An Architecture for Parallel Symbolic Processing Based on Suspending Construction

Eric R. Jeschke

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

April 1995

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Dr. Steven D. Johnson
(Principal Adviser)

Dr. David S. Wise

Dr. Dennis B. Gannon

Bloomington, Indiana
April 3, 1995.

Dr. Jonathan W. Mills

To Charmaine

ABSTRACT

Symbolic languages relieve the programmer from many resource management considerations that are required for traditional programming languages, such as memory and process management. This makes them promising candidates for solving a large, