.T. is efficient general purpose multiprocessors. The group is pursuing research towards this goal in three areas: languages, architectures, and applications. It is not enough to build a fast computer architecture if it is difficult to program the machine to use that power. We must have languages to exploit parallelism, and enough experience with application programs to be able to find the parallelism in a particular algorithm. Most of the group's effort is now concentrated in symbolic processing, and specifically the *Multilisp* language.

Multilisp is based on the Lisp dialect *Scheme* [19]. It shares with the latter an exclusive reliance on lexical scoping, rather than the dynamic scoping of most older Lisp dialects [41]. It also allows procedures to be passed freely as arguments or results of procedure calls, treating them as it would any other value.

Multilisp most unique feature is the manner in which it allows a user to explicitly indicate areas of potential parallelism in a particular algorithm [31]. This is in keeping with the philosophy that the programmer is still the best judge of where it is possible to exploit concurrency. One of the topics of research in the group is the possibility of building intelligent compilers that can make reasonable decisions about where to insert parallelism constructs. One such system has been built that was successful with functional Lisp code [27], but is not yet able to deal with arbitrary programs.

Many other concurrent Lisp languages are exclusively functional, that is, they do not allow side-effects. Multilisp takes the view that there are many applications that are easier to write and more efficient with side-effects. A programmer is encouraged to write mostly functional programs, with short sections that contain side-effects well insulated from the rest of the code. However, we are unwilling to restrict the user's ability by outlawing those mechanisms.

The final fundamental principle of Multilisp is that the language should shield the user from details of the underlying hardware. Multilisp is intended to be portable to a number of different architectures, both sequential and parallel. Any language that requires a user to explicitly partition code or data structures across the available processors will not be portable as the number of those processors changes. Since Multilisp presently runs on a number of different types of machines [8,9,6,43] initial indications are that that it has been successful in this goal.

2.4.2 Brief Description of Multilisp[3]

The most unique feature of Multilisp is the *future* construct. The form `(future <expr>)` immediately returns a future object, a distinguished token for the value of `<expr>`. It also spawns a process to compute that value. The future object is a promise that the result of `<expr>` will be available at some later time. A procedure

[3]For a more complete description of Multilisp, see [31] or [34].

can 'touch' a future object by attempting to read its value. This occurs when we operate on the future with an instruction that is strict in its arguments. At that point, if the value of the expression **<expr>** is still not determined, the task that touched it will be suspended until the future is resolved.

Variants of the future construct are the only way of expressing parallelism in Multilisp. At some point in the execution of a Multilisp program, there may be many concurrent tasks in existence. Each task was produced to calculate the value of a future's expression. We say that the *goal* of the task is to determine the value of a future. Once it has computed that value, the task re-starts all tasks that were suspended on that future, and ceases to exist. From that point on, the determined future is equivalent to any other Lisp object.

Many different implementation strategies are possible for futures, depending on how we wish to schedule the underlying tasks. Seeing the effects of those different scheduling strategies is one of the goals of this thesis. Section 4.1.5 contains a more detailed description of the Nusim implementation of futures.

Chapter 3

Machine Organization

This chapter presents a set of assumptions about the structure of a Multilisp machine that I used in simulating its behavior. In a multiprocessor such as this one, we must make a basic set of decisions about the global organization of the system, as well as of processing nodes within that organization.

3.1 Hardware Hierarchy

Figure 3.1 shows the hierarchy of subsystems in the multiprocessor that we are proposing. The highest level is the hardware organization of the system. In our case, this is a network of identical processing nodes connected together by a communications network. These processing nodes are composed of three components: the processor itself, some local memory associated with that processor, and a communications port to connect to the other nodes in the system.

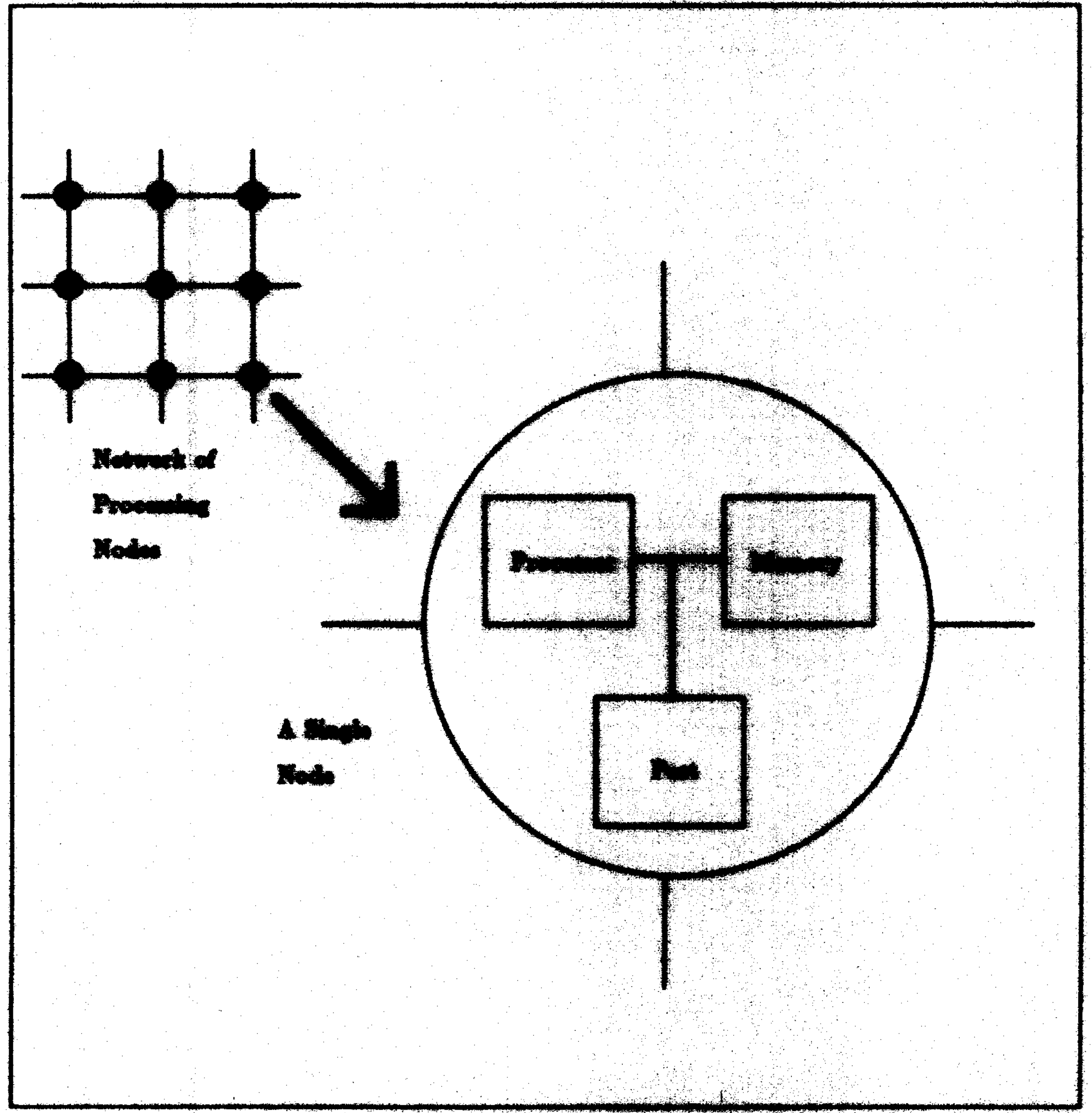

Figure 3.1: The hierarchy of subsystems in a multiprocessor.

3.1.1 System Organization

Issues

The system organization level determines what parallelism will be available in the system. The most fundamental decisions about how to design a multiprocessor are made here. Among them are:

- The number of processors in the system.
- The granularity of the processing nodes.

  These first two points are usually related, since we want to build a machine of a certain size. One approximation to the size of the machine is the area of silicon used to build it. This is the product of the size of a processing node (the granularity) and the number of nodes in the system.

  The size and granularity can range from two conventional mainframes on a common bus [2] to a million one-bit wide processors [36].

- The arrangement of memory and processing.

  We might choose to segregate processors and memories in a 'dance-hall' model, and connect them through a cross-bar switch [55], or to spread processing and memory evenly throughout the the system [36]. Most large systems associate some local memory with each processor, to try to reduce the amount of communication in the system.

- The degree of connectivity of the nodes.

  This determines the *diameter* of the network, that is how many hops are required to communicate from any one processor to any other point in the system. There is a tradeoff between the connectivity of the system and the cost and complexity of building the network. In some systems, each processor is directly connected to every other, either through a dedicated link [49], or through a shared broadcast bus [42]. Other structures only allow a node to communicate with its nearest neighbors [28,29].

- Shared memory versus message passing.

  Some programming models assume that all processors can read all memory in the system [31]. Others explicitly assume that processors communicate through message passing [37]. Some multiprocessing organizations are designed specifically to speed the form of communication used by a particular programming model [56,7,49]. In a multiprocessor designed specifically to support message passing, without any areas of shared memory, it is costly to emulate a shared memory mode of operation. In a practical machine, we would want dedicated hardware to handle requests for data, and avoid burdening a distant processor for each memory fetch.

  Similarly, inter-processor communication can be expensive in a shared memory machine. Here, processors synchronize by way of locks and semaphores in main memory. But a processor can waste many cycles 'spinning' on a particular lock, waiting for it to clear. If the lock is in distant memory, this puts a substantial load on the communications network. Again, hardware can alleviate some of the cost of this form of communication.

  In conventional computers, a processor 'owns' the connecting bus for the duration of a transaction to memory. However, in a multiprocessor, where many processors share the same communications path, it is too expensive to deny everyone access to the network for the duration of an access. This is especially true in a large system, where there is a long latency to distant memory. This is the reason for split-transaction buses, where a request for data is not immediately followed by the answer from memory. This type of read and write request can be considered a special case of message passing.

- Connections to I/O devices.

The problem of supporting a high input/output bandwidth is particularly important for machines that are intended to support users in a stand-alone manner. When the computer will be used in an interactive fashion, users are not willing to spend a long time loading a sizable amount of code and data onto the multiprocessor in order for it to run some complex program.

Discussion

Many of the assumptions that I have made in these topics are motivated by Prof. Halstead's proposal of a *Myriaprocessor* [28,29]. This is a view that large scale multiprocessors should be easily expandable and reconfigurable. He envisions a network of tens of thousands of identical processing nodes, all connected to their nearest neighbors. It should be easy to vary the number of processors in the system transparently to the user and the application program. In effect, we should be able to buy "Computing by the Yard" [1] to suit a particular application.

The processor that I propose here might not be suitable for such a myriaprocessor, but for a machine that we could build as the next step in that direction. I assume that this intermediate machine will have a modest number of processors — on the order of 500 nodes. The size of the system, and the need for it to contain a variable number of processors, constrains the network used to interconnect them.

With such a large system, it would be impractical to separate processors and memory by some large cross-bar switch [55]. The size and complexity of the switch would be too great. Every non-local access by a process would be forced to pay the maximum cost, by being routed through one common device. Such a design has just replaced the "Von Neumann Bottleneck" by one just as severe. Likewise, it would be impractical to build a fully-connected network of processing nodes. The cost and complexity of the wiring would dominate the design.

Instead, I propose a network of processing nodes, where the processor, memory

[1] This analogy is attributed to Steve Ward.

and switching mechanism are distributed in space. All nodes connect to nearest neighbors— the number of neighbors to be determined by the topology. A mesh in two dimensions would have each node connect to four other nodes, a three dimensional network connects six nearest neighbors. There is a tradeoff between the complexity of the network and the average latency as we move to higher dimensions.

This thesis is not concerned with the details of the communications system linking the processing nodes. It assume that any processor may read the memory of another node. What is important is how long that access takes. It is too early yet to be concerned with issues of loading of communications links or hot spots in the network.

This thesis assumes that the latency to distant parts of the system is relatively high. The speed of propagation of messages is fundamentally limited by the speed of light in such a large system. However, the system could have high throughput if it allows many requests to run through the network simultaneously. For these reasons, requests for data and the corresponding replies should take the form of short messages. The messages would traverse the network one hop at a time, moving from node to node.

A processor design may have to deal with the problem of processors sitting idle while waiting for replies to data requests. This could potentially waste a large percentage of the power of the system. It would be useful if processors could perform other useful work while waiting for a reply to some request. We will return to this point in Section 3.1.4.

With this design, we are attempting to exploit locality of reference in data requests, assuming that a computation running on a processor is likely to need data clustered close to that processor, rather than in distant memory. That way, the relatively low bandwidth connections between nodes will not saturate with messages. This is a risky matter in Lisp programs, which usually exhibit less locality than conventional languages. [43,48]. In this thesis, I tried to quantify the degree of locality available in our test programs, to see whether our assumptions are worthwhile.

In this model of the communication network, I have deliberately ignored the mechanisms by which messages are propagated through the system, the physical connections and the routing strategy. I also have not looked at the effect that contention for paths might have on bandwidth. I have ignored the effects of errors, lost packets, and fault tolerance in the nodes and communication channels. These topics each merit a discussion which is outside of the bounds of this thesis.

In this thesis, we do not consider the effects of memory management or virtual memory. While we do not presume any memory management facility in the processing nodes, there is no reason to assume that the address space of any processor need be the same as the system as a whole. It would be relatively easy to have the communications port in each processing node translate data references to system-wide addresses. Similarly, we will avoid the complexity of virtual memory by assuming that all data and code needed by an application is kept in real memory. Given the potential size of this multiprocessor, and the amount of memory that it can contain, this is not an unreasonable assumption.

Finally, any large machine will probably need to communicate with input/output devices. For the purposes of this thesis, I assume that we can use a front-end processor to load code and data into the system, and to support software debugging. The host might connect to all the processing nodes through some common broadcast net. That communication path would not be used during normal processing on the computer.

Other I/O devices can be associated with particular processing nodes, in effect, allowing memory mapped I/O. One processing node can control access to each I/O device, mediating requests from the rest of the system. We do not expect to use high bandwidth devices that might be a load on the system. Access to disk drives or secondary storage is only critical for paging purposes, which we do not expect to see in normal operation.

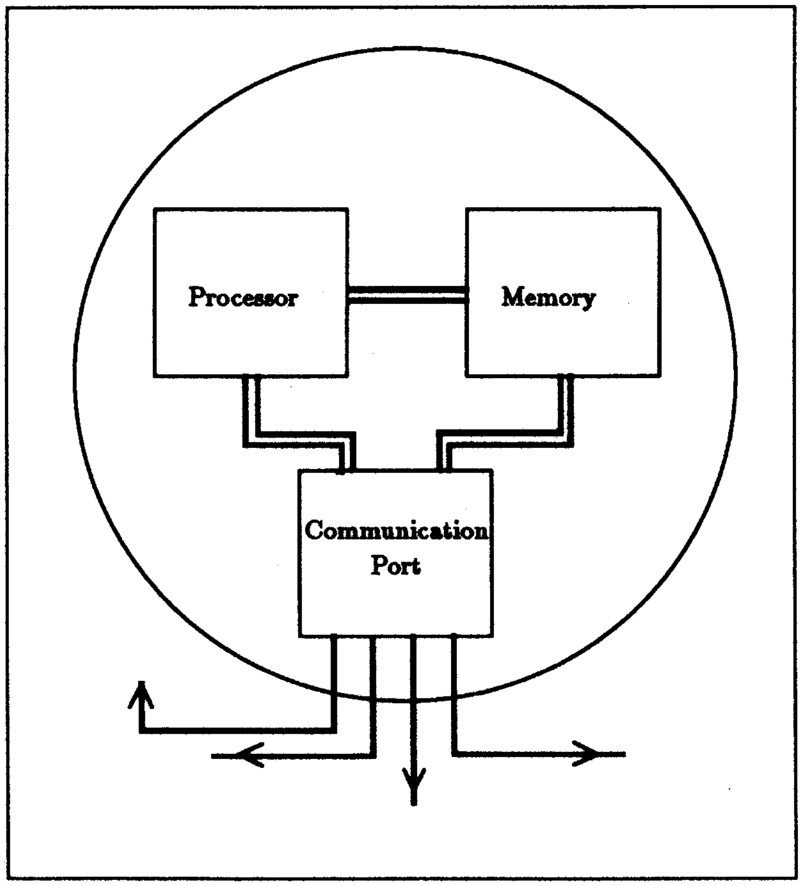

Figure 3.2: Structure of a processing node.

3.1.2 Processing Node

As mentioned in Section 3.1.1, the system proposed here is divided into identical processing nodes. A possible structure for a node is shown in Figure 3.2. The three components are the processor, local memory and a communications port.

Accesses to distant memory in a Multilisp machine may have considerable latency. Storing data locally rather than in distant memory will prevent paying this cost for some types of objects. The percentage of accesses that processors make to local memory rather than distant depends on the locality of reference of Multilisp programs. In this thesis, I have shown that there is considerable locality to be exploited in a range of Multilisp applications.

Communications Port

In such an MIMD machine, we expect that a processing node will run as an isolated unit much of the time. One role of the communications port in this machine is to decouple the operation of the processing node from the rest of the system. The port is the only link between processing nodes. A port in each node will connect through the network to the ports of some number of other processing nodes.

The communication port must be able to route messages from the local processor out to its destination. The port also translates incoming network messages into accesses to node local memory. In this machine, all routing decisions must be made within the communication ports. This decouples the design of the external network from the internals of the processing node. The communication port must be a fairly intelligent box in order to handle this functionality. Putting some intelligence in the port relieves the node processor of the burden of handling communications overhead.

Node Organization

The processing node is organized so that there are two paths to node memory. This is to support accesses both from the local processor and from distant requests. When there is a conflict for the memory, the local processor would have priority. Since the latency for local accesses is expected to be much less than that of distant accesses, it is more important that local accesses are unhindered.

Figure 3.2 also shows a direct path from the node processor to the communication port. This would carry interprocessor messages with low latency. While the Multilisp language does not explicitly use message-passing, there are some details of its implementation that would benefit from methods of synchronizing the processors.

3.1.3 Memory Hierarchy

The memory of a Multilisp machine would be divided into three classes. The highest speed memory would be internal to a processor. This includes processor registers and caches for code and data. This memory is typically quite small, since in any technology it is difficult to build large very fast memories. Node local memory is the next fastest, and should be large enough to contain all the code and data being referenced by a processor over the lifetime of a computation. Finally, the node memories of all other nodes in the system are accessible to the local processor as global, distant memory.

Processor Internal Memory

An instruction cache or instruction buffer is a standard way of speeding up code execution on a processor. Since code does not change in most high level languages, instructions in a cache will remain consistent with the global memory. Even a simple buffer can speed code fetches, by matching the speed of the processor to that of the memory system. The benefit of an instruction cache depends on the size of the cache, and on the types of instruction references that Multilisp programs make. The cache hit ratio is greatest when instruction references have a high degree of locality.

Another type of memory that is internal to a processor is a stack buffer. Current implementations of Multilisp are designed to run on stack machines. For such an implementation, a stack buffer would hold the top of the stack in high speed memory. Since most instructions currently fetch their operands from the stack, some type of buffer is necessary for acceptable performance.

Local and Distant Memory

Multilisp assumes that all processes in the system share a common address space. Some portion of that space would be mapped to local node memory, while the rest would require distant memory accesses. A goal of this division of memory would be to encourage programs to use local memory for as much data as possible. A program should only have to access global memory for variables that it shares with other processes, or for other necessary interprocessor communication.

In this thesis I have proposed a number of ways of encouraging this locality of access. For instance, all code and constants in a Lisp program could be loaded into the local memory of each processing node. Programs should default to allocating space out of local memory, and any private or short-lived Lisp objects should be stored locally as well.

Accesses to global memory are more difficult than fetches out of local memory, because no simple bus links processors to distant nodes. We have assumed that the communications network supports split transaction, packet or message based communications protocols [21]. Experience with other large scale multiprocessors suggests that accesses through the communications network might be an order of magnitude slower than those to local memory [5,25,15].

3.1.4 Processor Model

In order to accurately predict the performance of a Multilisp machine, we must make some assumptions about its processor architecture. Section 2.2 discussed some of the difficulties in running Lisp on a multiprocessor, and suggested hardware solutions to speed Lisp processing.

Tagged Architecture

One of the ways to speed execution of an untyped language like Multilisp is to run on a tagged architecture. Every data word in the system is tagged with a label that identifies the type of data in the word. Instructions must then check the tags of their operands to ensure that the object types match the operation to be performed. The processor must be able to trap to routines in microcode or assembly language to handle exceptional cases.

While this tag checking can be performed in software, it is much more efficient to have processor hardware check data tags. Studies have shown that even a small amount of processor support for tag checking can yield a significant improvement in performance [54]. In this thesis, we will assume that processors check the types of data objects in parallel with instruction execution. This means that in the normal case, instructions are executed at full speed. However, when data types do not match the operation to be performed, the processor aborts the instruction and traps to an exception handler.

Support for Multiple Tasks

The significant feature of Multilisp as a programming language is that it allows programmers to explicitly spawn parallel tasks. We will assume that processors contain a number of features to speed task handling.

First, processors must devote some effort to finding and loading executable tasks. This thesis discusses a number of algorithms that the processors may use to schedule tasks. We will assume that some of these functions are built into the hardware or the microcode of a processor. While tasks are loaded much less frequently in Multilisp than the instruction execution rate, the extra overhead in trapping to assembly code may be prohibitive. The most basic scheduling functions must exist at a low level in the processor. Support for futures in the processor must include a combination of data tag checking and task scheduling.

In our model of a communications network for a Multilisp machine, all accesses to distant memory have a long latency. One way of dealing with this latency is by keeping several processes loaded in a processor at any one time. So instead of sitting idle during this access period, a processor could switch to running another process. Some machines have tried to support multiple concurrent tasks, and to allow very fast context switching between the tasks [50,46]. This can be an expensive proposition, since it often requires allocating a different set of processor registers to each task. In this thesis, we have assumed that processors have this ability to multitask at a low level. We investigated the effect of this multitasking on parallelism and data accesses.

3.2 Justification

The preceding sections outlined some basic assumptions about the design of a Multilisp machine used in this thesis. But while all these features may be desirable for a Multilisp machine, if the resulting system is too large and complicated to be built, we have not accomplished anything. In this section I will argue that it is possible to build a machine with these characteristics using conventional technology.

There have been a few large scale multiprocessors built to date. The B.B.N. Butterfly [5], is built using 256 conventional microprocessors, tied to memory through a 'butterfly' switching network. All system memory is shared among all the processors. There are no fundamental engineering reasons why a 500 to 1000 processor machine could not be built using the same structure.

The processor architecture that we have proposed is different from most commercial microprocessors. It is similar to the processor of a CADR Lisp machine [40]. But some single chip processors based on this same architecture have been built. Texas Instruments has designed a Lisp machine on a chip that contains most of the functionality of the processor proposed here [57].

In any machine design, a large part of the complexity of the machine is not related to the number of chips that it contains, but instead the number of distinct chip types, and the patterns in which they are connected. Some large machines have been built using a few complex chips, connected in a regular structure [56,36]. The Connection Machine is built using a thousand VLSI chips, each of which contains 16 simple processors. A few high-density memory chips provide all the local memory required by group of processors. By matching the processor to the structure of the system, this design uses very few MSI support chips. It is reasonable to assume that a thousand complex single chip processors, could be connected in a similar regular structure.

Chapter 4

Simulation Method

4.1 The Nusim Simulator

4.1.1 Purpose

The core of this thesis is a study of the behavior of Multilisp programs, and of the effect of different architectural features on Multilisp execution. In order to collect this data, I wrote the *Nusim* simulator. Nusim was intended to be a flexible test bed for studying the architecture of symbolic multiprocessors. Though I will spend only one chapter describing this tool, realizing it occupied most of the time spent on this thesis.

Nusim is derived from and structurally similar to *XML*, the Multilisp emulator written by Robert Halstead and Juan Loaiza of the P.P.G. group at M.I.T. It was extensively rewritten and restructured to take the present form of Nusim. In addition, I added customizable routines that allow Nusim to simulate the operation of a processor, not just to emulate the Multilisp language. In fact, only a small amount of this flexibility was used in order to collect the data summarized in Section 5.5. However I hope that some of the additional features of Nusim will prove useful to others in the group, and to my own research in the future.

The sections that follow begin by defining a number of terms that will be used in later discussions of the Multilisp language and of Nusim. They also discuss how those components are handled in Nusim.

4.1.2 Concert

The *Concert* multiprocessor [9,33] was designed as a development system for parallel processing. Figure 4.1 shows a block diagram of the machine. It is composed of up to 34 Motorola MC68000 processors [1], and approximately 25 megabytes of memory. Concert is a tightly-coupled multiprocessor in which system global memory is shared between all processors.

Concert is divided into eight *slices*, each of which is a separate Multibus backplane [39]. A slice can hold between four and six processors. Each processor is given at least one 500K byte memory board as local memory. Local memory is visible to all processors in the slice, but not to processors on other slices. The processors and memory boards were a commercial design [3,4].

The slices of Concert are linked together by the *Ringbus*, a segmented shared bus. Several transactions can take place simultaneously on disjoint segments of the Ringbus. A central *Ringbus Arbiter* controls access to the Ringbus by each of the eight slices. It tries to provide fair access to the bus, and to support as many bus transactions as possible at any time. The Arbiter is a custom design by members of the P.P.G. group, as are the *Ringbus Interface Boards* that connect slices to the Ringbus.

4.1.3 Running on Concert

XML and Nusim run both on uniprocessors [8,6] and on Concert. The program itself only runs on a single processor. On Concert, one copy of the program runs on each processor in the system. Each processor that runs Nusim simulates one

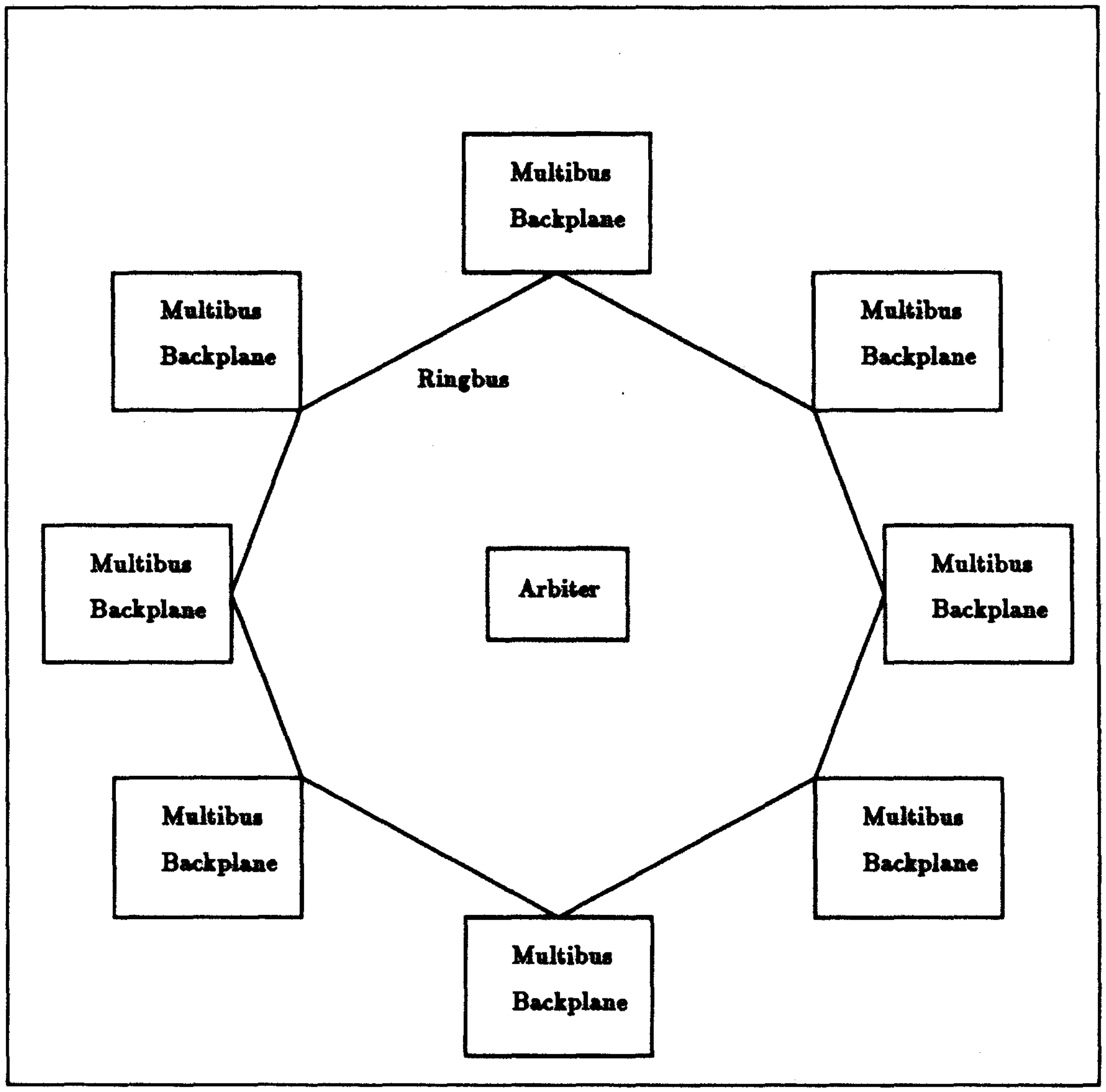

Figure 4.1: The Concert multiprocessor.

processing node of a hypothetical multiprocessor. While this limits the number of processing nodes that we can simulate with Nusim, it is much simpler than allowing each real processor to simulate several 'virtual' processors.

In Nusim processors communicate by side-effecting objects in shared global memory. For instance, there is a single global list of free memory blocks. A processor that requires more memory will lock that list, pop a block from the top of the list, and unlock it. This operation is duplicated for most resources in the system. Nusim makes no attempt to coordinate the operation of processors at a lower level than this.

4.1.4 MCODE

Both the XML implementation of Multilisp [30] and the Nusim simulator compile Lisp source code down to an 'assembly language' known as *MCODE*. MCODE is a machine language for a hypothetical stack machine. Most MCODE instructions are zero-address, that is, they pop their operands off the stack, and push the result on top of the stack. The stack is used for local data, arguments and the environment of a procedure, and for the control state of procedure calls. MCODE contains the usual mathematical operations, branching and calling instructions, instructions to access data structures in memory, and instructions to spawn parallel tasks. Appendix A describes MCODE instructions in more detail.

The Nusim program spends most of its time in a loop, reading MCODE instructions, and dispatching to the appropriate procedure to emulate each instruction. There are approximately 120 MCODE instructions in the current implementation of Nusim. MCODE instructions are currently encoded with one or two byte opcode, optionally followed by some bytes of immediate arguments to the instruction. (Appendix A describes these instructions, and how they are encoded).

Nusim allows a designer to easily add instructions, or to change the encoding of this machine language. The simulator uses a single dispatch table to describe the encoding of each instruction and what arguments it requires. At present, Nusim can run one of two different instruction formats. One is the format used by the *XML* emulator, the second is a more compact format, more suited to implementation on a real processor. In the interests of sharing the Multilisp compiler, assembler, and other system code, all the test cases described here used the standard *XML* format.

4.1.5 Tasks and Processes

In Multilisp, programmers explicitly label expressions that should be executed in parallel with the rest of the program by enclosing them in *futures*. Each concurrent thread of execution in Multilisp is referred to as a *task*. A task that is loaded and ready to run on a processor is referred to as a *process*. A typical task might be the size of a Lisp procedure. Multilisp programs do not fork off parallel tasks to execute single instructions, because the overhead involved in spawning each task would be greater than the potential benefit in speed. Of course, in typical Multilisp programs, very few of the expressions might be enclosed in futures, for the same reason.

When a Multilisp processor spawns a task, it could choose to continue running the parent task or devote computation to the child instead. The definition of the Multilisp language allows a user to choose several different constructs to express parallelism. These variants of the *future* instruction make different choices about which sub-task to run, and how much effort to devote to it. See [34] for more information. The test programs that we ran for this thesis only used the 'basic' *future* instruction. In Nusim, a form such as:

```
(funca (future (funcb op2)) op1)
```

will spawn a task to calculate the value of `(funcb op2)`, and begin running that task right away. The subtask is given a higher priority than the original parent task.

4.1.6 Task and Process Queues

While a Multilisp processor is running a child process to calculate the value of a future, it must save away the parent state. There are two places where the parent task could be saved: a processor's *task queue* or *process queue.*

The basic storage structure for tasks in Nusim is the *task queue.* Each processor in the Concert system maintains such a queue. In the absence of any additional mechanism, tasks that are spawned and not run are pushed onto the *task queue* of the processor that generated them. These task queues are accessible to all other processors in the system. Tasks move around the system when a processor steals a task from a distant processing node. Nusim treats the task queue as a *LIFO* queue, since processors always pop the most recent task off the top of the queue.

One of the design decisions that I wished to study with the Nusim simulator was whether it is worthwhile to have several processes loaded on a physical processor, each ready to run. That way, if one process stalls while waiting on a distant access, we can immediately switch to running another process. To this end, each processor in Nusim also maintains a *process queue.* A process on the queue is either running, or waiting to be run by the processor. While a processor's task queue is visible to every other processor in the system, the process queue is kept in private, local memory. The larger the size of the process queue, the easier it is for a processor to context switch between a number of active processes. However, this restricts the amount of parallelism in the system, since while one processor has many processes in its local queue, other processors may starve for work.

4.1.7 Task Handling

In Multilisp, tasks are spawned dynamically by a running program. Nusim devotes some effort to scheduling these tasks among the available processors. A common problem with systems that allow tasks to be produced dynamically is that of **too much** parallelism. Since each task requires memory, if we do not set a limit on the number of tasks produced, we would quickly run out of resources in the system. Nusim uses an unfair scheduling policy to limit the number of parallel tasks that a program will produce [30].

The basic strategy for scheduling tasks is as follows: When a processor spawns a task due to a *future*, the processor switches to running the new process. The parent process will be left on the local process queue, ready to run. If the process queue is full, we must spill some process out of this queue, turning it into a task that is put on the local task queue. In this way, the process queue acts as a 'cache' of recent tasks.[1] In Nusim, process queues contain only the most recently executed processes. Nusim swaps out the least recently run process from process queue to task queue.

If a processor has no processes to run, it must search task queues in the system. Since there is a task queue associated with each processing node in the system, we could choose a number of different search strategies when checking those queues. A typical strategy might be to check our own queue first. Then, if this queue is empty, we might search the task queues of nodes that are successively farther away from us. Once we find a task in one of the queues, we load it into our process queue, and then start running the process.

4.1.8 Process and Exception Handling

During the execution of a process, there are a number of things that could happen to disrupt the normal flow of execution. These *exceptions* include touching a future, creating a future object, Lisp errors, and various low-level interpretation errors. In Nusim, all of these errors are handled in the same manner.

[1]Of course, we may choose to make the process queue be of length one. In this case, the process that we are running will be the only one in the process queue. All other processes are off-loaded to a task queue. Since tasks queues are visible to the whole system, no processors should starve for lack of work.

Nusim maintains a structure with each process that is in the process queue. This *micro-state* structure contains the entire state of the process. If we get an exceptional condition while running a particular process, we load information about the error in the micro-state, and bubble back up to a single routine that handles all errors. This method of encapsulating the state of each process makes it easier to handle the error condition, and to restart the process afterwards. We expect that in a real processor, the micro-state for each process would be a set of registers. Switching processes is as simple as changing a single pointer to that register set.

4.1.9 Stacks and Environments

Chapter 3 described a possible memory hierarchy for a Multilisp machine. Since MCODE is designed to run on a stack machine, one of the components of that hierarchy is a *stack buffer*. This is intended to be a small high-speed memory local to the processor, along with some index registers. The Multilisp stack contains procedure call linkage information, stack frames for each procedure, procedure arguments and local variables. Since the stack buffer would likely be a small memory, it could only hold the top-most stack frames in the current procedure invocation tree.

Since programs may call arbitrarily many procedures, periodically the stack buffer will fill with stack frames. A Multilisp machine would need to flush old stack frames out to main memory. Similarly, as procedures return, the stack buffer will empty out, and stack frames must be paged back into the buffer.

Nusim simulates this stack buffer in software. On Concert, Nusim simulates a stack buffer 50 words long. A stack buffer is part of the micro-state for each process in the process queue. Since each process has its own stack buffer, it is not necessary to flush out any buffers on context switches. However, as the stack buffer overflows, Nusim saves older portions out to heap storage. There it maintains a linked list of stack *hunks*. In the current implementation of Nusim, the hunks are 3/5 the size of the buffer. Note that Nusim allocates space for stack hunks, as with any Lisp object, out of the local processing node's memory. This should encourage some locality of reference for the hunks.

Multilisp is a lexically scoped language. Every *let* statement or procedure declaration builds a block in which a number of local variables are defined. Most operations find their operands in those local variables to a procedure. Each of these lexical blocks in Multilisp is known as an *environment frame*. The environment frame contains all bindings for variables defined in this block. It also contains a pointer to the lexically enclosing block's environment frame. Every variable reference in Nusim refers to a slot in a particular lexical environment frame. Top level variables, or *globals*, are kept in a global symbol table, accessible to all routines.

In Multilisp, procedure values are first-class objects. However, since Multilisp is lexically scoped, an expression that builds a procedure object cannot simply return a pointer to the procedure code. It must point to the bindings of free variables that the procedure uses. This is known as a 'closure' in Multilisp.

Closures objects might be passed around a program as any other object would. A closure can exist long after the procedure that created it has returned. In fact, the entire call stack that existed at the time when the closure was created might disappear, but the lexically bound variable references in the closure still must be valid. So a closure must encapsulate both the code for a function and the lexical environment that existed when the function was created. For this reason, environments are allocated in the heap, where they will remain long after the call stack has been destroyed.

4.1.10 Memory Structure

Nusim attempts to simulate the ideas of shared global memory described in Chapter 3. It divides all global memory in the Concert system into equal sized parcels. In a real Multilisp machine, global memory would be divided among the processing nodes in the system. Thus, while global memory would be accessible to all the processors in the system, it would be fastest to access from the local processor.

In our implementations of Multilisp, a processor will access data from all parts of the system, but only allocate new data objects in its local memory. In simulating a potential machine, we could have simply divided all global memory on Concert among the available processors. This would represent the memory local to a processing node. However, because Concert is a small machine, and all processors do not use memory at the same rate, we would run out of local memory on a Concert node long before a real machine would.

In order to deal with this difficulty, and to simulate a processing node with variable amounts of memory, we do not statically allocate memory in Nusim. Instead, each processor that needs a block of memory obtains it from a global memory pool. We then consider that block of memory to be 'owned' by that processor.

4.1.11 Topology

As stated in Chapter 3, we have proposed a multiprocessor system for Multilisp which is composed of identical processing nodes connected together through some communications network. It is very expensive to build such a system using a single, fully connected communications network. More likely, processing nodes in the system will be organized into loose hierarchies, or directly connected to some subset of the remaining processors. Figure 4.2 shows some examples of possible topologies for a Multilisp computer system.

Due to the size of the system, it is likely that communication between nodes widely separated in this network will take a long time. Rather than slowing all accesses between nodes to take the same amount of time, we expect the latency of communication between processing nodes to increase as we step to more distant destinations. This is an important concept for a Multilisp computer, since it means that we should encourage communication between nodes that are 'close' together, and try to limit accesses to distant nodes. One of the main goals of this thesis was

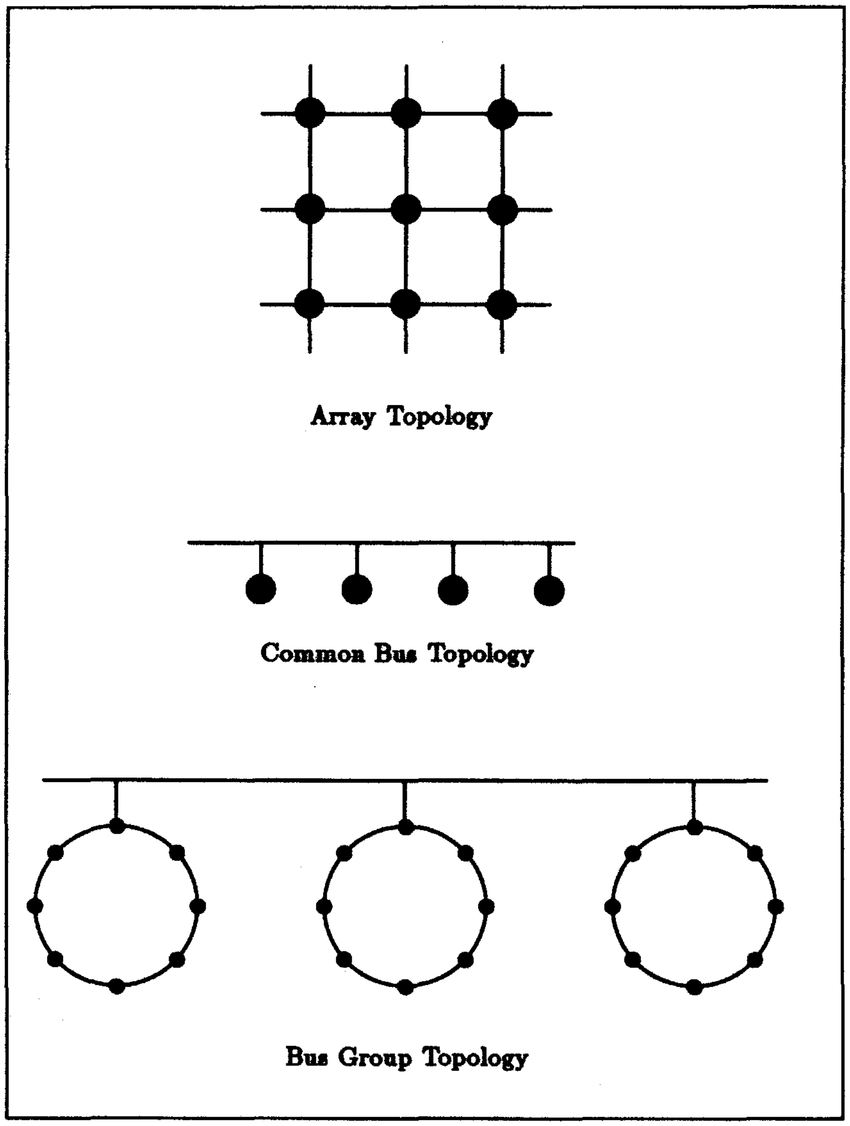

Figure 4.2: Some possible multiprocessor topologies.

to evaluate the effect of different topologies on the locality of reference of Multilisp programs.

Nusim takes a somewhat simplistic view of the possible organizations of processing nodes in a Multilisp machine. It does not care how individual nodes are connected together. Instead, the only thing that matters is the relative 'distance' of nodes from each other. This distance might be the number of hops across a communication network between two nodes. In some versions of a multiprocessor, processors would only be connected to nearest neighbors in some two (or three) dimensional grid. In another organization, processors would be clustered on local buses, which are then tied together through a global bus.[2] Processing nodes on the same local bus would be 'closer' than nodes on different buses. Finally, all the nodes in the system might be tied together through a large switch. In this case, all nodes would be equidistant.

In Nusim, each processor maintains a table of distances to all the other processors in the system. It also maintains a total ordering of those processors. This is done so that algorithms that search through the processing nodes for some resource can efficiently step from node to node. I have written Multilisp utilities to translate some different topologies to the form needed for Nusim.

4.1.12 Statistics Gathering

Nusim collects counts of most important events that occur in the system.[3] Each processor collects its own statistics locally for efficiency. One MCODE instruction turns statistics gathering on, another turns it off, and a third dumps the statistics from each processor in the system out to a file.

Statistics are enabled or disabled for each task in the system individually. The micro-state of each process contains a pointer to the area of memory where we collect

[2]This is the organization of the Concert system.

[3]As well as some unimportant ones!

statistics for that process. Since we could have several banks of memory allocated for statistics, we can change banks just by updating one pointer. This gives Nusim the flexibility to collect statistics for different tasks, or for different phases of the program, individually. See Appendix app:nusim-doc for a more detailed explanation of Nusim's statistics mechanism.

4.1.13 Implementation

Nusim is written in C. It consists of 12,000 lines of source code (400K bytes of source). The executable program is 160K bytes long. It also uses a library of Multilisp code to define the Multilisp compiler and run-time system. This library, written by Juan Loaiza and Robert Halstead, is an additional 7000 lines of code.

4.2 Variables

As stated in Chapter 1, the main focus of this thesis has been to measure the effect of different architectural variables on the locality of reference of different Multilisp programs. This section describes those variables, and the effect that they are expected to have on the behavior of applications.

4.2.1 Topology Variables

The most important variable affecting the locality of reference in Nusim is the topology of the organization that we are simulating. We can define the *diameter* of any topology as the maximum distance between any two nodes. Different topologies will have different diameters.

We would expect that, all other things being equal, the locality of reference of a Multilisp program will depend on the diameter of the underlying topology. This is certainly the case if the processor is ignorant of that topology. However, some functions of the processor architecture could be made more efficient if they exploited the organization of the external system.

One of the areas that shows the most promise of improvement is in obtaining resources from the other nodes in the system. A processor may run out of memory in the midst of creating a data object, in which case it might have to 'borrow' a block of memory from another node in the system. Temporarily, it might use this memory as if it were local. It would be better if the two nodes were close together in the system topology, to minimize expensive accesses across the system.

Another resource that processors must obtain more frequently is an executable task. Any time that a processor is idle, it searches task queues in the system for a task that it can run. In the current implementations of Multilisp, a processor will then load that task into its local process queue. However, tasks are spawned on a particular processing node, and could retain references to objects that were allocated in that node. In order to reduce the number of global accesses in the system, we might again prefer to grab tasks from nodes in the system that are close to our own.

Of course, the resource allocation functions of a Multilisp machine might use knowledge about the system topology in other ways. One problem in many multiprocessors is how to balance work across the machine. While we can increase the locality of accesses by encouraging work to stay local to an area, this may also decrease the amount of parallelism available in the system. Another danger in not distributing work across the system is that we may produce 'hot spots' in the communications network. Certain pathways may swamp with accesses, while other parts of the computer remain idle. So scheduling functions must strike a balance between increasing the locality of programs and increasing their distribution.

For the purposes of this thesis, I have tried to measure the tradeoff between locality and parallelism, by trying different task searching strategies and seeing their effect on the execution of application programs. The task searching strategies that I used either ignore the underlying topology of the machine that Nusim was simulating, or take advantage of that topology by searching local processing nodes first. Section 5.2.2 explains the searching algorithms in more detail.

4.2.2 Task Scheduling

A second class of decisions that affects the locality of accesses in a Multilisp program is how to distribute work between process queues and task queues. For example, we could give each processor a very large process queue. Then every process that was spawned by a future instruction would remain in that same process queue. Eventually, all the parallel tasks in the system could sit in the process queue in a single node. Since process queues are private to a processing node, this would not lead to much parallelism in running a particular program. However, it might allow us to have excellent locality of accesses!

Another example of task scheduling strategies is how many tasks we load from a task queue at one time. A processor that has spawned many tasks might have several of them in its task queue. If these tasks were created from the same instruction stream, they will likely share references to some data objects. Another processor that is idle and looking for tasks to run could grab several adjacent tasks from the first node's task queue. The number of tasks that it steals could affect the locality of the program.

Finally, there are many other possible algorithms for grabbing tasks out of a node's task queue. Figure 4.3 shows the parallel paths of execution through a typical program. A task which has been spawned off early in the running of a program, such as branch B, will later produce many other parallel tasks. In contrast, branch D, which is produced much later, will not spawn any new tasks. If another processor were looking for tasks to run, it could either choose an early task that could potentially produce much parallelism, or later tasks, which are more likely to produce results used by other running processes. In Nusim, we default to grabbing the most recently produced tasks.

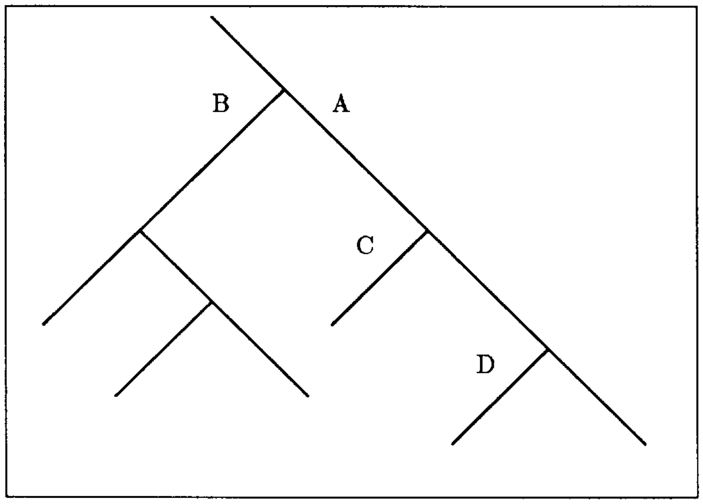

Figure 4.3: Paths of execution in a program. Path B is spawned early, path D is a 'leaf' task.

Chapter 5

Experiments and Results

5.1 Experiments

While most of the effort in this thesis was spent building the Nusim simulator, my primary research goal was to run the experiments described in this chapter. I ran several test programs under Nusim, varying several parameters of the architecture being simulated, as well as the external organization of the system. Under these different conditions, I tried to measure the locality of reference of Multilisp programs, and the extent to which that locality can be improved. All the programs discussed here were written or adapted by members of the Parallel Processing Group at M.I.T.

5.1.1 A Discussion of Benchmarking

When measuring characteristics of a language to be used in the architectural design of a machine, the most important consideration is to choose a mix of test cases that represents the applications that will be run in that language. This is especially difficult if the language is as young as Multilisp, in which few large applications have been written.

Of course, selecting a cross-section of programs does not mean that one should place equal importance on each of these benchmarks when evaluating different architectures. The most popular set of Lisp benchmarks currently in use has been run on a wide variety of Lisp languages, implementations, and machines [23]. The Gabriel Benchmarks are a set of small programs, each targeted towards a different aspect of a Lisp implementation. A benchmark that exercises function calling exclusively, such as *TAK* [23] may be less representative of Lisp programs than a function like *Browse*, which mimics some of the behavior of AI searching programs [23].

Most benchmarks suffer from another problem: a function that will be repeatedly run with different parameters, possibly on different machines, should be small and well understood. This is to shorten the running time of the program and to make data collection easier. However, a property of most Lisp application programs is their large program and data sizes. While small benchmarks may fit entirely within the caches of a particular machine, real Lisp programs often overwhelm the memory system.

For my test cases, I have tried to choose a selection of programs both from the 'classic' benchmarks as well as real applications that were written in Multilisp. The speed of our implementation limited the size of the data sets used as inputs to the test programs. However, some of the programs have a large code size. This should lead to a realistic mix of instructions and access types.

These tests are not intended to compare the present implementation of Multilisp against other Lisp implementations. Rather, they can be used to see the effect of changing parameters of an architecture that runs Multilisp. An algorithm can be coded many different ways, with widely different results. Even to compare two different Lisp implementations by using the same source code for the benchmarks is difficult, since each level of the hierarchy, from compiler down to machine architecture, has its own effect on system performance. Even the source code of the benchmarks might be different on different systems. Benchmarks that are coded in Common Lisp usually have to be re-coded to run in Multilisp.

The placement of *futures* in a Multilisp program has a great effect on the amount of parallelism obtained in running the program. In spite of tools developed by members of P.P.G. [27], deciding where to insert futures in code is still a time-consuming and somewhat arbitrary process. The small benchmarks used by the group have been carefully studied, to try to get the optimum parallelism out of each function. Larger programs are necessarily less understood. In spite of the difference in the 'quality' of the test cases used here, they should accurately reflect the potential parallelism of real Multilisp applications.

5.1.2 The Test Cases

The data presented in this chapter is from five different test programs. Two are small benchmarks from the popular literature, and the other three are applications written in Multilisp that were developed at P.P.G. The complete source code for each of the programs is given in Appendix C.

Quicksort

This is a version of *Quicksort* that was rewritten for Multilisp and made into a parallel application by adding futures in a few crucial locations. This benchmark is perhaps the best known and best understood program used by the P.P.G. group. Because of the care that has been put into its implementation, Quicksort has more parallelism than any non-trivial application.

In these benchmarks, Quicksort is used as an extreme test case: It runs many loops of a few short functions. Therefore the distribution of instructions for a run of Quicksort is skewed towards the instructions that occur within those inner loops. Also, because of the number of *futures* in the code, this application puts a greater strain on the fork, join, and task handling aspects of a Multilisp implementation.

For this series of tests, Quicksort was used to sort a list of random numbers. The only parameter to the program is the length of that initial list. Typical run times for a 700 element list, with Nusim running on 27 processors of Concert, range from 120 to 310 seconds.

Fboyer

The *Boyer* benchmark that has been widely used to compare Lisp implementations [23]. It is a simple theorem proving program. The program has two main parts: a *rewriter* and a *tautology checker*. The rewriter expands the original expression into a series of *IF* clauses. The tautology checker steps through this expanded expression to determine if the entire statement is true. It makes this analysis by maintaining lists of true and false statements. The components of any *IF* statement are checked against these two lists.

Fboyer is based on the original algorithm, but written in the *Scheme* language at B.B.N. Inc.[1] The writers made it into a parallel application by adding futures to the code. Members of the P.P.G. group improved on their implementation by slightly changing the placement of futures, and by speeding up particular primitives.[2] It is useful to note that these changes sped up the entire program by a factor of five.

This application is a more substantial one than Quicksort, and yet is smaller than the average Lisp program. Since Fboyer looks up clauses in a database of axioms that it has built up, most of the primitives in the program are list operations such as *car*, *cdr*, and *atom*, as well as property list operations such as *get*.

While Fboyer has several small basic inner loops, both for the rewriter and the tautology checker, it operates by recursively expanding an expression. This expression could be arbitrarily complex, consisting of levels of sub-expressions. This means that Fboyer should have more interesting patterns of execution than Quicksort, whose input is always a simple list. In particular, the number and sizes of

[1] Written by Seth Steinberg, 1986

[2] Specifically, *assq* and *equal* were re-coded for speed, and an *assq* was changed to *get-prop*. The B.B.N. implementation spent most of its time in *assq*. The version that I used was somewhat more realistic in its instruction mix. See Appendix C for details.

parallel tasks that are spawned during the run of the program should depend on the form of that input expression. In this way, the types of operations performed and the patterns of parallel tasks produced by Fboyer should be closer to that of real applications, even though it is a well understood benchmark.

The only parameter to the Fboyer benchmark is the input expression, which can contain any combination of logical, numeric, list structure, and conditional operators. For the purposes of my evaluation, I chose a simple test case:

```
'(implies (and (implies (f x) (g x)) (implies (g x) (h x)))
  (implies (f x) (h x)))
```

This expression returns true, of course. Usual runtime for Nusim on a 27 processor Concert multiprocessor ranges from 85 to 135 seconds.

Consim

Consim is a logic simulator program written by Elizabeth Bradley [12]. It simulates an arbitrary circuit as a finite state machine, that is, a block of combinational logic fed back through a set of registers. (See Figure 5.1). The combinational logic is expressed in terms of primitive boolean functions such as *NAND*, *NOR*, and *NOT* gates. There can be no feedback paths within the combinational logic, the only loop in the circuit being through a set of synchronous registers.

To simulate one cycle of the circuit, Consim initializes the inputs to the block of combinational logic and allows those signals to propagate through the rest of the gates. When all outputs have settled, the FSM registers are latched to start the next cycle. Consim represents each primitive gate in the circuit by a Multilisp function. More complex circuits are represented by the interconnection of several primitive functions.

Consim introduces parallelism into its simulation in two ways. First, selected logic gate primitives are enclosed in *future* instructions. This allows the simulator

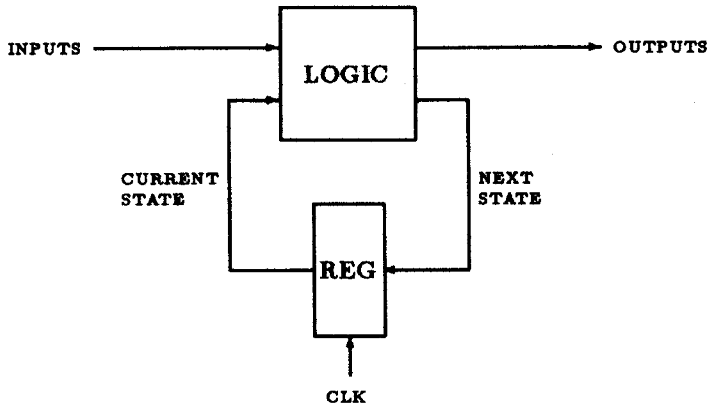

Figure 5.1: Finite State Machine Model

to mimic the operation of several gates at once. In this way, the parallelism of the simulator is exactly like that of the circuit itself. Second, Consim can run several cycles of the circuit concurrently. While it may seem that the data dependencies of the circuit would not allow any parallelism using this approach, some computation can usually proceed without all of the inputs to the circuit being valid.

This was the first 'real' application program that I tested. It is a moderately large application: the run-time code for the simulator amounts to approximately 1000 lines of Lisp code. In addition a user must write a high-level language description of his circuit, which is compiled down to Multilisp code. This can add several hundred lines of code to the simulation.

The simulator runs as a simple loop, calling different functional routines for the components of the circuit being simulated. The circuit is not simply a data structure that can be manipulated, but an interconnected group of Multilisp subroutines. This means that the form of the circuit has a great effect on the runtime properties of Consim. This was demonstrated in [13].

The test circuit that I used as an input to Consim was a four bit ALU, configured to act as a counter. It exploited parallelism both by running circuit components in parallel and by allowing cycles of the entire circuit to run concurrently. Because of this approach to parallelism and the large size of the circuit, it produced more parallelism than any other Consim simulation.

A second parameter was how long to run the simulator. Within Nusim running on a 27 processor Concert system, Consim typically took between 140 and 345 seconds to simulate the counter for 70 cycles.

Multilog

Multilog [51] is a simple query language interpreter, in the flavor of Prolog [20]. It maintains a database of assertions, and allows the user to query the database. Just as in Prolog, a user can ask the database whether a statement is true or false, or may ask what statements in the database match a particular pattern.

The main operations that Multilog performs are pattern matching, unification, and database manipulation. It also contains an evaluator and an interactive driver loop. In this way it is representative of a large class of Lisp software. Since the database storage and lookup manipulates streams [53] extensively, Multilog spends most of its time building and disassembling lists.

Multilog's author introduced parallelism into the program in two ways: one was to explicitly write the evaluator and pattern matcher using futures to fork off parallel searches. The other was to introduce futures into the streams used throughout the program. Since Multilog uses streams in all queries to the database, as well as inter-procedure communication within the program, this yields a large amount of concurrency.

Multilog is a large program — the source itself is almost 1000 lines of Lisp code, and the default database is another 250 lines. The dynamic operation of this program is complex, reflecting the influence of many small routines. However, since Multilog always runs the same operations of pattern matching and unification, the gross operation of the program should be independent of what queries it is trying to prove.

In the tests presented here, I ran Multilog on a query that attempted to trace a path through a graph. The graph consisted of eight nodes, connected together by links. The test case merely asked whether two nodes in the graph were connected by some set of links. Typical runtimes for this test case on a 27 processor Concert system ranged from 400 to 700 seconds.

Compiler

The final test case that I ran for this research was the Multilisp compiler itself. This is a large and complex piece of software, and is also the first large application that was written in Multilisp.

The compiler is composed of several phases. The *Reader* reads an expression from a file or from the user. A set of routines known as *Compile-expr* then compiles this expression into a symbolic assembly language by recursively compiling each sub-expression. Finally, the *Assembler* generates assembly binary MCODE from the symbolic assembly code. The specific part of the compiler that I instrumented was the central phase. This was partially because of the difficulties in instrumenting I/O accesses in Nusim, and also because Compile-expr seemed to be representative of a larger class of programs than either the reader or the assembler.

Compile-expr obtains concurrency by spawning a new task to compile each sub-expression. Thus the amount of parallelism that can be achieved depends on the complexity of the input expression. Statements that are 'wide', that contain many sub-expressions at the same level, produce many parallel tasks. However, these tasks will be short-lived unless the sub-expressions are also deep, giving the compiler something to work on.

Since the Multilisp compiler is a large piece of software, it has not been subject to the same amount of scrutiny as some of the smaller benchmarks that were written in the Parallel Processing Group. It is more difficult to find the optimum placement of futures in a large, heterogeneous program than in a smaller or more regular one. For this reason, the Multilisp compiler could probably be optimized somewhat to produce more parallelism, or to shorten its run time. However, in this respect, the Multilisp compiler may also be characteristic of other application programs. One hopes that the size of the program will compensate for the fact that futures may not always be well placed, since there should still be enough parallel tasks to saturate a target machine. The compiler took between 155 and 230 seconds to compile the test expression in Nusim.

5.2 The Variables

This section describes the variables that were varied in running application programs under Nusim. Nusim allows a user to vary selected low-level variables at run time. Using this facility, it was easy to write script programs to run through many invocations of a function, each time varying one parameter. Appendix B lists the architectural variables in Nusim, as well as how they can be modified.

5.2.1 Topology Types

For the tests described above, the most significant parameter of the Nusim emulator was the external organization of processor nodes that were being simulated. As described in Section 4.2.1, Nusim allows a user to connect nodes in arbitrary patterns, and to see the effect of those topologies. What follows is a description of the three basic topologies used in this thesis.

Line Topology

The simplest topology that I tested is known as the *line* topology. It simulates the organization of a set of processing nodes, each of which can only communicate with two adjacent nodes. The line is closed at either end to form a ring. See Figure 5.2.

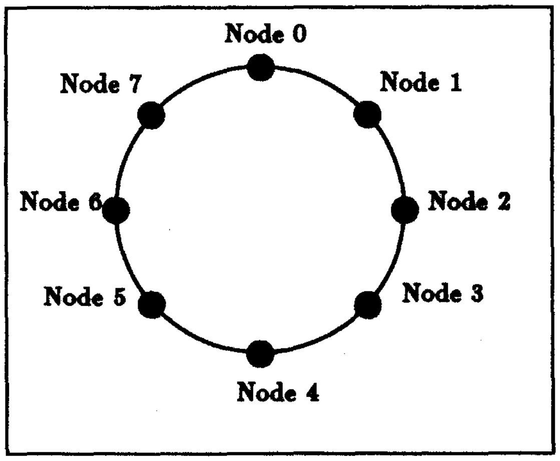

Figure 5.2: An example of a line topology.

In a line topology consisting of n nodes, any node can communicate with any other by stepping across at most $\lceil n/2 \rceil$ nodes. There is a *cost* associated with each of the accesses that a processor can make. A processor's local node memory is an access of distance 1, the two nodes on either side of it are at distance 2, and so on to the $n/2$ node. The 'diameter' of this topology, as defined in Section 4.2.1 is one half of the number of processors in the system.[3]

Segment Topology

The segment topology assumes that all processors in the system are separated into distinct groups. Figure 5.3 shows this distinction. The segments are tied together by a single global bus, while the processors within a segment share a local bus. This cuts down on the global bus traffic, and speeds up local bus accesses.

This organization is much like the one used by the Concert multiprocessor system. However, note that while I was simulating a segmented topology on a 'real' segmented topology, the two were not necessarily related. For instance, for most of the tests I ran, the Concert multiprocessor contained 27 processors, split among 7

[3]In other words, the diameter of the line topology is actually one half the circumference of the ring!

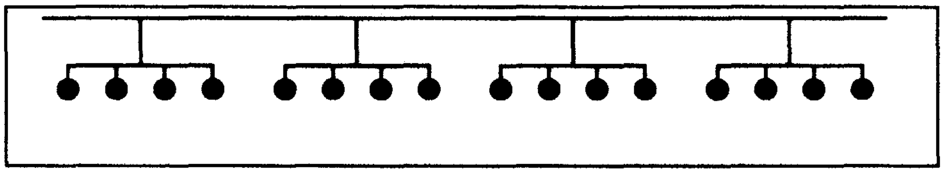

Figure 5.3: An example of a segmented topology.

slices. The system that I simulated, on the other hand, consisted of 27 processing nodes, divided into 4 segments. The reason for this distinction was to encourage sharing among processors in a segment. The simulated topology seemed to better reflect potential computer systems than the organization of Concert.

In the model of the segmented topology, accesses by a processor to its node's local memory are at distance 1. Other nodes in the same segment as the originator are at distance 2. Finally, accesses across the global bus to any of the processors in other segments are at distance 3. I could have classified these global accesses by the distance of the access. However, in the Concert system, there is no difference between the length of time required for an access between two adjacent segments, and the time between two widely separated segments. All accesses which require some part of the Ringbus are equally expensive. It seemed sensible to mimic this behavior of a real system when simulating such a topology.

Grid Topology

The grid topology has the greatest connectivity of the three topologies tested. It represents a two dimensional grid, in which each processing node is connected to four adjacent nodes. See Figure 5.4.

For the purposes of simulation, I tried to make the grid as square as possible. For example, I could have easily divided the n processors into a $2 \times y$ grid, but this would increase the diameter of the system. See Figure 5.5. In the $2 \times y$ case, a node has two paths to its nearest neighbor in one direction, but another node is $\lceil n/2 \rceil$ away. This topology looks too much like a line to be interesting. The

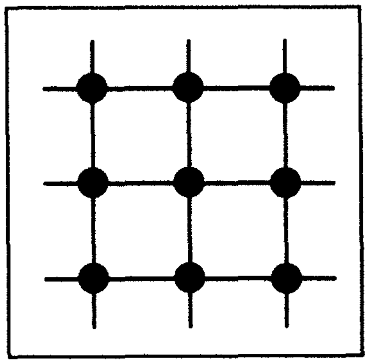

Figure 5.4: An example of a grid topology.

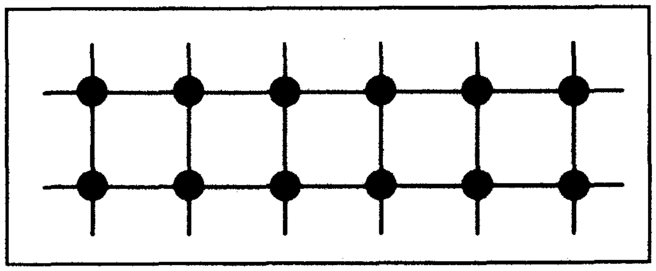

Figure 5.5: A grid topology that is too narrow.

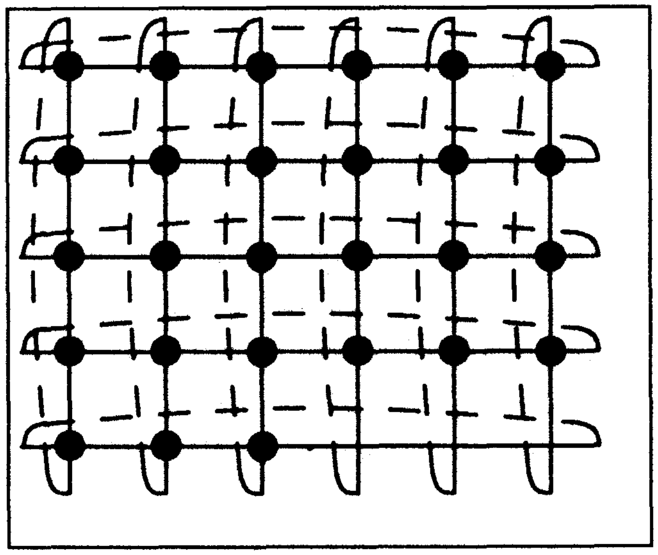

Figure 5.6: The grid topology used for this thesis.

number of processors in the system might not factor evenly into $n = x \times y$. The missing locations on the grid are treated as simple 'short circuits.' An access across those holes in the grid just skips over the intervening distance. For example, the 27 processor Concert used for most of these experiments was divided into a 5×6 grid.

Finally, a simple rectangular grid is an open topology, since the side nodes are only connected to three other nodes. For the grid topology used here, I connected together nodes on opposite sides of the rectangle. This makes for a closed network, and eliminates any strange edge effects. In fact, this topology represents what would happen if someone wrapped a fishing net around a torus. Figure 5.6 shows this arrangement.

5.2.2 Search Routines

Section 4.2.1 described some of the ways to exploit knowledge of the system topology in a potential Multilisp machine. Nusim can use this knowledge to manage resources. For instance, an idle processor must search for executable tasks among the other nodes in the system. The simulator can use several different search routines for this function. Switching between search routines shows the effect of local knowledge of the system topology.

I defined three basic search algorithms for the tests described here. Two of them ignore the topology being simulated. Since they do not encourage locality among groups of nodes, they should determine the lower bounds of locality of Multilisp applications. A third algorithm simply searches sequentially through the nodes in the system, trying successively more distant nodes.

Random Search

There are two random search algorithms: *Random* and *Close-Random*. Random mode will search each node in the system for a particular resource, but will check them in random order. If the algorithm fails to find a resource on a node, it will not check that node again. Close-Random mode is similar, but first checks the local node for the resource, before trying all other nodes at random.

Incrementing Search

This is a more deterministic method of searching the system. A processor starts by checking its local node. If the resource is not available locally, it will search ever more distant nodes. The other nodes are sorted into groups based on their distance from the local processor. If the processor cannot find the resource in one group, it will try a group at the next further distance. If a processor is trying to load several tasks from the system, it will steal all tasks from a particular group before moving to the next further group. Since the topology is always defined with respect to our local node, each processor usually has a different perspective on the system. Every node in the system is identified by a unique number. The search at a particular distance always starts with the lowest numbered node in that distance group.

5.2.3 Task Scheduling

I ran experiments with two of the parameters discussed in Section 4.2.2. The first is *Runsched*, a variable that sets the size of the processors' process queue. It is the maximum number of tasks that can be 'cached' locally within the processor, ready to run. For the tests described here, Runsched ranged from 1 to 4 processes.

The second variable in these tests is known as *Tasksched*. Once a processor finishes running all the processes in its process queue, it becomes idle. The Tasksched parameter sets the number of tasks that an idle processor tries to load into its process queue before starting to run any of them. If there are not enough free tasks in the system, the processor simply loads as many processes as it can into its queue. Note that the value of Tasksched must be less than or equal to the current value of Runsched.

I did not run test all combinations of Runsched and Tasksched. Preliminary experiments showed that Runsched has a much more significant effect on the locality of accesses in Multilisp and on the amount of parallelism attained by the program. The results of the experiments with Tasksched are not included in this chapter. I hope to study the effect of Tasksched and other scheduling parameters in more detail at a later date.

5.3 Data Gathered

The primary goal of this thesis is to quantify the locality of accesses in Multilisp programs. Thus all the statistics discussed here are intended either to quantify that locality of reference or to explain it.

To determine why a test program has a certain locality of reference, it is useful to see the distribution of accesses made by the program. Section 5.3.1 discusses how to classify the types of accesses in Multilisp. Before discussing locality, we need a way of quantifying the locality of different programs. Section 5.3.3 defines two such metrics of locality. Finally, since different benchmarks do different amounts of work, they must be normalized to some standard in order to compare their statistics. Section 5.3.2 discusses how to compare data from different benchmarks.

5.3.1 Access Types

Discussion

Multilisp programs deal with many types of data objects. For the purposes of this thesis, they are divided into several classes, depending on how the data will be used. The important distinction is between data that could be cached locally to a processor, and data that requires a global access.

Some data objects are truly global, that is, they are shared by several procedures and by different concurrent processes. However, if the value of one of these *global* objects is constant, it could be cached locally by a processor. For frequently used constants this greatly reduces the amount of global traffic. An example is Multilisp code, which is stored as a sequence of MCODE opcodes. Although in Multilisp it is easy to bind a variable name to a compiled object, programs do not rewrite the instructions within a code object.

A Multilisp machine must use a different approach with global objects whose value is updated during the run of the program. Since these *mutable* objects are shared, a processor cannot cache them locally unless it uses some technique to maintain cache consistency [42,54]. Some algorithms have been proposed that guarantee cache coherence at low cost, but they assume that all caches in the system are connected to a single common bus [17]. Here, the cost of maintaining cache coherence must be weighed against the benefit of speeding access to locally cached objects and reducing global communication. All global symbols in Multilisp are considered to be mutable objects. User data objects, such as lists and arrays, are also mutable.[4]

For this reason, one might wish to classify global mutable objects further by how often they are updated and how often they are read. Some shared and mutable objects in Multilisp programs are read much more often than they are written. Determining how often this occurs would help determine the potential benefits of local caching. The present version of Nusim cannot answer this type of question for shared objects. Several members of P.P.G. have expressed an interest in pursuing this work in the future.

Another way of classifying the data objects in Multilisp deals with how widely distributed are the accesses to an object. Some data objects, while not local to a particular procedure, may only be used by a few procedures or a few concurrent processes. Such objects must be accessible to the rest of the system, but will often only be touched by the processor that first allocated them.

For example, free variables in Multilisp programs are lexically bound. To find the binding of a free variable, a procedure must step through the environment frames

[4]Although *rplaca* is considered poor programming style!

of lexically enclosing procedures. A program may have spawned several concurrent tasks that refer to this free variable. The tasks might get halted and re-started, moved to a different processor, and still have to look up the variable binding in that environment frame.

For this reason, all environments in Nusim are now allocated in the heap. Objects in the environment of a process are then accessible to the entire system. However, most environment accesses are made to the local environment. Therefore, most accesses to an environment will be fast and local. An alternative might be to allocate environments on the stack of a process instead of in the heap. Environments would only get moved into the heap to allow free variable references. This would speed up most environment accesses at the cost of considerable complication at compile and at run time. Knowing how often a process refers to lexically enclosing environments would help determine whether this was a good trade-off.

We would like to see how often global accesses refer to these different classes of data in typical Multilisp programs. That should give some idea of the benefits of different approaches to speeding accesses to particular data types. My experiments have shown that the different classes of data objects respond differently to changing task scheduling parameters.

Classes of Access in Nusim

Every data access in Nusim is classified either by the type of object that it touches or by the reason for the access. These accesses are then divided into several broad classes using the criteria discussed above.

The classes of access are as follows:

- Constant data.

  This class includes accesses which load blocks of code onto each processor in the system. It also includes accesses to closure objects that may have been created at run-time by the program. Finally, it counts all accesses to Lisp constants that are included in the instruction stream.

- Data local to a processor.

  This class is for data that is allocated in the local memory of a processing node and is most often referred to by processes running in that same node. It includes such objects as stack hunks that are paged out of a local stack queue on the processor and into main memory. It also includes accesses to environment objects, whether they be accesses to the local environment of a procedure or to a lexically enclosing environment.

- Global data.

  The types of data in this class are those that would be difficult to cache locally to a processing node, since they are expected to be often modified. An example is a Lisp global variable. These are often used for interprocessor communication or as state variables for a computation. They do not lend themselves well to local caching. However, note that in Multilisp there is no way of declaring such a global variable to be constant. In the absence of such a declaration in the language, we must assume that all global variables are mutable.

  All Multilisp structured objects also fall into this category. This includes arrays, lists, and user-defined structures. A procedure may use such a structure as a local variable. In such a case, the object could possibly be cached locally, rather than be globally accessible. However, since Mulsim cannot determine whether objects are shared, I am making the simplifying assumption for this thesis of treating them as global, mutable objects.

- Future objects.

  I have placed future objects in a separate class because they are used in a unique manner in Multilisp programs. Futures are both the means to spawn tasks (*fork*), and to synchronize concurrent tasks (*join*). They are shared by different parallel tasks in the system. Since these tasks are likely to be running simultaneously, they will likely also run on separate processors. This means that futures may require more global communication per object than any other data type. Finally, the number of futures in the system gives an approximation to the maximum amount of parallelism available in a particular algorithm. This alone makes it an interesting data type to count separately.

5.3.2 Normalizing the Data

The types of accesses made by some algorithms depend on how long the program runs. Running the same test program on a larger test case will touch more data and usually generate more parallelism. Since each node in a multiprocessor has a limited amount of memory, this data will be spread over a greater area of the machine.

The problem of comparing different test programs is worse since each program makes a different number of accesses among a different distribution of object types. I use two methods to normalize the types of accesses made by different test programs. The first is to divide the types of references by the total number of references. This shows the percentage of accesses that fall into each of the different classes. The second is to divide the references by the number of instructions executed by the program.

Instead of counting all the instructions in a benchmark, I use the number of *completed* instructions run by a program as a baseline. Not all instructions in Nusim run to completion. When Nusim fetches the operands to a MCODE instruction, it will sometimes get an exceptional condition. For instance, Nusim will back out of an instruction that touches an undetermined future. Completed instructions are all the instructions executed by a process that do not get exceptions.

5.3.3 Locality of Access

The statistics of locality reported here are all based on counts of heap accesses. The accesses fetch Lisp objects, fetch links to those objects, or update the value of the objects. Nusim counts the accesses to data objects in each type class. It also tracks where those accesses occurred in the system topology. By the definitions of topology in Section 4.2.1, accesses to local memory are at distance 1, while nodes elsewhere in the topology are at successively greater distances.

For all the processors in the system, one could count the accesses that each made at different distances. One way of reporting the locality of accesses is simply by showing the percentage of accesses that occurred at each distance. Unfortunately, while this graphical method demonstrates the distribution of accesses for a particular test-case, it is not useful in comparing the results of several different test-cases. I have instead proposed two different measurements of the locality of access for a run of a particular program.

Local versus Distant

The first metric is simply the percentage of accesses made by all processors to local memory. Note that as mentioned in Section 4.1.10, in Nusim a procedure always creates objects in its own local memory. Thus a count of local accesses will indicate how often a process refers to objects that it has allocated, as opposed to those created by another task. A program that has 'perfect' locality of reference will make all its accesses to local memory.[5] Most Multilisp programs cannot reach this ideal, because when tasks are spawned in parallel, one task will often migrate to another processor. Thereafter, any object that one task uses to communicate with another will require a distant access.[6]

[5]This assumes that constant references are counted as local accesses.

[6]The alternative of keeping spawned tasks on the processor where they were created will indeed increase locality, but will not exploit the underlying multiprocessor organization.

The present implementation of Nusim does not try to cache the value of different global variables. It also does not count how often objects are shared by different tasks. The simple ratio of local accesses thus indicates the maximum amount of global communication that Multilisp programs require. Future Multilisp processors may require fewer global references if they cache the value of frequently used mutable variables.

Finally, quantifying the number of local accesses made by different algorithms will help determine the benefit of speeding references to local memory. If the majority of Multilisp algorithms make fewer local accesses than distant, the effort might be better spent speeding communication between processing nodes.

Mean Distance

The mean distance of accesses in Multilisp is the sum of the distances of each access in the system, divided by the total number of accesses. If memory accesses were randomly distributed throughout the system, the mean distance of access would be proportional to the diameter of the topology. For a given topology, this metric should also show the effect of task parameters that attempt to increase the locality of references.

Instead of merely measuring how many times processes refer to their own data objects, this metric shows the effect of clustering objects among a few nearby processors. In the example of a task that moves to a new processor after creating some objects, the distance of each access to those old objects is proportional to the distance that the task has moved. The task scheduling algorithms discussed in this thesis should discourage tasks from moving far from their node of origin.

If a program only referred to local data, it would show a mean distance of access equal to 1. This is not likely to happen in Nusim, because most programs refer to code chunks and global variables that are distributed around the system.

5.4 Results

5.4.1 Types of Accesses

Figure 5.7 shows the classes of fetches that are made by different application programs. Each column of the graph shows the accesses made by a different benchmark.[7] The lower stipple pattern represents constant accesses, as defined in Section 5.3.1. The second class represents global accesses to data that is shared by all nodes in the system. The next two classes are fetches of environment objects and stack hunks. We expect both of these classes of data to remain local to the processing node that allocated them. Finally, the uppermost pattern is a count of how often the program touches future objects.

This graph of fetches is normalized by the number of real instructions executed by each benchmark. The vertical axis represents counts of accesses per 1000 instructions. Thus, the Compile benchmark makes 200 accesses to 'constant' data per 1000 instructions. A benchmark may make more than one heap access per instruction. This is because some instructions, such as a subroutine call, require many heap accesses to load code blocks, stack hunks, and procedure arguments. Figure 5.8 is a similar graph of accesses, but represents data stores rather than fetches.

For both of these graphs, the benchmarks were run with a 'Random Choice' task searching algorithm. The size of the Process Queue, as specified by *Runsched*, was set to 1 process. Each benchmark was run on three different topologies. 'Line[14]' is a line topology of 27 processors, with a diameter of 14. 'Grid[5x6]' is a rectangular grid topology, which simulates an array of 30 processors, 5 high by 6 wide. Finally, 'Segm[4x7]' simulates a topology composed of 4 segments of 7 processors each.

[7] Note that 'PQSORT' is an abbreviation for the Quicksort benchmark.

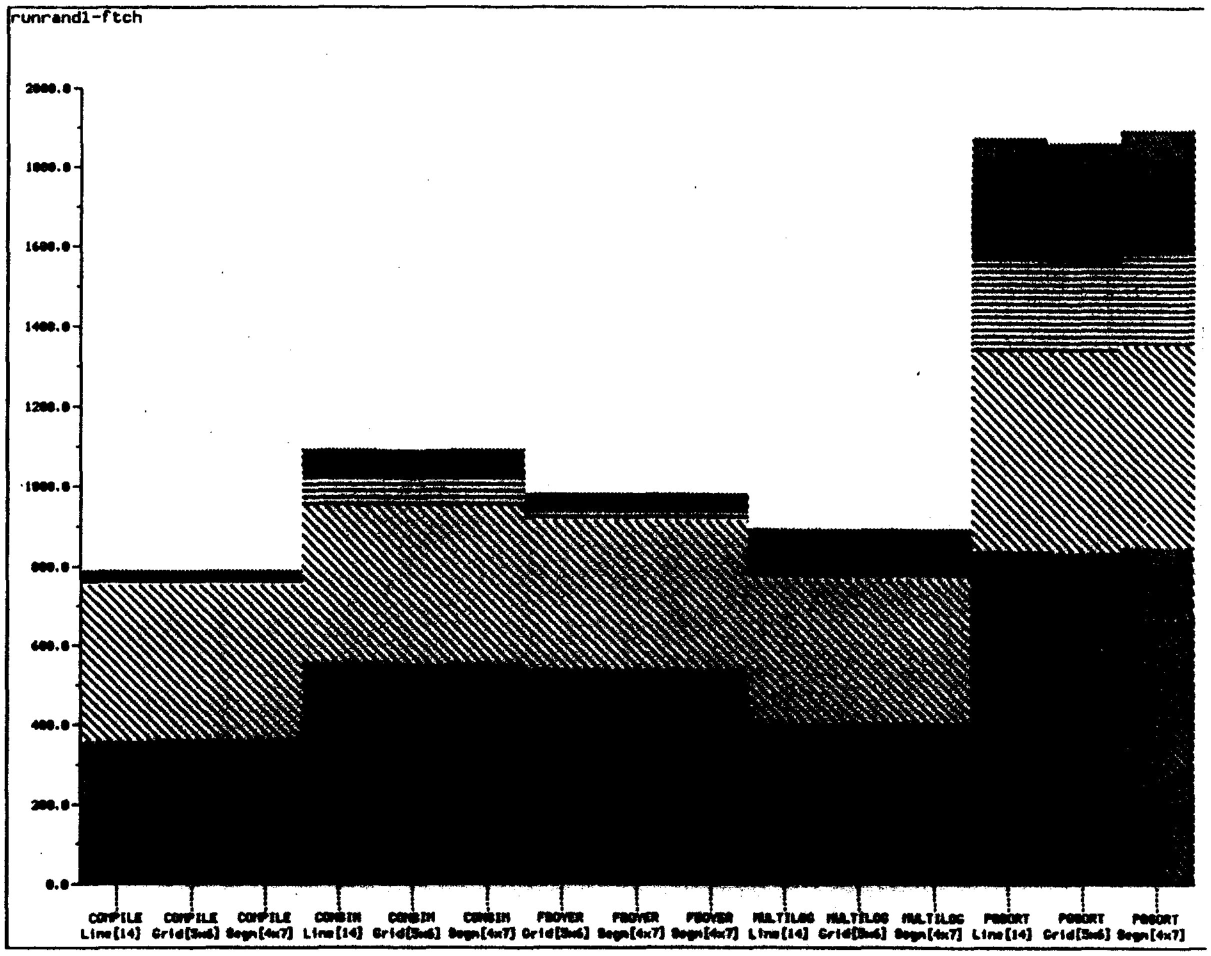

Figure 5.7: Basic data fetches. Graph shows data fetches per 1000 instructions for each benchmark. Accesses are sub-divided by class of data. From bottom to top: constants, global variables, environment fetches, stack hunks, and future touches. Benchmarks were run with random task search algorithm and Runsched= 1, on three different topologies.

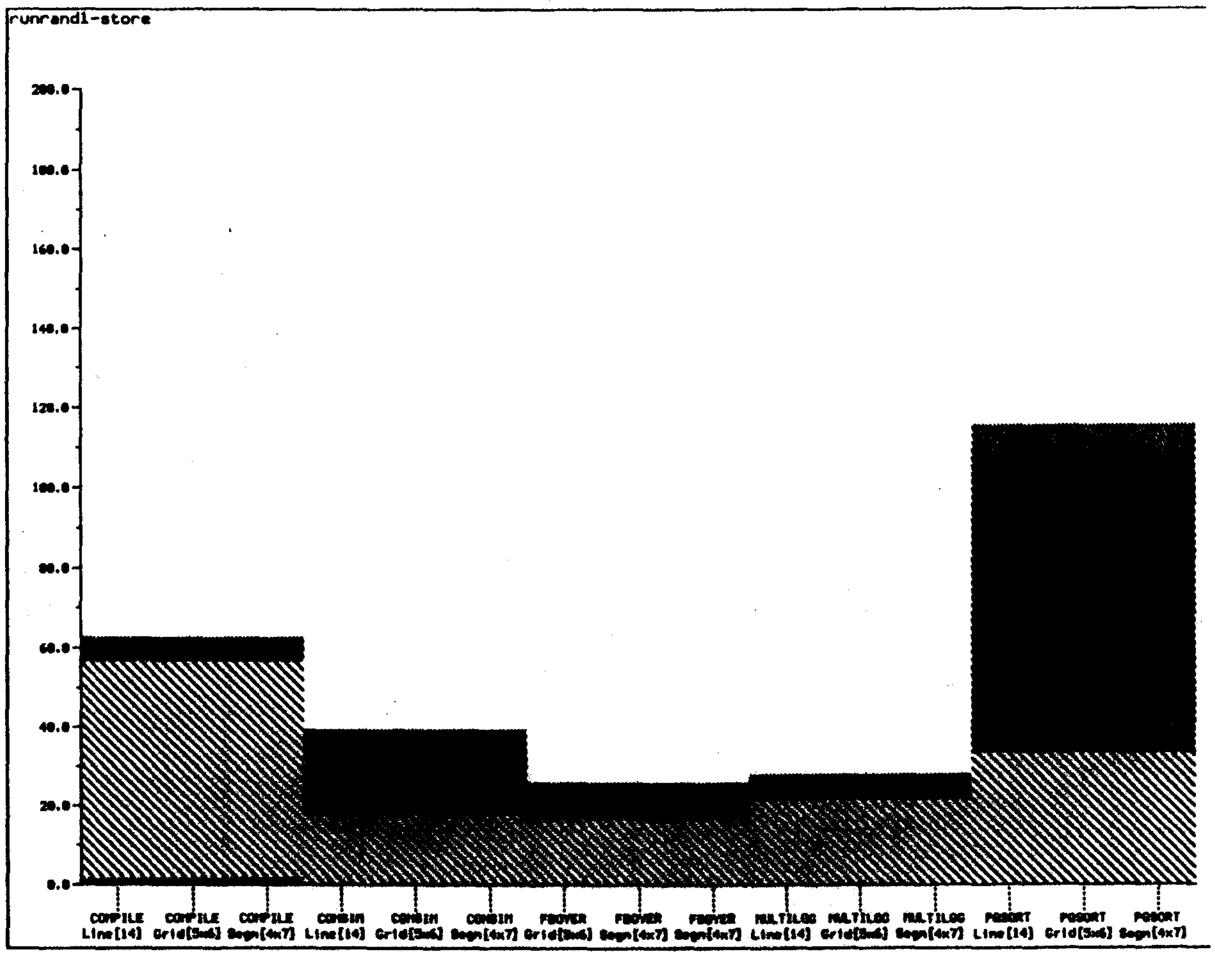

Figure 5.8: Data stores per 1000 instructions for each benchmark. Accesses are sub-divided by class of data. From bottom to top: global variables, environment stores, and futures created. Benchmarks were run with random task search algorithm and Runsched= 1, on three different topologies.

Discussion of Access Types

The data objects that a benchmark touches are largely independent of the topology upon which it runs. A deterministic program will always run the same set of MCODE instructions and will have to access the same data objects. However, the connectivity of a topology may affect the timing of different tasks in the program. Searching for executable tasks takes time, and a benchmark running on a topology with a large diameter may spend more time in this search operation. Slight changes in the timing of tasks will affect whether one task finishes before another, so that more tasks may wait on undetermined futures. As shown by the graphs, this effect is a minor one for the benchmarks tested here.

The five benchmarks accessed the heap at far different rates. The Quicksort benchmark was the most intensive, making 1.9 fetches per instruction. It exceeded the other benchmarks in all classes of access except environment fetches. At the other end of the spectrum, the Multilisp compiler made only 0.8 fetches per instruction.

This difference reflects the level of parallelism of the two benchmarks. *PQSORT* consists of a few small routines that run for many iterations. Most of those routines are spawned in parallel. The Compiler is a larger program that contains more substantial subroutines.

All the benchmarks accessed environment objects at approximately the same rate. Each of the programs made between 250 and 270 environment fetches per 1000 instructions. This constant rate of accesses reflects both the style in which our benchmarks are written and the type of code produced by the Multilisp compiler. All arguments and local variables of a routine are stored in the local environment. The compiler does not eliminate common sub-expressions. Instead, every time a procedure refers to one of its local variables, the compiler produces the same sequence of MCODE instructions to fetch the value from the environment. In typical Multilisp programs, the most frequent MCODE instruction fetches the value of an argument to a subroutine.[8]

Figure 5.9 compares the futures produced by each benchmark per 1000 instructions. The lower stipple pattern is the number of futures produced by the benchmark. Next is the number of times the future was touched after its value had already been determined. The top pattern is the number of times that a process touched an undetermined future and was forced to wait on it. The number of futures produced indicates the **potential** parallelism of the program, since each future causes a process fork. Two processes synchronize or *join* every time a process touches a future.

Quicksort generates more futures per instruction than any of the other benchmarks. Consim is next, followed by Fboyer, Multilog, and the Multilisp Compiler. Table 5.1 summarizes some of this data for each benchmark. Note that Multilog touches the futures that it has created much more frequently than any of the other benchmarks – close to 17 touches per future created. However, while Consim touches its futures much less, a higher proportion of those touches result in a process waiting on the value of the future. Consim is the only benchmark that has an average of more than one task waiting on the same future.

Note that Nusim does not indicate how those touches are divided among the future objects. It is possible that many futures may only get touched once, while one particular future might be touched many times. Additional instrumentation in Nusim would be useful in showing the range in number of touches per future.

There is one final way of comparing access types. Instead of normalizing the graphs of access types by the number of instructions executed by each program, we can normalize by the total accesses made by the program. This shows, in Figures 5.10 and 5.11, the percentage of accesses made by each benchmark in each of the different categories.

This analysis shows a few interesting patterns. The benchmarks all make ap-

[8]See Appendix C for MCODE instruction frequency.

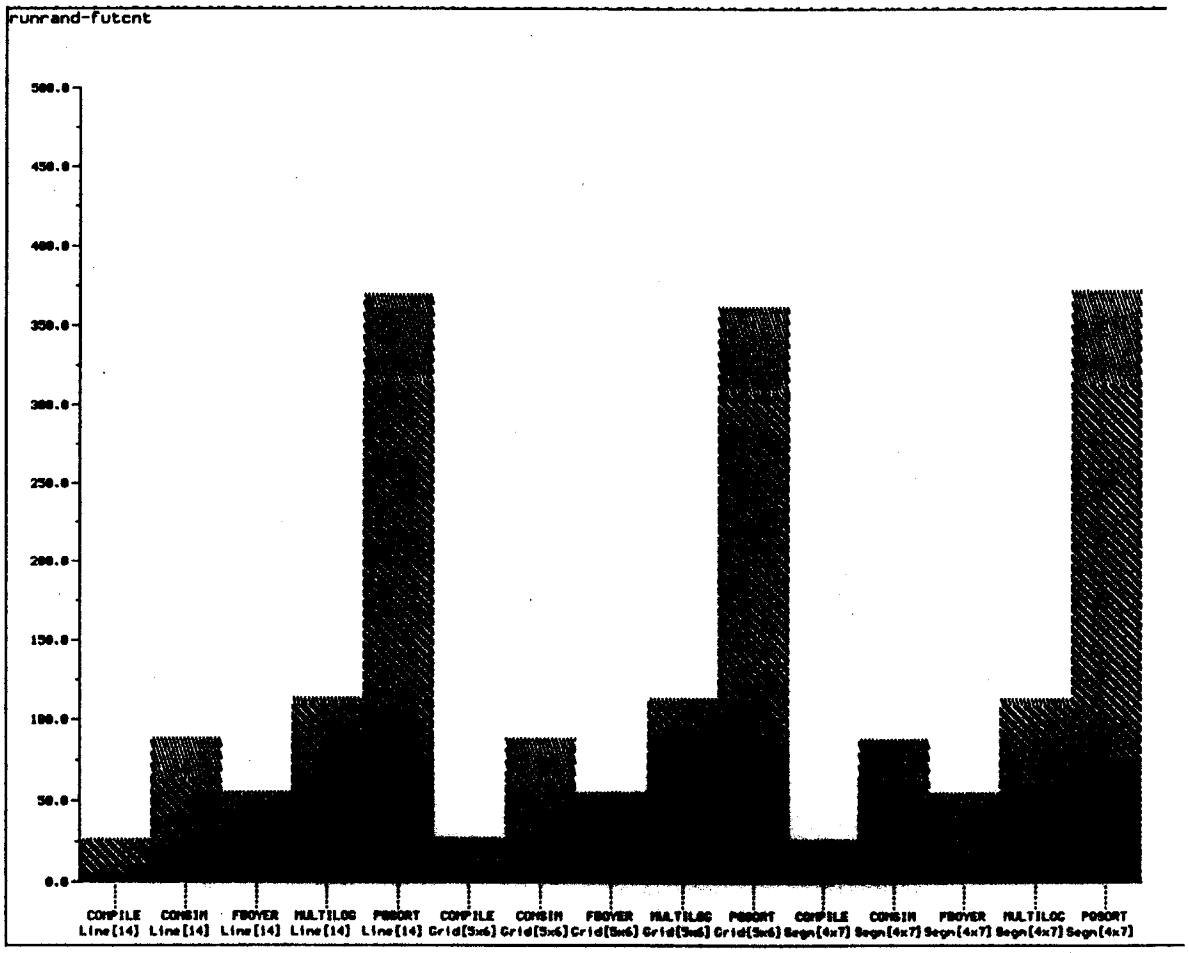

Figure 5.9: Futures created and touched per 1000 instructions for each benchmark. Lower pattern is futures created, middle is touches of determined futures, top is touches of undetermined futures. Benchmarks were run with random task search algorithm and Runsched= 1, on three different topologies.

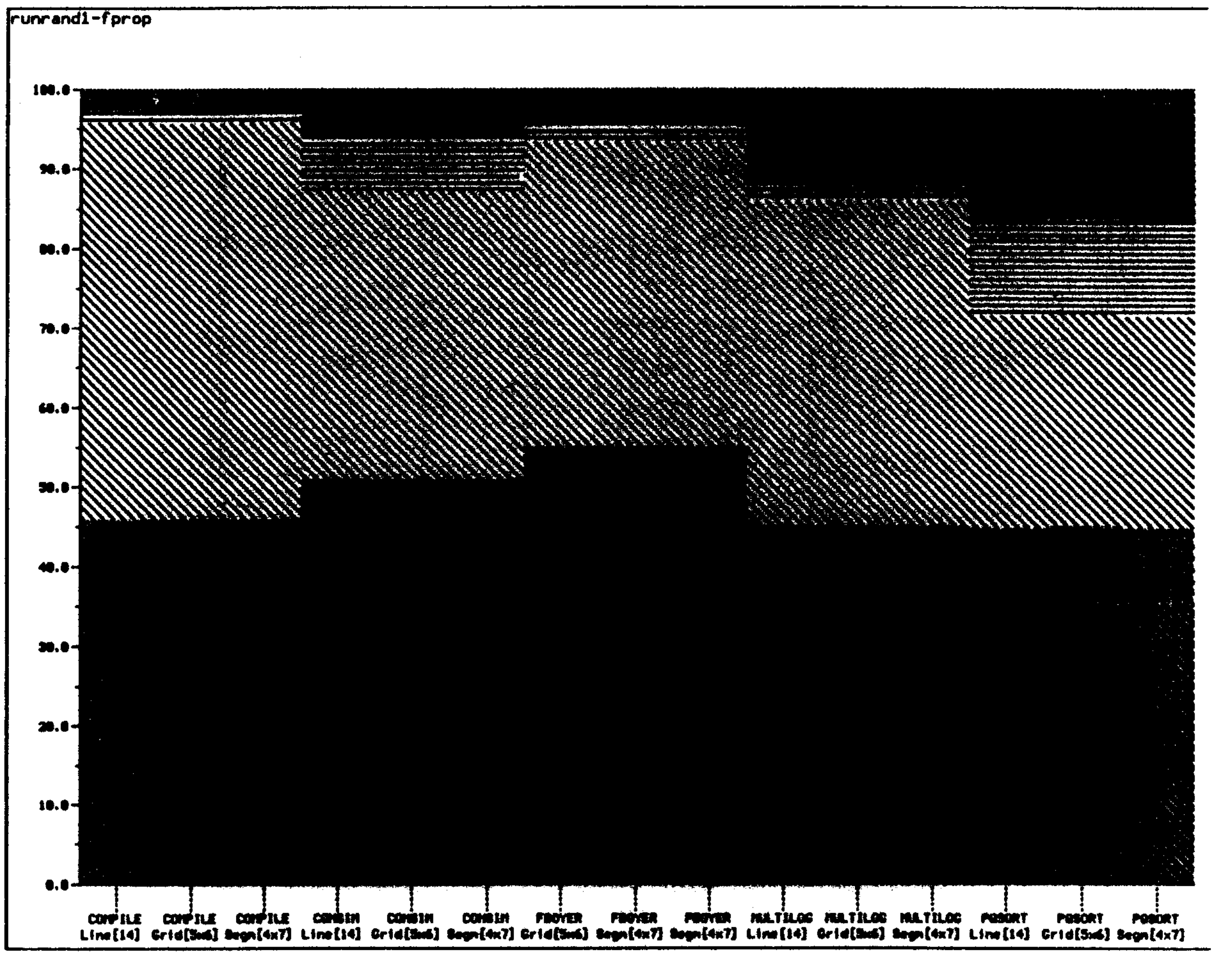

Figure 5.10: Proportion of fetches from different classes of data. From bottom to top: constants, global variables, environment fetches, stack hunks, and future touches. Benchmarks were run with random task search algorithm and Runsched= 1, on three different topologies.

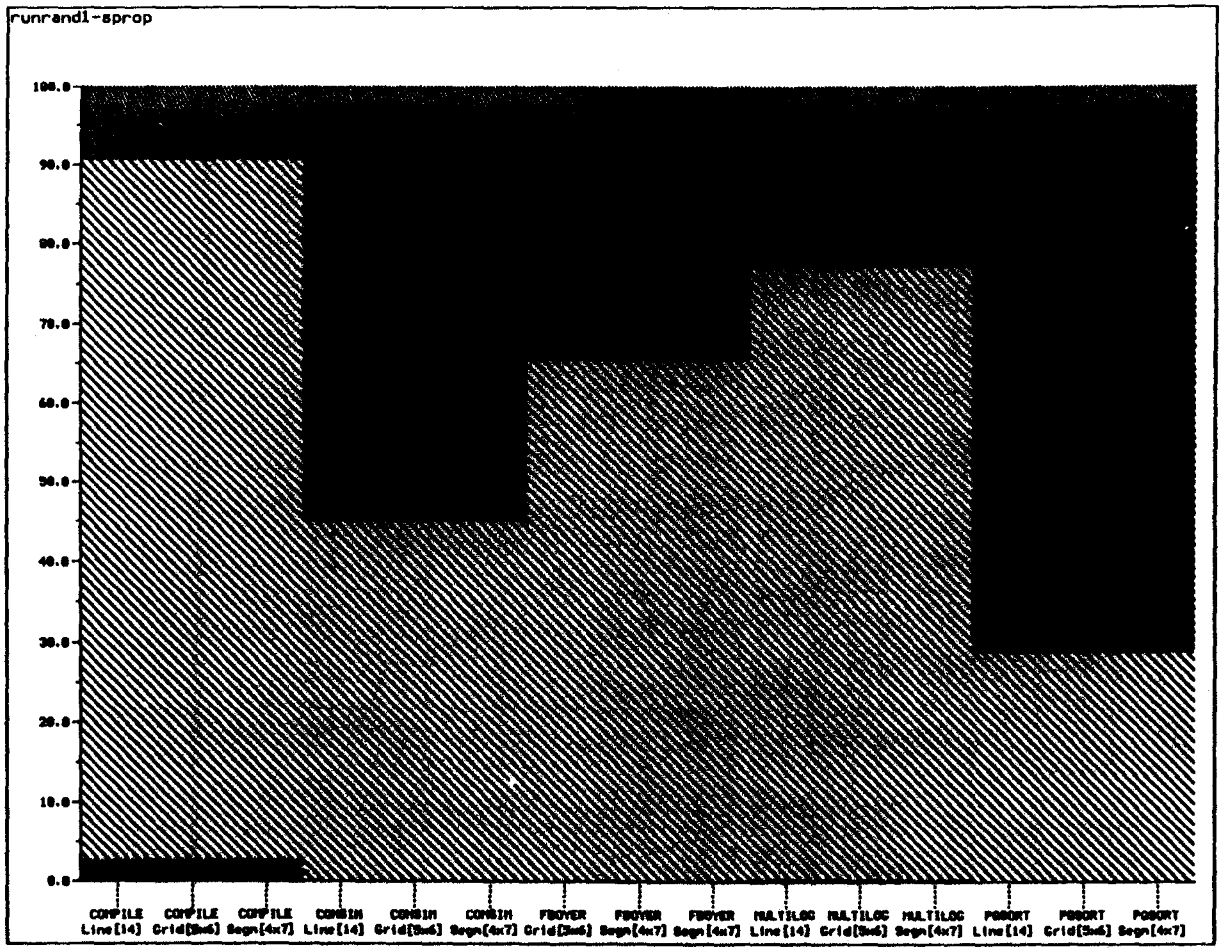

Figure 5.11: Proportion of stores to different classes of data. From bottom to top: global variables, environment stores, future stores. Benchmarks were run with random task search algorithm and Runsched= 1, on three different topologies.

Benchmark	Futures per 1000 instrs	Touches per Future	% Touches Undetermined
Quicksort	82	3.54	4.0%
Consim	22	3.18	37%
Fboyer	9	5.23	1%
Multilog	6.4	16.9	4.2%
Compile	5.9	3.54	3.8%

Table 5.1: Futures statistics for different benchmarks.

proximately the same proportion of their fetches from constant data. While the rate at which the programs fetched constants ranged between 200 and 600 words per 1000 instructions, constant fetches count for between 28% and 38% of the total fetches.

One other feature of the Compiler is apparent in Figure 5.8. While other benchmarks store almost nothing into global variables, the Compiler updates the value of global variables in 3% of its stores. The compiler also writes local environment variables much more often than the other benchmarks – More than 85% of its data writes are to local data.

5.4.2 Basic Locality of Reference

Two measures of the locality of accesses in a particular program are the mean distance of access and the ratio of local to distant accesses. The least locality of reference attainable by a program occurs when Nusim's task searching algorithm is purely random and ignorant of the underlying topology. Figures 5.12 and 5.13 show the locality of reference for the five benchmarks with a random task searching algorithm and a processor queue of length 1. Each of the programs was run on three different topologies.

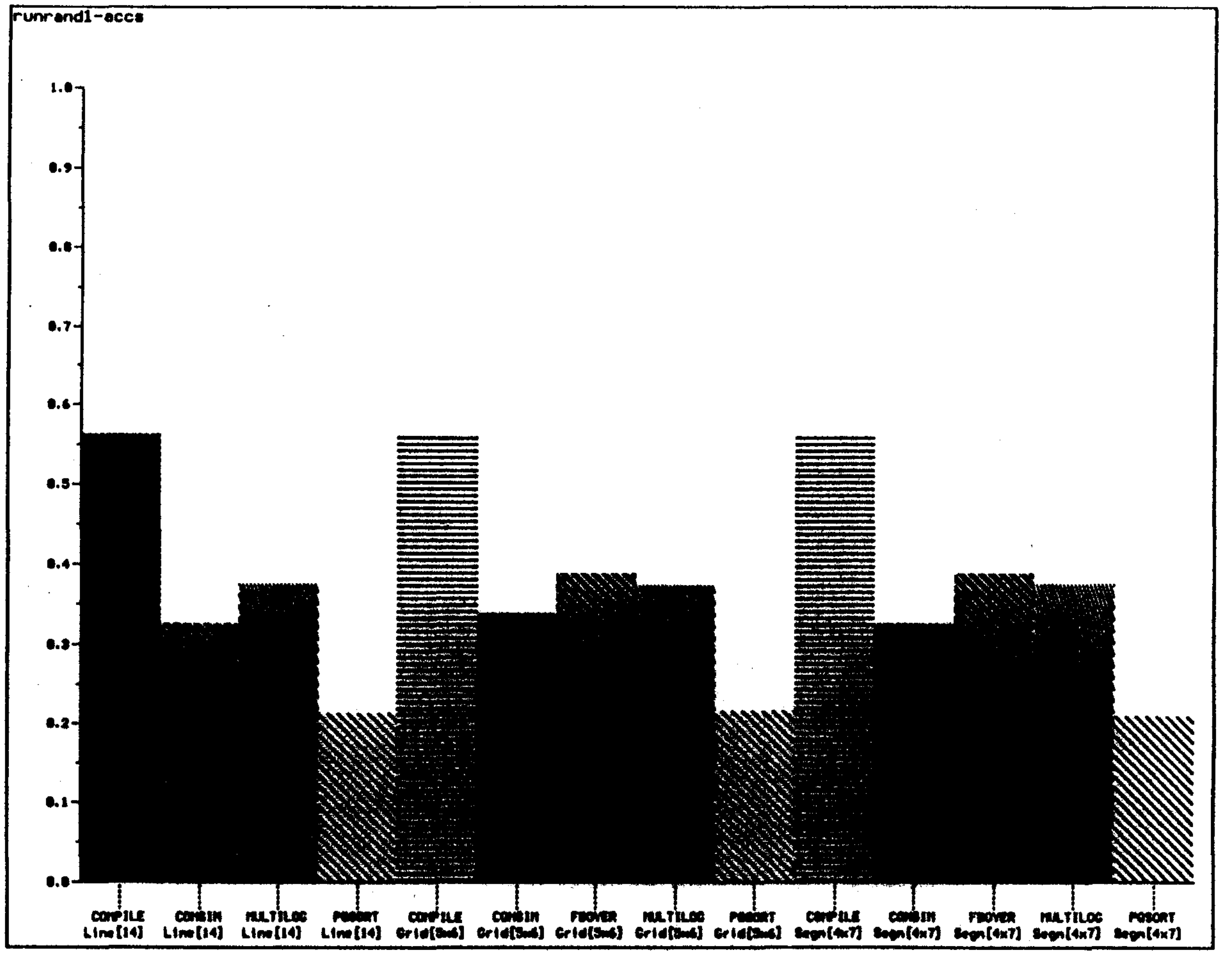

Figure 5.12: Accesses made to local memory, as a fraction of all memory accesses, made by each benchmark. Benchmarks were run with random task search algorithm and Runsched= 1, on three different topologies.

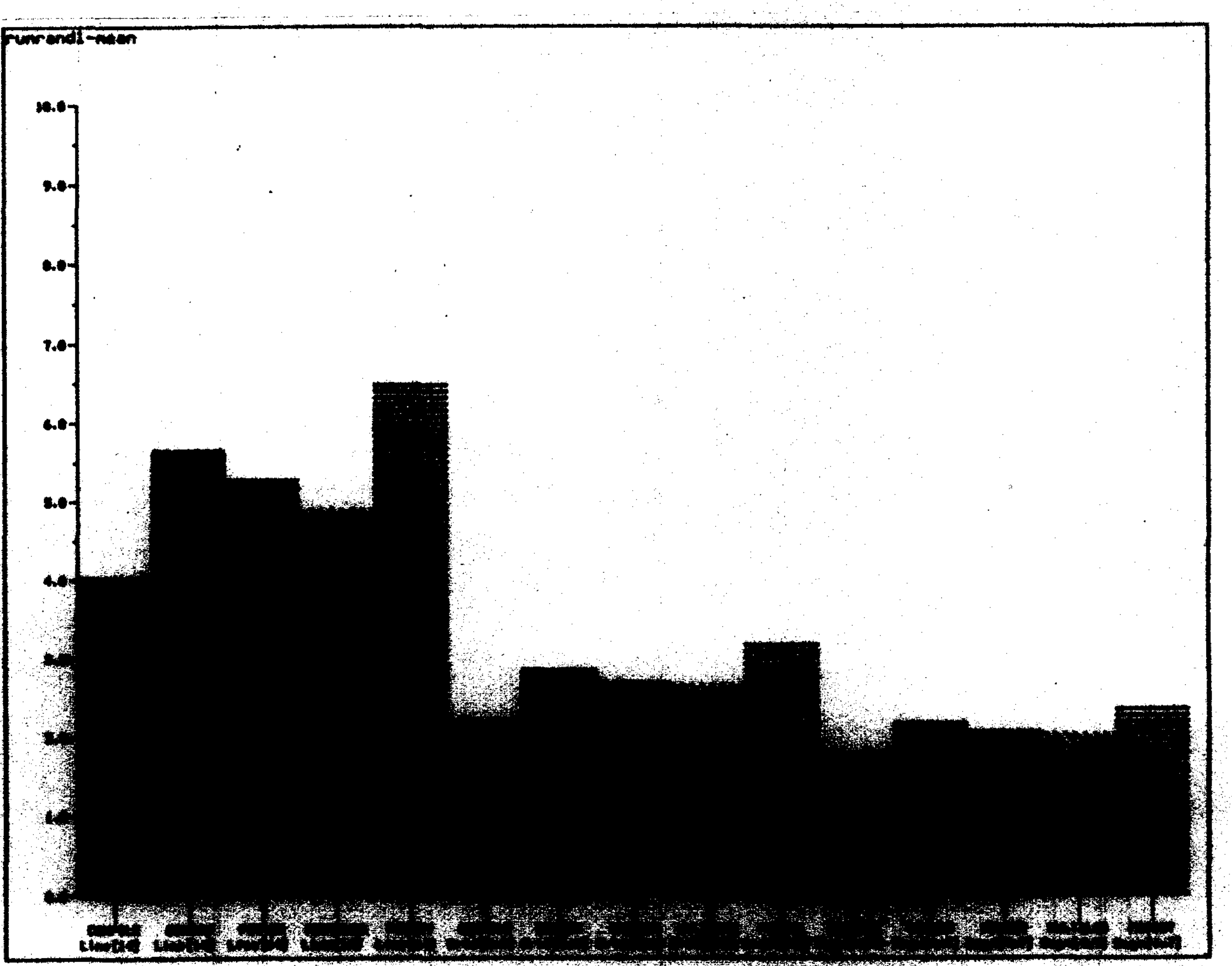

Figure 5.13: Mean distance of data access for each benchmark. Benchmarks were run with random task search algorithm and Runsched= 1, on three different topologies.

Basic Ratio of Local Accesses

Of the accesses made by a program, the percentage that touch data in local memory indicates how many of the objects that the program accesses were created locally. Figure 5.12 shows this ratio of local accesses to all accesses. It counts both data fetches and stores to all classes of data objects. The graph shows the locality of all three topologies simulated, though there is little variation in locality for a particular benchmark across several different topologies.

By this measure, the Multilisp Compiler exhibits the greatest locality. Some 57% of accesses in the Compiler are to local memory. Fboyer and Multilog have close to the same percentage of local accesses, at 39% and 38%. Consim makes fewer local references at 33%, while Quicksort has the lowest percentage, with only 21% local accesses.

A reason for this variation in locality is shown in Figure 5.10. The Compiler fetches close to 50% of its data from environment frames. This is a greater proportion than for any of the other benchmarks. In contrast, Quicksort makes relatively few environment accesses, instead touching a higher proportion of constants, futures and stack hunks. Consim fetches constant data 38% of the time. However, it makes a higher proportion of environment accesses than Quicksort. Consequently, it exhibits better locality.

Though environments are allocated in local memory, some environment fetches made by a program might not be from local memory. When a processor grabs a task from a distant node, the task's environment frame stays on that distant node. However, every time one procedure calls another in Multilisp, Nusim creates a new environment frame. Any task that does a few function calls will allocate several functions in local memory. References to variables in these environment frames will be local. So local environment accesses will be common for most programs.

Topology	Diameter	Fair access distance
Line[14]	14	7.74
Grid[5x6]	6	3.68
Segm[4x7]	3	2.71

Table 5.2: Fair access distance of three topologies.

Basic Mean Distance

The mean distance of access of the benchmarks tested varies both with the program itself and with the topology that Nusim simulates. We will first consider the variation due to the different programs. See Figure 5.13.

In agreement with their percentage of local accesses, the programs with the lowest mean distance of access are (in order) the Compiler, Multilog, Fboyer, Consim, and Quicksort. The only anomaly in this graph is that while Fboyer has a higher percentage of local accesses than Multilog, it also has a slighly larger mean distance of access. We will return to this contradiction shortly.

Topology and Random Search

As mentioned above, when Nusim uses a purely random task searching algorithm, the topology simulated by Nusim has little effect on the percentage of local accesses made by each benchmark. However, it does affect the mean distance of access for each benchmark. Table 5.2 shows the expected mean distance of access for each of the topologies, if accesses were randomly distributed throughout all memory. This is known as the *fair access distance* for a particular topology.

The line topology has the poorest locality of the three tested here. None of the benchmarks tested came close to this worst-case limit, even though Nusim was using a random task scheduling algorithm. The mean distance of reference for most of the benchmarks ranged from 60% to 90% of the fair access distance. Figure 5.12 showed that programs make a significant fraction of their accesses to local memory.

Topology	Diameter	Non-local fair access distance
Line[14]	14	8.00
Grid[5x6]	6	3.78
Segm[4x7]	3	2.78

Table 5.3: Fair access distance for non-local memory.

Benchmark	Line[14]	Grid[5x6]	Segm[4x7]
Compile	7.95	3.78	2.77
Consim	7.92	3.79	2.78
Fboyer	8.00	3.77	2.78
Multilog	7.24	3.65	2.67
Quicksort	8.01	3.79	2.77

Table 5.4: Mean distance of access for non-local accesses.

An interesting question is then whether the locality shown in the mean distance of access for these benchmarks is simply due to the ratio of local to distant accesses.

Table 5.3 shows the fair access distance for the three topologies, ignoring local memory. This would be the mean distance of access if processors accessed only distant memory, and if the accesses were randomly distributed through this distant memory. For comparison, Table 5.4 shows the mean distance of non-local accesses made by the five benchmarks. These two tables show that non-local accesses made by each of the benchmarks are randomly distributed through distant memory.

This data shows that there is considerable locality to be exploited in Multilisp programs even when they do not take advantage of the underlying topology. However, all of this locality is due to processors accessing local memory proportionally more often than distant memory.

Explanation of Mean Distance

Figure 5.14 shows the mean distance of access of different types of data for each of the benchmarks. Due to the way that Nusim collects locality information, the breakdown of data types is not the same as division by class of data presented in Section 5.3.1. From bottom to top, the locality data types used here are: *code fetches, structure stores, structure fetches, environment accesses, stack fetches, and future touches.* The major difference is that structure stores and fetches include several types of data accesses. Most structure stores are simply updating the value of determined futures. Only the Compiler updates a significant number of global variables. Structure fetches include fetches of constants, mutable variables, and several other classes. Note that though each different type of access has a different mean distance of access, they may not all count equally in the mean distance of access of the program as a whole. This graph does not show the relative number of accesses in each category.

A few things are apparent from this graph. The data type with the poorest locality of reference is the future object. For all the benchmarks, touching a future object has approximately the same mean distance of access as the fair access distance of the topology. This means that futures are distributed fairly randomly throughout memory.

For most benchmarks, the locality of reference is poorest for futures, followed by code, structure fetches, stack accesses, structure stores, and environment fetches. Multilisp code should have a high mean distance of access, since all processors in the system share the same code. Similarly, since environment objects are allocated in local memory, most environment fetches should also be local. However, this depends on the lifetime of a process, and how often tasks move to different processors.

The two programs with the poorest locality of reference are Quicksort and Consim. Table 5.1 showed that these two benchmarks created more future objects than any other program. They also have a higher rate of touching undetermined futures.

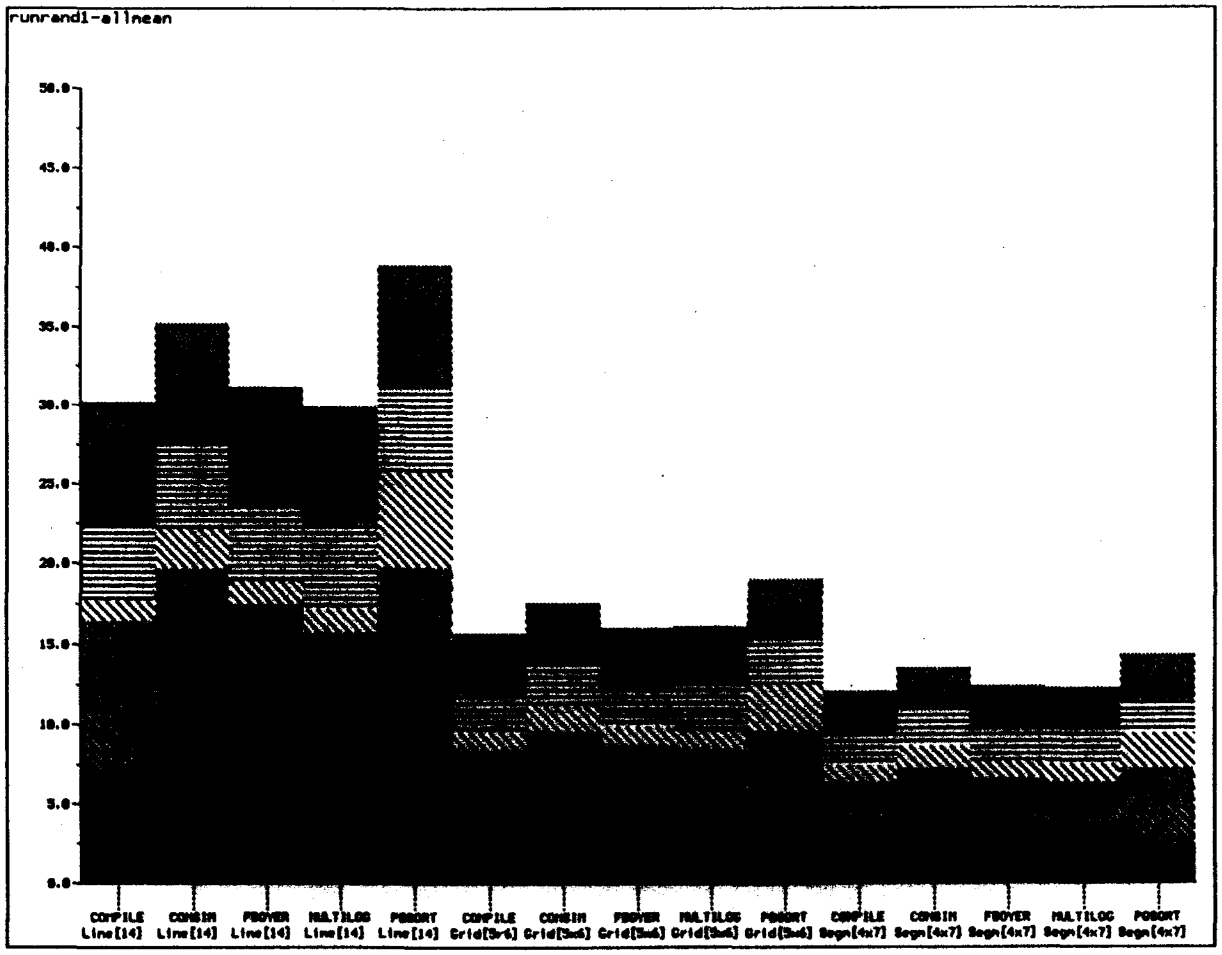

Figure 5.14: Mean distance of access for each benchmark, by type of access. From bottom to top, the patterns represent: code accesses, structure stores, structure fetches, environment accesses, stack fetches, and future touches. Benchmarks were run with random task search algorithm and Runsched= 1, on three different topologies.

As seen in Figure 5.14, these programs are also notable for the high mean distance of structure stores. For both benchmarks, almost all of these stores write the value of a determined future. It seems that for these two benchmarks, both touching and writing the value of a future are distant accesses.

Quicksort is also unusual in that the locality of environment fetches is much poorer than for any other benchmark. As mentioned above, this could occur if many tasks moved around from processor to processor. An environment created by a process on one processor would remain in that node's local memory after the process had moved to another processor. As shown Section 5.4.3, Quicksort fetches more tasks from distant processors than any of the other benchmarks.

Finally, there is the discrepancy between the mean access distance of Multilog and Fboyer. Fboyer has a larger mean distance of access, even though it fetches a higher percentage of data from local memory. The most significant difference between Multilog and Fboyer is that the former touches more futures, while the latter accesses more global shared data. If touching a future were a more local access than fetching global variables, this would explain the difference between the two benchmarks. However, futures are more randomly distributed than global variables. The actual reason for the poorer locality of Fboyer is that code accesses for this program are less local than for Multilog. In fact, the mean distance of code accesses for Fboyer is worse than for any of the other benchmarks. This may somehow reflect either the size of the Fboyer code, or the pattern in which procedures get called. It is probably not worth spending much time trying to improve the locality of reference for code, since future Multilisp implementations will probably cache code locally to a processor.

5.4.3 Local Knowledge of Topology

Effect on Data Fetching

The types of data objects fetched by the benchmarks does not vary much with the type of task searching algorithm used by Nusim. This is not surprising, since when the same tasks are running in the system, they refer to the same types of data. Where a particular task runs in the system does not seem to have much effect.

Quicksort made slightly fewer data fetches with an intelligent task searching algorithm than with a random algorithm. It accessed the different classes of data in the same proportions though. It is possible that this is only due to a difference in the timing of tasks. As discussed in Section 5.4.1, changing the timing of task searching may change the number of futures that were determined when touched.

Effect on Task Fetching

By using different task searching algorithms in Nusim, the processors can be more knowledgeable about the external topology of the system. This can improve the locality of reference of Multilisp programs. The immediate effect of using an 'incrementing' task search algorithm instead of a 'random' algorithm is to grab tasks from nearby processing nodes.

Just as the mean distance of data access quantifies the locality of reference of a program, the *mean distance of tasks* measures the average distance from which processors fetch tasks. Figures 5.15 and 5.16 compare the mean distances at which processors fetch tasks using 'random' and 'incrementing' task searching algorithms. Note that the mean distance of access in the random case is almost exactly the fair access distance of each topology calculated in Section 5.4.2. For the line topology, if Nusim uses a random search algorithm, the mean distance of task fetching is 7.74.

Using an incrementing search algorithm can have a dramatic effect on the mean distance of task fetching. Figure 5.16 shows that the new mean task distance for all

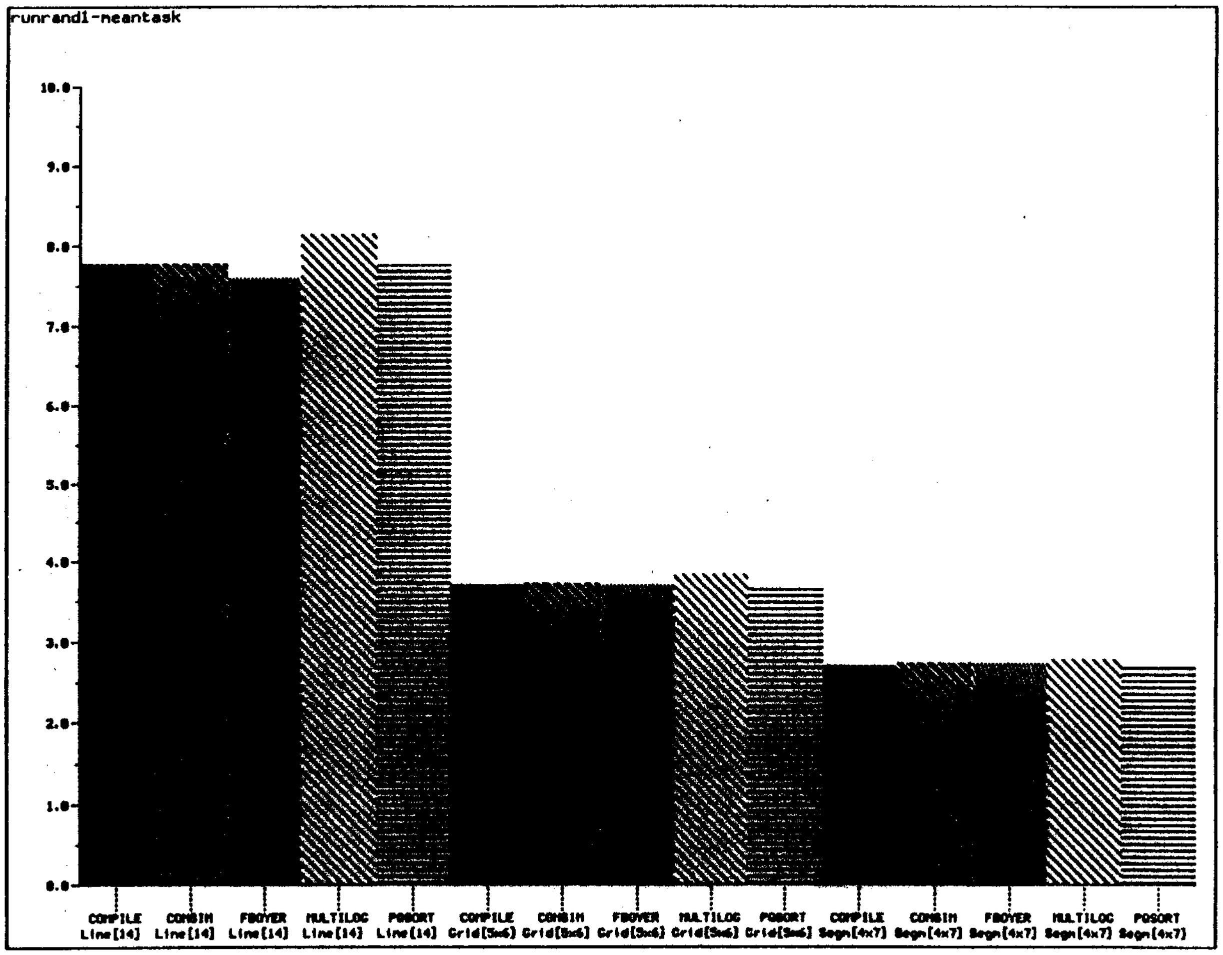

Figure 5.15: Mean distance from which tasks were grabbed. Benchmarks were run with random task search algorithm and Runsched= 1, on three different topologies.

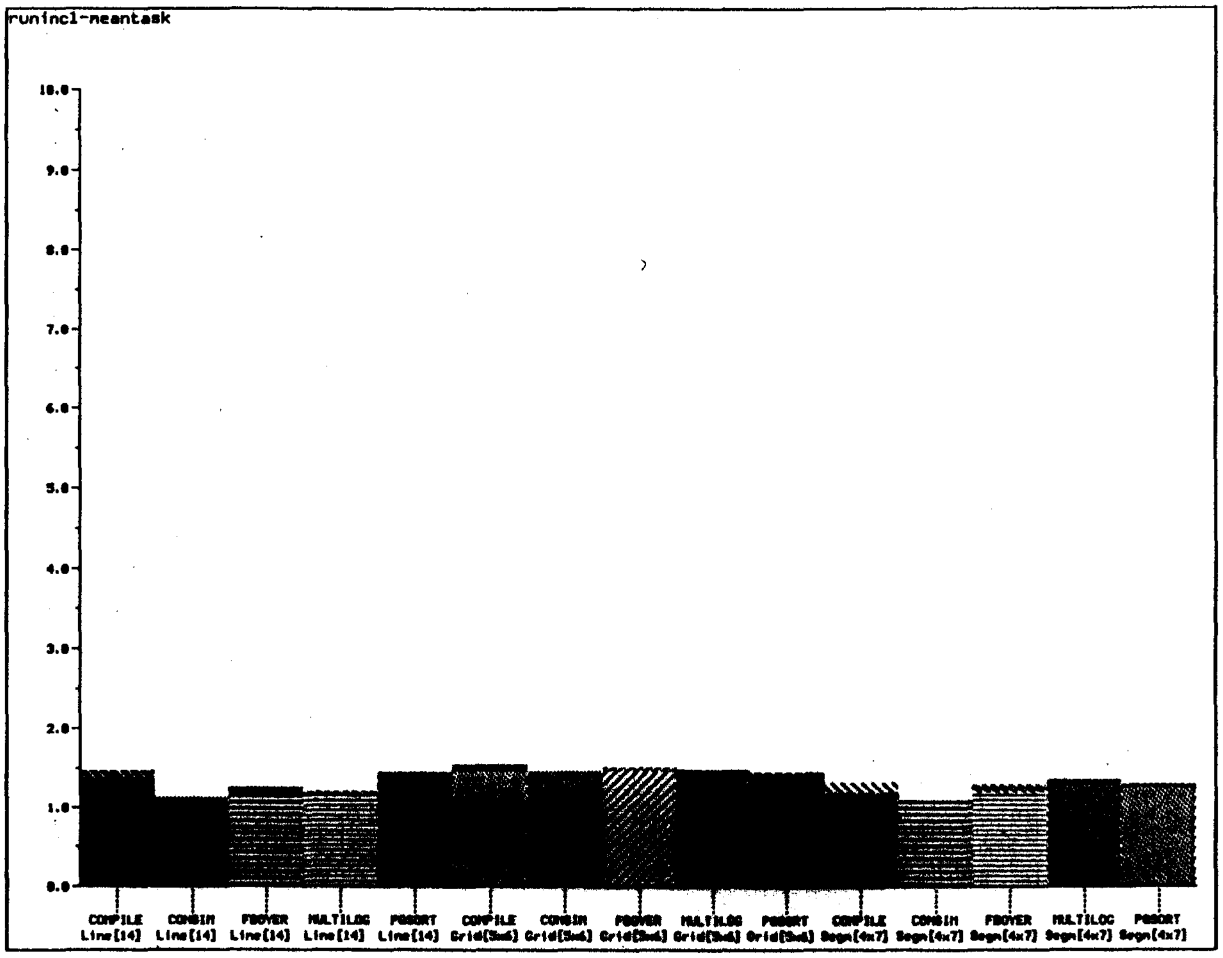

Figure 5.16: Mean distance from which tasks were grabbed. (Top pattern in each column shows variation in mean distance between runs of the benchmark). Benchmarks were run with incrementing task search algorithm and Runsched= 1, on three different topologies.

topologies ranges between 1 and 1.5. If all tasks were fetched from processor's local task queues, the mean task distance would be 1. Whereas before only 5% of the tasks that a processor fetched came from its local task queue, with the new search algorithm the proportion ranges between 60% and 90%. See Figure 5.17.

Finally, note that in Figure 5.16, the mean distance from which a processor obtains tasks is greatest for the grid topology. This is surprising — one might expect that the mean task distance would still be proportional to the fair access distance of the topology. However, there seems to be another factor affecting the mean task distance which depends on the shape of the underlying topology.

It would be interesting to experiment with other topologies that all had the same fair access distance, but had different degrees of connectivity. Every node in a line topology has two nearest neighbors, while in a grid each node has four neighbors. In the segmented topology each node is surrounded by a group of processors, all of which are its 'neighbors'. It is possible that the data in Figure 5.16 shows the effect of processors competing for tasks. Additional experiments would show whether this is related to the connectivity of the topology, or some other parameter.

Effect on Local Accesses

An incrementing task search algorithm greatly increases the likelihood that a processor will find a task on its own task queue. Unfortunately, this increased locality of tasks does not produce as significant an increase in data locality.

Figure 5.12 and 5.18 show the ratio of local data accesses to all accesses using the random and incrementing task search algorithms. Most of the benchmarks made 10% more accesses to local memory using the incrementing algorithm, although Quicksort improved by 22%, and the Compiler showed only a 5% improvement. In random mode, the number of local accesses does not depend on the topology. When the search algorithm exploits the topology of the system, the number of data accesses to local memory varies with the connectivity of the system. Note that in

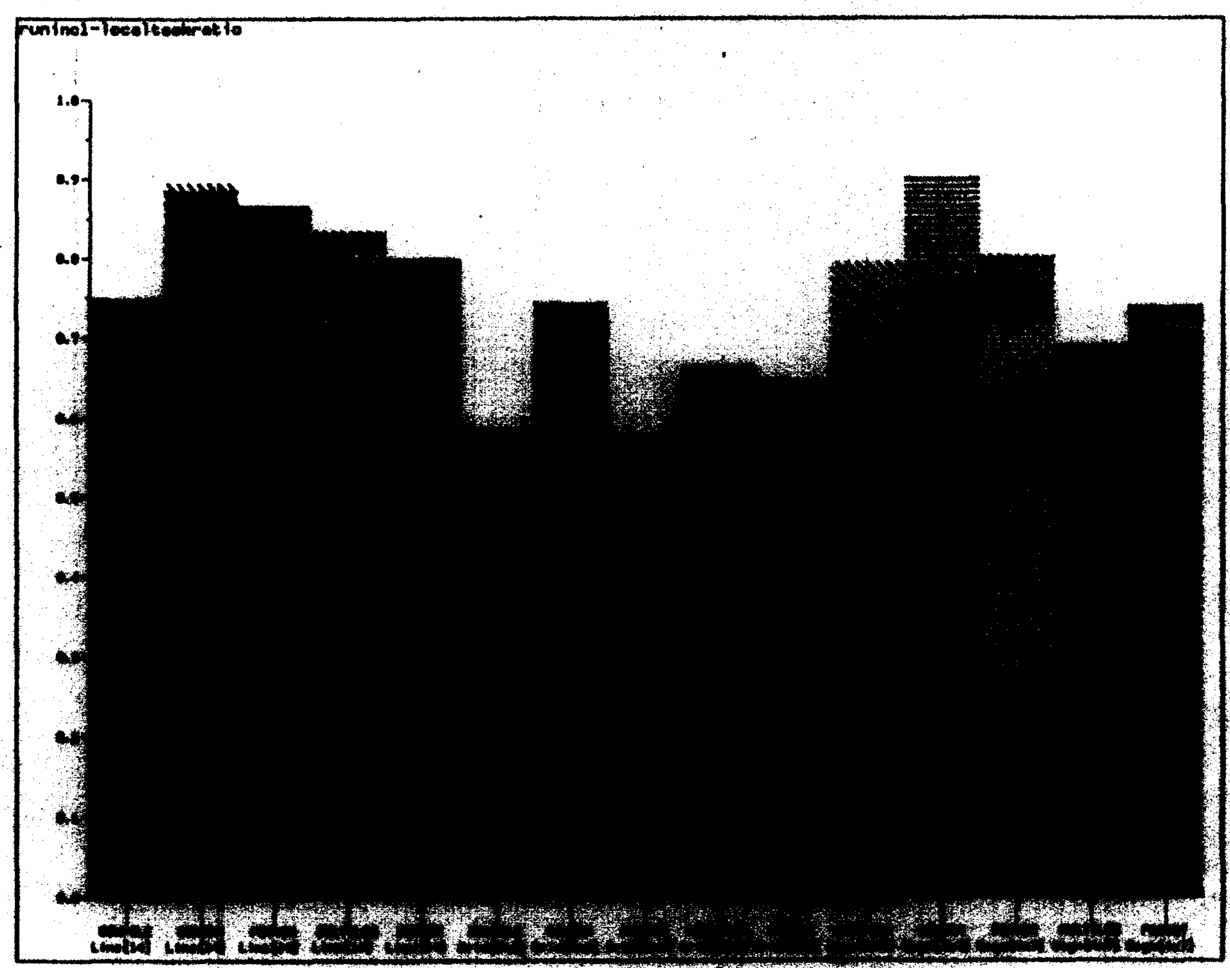

Figure 5.17: Tasks fetched from local task queue, as a fraction of all task fetches. (Top pattern in each column shows variation between runs of each benchmark). Benchmarks were run with Runsched= 1, on three different topologies.

most cases, a program will make the highest percentage of local accesses on the line topology, and the least on the grid topology. This agrees with previous observations of mean task distance. Further experiments with other topologies might show what causes this trend.

Effect on Mean Distance of Access

The effect of task scheduling on the mean distance of tasks is more significant than its effect on mean distance of data accesses. If tasks referred only to data stored in their node of origin, the behavior of data fetches would mimic that of task object fetches. In fact, there is not that much coupling between tasks and data accesses.

While the mean distance of data access decreases with a more intelligent task search algorithm, the improvement is not spectacular. Figure 5.19 shows the mean distance of references for different topologies using an incrementing search algorithm. Figure 5.20 compares the mean distances of random search and incrementing search. The latter ranges from 40% to 95% of the mean distance of the random case.

Changing to an intelligent search algorithm has the most effect on the Quicksort program. With the new algorithm, the Quicksort benchmark has the best overall locality of reference, whereas before it had the worst! The mean distance of access is a volatile measure of locality. The change in mean distance due to changing search algorithms depends on the underlying topology, and varies for different runs of a benchmark. Both Multilog and Consim also often improve with the intelligent task search algorithm.

Figure 5.21 shows the improvement in mean distance for each different type of access. Each stipple pattern shows the mean distance for incrementing search divided by mean distance for random search. So a bar that is less than 1 high means that the locality has improved for that access type. The access types are the same as in Figure 5.14. Note that this graph compares two runs of each program, not the

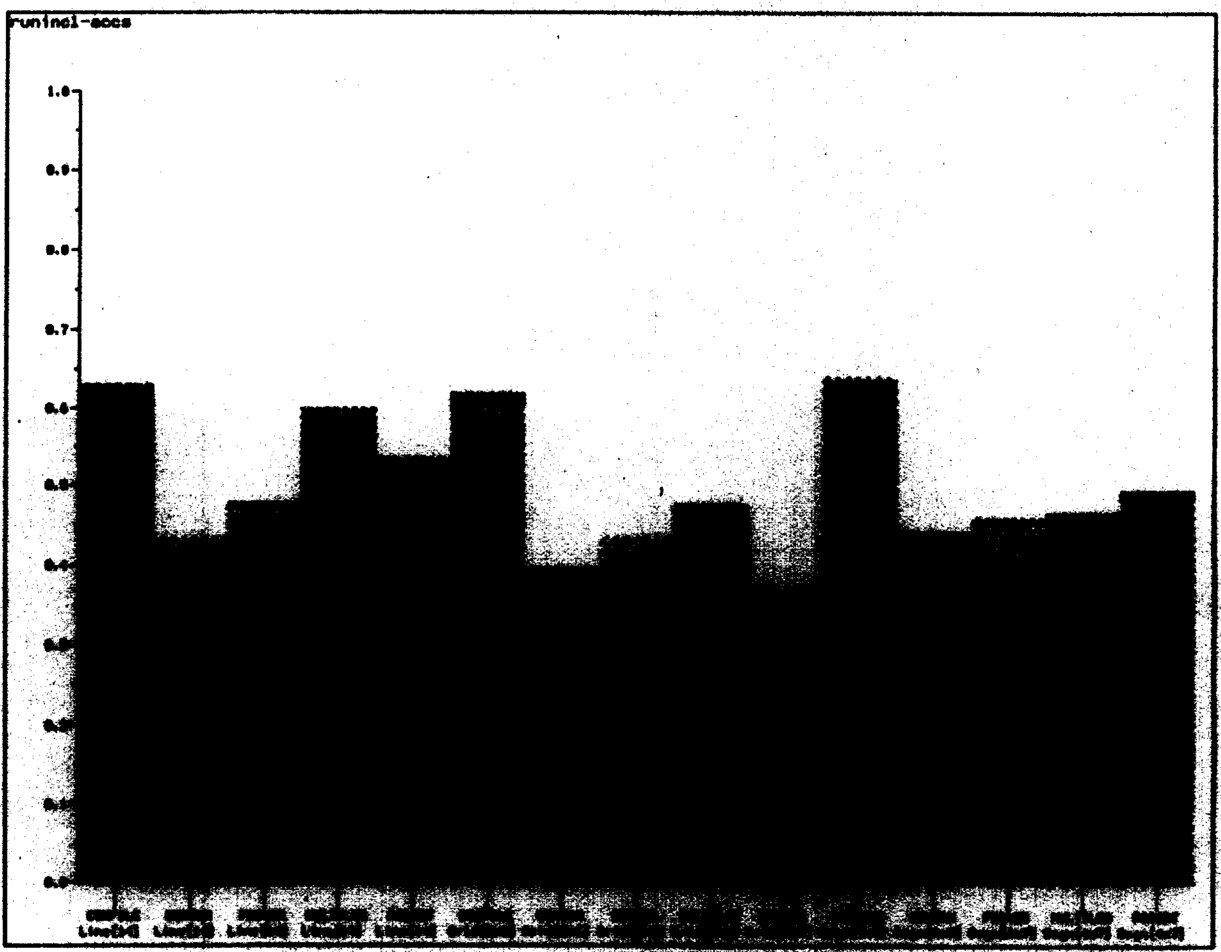

Figure 5.18: What ratio of accesses were made to local memory by each benchmark. (Top pattern in each column shows variation between runs of the benchmark). Benchmarks were run with incrementing task search algorithm and Runsched= 1, on three different topologies.

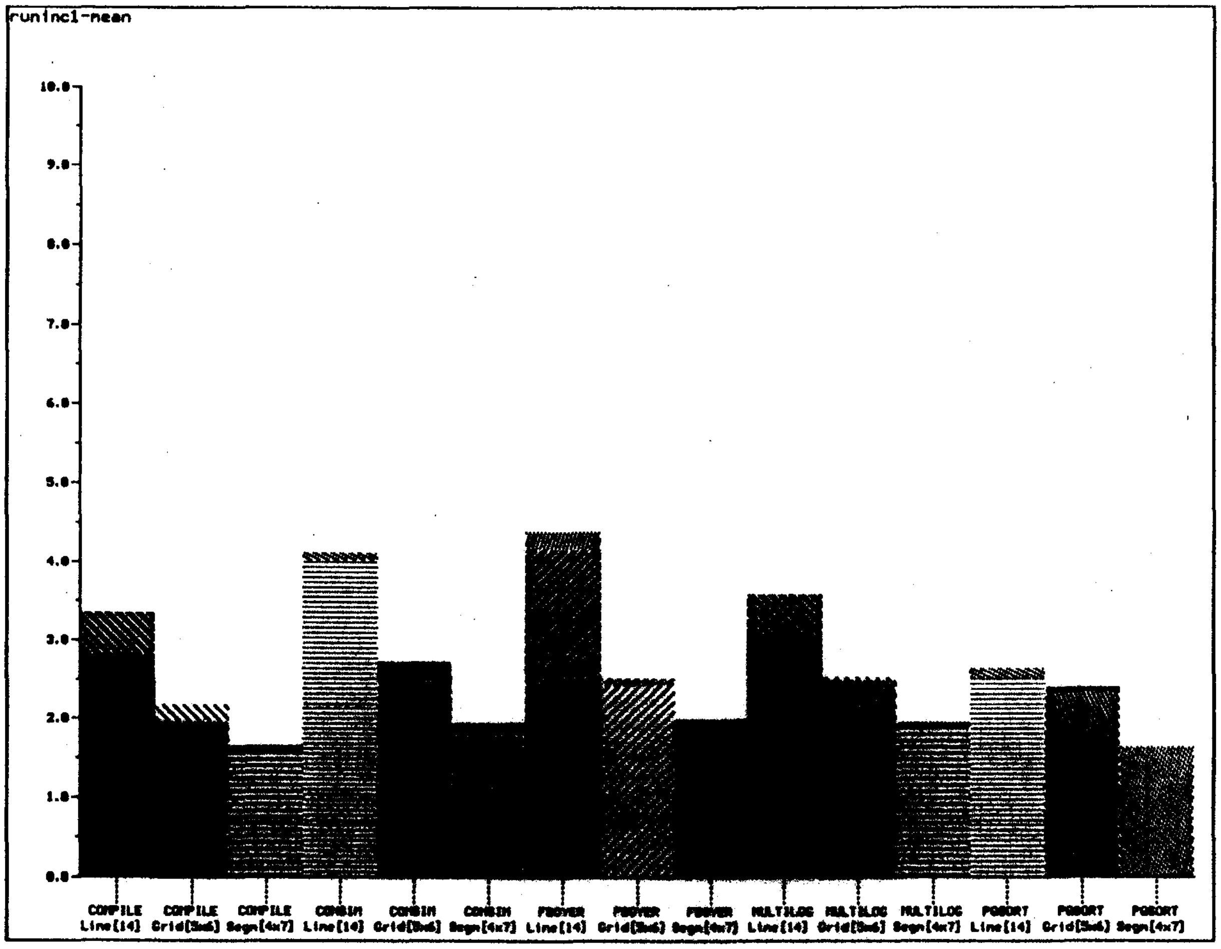

Figure 5.19: Mean distance of data access for each benchmark. Benchmarks were run with incrementing task search algorithm and Runsched= 1, on three different topologies.

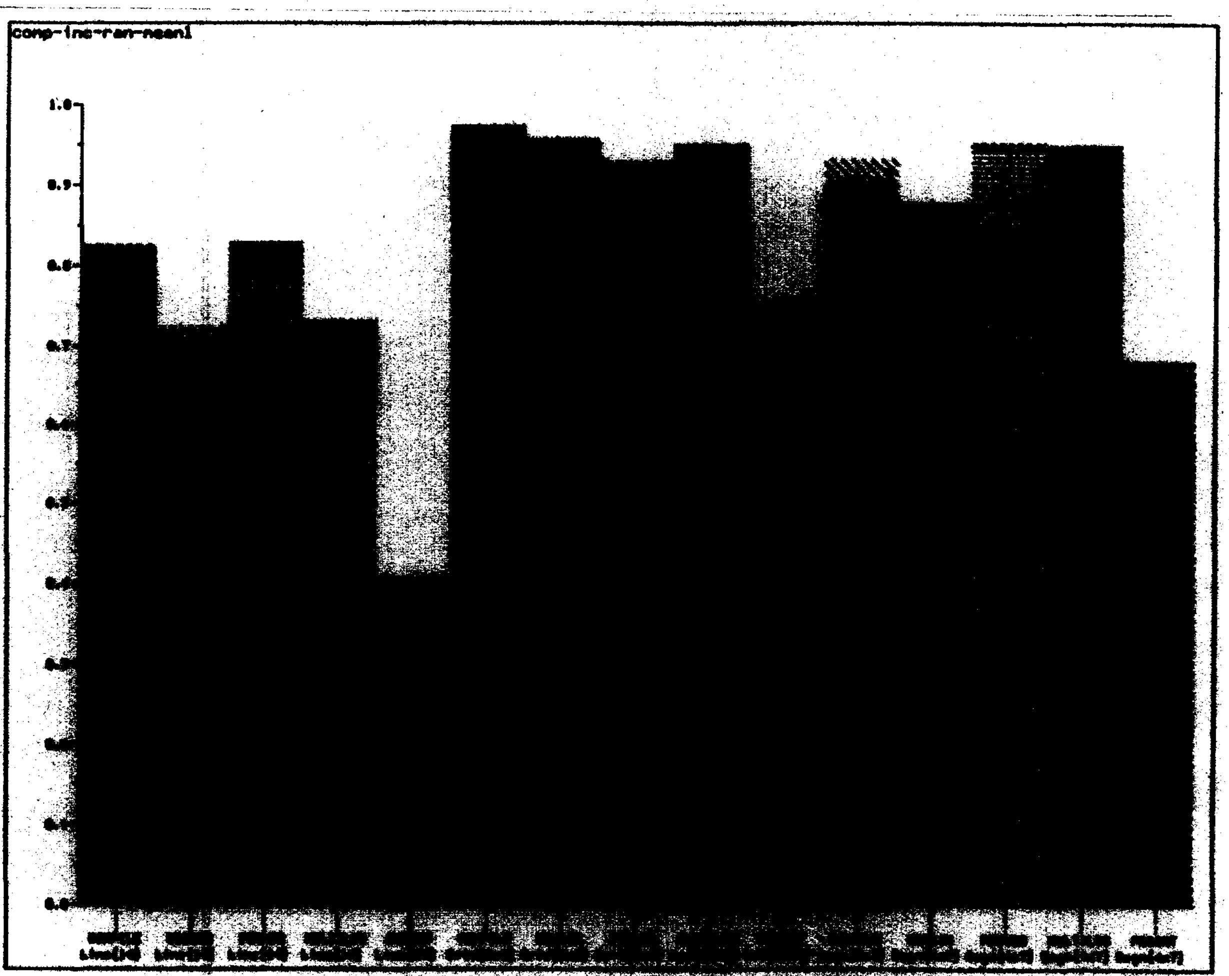

Figure 5.20: Ratio of mean distance of access with incrementing task search to mean distance with random search. Benchmarks were run with Runsched= 1, on three different topologies. (Top pattern on each column shows variation in mean distance for different runs).

average values of several runs. The locality can vary significantly from run to run.

For all benchmarks, the locality of stack accesses, future touches, and structure stores improved with an intelligent task fetching algorithm. Structure fetches also improved for most benchmarks.

However, for the Quicksort benchmark, all types of accesses had better locality using an incrementing search algorithm. The locality of code fetches improved significantly. For one run of Quicksort, environment fetches also improved. Since code, environment, and future objects count for most of the accesses made by Quicksort, the program had much better locality with intelligent task fetching. The reason that Quicksort showed such a significant improvement may be due to the fact that the program fetches more tasks out of task queues than any other. (See Section 5.4.4). Several code, stack and environment fetches are required to load a task onto a processor. If the accesses that load a task are more local when the task is more local, the cost should be greater for a program that fetches many tasks.

We have seen that future accesses had the worst locality when Nusim used a random task search algorithm. Code accesses, structure fetches and stack accesses also had poor locality. We expect that new implementations of Multilisp will cache code locally, and eliminate many global accesses. The results of Figure 5.21 have shown that an intelligent task searching algorithm can do much to improve the locality of reference of stack and future accesses. For programs like those studied here, in which future fetches account for a significant fraction of the data accesses, this improved locality can make a significant difference.

5.4.4 Task Scheduling Parameters

The most significant task scheduling parameter in Nusim is the size of a processor's Process Queue, as set by the variable *Runsched*. A larger process queue will allow a processor to 'cache' more tasks locally ready to run. This might increase the locality of reference for data objects that are shared by those processes, but at the

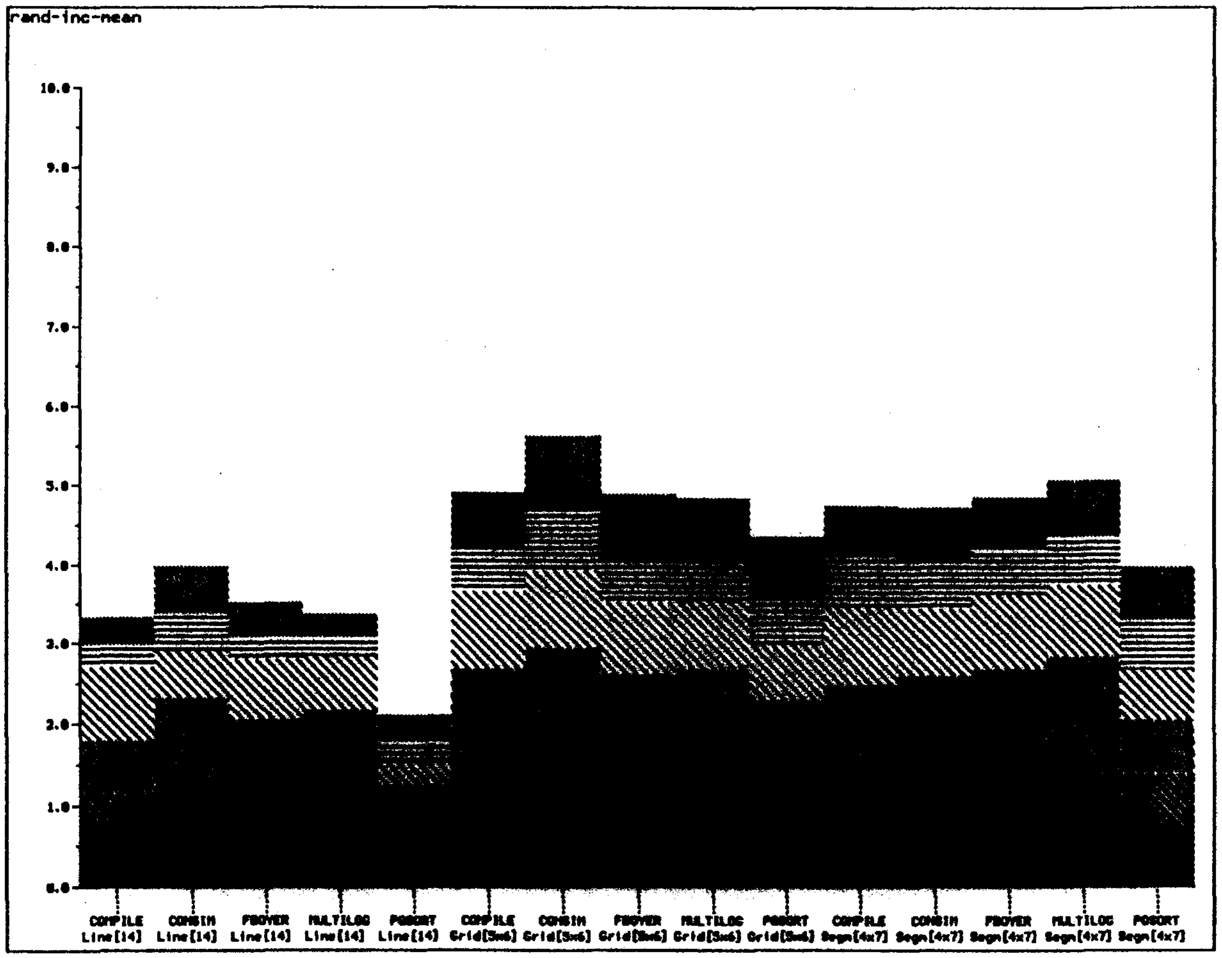

Figure 5.21: Ratio of mean distance of data access with incrementing task search versus random task search. Accesses are divided by data class. From bottom to top, the patterns represent: code accesses, structure stores, structure fetches, environment accesses, stack fetches, and future touches. Benchmarks were run with Runsched= 1 on three different topologies.

cost of decreased parallelism.

This section discusses the effect of Runsched on parallelism and its effect on locality of reference.

Effect on Task Fetching

Figure 5.22 compares the number of tasks fetched from task queues during the execution of different benchmarks on the line topology. It is normalized to show counts of tasks fetched per 1000 instructions executed by each benchmark. (For instance, the Quicksort benchmark fetches approximately 100 tasks per 1000 instructions at Runsched= 1). The lower shading on each column counts the tasks fetched out of a processor's local task queue, while the upper pattern shows how many were fetched out of distant queues.

Note that at greater values of Runsched, each benchmark fetches successively fewer tasks out of task queues. This is expected, since the process queues will act as LIFO queues for tasks spawned on each processor. In Nusim, a task that is forked by a future will be put on the process queue whenever possible. Similarly, a processor will first try to run processes out of its local process queue before searching any task queues.

Increasing the value of Runsched has a similar effect on the number of tasks fetched by each benchmark. At each successive value of Runsched, most benchmarks fetch approximately half as many tasks as before, although the number of tasks fetched by Quicksort does not vary much with Runsched after Runsched=2. At high enough values of Runsched, some benchmarks fetch few tasks from the task queues – they run almost entirely within the process queues.

Task Locality

The locality of tasks indicates how far processors reach to grab executable tasks from task queues in the system. The measures used to quantify the locality of tasks

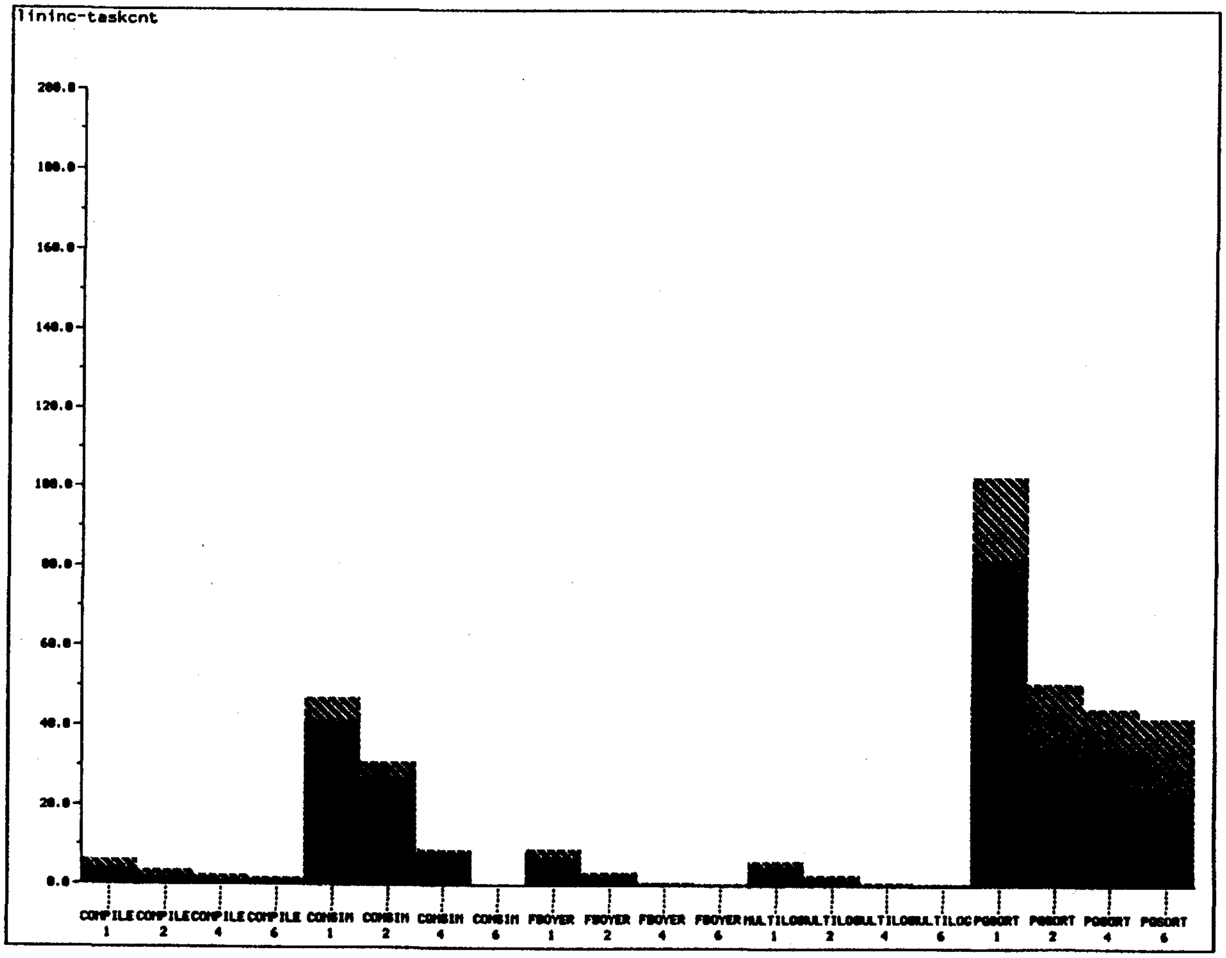

Figure 5.22: Number of tasks fetched from task queues per 1000 instructions. Lower trace is tasks fetched from the local task queue, upper trace is from distant task queues. Benchmarks were run using incrementing task search, on line[14] topology, with different values of Runsched. Each column is labelled with the name of the benchmark and the value of Runsched.

are the same as used for locality of data references: the percentage of references that are local, and the mean distance of references.

Figure 5.23 shows the ratio of tasks fetched out of local task queues to all task fetches. For most of the benchmarks tested, as Runsched increased, the processors fetched a higher percentage of tasks out of distant task queues. The only exception was Consim, which fetched no tasks at all at high values of Runsched. This indicates that a process queue six entries long can contain all the parallel forking of the Consim program.

We can speculate about why processors must grab tasks out of distant task queues at larger values of Runsched. We have seen that the number of available tasks decreases as Runsched increases. When there is little parallelism in a particular program, many processors in the system will sit idle, looking for work. They may have to search more distant task queues to find executable tasks. Meanwhile, if there is a lack of executable tasks in the system, no task will stay in a task queue very long. It will quickly be grabbed by neighboring processors. So once a processor has finished running a process, and starts looking for new tasks to run, it will rarely find any in its own task queue. While this is a simplified model of how the system might behave when there is a shortage of executable tasks, it fits the behavior of Nusim at high values of Runsched.

Figure 5.24 shows the mean distance from which processors fetched tasks on the line topology. This mean task distance increases as Runsched increases.[9] This increasing distance may simply be because of the decreasing ratio of local task fetches.

Data Fetching

The data objects created by programs are independent of the value of Runsched. However, the **number** of data fetches per instruction of the program varies with

[9]Note that Consim at Runsched= 6 is not fetching anything out of task queues.

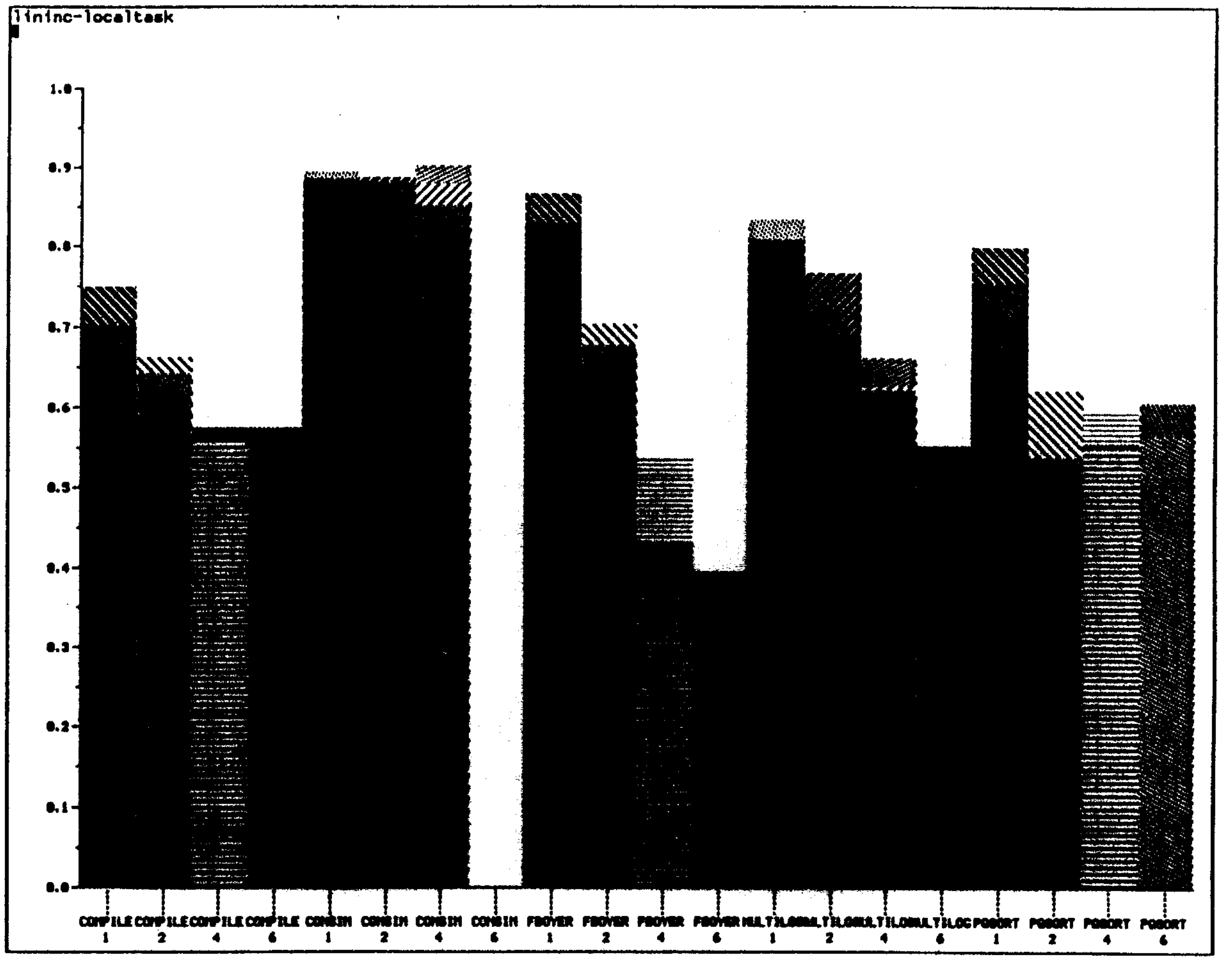

Figure 5.23: Ratio of tasks fetched out of local task queues to all task fetches. Benchmarks were run using incrementing task search, on line[14] topology, with different values of Runsched. Each column is labelled with the name of the benchmark and the value of Runsched. (Upper patterns in each column show variation between runs of a benchmark).

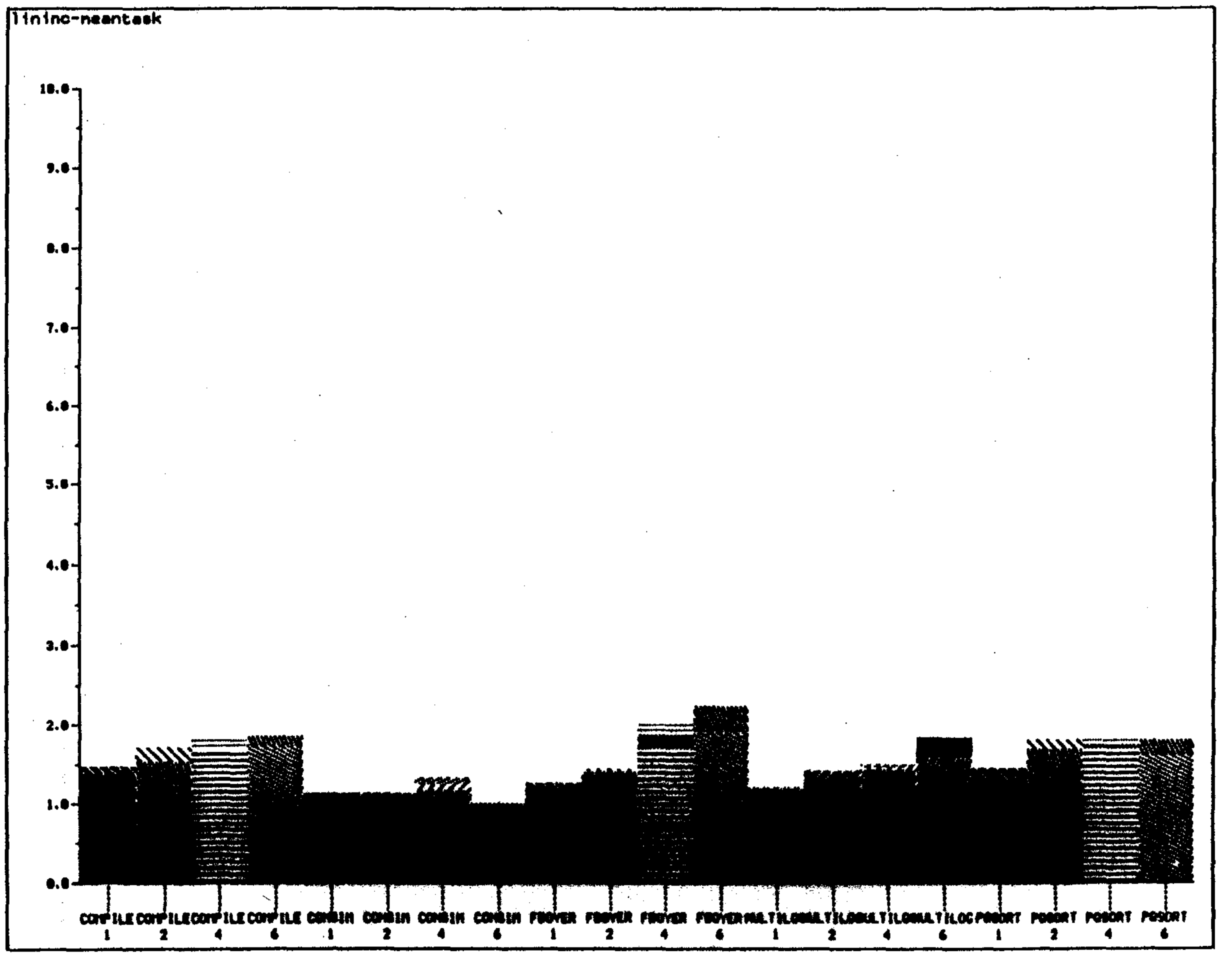

Figure 5.24: Mean distance of task fetches. Benchmarks were run using incrementing task search, on line[14] topology, with different values of Runsched. (Upper patterns in each column show variation between runs of a benchmark).

the size of a processor's process queue. Figure 5.25 shows this relationship. At successively higher values of Runsched, each of the benchmarks makes fewer data fetches. The Consim benchmark makes 20% fewer data fetches at Runsched = 6 than at Runsched = 1.

The proportion of accesses in each of the different data classes is shown in Figure 5.26. The programs make proportionally fewer stack and constant references at higher values of Runsched. This is most evident for Consim and Quicksort, the programs that showed the greatest decrease in number of data accesses. For these two benchmarks, the proportion of global fetches actually increased with Runsched. This indicates that programs are making fewer stack and constant accesses, but fetching approximately the same number of global variables and structured objects.

Future Accesses

Figure 5.27 tracks the number of futures touched by the different benchmarks, normalized per 1000 MCODE instructions. The lower stipple pattern is the number of futures produced, the middle pattern shows how many times futures were touched after they were determined, and the top pattern is how many times undetermined futures were touched. The number of futures produced by each program and the number of times those futures are touched does not vary much as Runsched changes. What varies is the number of times that a process touches an undetermined future.

Table 5.5 shows the number of times that each benchmark touched undetermined futures at different values of Runsched. These counts can vary significantly for different runs of a program.[10] The Compiler and Quicksort programs both touch the most undetermined futures at Runsched= 2, while Fboyer waits on the most futures at Runsched= 3. However, for most of the benchmarks, as Runsched increases, the number of undetermined futures decreases. In an extreme case, such as Consim

[10]Because of the limited number of runs of these benchmarks, I was not able to determine the standard deviations of these measurements.

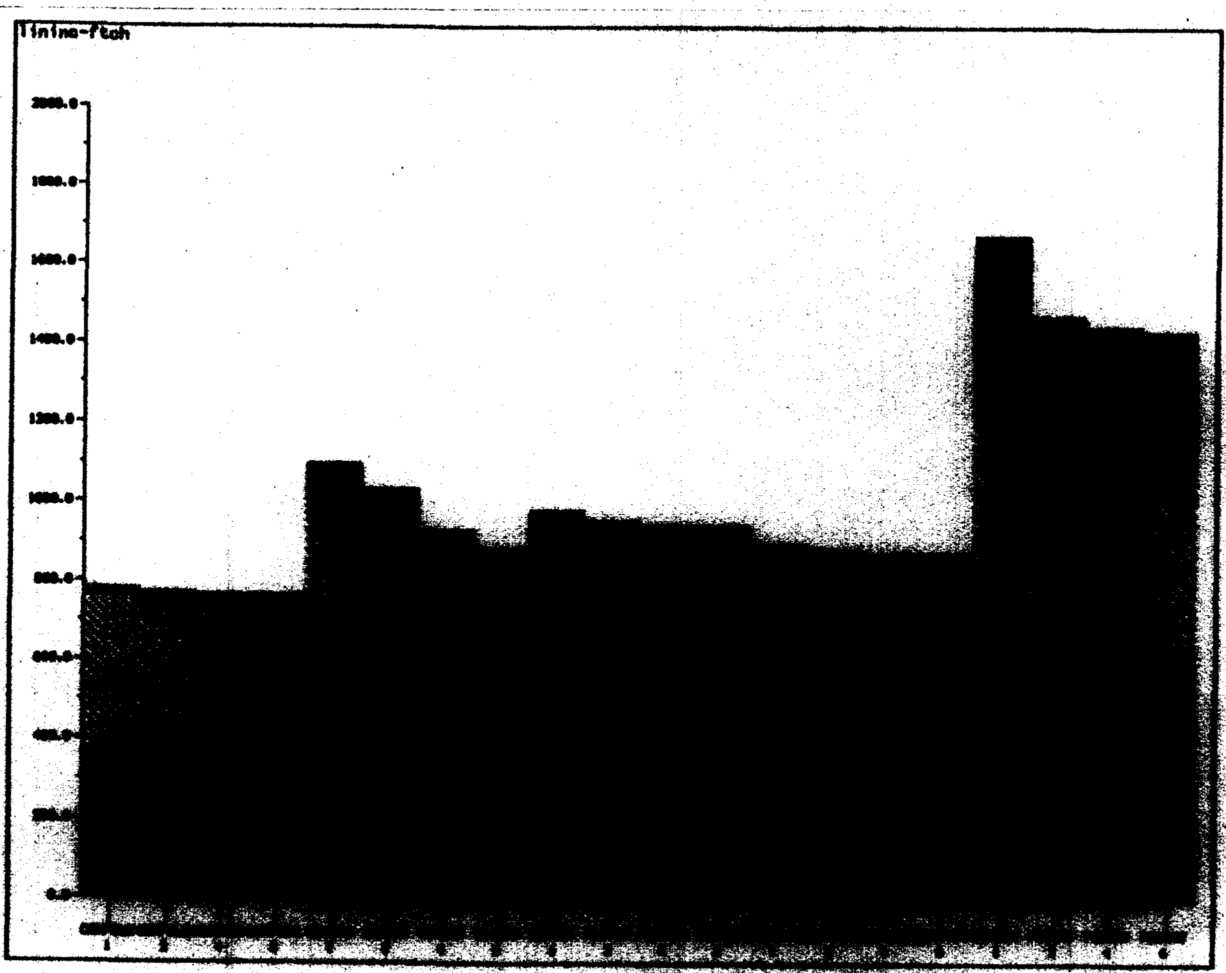

Figure 5.25: Data fetches per 1000 instructions for each benchmark. Each pattern shows a different class of data. From bottom to top: constants, global variables, environment fetches, stack hunks, and future touches. Benchmarks were run with incrementing task search algorithm, on line[14] topology, for different values of Runsched.

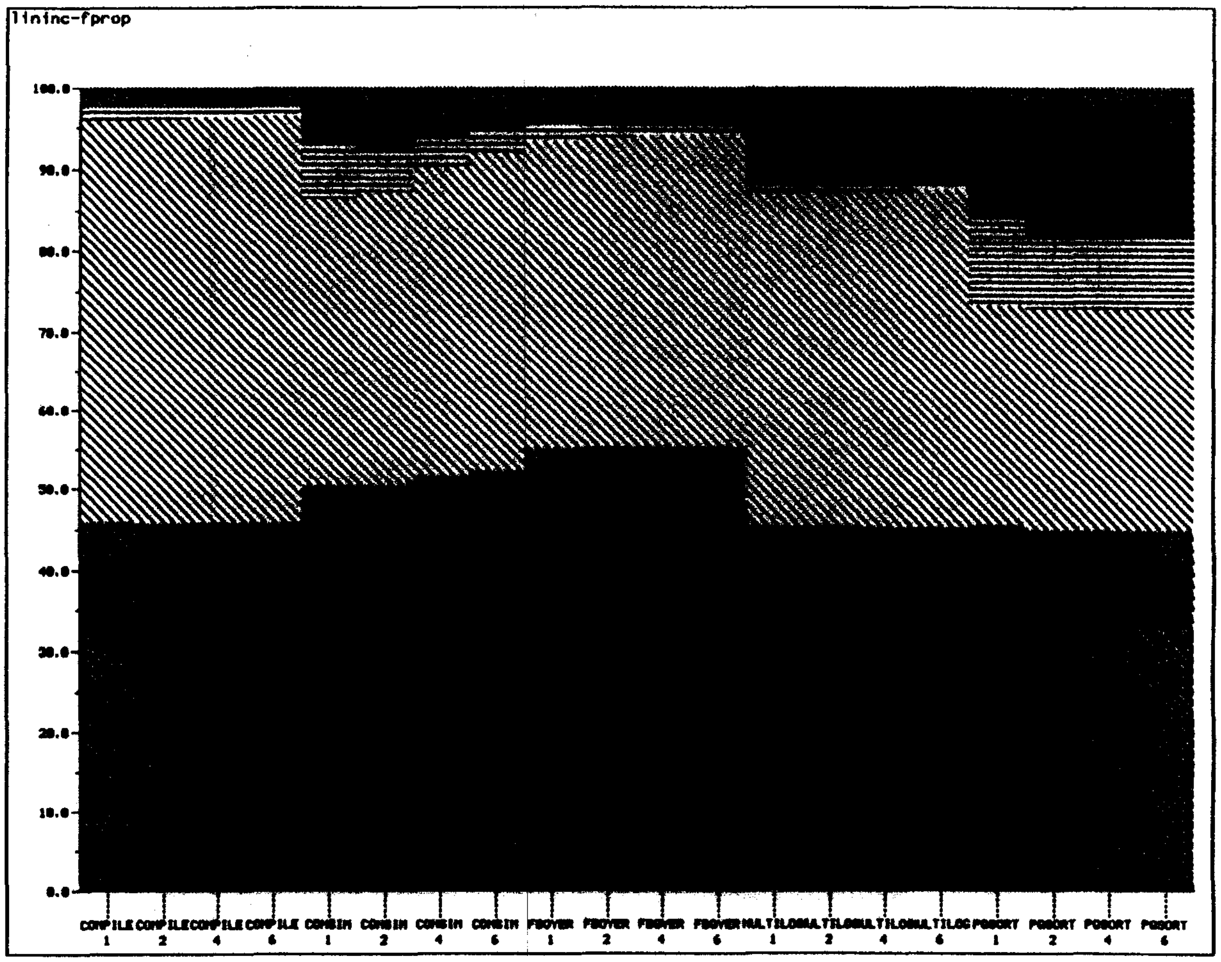

Figure 5.26: Proportion of fetches of different classes of data. From bottom to top: constants, global variables, environment fetches, stack hunks, and future touches. Benchmarks were run with incrementing task search algorithm on line[14] topology for different values of Runsched.

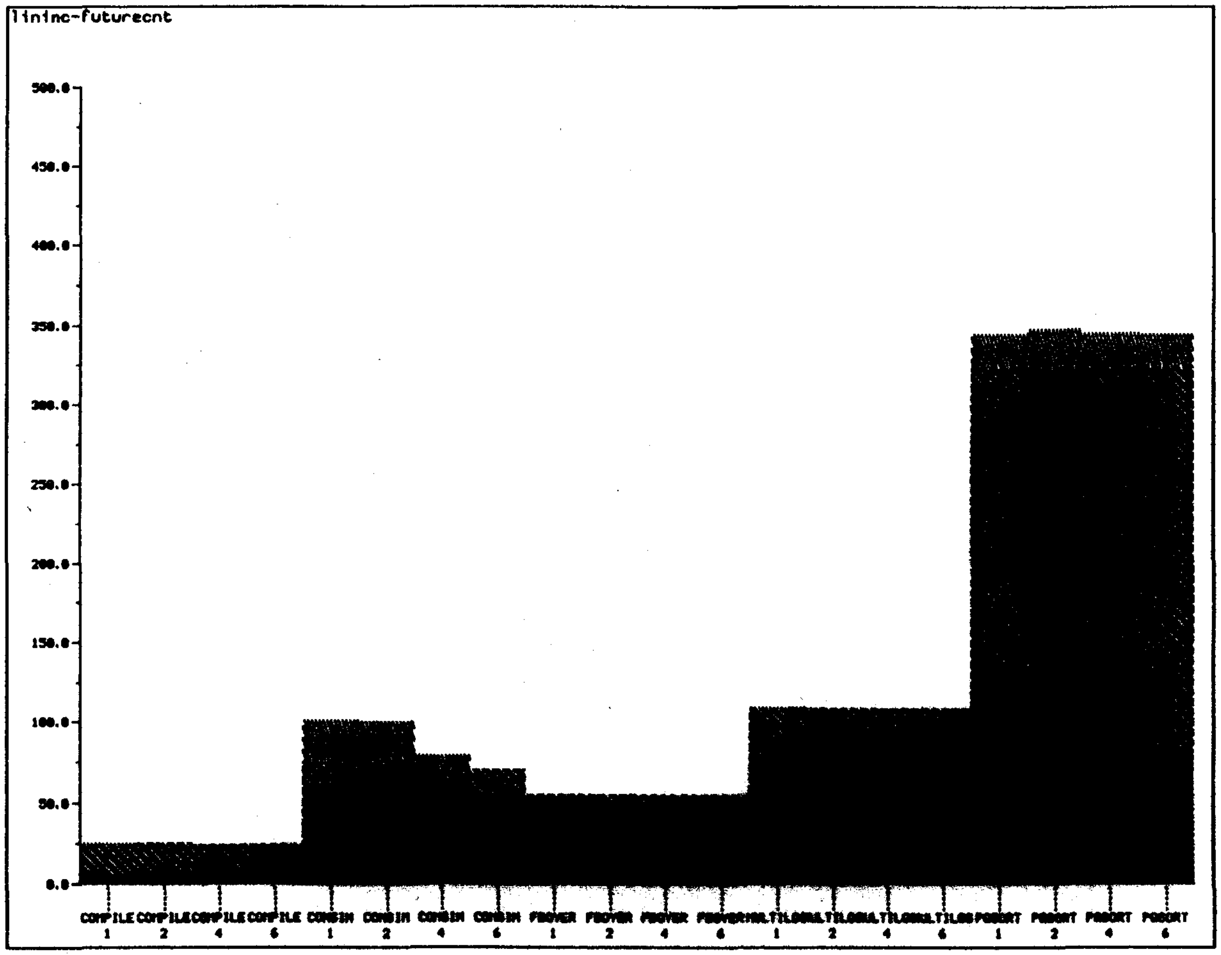

Figure 5.27: Futures created and touched per 1000 instructions. Lower pattern is futures created, middle is touches of determined futures, top is touches of undetermined futures. Benchmarks were run with incrementing task search algorithm on line[14] topology for different values of Runsched.

Benchmark	Runsched			
	1	2	4	6
Compile	0.28	0.30	0.22	0.20
Consim	24.7	24.2	8.52	0.0
Fboyer	0.18	0.15	0.27	0.08
Multilog	0.76	0.54	0.22	0.09
Quicksort	20.5	24.9	21.4	20.1

Table 5.5: Undetermined future touches per 1000 instructions at different values of Runsched.

at Runsched= 6, futures are always determined before they are touched. We have already seen that Consim runs completely on one processor if the process queue is large enough.

Effect on Parallelism

Increasing the value of Runsched has several effects on the operation of the program. Processors fetch fewer tasks from task queues around the system. Programs also fetch less data with increasing Runsched. Finally, most programs touch fewer undetermined futures at higher values of Runsched. These three observations indicate the effect of Runsched on parallelism.

Programs make fewer data fetches when they fetch fewer tasks out of task queues. Every time a processor pushes a process out of its process queue and saves it in its task queue, it must allocate space for the task state in the heap. Later, the processor that loads the task from the task queue to a process queue must make a number of data fetches to read in the task's state. While a processor is running processes in its process queue, it does not have this additional data fetch overhead.

Programs also make more data fetches when they touch undetermined futures. Every time a process touches an undetermined future, it must queue up and wait on that future. Later, when another process evaluates the future, the determining process will restart all waiting tasks. Each task will eventually be loaded into a process queue, which requires several stack and code fetches and an additional environment fetch.[11] This is the extra cost of undetermined futures.

While touching fewer undetermined futures at high values of Runsched may decrease the number of fetches that a program makes, it is likely that fetching fewer tasks has more of an effect. Note that Quicksort touches 5 fewer undetermined futures per 1000 instructions at Runsched= 1 than at Runsched= 2. However, the program makes 205 more data fetches per 1000 instructions at Runsched= 1 than at Runsched= 2. Meanwhile Quicksort fetched 56 more tasks out of task queues at Runsched= 1 as it had at Runsched= 2. Given the number of data fetches required to load a task, it is clear that task fetching is the dominant force here.

It is difficult to see what effect Runsched has on the parallelism of a program. A task is forked for every future instruction executed. For deterministic programs, the number of future instructions and the number of processes created is independent of scheduling decisions. However, the number of tasks in existence at any one time could depend on the scheduling strategy.

For instance, suppose a program forked off tasks in the manner shown by Figure 5.28. At each fork, Nusim spawns a task, and then runs branch 'A' of the fork. The branch 'B' not chosen could potentially have spawned the same number of tasks as 'A'. If another processor was able to run task 'B', twice as many tasks could run in the system simultaneously. In this example, a process queue 5 processes long would be big enough to hold task 'B' as well as all the tasks spawned by branch 'A'. None of those tasks will get put in a task queue, so they will not be available to the other processors in the system. In effect, a large process queue can limit the number of tasks produced per unit time by the program.

Nusim does not produce a parallelism profile that shows the tasks in existence

[11]See Appendix B for the implementation of these operations.

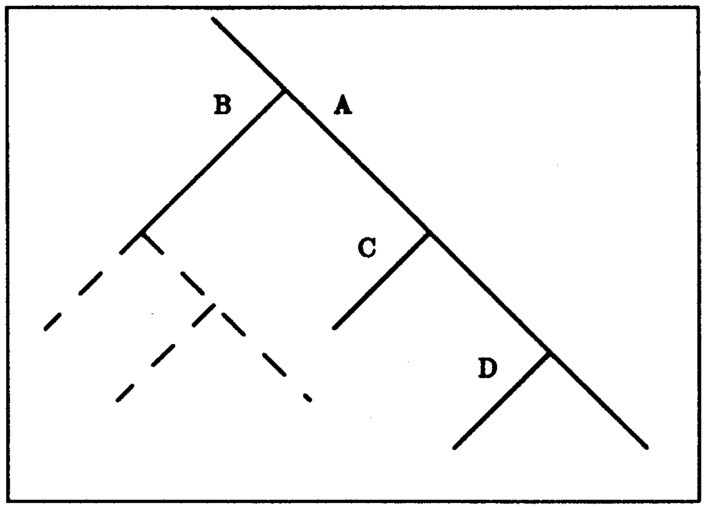

Figure 5.28: Possible order of tasks forked by a program. Path A has been run, path B has not.

at a particular time. Instead, there are only indirect ways of seeing the effect of reduced parallelism in Nusim. The first is to look at the execution time of each benchmark. If processors are idle for lack of available tasks, the run time of the program should increase. Unfortunately, many other factors also affect the run time of a program. It was not possible to extract useful information out of run-time statistics.

A second measure of parallelism tests the imbalance of work among the processors in the system. Nusim collects data from each processor individually. If work is evenly distributed among the processors, each runs approximately the same number of MCODE instructions. If there is a large variance in the number of instructions run by different processors, it could be because some processors are starving for tasks to run. Figure 5.29 shows the deviation between processors as a percentage of instructions executed by a program. Consim at Runsched= 6 has the most deviation, since only one processor is executing all the code. Multilog and Fboyer also show large deviations at high Runsched. By this measure, the amount of parallelism in Consim and in Fboyer decreases with increasing Runsched.[12]

Local Data

As Runsched increases, processes fetch tasks out of more distant task queues. However, data fetches show the opposite trend: programs fetch more data objects out of local memory at high Runsched. Figure 5.30 shows the percentage of data accesses that are local for the different benchmarks as a function of Runsched. In each case, the trend is to make more local accesses at higher values of Runsched. The locality increases by between 5% and 40% as Runsched varies between 1 and 6.[13] For most of the benchmarks, the proportion of local accesses increases by less than 10%.

Mean Distance of Data Access

We should expect that the mean distance of data access would agree with the results for local percentage of accesses presented above. But while the general trends in mean distance of access show that locality increases with increasing values of Runsched, this measure of locality is more volatile.

Using an intelligent task search algorithm on the grid topology, Figure 5.31 shows a trend to greater locality at higher values of Runsched. The locality increases by approximately 10% for most of the benchmarks as Runsched increases from 1 to 6. The exceptions are Consim and Fboyer. At high values of Runsched, both of these programs show a tendency to run completely sequentially on a single processor. In this limiting case, the mean distance of data access falls to 1, since all fetches will be made to local memory.

[12]Note that since this measure does not directly measure the number of parallel tasks in existance in the system at any time, it is only an approximate indication of parallelism.

[13]Though, as already noted, Consim at Runsched= 6 is a special case.

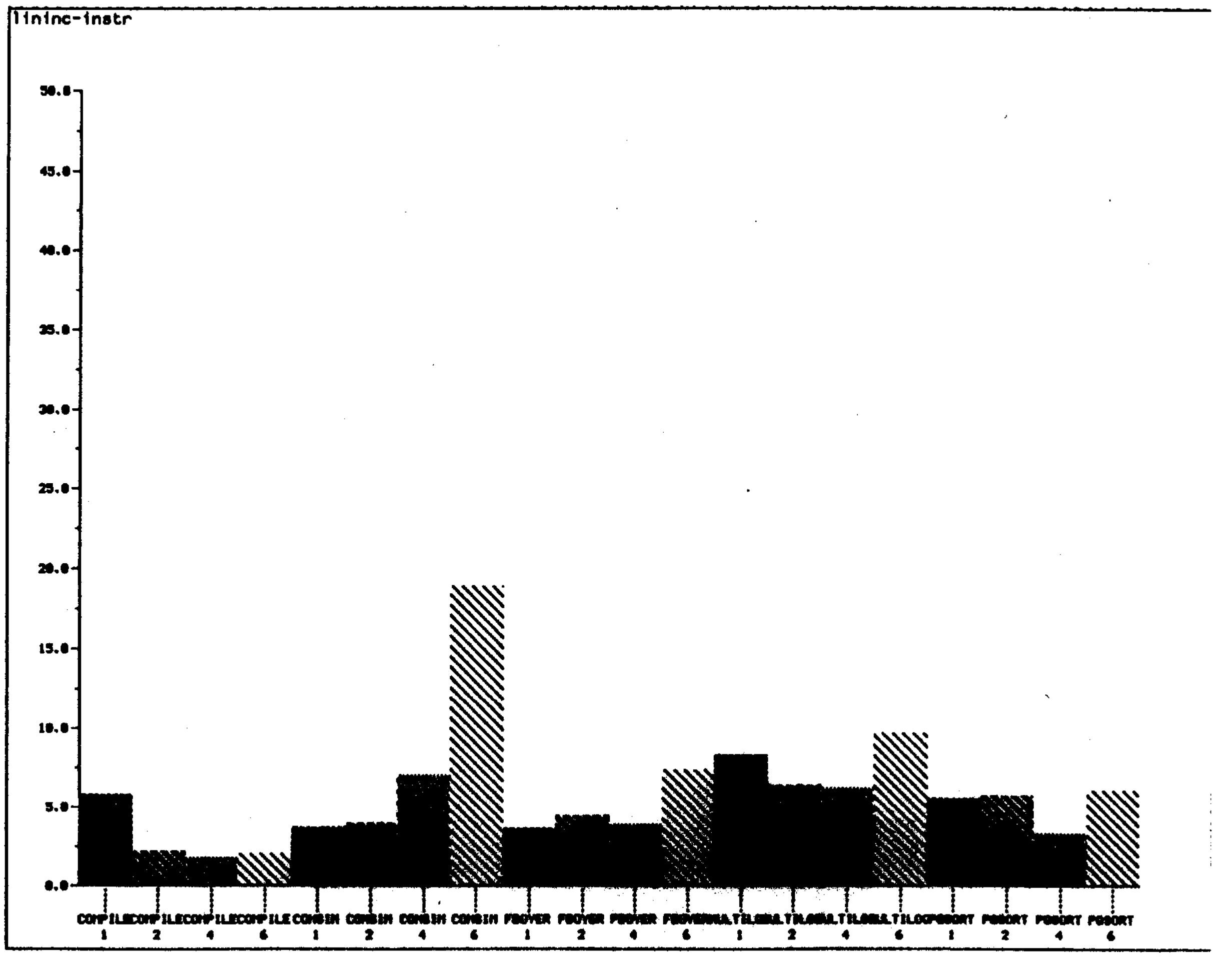

Figure 5.29: Inter-processor variation in instructions executed per processor. The deviation is shown as a percentage of total instructions. Benchmarks were run with incrementing task search algorithm and different values of Runsched on line[14] topology.

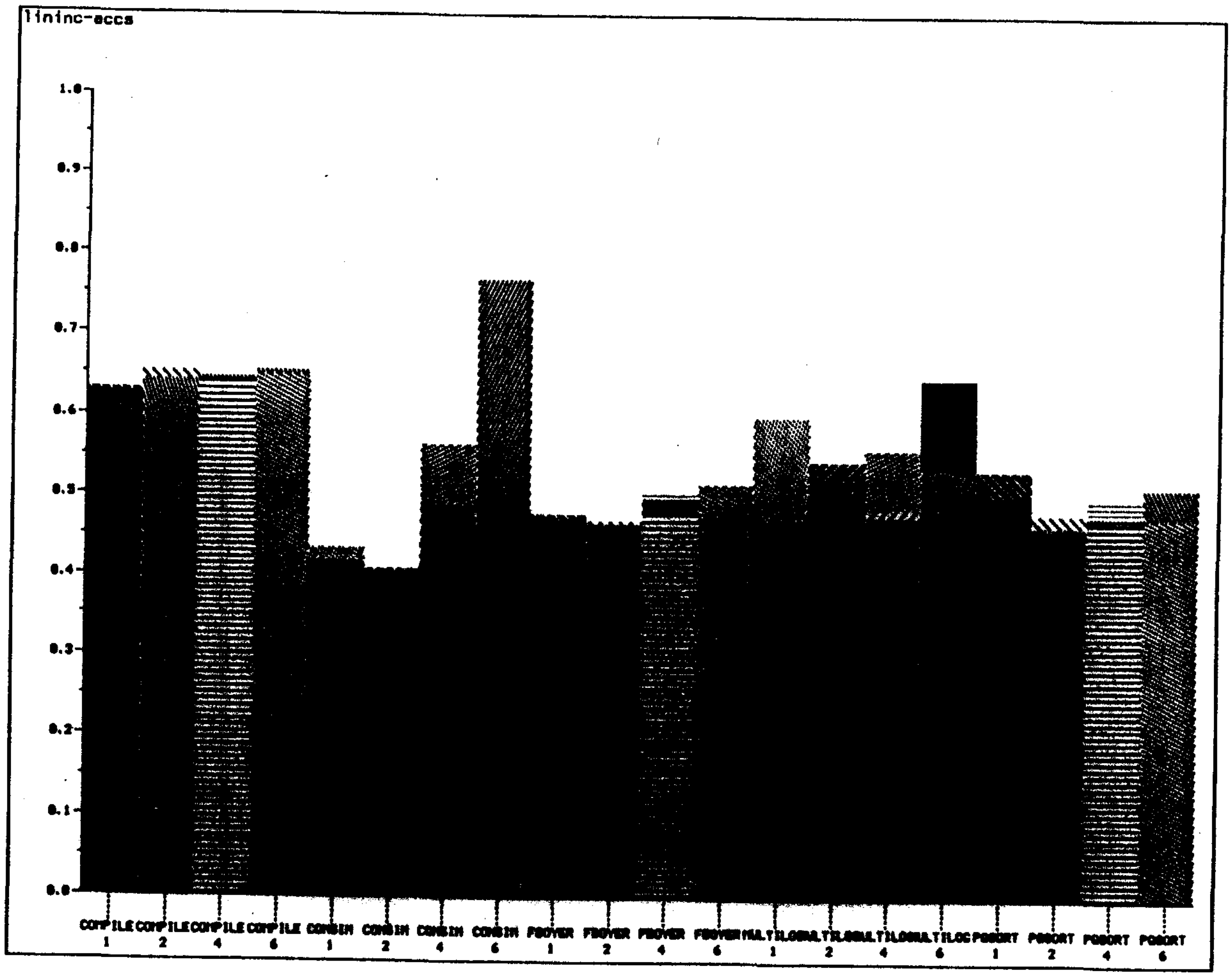

Figure 5.30: Ratio of local memory accesses to all accesses. (Top pattern in each column shows variation between runs of the benchmark). Benchmarks were run with incrementing task search algorithm on line[14] topology for different values of Runsched.

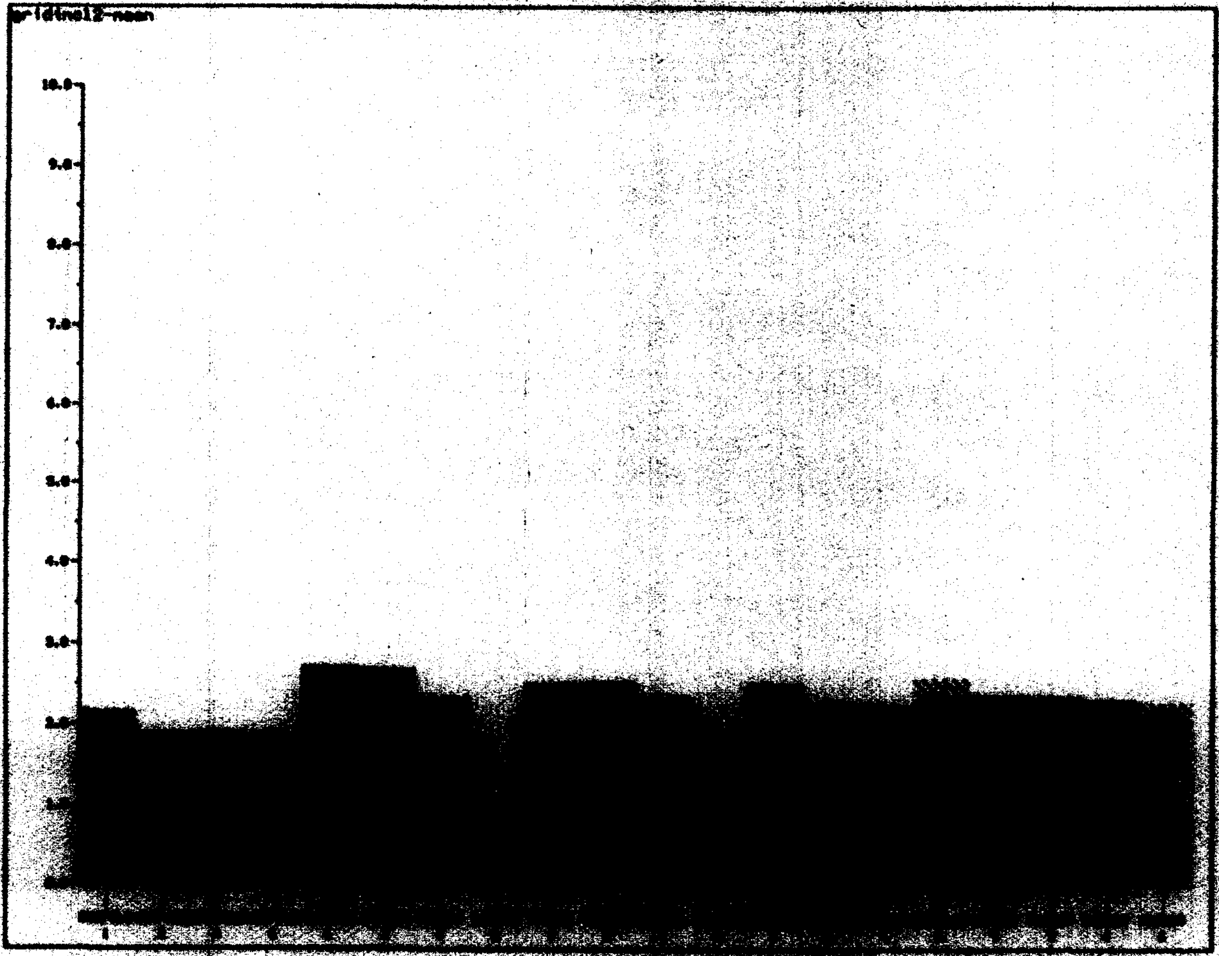

Figure 5.31: Mean distance of data access versus Runsched. Benchmarks were run on grid[5x6] topology using incrementing task match algorithm for different values of Runsched.

Discussion of Locality Improvement

Increasing the size of process queues improves the locality of most types of data accesses. Figure 5.32 shows the mean distance of access for the grid topology, broken down by type of access. For most benchmarks, as the value of Runsched increases, the locality of future fetches and future stores increases as well. For these benchmarks, the locality of future fetches was 10% to 45% better at Runsched= 6 than at Runsched= 1.

Other types of access may become more local at high values of Runsched, though the effect varies from program to program. Structure fetches sometimes improved as Runsched increased. Stack and environment fetches did not improve much for most of the benchmarks tested. The locality of code fetches did not improve at except for test cases where the benchmark ran almost entirely on one processor. In the cases of Consim and Fboyer at Runsched= 6, the processors that run the program seem to be very close to the code.

This data seems to show that certain scheduling decisions can improve the locality of accesses in Multilisp. In particular, future accesses show a great potential for improvement. As discussed earlier, futures were randomly distributed in a system that used a random task search algorithm. Changing to an intelligent task search improved the locality of these accesses. Increasing the size of process queues improved the locality even more.

The results are similar for the segmented topology. On the line topology, the mean distance of data access is much more variable. Using a random task search algorithm, mean distance decreases with increasing Runsched, as expected. However using an incrementing search algorithm, for some benchmarks the mean distance decreases with increasing Runsched, while for others the opposite is true. See Figure 5.33. This graph shows that the mean distance for code accesses is much more variable with the line topology than with other topologies. In most cases, the locality of these code accesses seems to be perturbing the mean distance of access

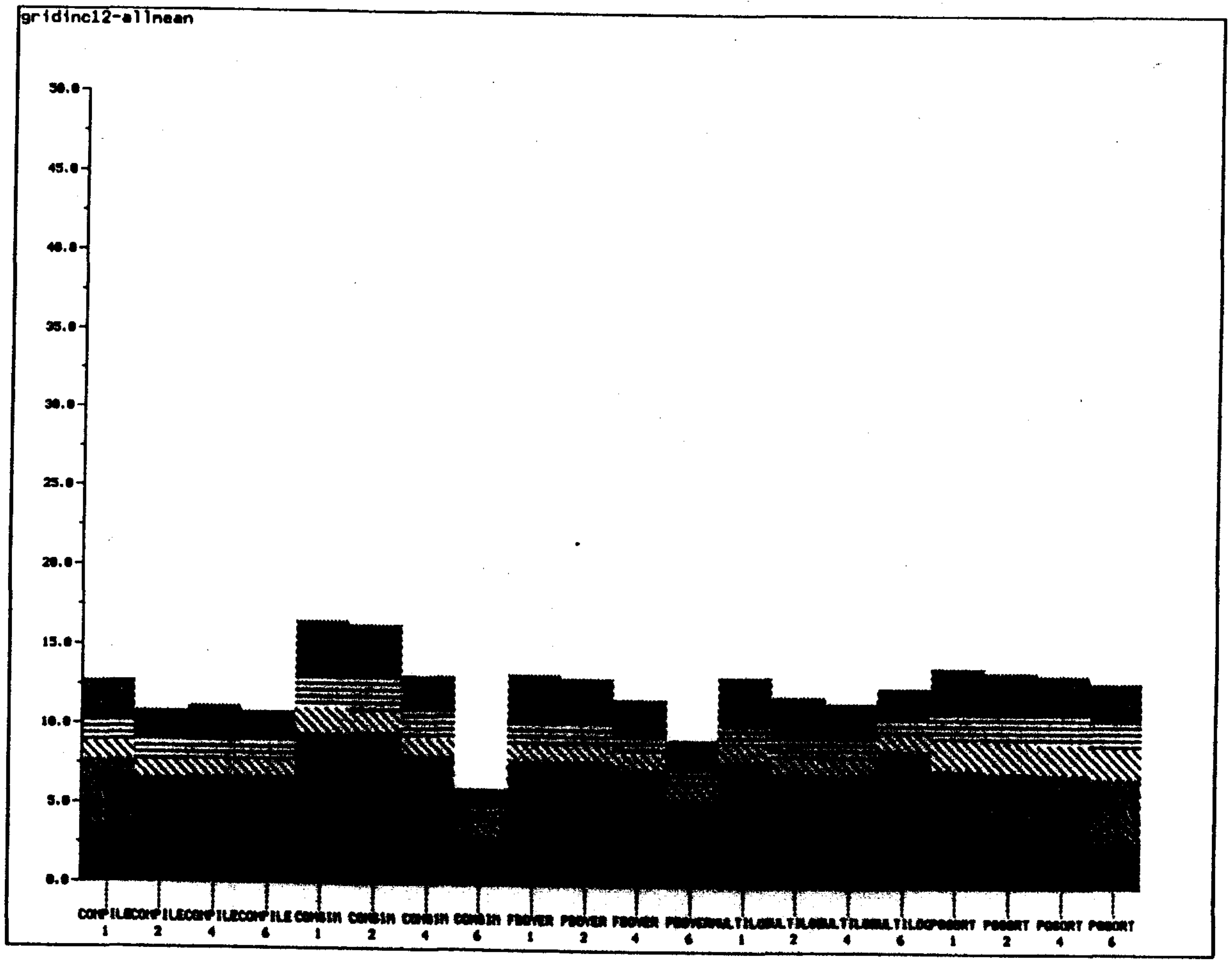

Figure 5.32: Mean distance of access for each benchmark, by type of access. From bottom to top, the patterns represent: code accesses, structure stores, structure fetches, environment accesses, stack fetches, and future touches. Benchmarks were run on grid[5x6] topology using incrementing task search algorithm for different values of Runsched.

under the line topology. It is not clear what is causing this wide variation in the locality of code. We should run more experiments with different topologies to see how much code locality varies with Runsched. However, it is also not clear how significant this effect will be in a real machine.

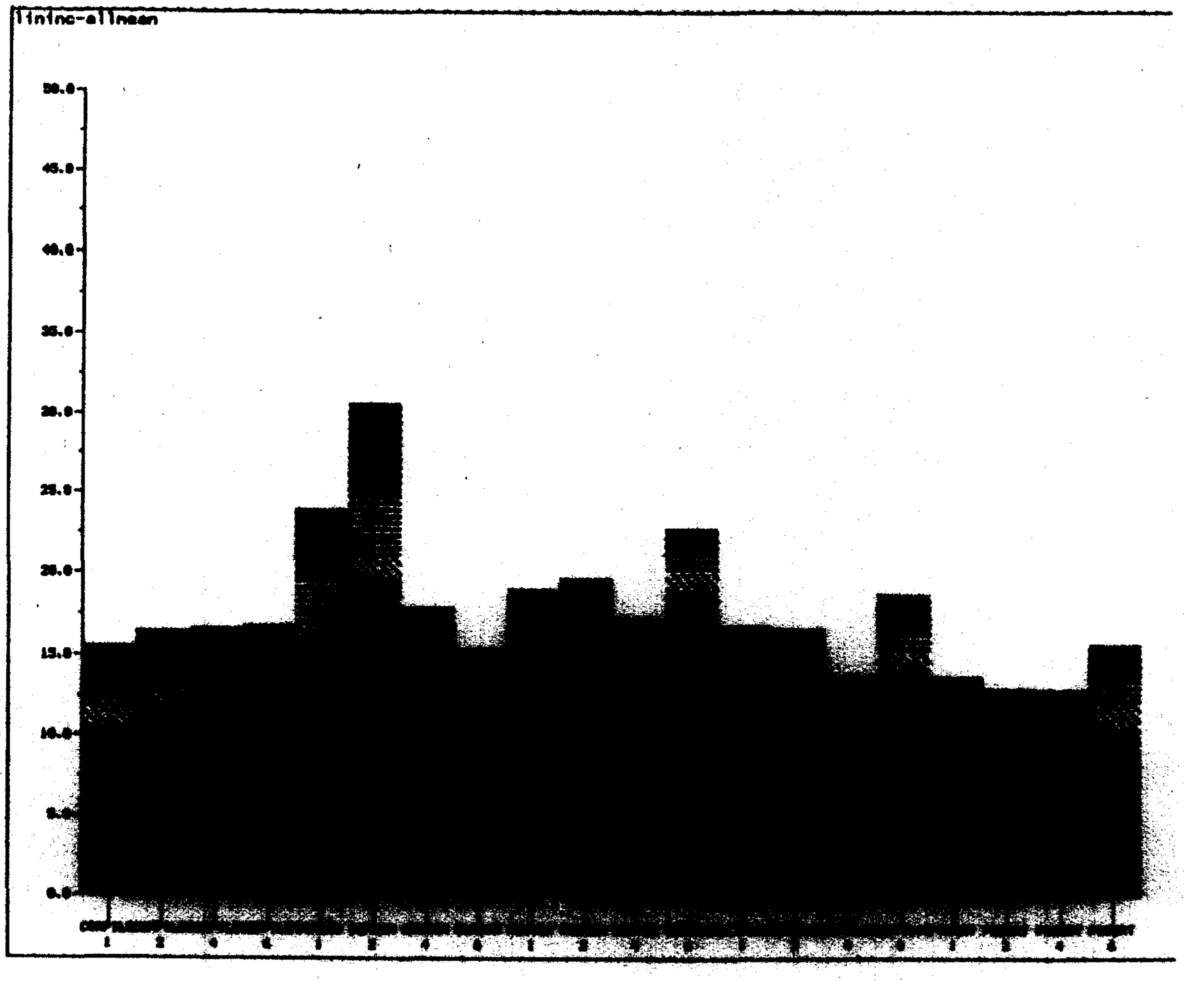

Figure 5.33: Mean distance of data access, by type of access. From bottom to top, the patterns represent: code accesses, structure stores, structure fetches, environment accesses, stack fetches, and future touches. Benchmarks were run on a line[14] topology using incrementing task search algorithm for different values of Runsched.

5.5 Summary

5.5.1 Basic Data and Locality

The results in this section were obtained using a random task search algorithm, and with Runsched= 1. Programs should have the poorest locality of reference under these conditions.

Data Accesses

- The types and proportions of data touched by a benchmark are largely independant of topology.

- The number of accesses per instruction varies by more than a factor of two for the different benchmarks.

  Quicksort was the most intensive benchmark, with 1.9 fetches per instruction. The Multilisp Compiler made the fewest fetches, with 0.8 per instruction.

- All of the benchmarks accessed environment objects at approximately the same rate.

  The programs fetched data from the environment at an average rate of once every 4 instructions.
  Hypothesis: This may be because of the characteristics of compiled code, and the style in which the benchmarks were written.

- Constant fetches were the same proportion of accesses for all the benchmarks.

Futures

- The number of futures produced by the benchmarks varies by more than a factor of 13.

Quicksort creates the most futures, at a rate of 82 per 1000 instructions. The Multilisp Compiler and Multilog produce only 6 futures per 1000 instructions.

- There were 4 times as many future touches as futures created in most of these benchmarks.

  The exception was Multilog, which touched futures 17 times for every future created.

- For most benchmarks, less than 5% of future touches hit an undetermined future.

  The only exception is Consim, in which one third of future touches hit undetermined futures.

Task Locality

- All benchmarks fetch tasks at random.

  The mean distance of task fetches is just the fair access distance. Processors fetch only 5% of their tasks from local memory.

Data Locality

- Most benchmarks fetch 30% to 40% of data out of local memory.

  The only exception is the Compiler, which makes 57% of its accesses to local memory.

- The mean distance of data accesses for these benchmarks is 60% to 90% of the fair access distance.

  The Compiler has the lowest mean distance, followed by Multilog, Fboyer, Consim, and Quicksort.

  This low mean distance of data access is due to the percentage of local accesses made by the benchmarks. Non-local data accesses are randomly distributed through distant memory.

- Futures are randomly distributed in memory.

  Future touches have the poorest locality, followed by code accesses, structured fetches, stack hunks, structured stores, and environment fetches.

5.5.2 Effect of Smart Task Search

For the results in this section, Nusim used an incrementing task search algorithm, and Runsched= 1.

Data Accesses

- The task searching algorithm has little effect on the types of data fetched by the algorithms.

Task Locality

- Processors fetch 60% to 90% of tasks out of local task queues.

  This compares to 5% of tasks fetched from local task queues using the random task search algorithm.

- Mean distance of task access is 1.2 to 1.5.

  The mean distance of task access was the fair access distance of a topology using random task search.

- Mean distance of task access is no longer proportional to the diameter of the underlying topology.

  Mean task distance is greatest for the grid topology.
  Hypothesis: The mean task distance may be related the connectivity of a topology. More tests with different topologies might confirm this.

Data Locality

- Most programs make 10% more accesses to local memory with a smart task search algorithm than with random search.
- The mean distance of data access is 40% to 95% of the mean distance with a random search algorithm.
- Stack accesses, future fetches and touches showed the greatest improvement in locality.

5.5.3 Effect of Runsched

This section shows the effect of varying Runsched on program behavior. Data was collected using an incrementing task search algorithm.

Data Accesses

- Programs made fewer data fetches at greater values of Runsched.

  Most benchmarks made 20% fewer fetches at Runsched= 6 than at Runsched= 1.

- The benchmarks fetched fewer stack hunks and less constant data at high values of Runsched.

  There was little change in the number of global variables and structures fetched.
  Hypothesis: Most of this effect may be due to programs fetching fewer tasks at high values of Runsched.

Futures

- Programs created the same number of futures but touched fewer undetermined futures at large values of Runsched.

Task Locality

- Programs fetch fewer tasks out of task queues at high values of Runsched.

  Some programs fetch no tasks out of task queues when the process queues are large enough. A process queue six entries long can hold all the parallel forking of the Consim program.

- Programs fetch a lower percentage of tasks out of local task queues at large Runsched.

- The mean distance of task fetching increases as runsched increases.

  Hypothesis: Processors may be contending more for tasks when there is less parallelism in the system. Running simulations on other sorts of simulators may show the effect of contention for resources in different topologies.

Data Locality

- Benchmarks fetched a higher proportion of data from local memory at higher values of Runsched.

  The percentage of data fetched from local memory increased by 10% between Runsched= 1 and Runsched= 6.

- The locality of data accesses improved as Runsched increased.

  The mean distance of data access was 10% lower for Runsched= 6 than for Runsched= 1.

- Future objects showed the greatest improvement in locality as Runsched increased.

  The mean distance of future accesses was 10% to 45% better for Runsched= 6 than Runsched= 1. The locality of structured data fetches also improved somewhat as Runsched increased.

Chapter 6

Conclusion

6.1 Review of Goals

The initial goals of this thesis, as stated in Section 1.1.1, were to quantify the communication requirements of one model of symbolic computing.

The first objective was to find out what types of data are accessed by Multilisp programs. Some classes of data objects can be cached locally to a processing node, while other data must be accessible to all processors. The number of data accesses in each of the different classes then helps quantify the communication load of a Multilisp program.

A second objective of this thesis was to see how data is distributed in a multiprocessor system. This locality of reference indicates whether data is clustered near processors that use the data. The greater the locality of reference in a system, the lower the load on the global communications network will be. Determining the locality of different types of data objects shows where the greatest potentials are for reducing global communication.

The final goal of this research was to see how scheduling decisions can affect the locality of reference. One objective was to determine whether the locality of data access of a program can be improved by knowing the topology of the system.

Another was to see what effect changing parameters of the processor architecture would have on locality. This research could be used to predict the effect of different architectural decisions on the global communication requirements of symbolic programs.

6.2 Results

The detailed observations of running benchmark programs on Nusim were presented in Section 5.4. Those results were also summarized at the end of Chapter 5. What follows is a brief discussion of how those results relate to the original goals of the thesis.

The programs studied in this research exhibited some locality of reference, even when Nusim was ignorant of the underlying system topology. Processors accessed local memory proportionally more often than distant memory. This is because new data produced by programs is always allocated in local memory. That newly allocated data is likely to be referred to again by the task that created it.

The locality of reference of Multilisp programs improves when task scheduling algorithms exploit knowledge of the underlying topology. The task scheduling routines used in this study tried to increase the locality of task fetching by searching nearby processing nodes for executable tasks. This was successful technique in that it reduced the mean distance of task fetching in the system by 50% to 80%. This intelligent task search algorithm did not have as great an effect on the locality of data accesses. However, the locality of data references for the benchmarks tested improved by 5% to 60%.

Finally, increasing the value of Runsched increases the locality of data fetches, but decreases that of task fetches. The immediate effect of increasing Runsched is to decrease the parallelism of Multilisp programs. Processors fetched fewer tasks, and had worse locality of task fetching for increasing values of Runsched.

However, the locality of most types of data objects increases as Runsched increases. This improvement in locality is greatest for future objects. So it seems that it is possible to increase the locality of data references while reducing the parallelism available in a particular program.

6.3 Additional Questions

6.3.1 Contention for Tasks

The data presented in Section 5.4 showed some of the effects of parallelism on task fetching for different topologies. But this data did not explain what mechanisms caused this behavior. One specific question is what effect contention for tasks has on the the mean distance of task grabbing for different topologies. There are indications that the task fetching behavior when processors contend for tasks is somehow related to the connectivity of the topology being simulated.

Nusim is not the ideal vehicle for studying these effects, since in running different types of benchmarks it only allows indirect control over the number of tasks produced by a program. It also does not keep track of contention for tasks by processors in the system. However, a new simulator being developed in the Parallel Processing Group at M.I.T. may be able to show this behavior more clearly [35]. This event based simulator will allow users to simulate processors as task generators and consumers, and to see how contention for tasks is related to the topology of the system.

6.3.2 Implementation Issues

Nusim is one possible implementation of a symbolic language on a multiprocessor. In this thesis, I tried to investigate the part of the behavior of Multilisp programs that is independent of the implementation of the language. I hope that the trends shown here for the effect of scheduling decisions on locality of reference are true for a larger class of programs and implementations than five benchmarks running under Nusim. However, some of the basic decisions that were made in the implementation of Nusim may affect the data presented here.

Use of Local Memory

One important part of the implementation of Nusim is that it assumes that local memory is faster to access than distant memory.[1] For this reason, Nusim always allocates new data objects that were created by a processor out of the processor's local memory. This is a preliminary effort to improve the locality of reference of Multilisp programs. Because new data is allocated out of local memory, programs running under Nusim always show some locality of data accesses.

The decision to allocate new data out of local memory is a sensible one in a system like the one proposed in Chapter 3, in which local memory is faster to access than distant memory. It also has the effect of decreasing the load on the global communication network. This is a desirable goal if the network is likely to be a bottleneck in the system. However, it is possible to imagine multiprocessor organizations in which this is not the case [25].

Process Scheduling

Another important feature in the implementation of Multilisp is its 'unfair' scheduling strategy [30]. When a future forks a process, this algorithm always runs the child process first, before returning to the parent. An alternative would be to continue running the parent process, and allow the child to migrate to another processor. This different sort of scheduling strategy might affect the parallelism available in Multilisp programs. It could also affect the locality of task fetching by processors,

[1] Perhaps a more fundamental assumption is that the system has both local and distant memory, and a single shared address space.

and possibly the locality of data references as well. Nusim can use several different scheduling strategies for future tasks. A topic for further experiments is how these strategies for running parent and child tasks affect the results shown in this thesis.

Weights of Data Accesses

Appendix B details how Nusim was instrumented to track different types of data accesses, and where this data was located. There is no need to describe that function here, except to say that there are many ways that Nusim **could** have counted different data accesses.

For instance, fetching the value of a mutable global variable is assumed to have the same cost fetching a stack hunk. In fact, for a processor that uses a stack buffer and must load and unload stack hunks from that buffer, this might not be true. A processor might fetch each word in the stack hunk that is loaded into the stack buffer. Or in a different implementation, a processor might be able to load an entire stack hunk across the network in one access. These two approaches might impose a greater load on the communications network than a simple variable fetch. Rather than assign a different 'weight' to stack hunk fetches corresponding to the expected implementation on a future machine, I simply counted the stack fetches. It is then possible to scale these results to different implementations by multiplying the number of stack loads by the expense of that stack access.

Another important point about the results of this thesis is that I have not presented any information about the rate of data **allocation** by different Multilisp programs. All data is allocated in a processor's local memory, so it should not affect the global communication cost for Multilisp programs. However, it does provide an interesting data point in rating the relative importance of local memory versus global communication. Presumably, since all allocation requires memory accesses, programs that allocate a lot of data demand fast accesses to local memory.

Instruction Set and Compiler

One of the observations in Section 5.4.1 was that all the programs tested accessed environment objects at approximately the same rate. This may be a result of the type of code produced by the Multilisp compiler. As mentioned earlier, if the compiler did common sub-expression elimination, many of the instructions that read values out of the local environment would be eliminated. This would certainly affect the division of accesses among different data types. It might also affect the measures of locality of reference seen in this thesis. It is unclear whether any of the global data references counted by Nusim could be optimized out by a better compiler.

A different compiler might be able to affect the execution of Multilisp code in another way. It may be possible to do some analysis of program structure at compile time. A compiler might then be able to offer hints about scheduling tasks to the run-time system. For instance, a compiler could tell whether data allocated by a process was shared with other processes. Data private to a process might be allocated in a processor's local memory, but then the one process that refers to it should not be allowed to move to another processor. Similarly, the compiler could indicate whether a child task shares much data with its parent. In such a case, it may be better for locality of reference not to allow the child to migrate to another processor.

A different set of questions concerns the efficiency of an implementation of Multilisp. One hopes that the data collected for this thesis does not fundamentally depend on the run time characteristics of programs. For instance, a more efficient implementation of Multilisp might be able to run instructions at a higher rate. The instructions would still refer to the same data objects, so processors would load the communications network more heavily over the life of the program. However, in a different implementation of Multilisp, the ratio of processor speed to global access time might change. It may not be possible to speed up data fetches much. This difference in the relative rates of operations in Multilisp could change the run time scheduling of tasks in the system. It remains to be seen whether this would have any effect on the parallelism of programs, or on the locality of reference. Since Nusim provides a fairly simple mechanism for changing the simulated speed of different operations, I intend to check this conjecture at a later date.

Garbage Collection

One final important aspect of the Nusim implementation of Multilisp is its garbage collector. In Nusim, the garbage collector operates concurrently with program operation [30]. It has two interesting effects on the operation of Multilisp. The first is that the garbage collector replaces determined futures by the value of the future. That means that a future that is in existence for a long time is likely to be turned into a simple Lisp data object. It is unclear how much this affects the counts of future touches reported in Section 5.4.1.

The second property of garbage collector is that it moves data objects around in the system. Since the garbage collection is distributed, a particular processor might follow pointers from an object in its local memory to another data object in old space.[2] The object being pointed to is then moved from old space to the new space of the local processor. If the object was previously in old space in another processing node, it will be moved across the system to a new node. A processor is likely to move data objects that are referred to by the processes that it is running. In general, this means that data will tend to be moved by the garbage collector to be closer to tasks that use that data. This should improve the locality of reference of those tasks.

It is not clear how these properties of the Nusim garbage collector affect the data presented here for the locality of reference of Multilisp programs. Preliminary experiments by other members of the Parallel Processing Group have not shown

[2] See [11] for a more detailed explanation of these terms.

any direct evidence that the Nusim garbage collector improves locality of reference [45]. I intend to do another set of experiments to quantify the effect of the garbage collector. That would show whether new implementations of Multilisp, possibly with different garbage collectors, should expect to have the same behavior as was shown in this thesis.

6.4 Evaluation

One of the major motivations of this research was to help guide architectural decisions in the design of future symbolic multiprocessors. To that end, this thesis tried to answer some specific questions about data access and communication in Multilisp programs.

Section 6.2 shows that in this respect, the thesis has accomplished its goals. I have been able to quantify the communication requirements of a number of different Multilisp programs. I have also been able to see how that communication load varied with different scheduling decisions and processor designs.

This data was collected by building the *Nusim* simulator and running a number of different Multilisp applications under it. This seems to have been a successful technique for studying architectural issues. This research was possible in part because Nusim was a tool specifically designed to measure this type of data.

Evaluation of Nusim

The Parallel Processing Group at M.I.T. has some tools for studying the behavior of Multilisp programs. These tools were used primarily to show the parallelism available in different algorithms. They were also the primary means of evaluating the efficiency of Multilisp implementations. Programmers have used these tools to tune particular algorithms to try to extract the maximum parallelism from a program. Seeing the effect of future placement in programs has taught the members of the group much about the nature of parallelism in symbolic programs.

However, these tools were not used to help guide architectural decisions. They all studied the behavior of Multilisp code at the language level, not the interaction between the language, its implementation, and the underlying machine organization. *Nusim* was a first attempt at looking at these issues.

There are three significant features of Nusim that made it easier to study the architecture of Multilisp machines. The first was Nusim's ability to run Multilisp programs without modification. Second was the ability to modify parameters of the architecture being simulated at run time. The third feature was that Nusim actually simulates a multiprocessor by running on an existing multiprocessor.

In order to see the effect of architectural decisions in Multilisp machine, it is not enough to model the behavior of Multilisp processors. This thesis has shown that there is a wide variation in the behavior of different Multilisp programs. Rather than hypothesizing how Multilisp processors might behave, a better approach is to actually measure the characteristics of programs in operation. Since Nusim evolved from an existing Multilisp emulator, it can run any Multilisp application. It imposes a realistic communication load on the system because it must access the same data that the program would.

Nusim differs most from previous implementations of Multilisp in that it simulates the architecture of a machine at some level. Being able to easily modify the characteristics of that architecture was vital for my research. I could not have have run as many types of experiments if each test required reloading some part of the software. One approach to simulating different topologies might have been to write a version of Nusim for each topology. A less time-consuming approach would have been to configure the characteristics of the architecture when Nusim started running. The final version of Nusim that I used allowed me to vary many different parameters of the architecture during a run of the program. This made it easy to set up automatic test runs.

The final important feature of Nusim is that it ran on a real Multiprocessor system. It is possible to simulate a multiprocessor by running on a uniprocessor. Any processor can time-share between a number of tasks. However, it is difficult to know how different processors will interact in a real system. Programming on a uniprocessor may introduce some assumptions about those interactions. In a real multiprocessor, accesses by different processors may collide in the network. Processors also collide when competing for resources. While running one multiprocessor may not duplicate the type of interactions that could occur in another parallel machine, it is likely to be more realistic than running on a uniprocessor. The fact that some of the interprocessor behavior seen in this thesis is still unexplained indicates that this approach to simulation was successful.

6.4.1 Evaluation of Research

The data that was collected for this thesis will be useful for architectural discussions in the Parallel Processing Group at M.I.T. We now have some data on the communication load imposed by different Multilisp programs. We know the types of data that these programs use, and how often the programs refer to that data. We have additional information on the production of futures in parallel programs, and on how those futures are touched. We have seen how many tasks are produced by a parallel application, and how processors in a multiprocessor compete for those tasks. Finally, we have seen how this behavior depends on different characteristics of the underlying computer architecture.

Where possible, Nusim measures the aspects of Multilisp programs that are independent of the Nusim implementation. The number of tasks produced by an application, and the number of data objects touched, should be invariant. This data should prove useful in evaluating other possible implementations of Multilisp.

A major topic of research at P.P.G. in the next few years will likely involve designing a new type of symbolic multiprocessor. We will try to model and simulate different subsystems of that computer in order to evaluate different designs. Some work in general purpose simulators has already begun [35]. The data on communication in Multilisp that was collected in this thesis should play an important role in that design process. It sets a bound on the global communication needed by Multilisp programs, and suggests some means of reducing that load.

If unexpected results are a measure of the worth of research, this thesis has been worthwhile. In the field of computer architecture, I hope that this thesis has provided some answers, but more importantly I hope that it has suggested some interesting questions.

Appendix A

The MCODE Machine Language

What follows is a brief listing of the MCODE instructions used by Nusim. These operations are described in more detail in [34]. The purpose of this listing is just to provide a flavor for the type of instructions available in this machine language.

MCODE instructions generally consume one or more values from the stack and may produce one or more values that are pushed on the stack. Additionally, many MCODE instructions take 'in-line' parameters, arguments which are encoded in the instruction stream with the opcodes. These are commonly simple integers, such as the number of items to pop off the stack. Instructions may also read 'constant values', which are known at compile time. These are kept as a sequence of values in the code object for a Multilisp function. Value arguments may be arbitrary Lisp objects.

In the listing that follows, an instruction stream argument is represented by [arg], while value stream arguments are represented by {val}. All other instructions are assumed to pop arguments off the stack. Each opcode is labelled with its index. The instruction indices are not necessarily contiguous.

MCODE Instructions

Index	Opcode	Args	Opcode Description
5	POP	[n]	pop items off stack
6	GVAL	{sym}	push global value of symbol
7	SGVAL	{sym}	set global value of symbol from pop()
8	COPY		push extra copy of top of stack
9	RTN		return from call, value is on stack
10	CONS		make conscell
11	FEVAL	[level,offset]	get value from environment at
12	SEVAL	[level,offset]	set pop() into environment slot
13	CALL	[nargs, flag]	push some args and call closure
14	CALLRTN	[nargs, flag]	tail-recursive call
16	PUSHENV	[#slots] {doc}	create new env frame, push onto frames
17	CLOSE	{doc,close,espec}	make a closure from code, doc, and espec
18	PUSHVAL	{what}	push a value from code, onto stack
19	LABEL		make label value
20	PUSHNVAL	{what}	same as pushval but does not fetch value
21	THROW		push value, push label, 'throw' value
22	ENDFUTURE		fill in value of future and restart tasks
23	ARRAYSET		set an element of an array
25	GETSTRUCT	[type,off]	get val at offset in some struct
26	SETSTRUCT	[type,off]	set val at offset in some struct
27	TYPEEQ	[type]	check type of object with type in ops
28	EQSETSTRUCT	[type,off]	setstruct if eq, atomic operation

Index	Opcode	Args	Opcode Description
29	OLDSETSTRUCT	[type,off]	setstruct return old, atomic operation
30	TYPECAST	[type1,type2]	change type to value in instructions
31	IGOTO	[IPC,VPC]	goto, instr, value pc are 2-byte items
32	ITGOTO	[IPC,VPC]	in the instruction stream
33	IFGOTO	[IPC,VPC]	IGOTO if pop() is nil
34	IFORK	[IPC,VPC]	fork, new thread is IGOne TO
35	PUSHNIL		push nil on the stack
36	PUSHNUM	[num]	push a 1-byte signed number on the stack
40	UNDETERMINEDP		check if future has value
41	GETFSTRUCT	[type,off]	getstruct without forcing futures
42	PRINT		print a lisp object on a file
43	PRINC		print without slashification
44	SYMVAL		get the global value of a symbol
51	SETSYMVAL		set value of symbol
52	EXCEPTIO		return non-nil if passed an exception
53	EXC2LIST		typcast exception into a list
55	NFUTURE	[IPC,VPC]	make a future, priority to future
56	NDFUTURE	[IPC,VPC]	make a future, priority to main thread
57	NDELAY	[IPC,VPC]	make a delay
61	QUIT		terminate current process
62	POPENV		pop one frame off the environment
63	STAT		return current statistics

Index	Opcode	Args	Opcode Description
64	MAKEMARKER		return MARK
65	IFASTREAD		do a FastRead() for setup
66	HIST		return histogram of task stats
67	CURTASK		get a picture of the current task
68	GTIME		return the current elapsed time
69	THEENV		return the current working environment
70	INFO		return info stats
71	INFOON		enable info stats
72	INFOOFF		disable info stats
73	SHOWFREQ		return inst use frequency
74	RESETTLINES		reset time lines on all processors
81	TOUCH		touch a future (a noop)
82	GETCH		get char from file as fixnum
83	NULL		null() predicate
84	SIGNAL		signal an exception
85	MINUS		reverse sign of a number
86	INTERN		intern(string)
87	MAKESTRING		make-string(n)
88	STRLEN		string-length(string)
89	IOFLUSH		flush characters from file
90	LABGO		go to label value
91	CARCDR		push car, then cdr of TOS

Index	Opcode	Args	Opcode Description
92	ANYCHAR		ee if any input is ready
93	CLEAROPTION		n option flag,return it's old val
94	PLIST		get the property list of symbol on stack
95	LFCLOSE		close a lisp file
96	BOUNDP		return true if symbol is bound
97	MARRAY		make an array object of a certain size
98	ARRAYLEN		return the length of an array
99	FFLUSH		flush a file
100	NSUSPEND		suspend a task, calling a function with it
101	TYPE		return the type of an object
102	SETOPT	[posn]	initialize an optional argument
103	SUSPEND		suspend a task, calling a function with it
104	ACTIVATE		activate a previously suspended task
105	FIX		truncate floating-pt to fixnum
106	FLOAT		convert fixnum to flonum
107	STRUCTNAME		return the name of a structure
108	STRUCTSIZE		return the size of a structure
109	DUMPTLINES		dump all time lines to a file
110	RECEVENT		record event on current processor
122	PLUS		add numbers
123	GT		greater-than predicate
124	EQ		eq predicate

Index	Opcode	Args	Opcode Description
125	DIFF		subtract numbers
126	TIMES		multiply numbers
127	QUOT		divide numbers
128	REM		remainder of numbers
129	NTHSTR		nth-string(string,n)
130	UNGETCH		put a char (fixnum) back into file
131	EXER		exercise primitives for performance meas
132	PUTCH		output fixnum from stack to file
133	NUMEQ		numerical equal test
134	ARRAYREF		get an element of an array
135	MAKESTRUCT		make a lisp structure
136	ADDRREF		indexed reference based on C pointer
137	NACTIVATE		activate a previously suspended task
138	FASTRSTRING		read a string from a file
139	SETSPECIAL		set an inst as special for info collection
161	SETNTHSTR		set-nth-string(string,n,value)
162	LFOPEN		open a file in a particular mode
163	STRUCTREF		reference a slot of a structure
164	ADDRSET		indexed set based on C pointer
169	MAKEESPEC		return a new espec
170	SCANTOK		scan a token from a file
171	MAKECLOSURE		return a new closure
172	STRUCTSET		set a slot in a structure

Appendix B

Statistics Collection in Nusim

B.1 Using Nusim

B.1.1 Lisp Functions

The following Lisp functions are unique to the Nusim implementation of Multilisp. They are used to enable statistics processing, to dump out statistics to a file, and to set the values of some Nusim scheduling variables.

Nusim collects information in a *statistics structure*. The statistics information includes counts of different types of accesses and where they occured, types of instructions executed, time spent in different phases of Nusim, and many other variables. A complete description is in [44]. Nusim maintains an array of statistics structures. Different phases of the program can dump statistics in a different structure of this array. For instance, Nusim can step to a new statistic structure after every garbage collector flip. This is useful because the division of memory blocks among processors changes after each flip.

(info file n)

Dumps statistics information from statistics structure n into the file. Array 0 is where info stats go when info collection is turned off. The schedule variable *globalinfo* sets where info stats go when info is turned on. Allowable range for *n* is 0 to **info-locations** (which is presently 10), or negative. A negative index prints out 'Garbage Collection' and 'Cost of Info Collection' statistics.

(clearinfo n)

Clears the info structure *n*. Same rules as above.

(infoon)

Turns on info collection by using the value of variable *globalinfo* as an index to a statistics structure. Hereafter, run-time statistics will use this new statistics structure.

(infooff)

Turns off info collection by setting the current statistics structure back to structure 0.

(getsched string)

How to look at schedule variables that the system uses. Eg: `(getsched "minsched")`

(setsched string value)

Sets the value of a scheduling variable to be *value*.

B.1.2 Scheduling Variables

In Nusim, each processor alternates between running processes and loading tasks. While running, a processor may switch between active processes in the *process queue*. A processor runs a *quanta* of instructions as an uninterrupted unit. Each process is allowed to run for at most *runfor* of these quanta. The processor does some amount of garbage collection after each quanta. After the process has run *runfor* quanta, the processor switches to another active process in the process queue.

After at most *totalrun* quanta of running processes, the processor enters a phase of loading tasks. A processor that has run all its processes and is otherwise idle also enters the task loading phase.

Processors attempt to load processes so that the minimum number of processes in the process queue is greater than or equal to *minsched*. Processors only load tasks out of *task queues* during this phase. Programs spawn processes by executing a future instruction. These newly spawned processes are stored in the process queue, possibly bumping an older task out into a *task queue*. The parameter *runsched* sets a limit on the size of the process queue due to process spawning, and *tasksched* sets the size limit due to task grabbing.

parcelchoice

How to pick a node from which to grab a parcel of memory. Choices are:

random-choice	Choose a node at random.
closrand-choice	Try our node first, then choose randomly.
increment-choice	Start at our node and move a further distance away each time we fail to get the resource. Reads a topological description that we have built up for the system we want to model.

taskchoice

How to pick a node from which to grab an executable task. Choices are the same as for *parcelchoice*.

minsched

A processor tries to maintain at least this many processes in its process queue at all times. If number of processes in the process queue is less than this, excessive searching for tasks may occur so it may be best to keep *minsched*= 1.

tasksched

When a processor loads tasks into its process queue, it will try to grab enough tasks to have *tasksched* processes in the process queue.

taskgrab

The maximum number of tasks that a processor will steal take from each task queue at any one time.

iosched

One processor in the system is designated I/O processor. It loads I/O tasks from a single system-wide I/O task queue. The *iosched* variable sets the maximum number of tasks that the I/O processor will grab from the I/O task queue.

runsched

When a processor executes a *future* instruction, it forks a child task and pushes the parent back into the process queue. This variable sets a limit on how many processes may be in the process queue due to forks. If there are allready *runsched* processes in the queue, the processor bumps the oldest process out into its task queue.

runquanta

How many instructions to run at a time before checking the status of garbage collection.

runfor

How many quanta a process runs before we let the next process in the process queue run. Schedule them round-robin. If this is set very high, (the default), then every process runs to completion.

totalrun

Run processes for this many quanta before trying to load new tasks into the process queue.

globalinfo

This is the index of the statistics structure that will be used if info collection is turned on.

info-increment

If this flag is non-zero, then processors increment *globalinfo* after each garbage collection flip.

stopncopy

Force garbage collection to behave as a stop and copy system rather than incremental.

movespeed

The speed at which incremental garbage collection sweeps through memory between quanta of running instructions.

B.2 How Nusim Counts Accesses

Some Nusim counters

Every memory access is counted as a fetch or store of some type. Nusim tracks different types of accesses by incrementing slots in a statistics structure. Rather than reproduce the entire structure here, I will name and describe a few significant slots.

code-fetch

This slot counts code object fetches. In the Nusim implementation of Multilisp, a code object contains the code for a Multilisp procedure. The code object points to a block of MCODE instructions and to a set of constant values used by the instructions.

instr-fetch

A block of instructions contains the actual MCODE byte-code for a procedure or expression. Fetching a block of instructions is conceptually like loading those instructions into a processor's instruction buffer. Once the block has been loaded, a processor can run all of the instructions in that procedure without global accesses to fetch each instruction.

env-link-fetch

In Nusim, lexically scoped environments are stored as a series of environment frames. Each frame contains a pointer to its lexical parent. In order to fetch a value out of a lexically enclosing environment, a process must step through these links to the appropriate environment frame, then fetch a value out of a slot in that frame. Nusim counts the environment link accesses as *env-link-fetch*. Fetches from the local environment do not require any link accesses. Note that *env-link-fetch* counts environment link accesses necessary to store into an environment slot, as well as to read a value out of a slot.

env-obj-fetch

This is a count of environment value fetches.

env-obj-store

This is the corresponding count for environment value stores.

hunk-fetch

Stack hunks are chunks of the stack that have been pushed out of a processor's stack buffer. When a process loads a new task, it must load a new stack into its stack buffer. This is counted as *hunk-fetch*.

scache-save

This variable counts the number of stack hunks that are pushed out of the stack buffer into memory.

scache-load

This is a count of stack hunk loads into the stack buffer.

scache-hdr-fetch

The head of each stack hunk contains two words of information. Nusim reads these header words on each stack hunk load.

stack-deep-fetch

Nusim occasionally fetches a word out of the stack without loading in an entire stack hunk. This 'deep' fetch fetches a value out of a hunk in the heap.

future-touch

This variable the number of times that futures are touched. Most instructions implicitly touch their operands, to make sure that the operands are not futures. Any time that an instruction stumbles across a future, it is counted as a future touch.

future-val-fetch

When an instruction touches a future in Nusim, it jumps to an exception handling routine. If a future has been determined, the exception handler merely fetches the value of the future, incrementing *future-val-fetch*. If the future is not determined, the process that touched it must wait on the future.

future-val-store

A function that determines the value of a future writes the value into a slot in the future object. *Future-val-store* counts those stores.

symbol-val-fetch

This is a count of reads of global symbols.

static-value-fetch

In Multilisp, constant values used by the program are compiled in with the code. These values could be any Lisp object. Typical constant values are numerical constants and symbol names. These constants are kept in the code object for a procedure. The variable *symbol-value-fetch* counts references to those values.

cons-fetch

This variable counts fetches of the *car* or *cdr* of a cons-cell.

array-fetch

This variable counts array references.

struct-fetch

This variable counts references to user-defined structures.

string-fetch

This variable counts string references.

closure-fetch

This variable counts fetches of closures.

B.2.1 Cost of some operations in Nusim

Most simple data fetches in Nusim increment one of the counting variables described above. Some basic operations in Nusim fetch several different types of data. This section describes those operations, and how they are counted using the variables defined above. (The increment for a counter variable is 1 except where noted).

Operation	Counter	Increment
Save stack hunk	scache-save	1
Load stack hunk	scache-load	1
	scache-hdr-fetch	2
Call subroutine	code-fetch	1
	instr-fetch	1
	env-link-fetch	1
Return from subroutine	instr-fetch	1
Load process from task queue	code-fetch	1
	instr-fetch	1
	env-link-fetch	1
	hunk-fetch	1
	scache-load	1
	scache-hdr-fetch	2
Touching a determined future	future-touch	1
	future-val-fetch	1
Touching an undetermined future	future-touch	1
	scache-save	1
Determining a future's value	future-val-store	1
Reading out of the environment	env-link-fetch	[lexical level]
	env-obj-fetch	1
Storing into the environment	env-link-fetch	[lexical level]
	env-obj-store	1

Garbage Collection flip	instr-fetch	¡ [# processes in queue]

B.2.2 Nusim counters used for results

In Chapter 5, I presented several types of data graphs. Two graphs distinguished different types of data accesses. Data fetches and stores were divided into classes of objects. The mean distance of reference was divided into a different set of access types. Chapter 5 also showed data on the locality of reference for all access types. This section explains which Nusim variables contributed to each of these data graphs.

Data Fetch Classes

Data Class	Nusim Variables
Constant	code-fetch
	instr-fetch
	closure-fetch
	string-fetch
Global	symbol-val-fetch
	cons-fetch
	array-fetch
	struct-fetch
Environment	env-link-fetch
	env-obj-fetch
Stack	scache-hdr-fetch
	hunk-fetch
	stack-deep-fetch
Future	future-touch

Data Store Classes

Data Class	Nusim Variables
Global	symbol-val-store
	cons-store
	array-store
	struct-store
	closure-store
Environment	env-obj-store
Future	future-val-store

Data Types for Locality

Data Class	Nusim Variables
Code	code-fetch
	instr-fetch
Structure Stores	future-val-store
	array-store
	struct-store
Structure Fetches	future-val-fetch
	symbol-val-fetch
	env-link-fetch
	hunk-fetch
	cons-fetch
	array-fetch
	struct-fetch
	string-fetch

Environment	env-obj-fetch
	env-obj-store
Stack	scache-load
Futures	future-touch

Locality Measurements

The following Nusim data types are used to compute the mean distance of access and the percentage local access for all data accesses.

code-fetch
instr-fetch
future-val-store
array-store
struct-store
future-val-fetch
symbol-val-fetch
env-link-fetch
hunk-fetch
cons-fetch
array-fetch
struct-fetch
string-fetch
env-obj-fetch
env-obj-store
scache-load
future-touch

Appendix C

Test Programs

C.1 Compile-Expr

C.1.1 Source Code

The Multilisp compiler is a large body of code. The function *compile-expr* calls several other routines in order to compile an expression to symbolic assembly code. A few of those functions are reproduced here.

```
; Copyright (c) 1984.  Robert H. Halstead, Jr. and Juan R. Loaiza.

; Compile expr in the enviroment env and push it on the stack before
; the code in cont.
;
(defun compile-expr (expr cont env &aux op args prim-code)
  (future
    (cond ((or (numberp expr) (null expr) (stringp expr))
   (cons-code '(pushval ,expr) cont))
  ((symbolp expr) (get-var-val expr cont env))
  ((atom (setq op (car expr)))
   (setq args (cdr expr))
   (cond ((setq prim-code (get op 'prim-form))
  (lexpr-funcall prim-code cont env args))
 ((in-env op env) (compile-apply expr cont env))
 ((setq prim-code (get op 'primitive-code))
  (or (= (car prim-code) (length args))
      (error "; wrong # of arguments" expr))
  (compile-arglist args
   (append-code (cdr prim-code) cont)
   env))
 ((setq prim-code (get op 'vx-multilisp-macro))
  (compile-expr (apply prim-code args) cont env))
 ((setq prim-code (get op 'multilisp-macro))
  (compile-expr (apply prim-code args) cont env))
 ((compile-apply expr cont env)))
  ((compile-apply expr cont env)))))

; Pushes in front of cont an expression that will reference var in env.
;
(defun get-var-val (var cont env &aux where)
  (cond ((setq where (in-env var env))
 (cons-code `(eval . ,where) cont))
((cons-code `(gval ,var) cont))))

; Pushes in front of cont an expression that will set var in env to
; whatever is currently on the top of the stack.
;
(defun set-var-val (var cont env &aux where)
  (cond ((setq where (in-env var env))
 (cons-code `(seval . ,where) cont))
((cons-code `(sgval ,var) cont))))

; Push the function and its arguments onto the stack and do a call with
; the number of arguments that were supplied.
;
(defun compile-apply (expr cont env)
  (compile-arglist expr
       (call-it (length (cdr expr)) cont)
       env))

(defun compile-arglist (args cont env)
 (future
  (cond ((null args) cont)
((compile-expr (car args)
       (compile-arglist (cdr args) cont env)
       env)))))
```

C.1.2 Test Data

The test case for running compile-expr is one large function definition, *one-big-fn*. It contains four macro definitions, and sixteen smaller function definitions. Four more functions are defined within these second level routines. *One-big-fn* contains approximately 8500 bytes of Multilisp source code. The compiled MCODE representation for *one-big-fn* is 6400 bytes long.

C.1.3 Instruction Mix

The MCODE instructions executed by *compile-expr* while compiling this test case were as follows:

Instruction	Count
EVAL	144803
ITGOTO	55896
GETSTRUCT	50324
POP	39981
NULL	33629
SEVAL	29476
RETURN	17620
CALL	17620
TYPEEQ	17433
GVAL	16411
IGOTO	14843
PUSHNIL	13266
PUSHNUM	11215
COPY	10032
CALLRTN	9695
CONS	8517
PLUS	8059
EQ	8053
PUSHVAL	7407
CLOSURE	3227
DETERMINE	3154
FUTURE	3154
PLIST	2907
SETNTHSTR	2373

Instruction	Count
ARRAYREF	1268
SETOPT	1226
ARRAYSET	952
NUMEQ	446
TOUCH	317
DIFF	271
FIX	240
GT	153
QUOT	147
REM	147
MARRAY	120
NTHSTR	73
TYPECAST	62
MAKESTRING	37
SETSTRUCT	34
MAKEESPEC	31
TIMES	15
PUSHENV	10
POPENV	10
STRLEN	6
SGVAL	3
INTERN	3
BOUNDP	3
INFOOFF	1

C.2 Consim

C.2.1 Source Code

A large part of the Consim environment is a compiler that translates a high level description of a circuit down to parallelized Multilisp code. The function *psim* actually runs the simulation. It spawns many cycles of the circuit in parallel, passing the output of one cycle to the input of the next.

```
(defun psim (sc-proc mstate ckt-in cycles current-cycle)
       (let ((elt (future (sc-proc mstate
                                       (future (car ckt-in))
                                       current-cycle))))
            (if (= cycles current-cycle) (list elt)
                (psim sc-proc
                      (future (cadr elt))
                      (cdr ckt-in)
                      cycles
                      (+ current-cycle 1)))))
```

C.2.2 Circuit Simulated

The circuit simulated for these experiments was a four bit ALU, configured to act as a counter.

```
(defun alusgf (mstate ckt-in cyc-num)
  (let* ((g0885 mstate)
         (a3 (future (field 1 g0885)))
         (a2 (future (field 2 g0885)))
         (a1 (future (field 3 g0885)))
         (a0 (future (field 4 g0885)))
         (b3 (future (field 5 g0885)))
```

```
        (b2 (future (field 6 g0885)))
        (b1 (future (field 7 g0885)))
        (b0 (future (field 8 g0885)))
        (~cin (future (field 9 g0885)))
        (g0886 ckt-in)
        (s3 (future (field 2 g0886)))
        (s2 (future (field 3 g0886)))
        (s1 (future (field 4 g0886)))
        (s0 (future (field 5 g0886)))
        (~b3 (future (f-not b3)))
        (~b2 (future (f-not b2)))
        (~b1 (future (f-not b1)))
        (~b0 (future (f-not b0)))
        (~m (future (f-not (future (field 1 g0886)))))
        (t22 (future (f-nor3 (future (f-and2 ~b3 s1))
    (future (f-and2 s0 b3)) a3)))
        (t23
          (future (f-nor2 (future (f-and3 b2 s3 a2))
(future (f-and3 a2 s2 ~b2)))))
        (t26 (future (f-nor3 (future (f-and2 ~b1 s1))
    (future (f-and2 b1 s0)) a1)))
        (t27
          (future (f-nor2 (future (f-and3 b0 s3 a0))
(future (f-and3 a0 s2 ~b0)))))
        (t28 (future (f-nor3 (future (f-and2 ~b0 s1))
    (future (f-and2 s0 b0)) a0)))
        (t21
          (future (f-nor2 (future (f-and3 b3 s3 a3))
(future (f-and3 a3 s2 ~b3)))))
        (t24 (future (f-nor3 (future (f-and2 ~b2 s1))
    (future (f-and2 s0 b2)) a2)))
        (t25
          (future (f-nor2 (future (f-and3 b1 s3 a1))
(future (f-and3 a1 s2 ~b1)))))
```

```
        (f0
          (future (f-xor2 (future (f-and2 t27 (future (f-not t28))))
   (future (f-nand2 ~m ~cin)))))
        (f3
          (future (f-xor2 (future (f-and2 t21 (future (f-not t22))))
   (future (f-nor4 (future (f-and5 ~cin t27 t25 t23 ~m))
   (future (f-and4 t25 t23 t28 ~m))
   (future (f-and3 t23 t26 ~m))
   (future (f-and2 t24 ~m)))))))
        (f2
          (future (f-xor2 (future (f-and2 t23 (future (f-not t24))))
   (future (f-nor3 (future (f-and4 ~cin t27 t25 ~m))
   (future (f-and3 t25 t28 ~m))
   (future (f-and2 t26 ~m)))))))
        (ckt-out
          (list (future (f-nand2 (future (f-nand5 t21 t23 t25 t27 ~cin))
  (future (f-nor4 t22
  (future (f-and2 t21 t24))
  (future (f-and3 t21 t23 t26))
  (future (f-and4 t21 t23 t25 t28))))))))
        (f1
          (future (f-xor2 (future (f-and2 t25 (future (f-not t26))))
   (future (f-nor2 (future (f-and3 t27 ~cin ~m))
   (future (f-and2 t28 ~m)))))))
        (eq (future (f-and4 f3 f2 f1 f0)))
        (~eq (future (f-not eq)))
        (zero (future (f-and2 eq ~eq)))
        (nstate (list f3 f2 f1 f0 zero zero zero zero zero)))
    (list cyc-num nstate ckt-out)))
```

C.2.3 Instruction Mix

The MCODE instructions executed by *Consim* while simulating a four bit ALU were as follows:

Instruction	Count
EVAL	36163
ITGOTO	15116
PUSHNUM	13273
NUMEQ	10866
GVAL	10608
GETSTRUCT	10491
NULL	8409
CALLRTN	7406
PUSHNIL	6480
CONS	5920
POP	4766
RETURN	4361
CALL	4361

Instruction	Count
DETERMINE	3238
FUTURE	3238
COPY	2242
DIFF	1840
SEVAL	1521
SETOPT	1122
TYPEEQ	1039
PLUS	472
CLOSURE	1
PUSHVAL	1
SETSTRUCT	1
INFOOFF	1

C.3 Fboyer

C.3.1 Source Code

The Boyer-Moore benchmark has been used to test a number of Lisp implementations ??. The main parts of the program are a tautology checker and a term rewriting routine.

```
; Boyer-Moore Theorem prover - works by moby expansion
; rewritten in Scheme by Seth Steinberg 1986
; Modified to remove useless future
; in tautology? and add a future in apply-subst-list.
;                                          Randy Osborne Nov. 12/86

; Tautology detection checks forms (if predicate consequent alternate)
; If the predicate is known true then we just check the consequent
; If the predicate is known false then we just check the alternate
; Otherwise we see if the consequent is true assuming the predicate is true
; and that the alternate is true assuming the predicate is false

(define true #t)
(define false #f)

(define (taut? form)
  (tautology? (rewrite form) nil nil))

(define (tautology? form true-list false-list &aux temp)
  (cond ((known-true? form true-list) true)
((known-false? form false-list) false)
((eq? (car form) 'if)
 (cond ((known-true? (cadr form) true-list)
(tautology? (caddr form) true-list false-list))
       ((known-false? (cadr form) false-list)
(tautology? (cadddr form) true-list false-list))
```

```
       (else
 (setq temp (future (tautology? (cadddr form)
true-list
(cons (cadr form) false-list))))
 (and
   (tautology? (caddr form)
       (cons (cadr form) true-list)
       false-list)
   temp))))
(else false)))

(define (known-true? form true-list)
  (if (equal form '(t))
      true
      (member form true-list)))

(define (known-false? form false-list)
  (if (equal form '(f))
      true
      (member form false-list)))

; Rewriting matches a form against the list of lemmas associated with the car
; of the form and first rewrites the remainder of the form before
; finding the first lemma which matches and expanding it accordingly.
(defun rewrite (form)
    (if (atom form)
form
(rewrite-with-lemmas (cons (car form) (rewrite-args (cdr form)))
     (find-lemmas (car form)))))
(define (rewrite-args args)
  (if (null? args)
      nil
      (cons (future (rewrite (car args))) (rewrite-args (cdr args)))))
```

```
(define (rewrite-with-lemmas form lemmas)
  (if (null? lemmas)
      form
      (let ((subst-list (one-way-unify-util form (cadar lemmas) nil)))
(if (not (eq? subst-list 'failed))
    (rewrite (apply-subst subst-list (caddar lemmas)))
    (rewrite-with-lemmas form (cdr lemmas))))))

;; Weak unification works by a recursive pattern match.
;;
(define (one-way-unify-util form pattern frame)
  (cond ((eq? frame 'failed) 'failed)
((atom pattern)
 (let ((already-matched (fast-assq pattern frame)))
   (cond (already-matched ; if matched verify rematch
  (if (equal form (cdr already-matched)) frame 'failed))
 (else
  (cons (cons pattern form) frame)))))
((atom form) 'failed)
((eq? (car form) (car pattern))
 (one-way-unify-list (cdr form) (cdr pattern) frame))
(else 'failed)))

(define (one-way-unify-list form pattern frame)
  (if (null? form)
      frame
      (one-way-unify-list (cdr form) (cdr pattern)
  (one-way-unify-util (car form) (car pattern) frame))))

;; Very simple substituter used by rewrite with the result of the unification.
;;
(defun apply-subst (subst-list form)
    (if (atom form)
(let ((value (fast-assq form subst-list)))
  (if value (cdr value) form))
(cons (car form) (apply-subst-list subst-list (cdr form)))))

(define (apply-subst-list subst-list form)
  (if (null? form)
      nil
      ; added future here (R.O.)
      (cons (future (apply-subst subst-list (car form)))
    (apply-subst-list subst-list (cdr form)))))

(define (add-lemma lemma)
  (cond ((and
   (not (atom lemma))
   (eq? (car lemma) 'equal)
   (not (atom (cadr lemma))))
 (push lemma (get (caadr lemma) 'lemmas)))
(else
 (print '(Bad lemma form ,lemma)))))

(define (find-lemmas key)
  (get key 'lemmas))

;;; Speeded-up versions of assq and equal:
;;
(define (fast-assq key lst)
  (until (((null lst) nil)
  ((eq key (caar lst)) (car lst)))
    (setq lst (cdr lst))))

(defun equal (arg1 arg2)
  (until (((eq arg1 arg2))
  ((atom arg1)
   (cond ((numberp arg1)
  (if (numberp arg2) (= arg1 arg2)))
((stringp arg1)
 (if (stringp arg2) (string-equal arg1 arg2)))
((structurep arg1)
 (if (structurep arg2) (structure-equal arg1 arg2)))))
 ((atom arg2) nil)
 ((not (equal (car arg1) (car arg2))) nil))
   (setq arg1 (cdr arg1))
   (setq arg2 (cdr arg2))))
```

C.3.2 Test Data

Fboyer uses a data base of 106 lemmas to rewrite the input expressions into a form containing only *if* statements. For space reasons, the data base is not included here.

The test case that was used for the runs of Fboyer in this thesis is:

```
(implies (and (implies (f x) (g x)) (implies (g x) (h x)))
         (implies (f x) (h x)))
```

C.3.3 Instruction Mix

The MCODE instructions executed by *Fboyer* in proving this test case were as follows:

Instruction	Count
EVAL	48913
GETSTRUCT	23526
ITGOTO	20852
NULL	14950
GVAL	13302
RETURN	8959
CALL	8959
TYPEEQ	6218
EQ	4923
PUSHNIL	4749
PUSHVAL	4600
CALLRTN	4313

Instruction	Count
POP	3767
CONS	3266
SEVAL	3107
IGOTO	2547
DETERMINE	1633
FUTURE	1633
PLIST	1245
PUSHENV	738
POPENV	738
COPY	727
INFOOFF	1

C.4 Multilog

C.4.1 Source Code

The Multilog program is designed to be an interactive system. It includes a query-driver loop, which reads commands from the user and dispatches to the appropriate database manipulation function. It allows users to load and save databases, to add and delete clauses.

The code that follows is a sample of routines for evaluating assertions. A more complete description of the code for Multilog can be found in [51].

```
; This is the main evaluating mechanism for the interpreter in the case
; of a normal query. If the query (or part of it being evaluated) is
; predicated by some operator such as AND, OR, NOT, or LISP-VALUE,
; then qeval will detect this, retrieve the appropriate function name
; from a symbol table (created and used through put's and get's)
; and apply this appropriate function to the rest of the input
; expression. Otherwise, the asserted? function is called in the case
; of a simple query with no predicating operators.

(define (qeval query environment-stream)
 (let ((qproc (get 'qeval (type-of query))))
   (if (null qproc)
       (asserted? (make-arg-list query)
                  environment-stream)
       (qproc (contents query) environment-stream))))

; The asserted? procedure handles simple queries. It takes an
; argument list which contains a single query and a stream of environments
; to be extended by database matches of that single query. These extensions
; are found by finding explicit assertions in the database and applying rules.
; The result returned is that extended environment.

(define (asserted? a environment-stream)
  (flatten-stream
    (append-streams (future(map (lambda (environment)   ; PARALLEL
                                  (find-assertions (pattern-of a)
                                                   environment))
                                environment-stream))
                    (future(map (lambda (environment)   ; PARALLEL
                                  (apply-rules (pattern-of a)
                                               environment))
                                environment-stream)))))

; Pconjoin is the procedure which handles the parallel AND's presented to the
; system. It evaluates successive conjuncts in the environment stream
; returned by evaluation of the previous conjuncts. It returns the final
; stream of extended environments after evaluation of all of the conjuncts.

(define (pconjoin conjuncts environment-stream)
  (cond ((empty-conjunction? conjuncts)
         environment-stream)
(1 (pconjoin (future(rest-conjuncts conjuncts)) ; PARALLEL
                    (future(qeval (first-conjunct conjuncts)
                                  environment-stream))))))

; Pdisjoin is the procedure which handles the parallel OR's presented to the
; system. Evaluation of a successive disjunct does not depend on any of the
; variable bindings from evaluation of previous disjuncts, so the procedure
; need merely merge the extended environment streams each formed from
; evaluation in the context of the original environment stream.

(define (pdisjoin disjuncts environment-stream)
 (cond ((empty-disjunction? disjuncts)
             (the-empty-stream))
       (1
        (append-streams (future(qeval (first-disjunct disjuncts)       ; PARALLEL
                                      environment-stream))
```

```
                         (future(pdisjoin (rest-disjuncts disjuncts)
                                          environment-stream))))))

;  Unify-match is the main unification algorithm, which takes two patterns
;  as inputs and an environment, and returns either an extended environment
;  or 'failed.


(define (unify-match p1 p2 env)
 (cond ((not (or (consp env) (null env))) 'failed)
       ((equal p1 p2) env)
       ((atom p1)
        (cond ((atom p2) 'failed)
              ((var? p2) (extend-if-possible p2 p1 env))
              (1 'failed)))
       ((var? p1) (extend-if-possible p1 p2 env))
       ((atom p2) 'failed)
       ((var? p2) (extend-if-possible p2 p1 env))
       (1 (unify-match (cdr p1)
                       (cdr p2)
                       (future(unify-match (car p1)      ; PARALLEL
                                           (car p2)
                                           env))))))

;  The basic pattern matcher takes a pattern, a data object, and an
;  environment and returns either the symbol 'failed or an extension of the
;  given environment if such extension would be possible.  The pattern matcher
;  checks the pattern against the data, symbol by symbol, and returns an
;  extended environment, the original environment or the symbol 'failed
;  depending on the result of that check.  Extensions to the environment must
;  be consistent with current bindings.

(define (pattern-match pat dat environment)
 (cond ((not (or (consp environment) (null environment))) 'failed)
        ((and (numberp pat) (numberp dat))
         (cond ((= pat dat) environment)
               (1 'failed)))
        ((atom pat)
         (cond ((eq pat dat) environment)
               (1 'failed)))
        ((var? pat)
         (future(extend-if-consistent pat        ; PARALLEL
                                      dat
                                      environment)))
        ((atom dat) 'failed)
        (1 (pattern-match (cdr pat)
                          (cdr dat)
                          (future(pattern-match (car pat)        ; PARALLEL
                                                (car dat)
                                                environment))))))

;  The following procedure checks if it is possible to extend the input
;  environment with the given var to dat binding.  If there is no binding
;  currently in the environment for the variable, then the binding is simply
;  added.  Otherwise, extend-if-consistent matches in the environment the
;  data against the variable binding value.  This will return either 'failed
;  if the extension would be inconsistent because the pattern match would fail,
;  or the original environment if the extension would be acceptable.

(define (extend-if-consistent var dat environment)
  (let ((value-cell (binding-pair var environment)))
    (if (null value-cell)
        (extend var dat environment)
        (pattern-match (value-in value-cell) dat environment))))

;  Find-assertions takes as input a pattern and an environment.  It returns
;  a stream of environments found by extending the original environment by a
;  database match of the given pattern.
```

```
(define (find-assertions pattern environment)
 (map-nofail (lambda (datum)
               (pattern-match pattern datum environment))
             (fetch-assertions pattern environment)))
```

C.4.2 Test Case

For the runs of Multilog described in this thesis, I used a simple test case. The graph *path.out* describes points connected by edges. The assertion that Multilog tested was whether there is a path from point *a* to point *i*.

```
;; The data base for this test
;;
(setq path.out
 '(
   (edge h i)
   (edge a h)
   (edge a b)
   (edge b c)
   (edge c d)
   (edge a c)
   (edge a g)
   (edge g d)
   (edge a f)
   (rule (path (? x) (? y))
         (por (edge (? x) (? y))
              (pand (edge (? x) (? i)) (path (? i) (? y)))))
   )
 )

;; The assertion to prove
```

```
;;
(setq query-test '((path a i)))
```

C.4.3 Instruction Mix

The MCODE instructions executed by *Multilog* in proving this assertion were as follows:

Instruction	Count
EVAL	105774
ITGOTO	62675
TYPEEQ	38703
GVAL	23844
GETSTRUCT	23213
RETURN	18966
CALL	18966
NULL	18538
COPY	15069
POP	15041
EQ	9665
CALLRTN	6058
PUSHNIL	5875
PUSHNUM	3774
PUSHVAL	2997
SEVAL	2139
DETERMINE	2096
FUTURE	2096
PUSHENV	1157
POPENV	1157

Instruction	Count
IGOTO	903
CONS	770
NUMEQ	588
PLUS	373
SETNTHSTR	366
CLOSURE	183
PLIST	122
SETSTRUCT	40
NTHSTR	29
SETOPT	17
PUTCH	15
SGVAL	11
MAKESTRING	7
INTERN	6
STRLEN	6
FFLUSH	5
PRINT	3
PRINC	2
INFOOFF	1

C.5 Quicksort

C.5.1 Source Code

Quicksort is a well known algorithm for sorting a list of numbers. The version shown here uses futures extensively.

```
; Copyright (c) 1984.  Robert H. Halstead, Jr. and Juan R. Loaiza.
; Quicksort programs in Multilisp, RHH, April 1984.

(defmacro bundle-parts (left right)
  `(cons ,left ,right))

(defmacro left-part (bundle)
  `(car ,bundle))

(defmacro right-part (bundle)
  `(cdr ,bundle))

(defun pqsort (l) (pqs l nil))

;; Recursive parallel quick sort routine
(defun pqs (l rest &aux parts)
  (if (null l)
      rest
      (setq parts (ppart (car l) (cdr l)))
      (pqs (left-part parts)
           (future (cons (car l) (pqs (right-part parts) rest))))))

;; Partition the list in parallel
(defun ppart (elt l &aux cdrparts)
  (if (null l)
      (bundle-parts nil nil)
      (setq cdrparts (future (ppart elt (cdr l))))
      (if (> elt (car l))
          (bundle-parts (cons (car l) (future (left-part cdrparts)))
                        (future (right-part cdrparts)))
          (bundle-parts (future (left-part cdrparts))
                        (cons (car l) (future (right-part cdrparts)))))))

;; Function to generate a list of 'n' random numbers
(defun g (n)
  (if (<= n 0)
      nil
      (cons (- (rand 2000) 1000) (g (- n 1)))))

(define ranseed 12345)

(defun rand (&optional max)
  (setq ranseed (% (+ (* ranseed 54321) 75319) 2000000))
  (if max (% ranseed max) ranseed))
```

C.5.2 Test Data

All of the runs of Quicksort reported in this thesis sorted a 700 element list of random numbers.

C.5.3 Instruction Mix

The MCODE instructions executed by *Quicksort* in sorting a 700 element list were as follows:

Instruction	Count
EVAL	52126
GETSTRUCT	35939
PUSHNIL	18587
DETERMINE	17562
FUTURE	17562
ITGOTO	13416
CONS	12390
NULL	7748
GVAL	7222
POP	7221

Instruction	Count
SEVAL	7220
RETURN	6709
CALL	6709
GT	5683
TYPEEQ	534
CALLRTN	513
PUSHNUM	513
IGOTO	512
PLUS	512
INFOOFF	1

Bibliography

[1] *16-Bit Microprocessor User's Manual.* Motorola Inc., Englewood Cliffs, N.J., 1982. 68000 Manual.

[2] *Cray-2 Computer System Functional Description.* Cray Research, May 1985. HR-2000.

[3] *DBC68K Hardware Reference Manual.* Microbar Systems Inc., Palo Alto, CA, 1.0 edition, May 11 1982.

[4] *DBR 50 Hardware Reference Manual.* Microbar Systems Inc., Palo Alto, CA, 1.0 edition, November 2 1982.

[5] *Development of a Butterfly Multiprocessor Test Bed.* Quarterly Technical Report 5872, Bolt Beranek and Newman Inc, 1985.

[6] *Nu Machine Technical Summary.* Texas Instruments Corp., Irvine, California, 1984.

[7] *Occam Programming Manual.* INMOS Ltd., November 1984.

[8] *VAX Architecture Handbook.* Digital Equipment Corp., 1981.

[9] Thomas Anderson. *The Design of a Multiprocessor Development System.* Technical Report TR-279, Laboratory for Computer Science, M.I.T., Cambridge, Mass., September 1982.

[10] Arvind and Robert E. Thomas. *I-Structures: An Efficient Data Structure for Functional Languages.* Technical Report MIT/LCS/TM-178, MIT Laboratory for Computer Science, October 1980.

[11] H. Baker and C. Hewitt. *The Incremental Garbage Collection of Processes.* Memo 454, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass., December 1977.

[12] Elizabeth Bradley. *Logic Simulation on a Multiprocessor.* Technical Report TR-380, M.I.T. Laboratory for Computer Science, Cambridge, Mass., November 1986.

[13] Elizabeth Bradley and Robert H. Halstead, Jr. Simulating logic circuits: a multiprocessor application. 1987. To be published in 'Journal of Parallel Processing'.

[14] Petar Brajak, Sasa Presern, and A.P. Zeleznikar. Rationale and concepts for the Parasys parallel processor architecture. Seminar given at MIT Lab for Computer Science, April 15 1987. Authors are with Iskradelta Inc., Jozef Stefan Institute, and Edvard Kardelj University, Yugoslavia.

[15] W.C. Brantley, K.P. McAuliffe, and J. Weiss. RP3 processor-memory element. In *Proceedings of 1985 International Parallel Processing Conference*, pages 782–789, August 1985.

[16] R.A. Brooks, R.P. Gabriel, and G.L. Steele. An optimizing compiler for lexically scoped Lisp. In *Proceedings of the 1982 ACM Compiler Construction Conference*, June 1982.

[17] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. In *IEEE Transaction on Computers*, pages 1112–1118, December 1978.

[18] D. W. Clark and W. D. Strecker. Comments on 'The Case for the Reduced Instruction Set Computer'. *Computer Architecture News*, 8, October 1980.

[19] William Clinger. *The Revised Revised Report on Scheme, or an Uncommon Lisp*. Memo 848, M.I.T. Artificial Intelligence Laboratory, Cambridge, Massachusetts, August 1985.

[20] W.F. Clocksim and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, 1981.

[21] William J. Dally. Message-driven processor: preliminary architecture. November 1986. Working Paper: CVA-1.

[22] J.K. Foderaro and K.L. Sklower. *The FRANZ Lisp Manual*. Technical Report, University of California, Berkeley, April 1982.

[23] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. *MIT Press Series in Computer Science*, M.I.T. Press, Cambridge, MA, 1985.

[24] L.R. Goke and G.J. Lipovsky. Banyan networks for partitioning multiprocessor systems. In *Proceedings of the First Annual Symposium on Computer Architecture*, pages 21–28, 1973.

[25] John Goodhue. The Monarch multiprocessor. MIT VLSI Seminar, April 14 1987. Author is employed by B.B.N. Inc.

[26] Alloan Gottlieb, Ralph Grishman, Clyde Kruskal, Kevin McAuliffe, Larry Rudolph, and Mac Snir. The NYU Ultracomputer — designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, 1983.

[27] Sharon Gray. *Using Futures to Exploit Parallelism in Lisp*. Master's thesis, M.I.T., Cambridge, MA, February 1986.

[28] Robert H. Halstead, Jr. *Architecture of a Myriaprocessor.* La Jolla Institute, La Jolla, California, 1981.

[29] Robert H. Halstead, Jr. Architecture of a myriaprocessor. *IEEE COMPCON Spring 81*, 299–302, February 1981.

[30] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. *ACM Symposium on Lisp and Functional Programming*, August 1984.

[31] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[32] Robert H. Halstead, Jr. Parallel symbolic computing. *IEEE Computer*, 19(8):35–43, August 1986.

[33] Robert H. Halstead, Jr., T. Andersor, R. Osborne, and T. Sterling. Concert: design of a multiprocessor development system. In *13th Annual Symposium on Computer Architecture*, pages 40–48, Tokyo, June 1986.

[34] Robert H. Halstead, Jr., Juan R. Loaiza, and Moses H. Ma. The Multilisp manual. September 1986. PPG Group Working Paper.

[35] Stephan Herron. *A General-Purpose Architecture Simulator.* Bachelor Thesis, M.I.T., Cambridge, MA, June 1987.

[36] Daniel W. Hillis. *The Connection Machine. MIT Press Series in Artificial Intelligence*, M.I.T. Press, Cambridge, MA, 1985.

[37] C.A.R. Hoare. Communicating sequential processes. *C.A.C.M.*, 21(8):666–667, 1978.

[38] J. Holloway, Jr. G.L. Steele, G.J. Sussman, and A. Bell. *The SCHEME-79 Chip.* Memo 559, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass., January 1980.

[39] IEEE Task P796/D2. Proposed microcomputer system 796 bus standard. *IEEE Computer*, 13(10):89–105, October 1980.

[40] Thomas Knight, David Moon, John Holloway, and Guy Steele. *CADR*. Memo 528, M.I.T. Artificial Intelligence Laboratory, Cambridge, Massachusetts, June 1979.

[41] J. McCarthy, P. Abrahams, D. Edwards, T. Hart, and M. Levin. *LISP 1.5 Programmer's Manual*. M.I.T. Press, Cambridge, Mass., 1962.

[42] Louis Monier and Pradeep Sindhu. The architecture of the Dragon. In *IEEE Spring CompCon*, pages 118–121, 1985.

[43] David A. Moon. Symbolics architecture. *IEEE Computer*, 20(1), January 1987.

[44] Peter Nuth and Randy Osborne. Nusim-doc.text. P.P.G. Internal Document.

[45] Randy Osborne. Personal Communication, May 1987.

[46] Gregory M. Papadopoulos. *An Engineering Implementation of the Tagged-Token Dataflow Machine*. Computation Structures Group Memo 270, M.I.T. Laboratory for Computer Science, 1987.

[47] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfleder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): introduction and architecture. In *Proceedings of 1985 International Parallel Processing Conference*, pages 764–771, August 1985.

[48] Andrew R. Pleszkun and Matthew J. Thazhuthaveetil. The architecture of Lisp machines. *IEEE Computer*, 20(3):25–35, March 1987.

[49] C.L. Seitz. The cosmic cube. *C.A.C.M.*, 28(1):22–33, January 1985.

[50] Burton Smith. *The Architecture of the HEP*, pages 41–55. *MIT Press Series in Scientific Computation*, M.I.T. Press, 1985.

[51] Susan Solomon. *A Query Language on a Parallel Machine.* Bachelor Thesis, M.I.T., Cambridge, MA, June 1985.

[52] S. Sugimoto, K. Agusa, K. Tabata, and Y. Ohno. A multi-microprocessor system for concurrent Lisp. In *Proceedings of International Conference on Parallel Processing*, June 1983.

[53] Gerald Sussman and Harold Abelson. *Structure and Interpretation of Computer Programs.* M.I.T. Press, Cambridge, Mass., 1984.

[54] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR lisp architecture. In *Thirteenth International Symposium On Computer Architecture*, June 1986.

[55] Vito A. Trujillo. System architecture of a reconfigurable multimicroprocessor research system. In *1982 International Conference on Parallel Processing*, 1982.

[56] Colin Whitby-Strevens. The Transputer. In *12th Annual International Symposium on Computer Architecture*, pages 292–300, June 1985.

[57] Alexander Wolfe. TI puts its Lisp chip into a system for military AI. *Electronics*, 60(6):95–96, March 19 1987.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-395	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125 and N00014-84-K-0099
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS *(City, State, and ZIP Code)* 545 Technology Square Cambridge, MA 02139	7b. ADDRESS *(City, State, and ZIP Code)* Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
COMMUNICATION PATTERNS IN A SYMBOLIC MULTIPROCESSOR

12. PERSONAL AUTHOR(S)
Nuth, Peter R.

13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM ______ TO ______	14. DATE OF REPORT *(Year, Month, Day)* 1987 June	15. PAGE COUNT 188

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Data reference, Lisp, locality, multiprocessor, symbolic processing

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

An important design decision for large scale multiprocessors is the balance of processor power to communication network bandwidth. In order to evaluate different design alternatives, it is necessary to be able to predict the load imposed on the network by a programming model.

This thesis quantifies that communication load for a model of parallel symbolic computing using the Multilisp language. An organization of a shared memory multiprocessor for Multilisp is proposed. The Nusim architectural simulator is built to model that organization. Several Multilisp application programs are run under Nusim, and the communication requirements of each program is measured. The locality of reference of memory accesses for the benchmarks is determined for three proposed multiprocessor topologies. The effect of scheduling decisions in increasing locality of access and in reducing global communication is studied. The thesis concludes with implications of scheduling policies on the design of parallel computer systems.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE *(Include Area Code)* (617) 253-5894 — 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

☆U.S. Government Printing Office: 1985—607-047

Unclassified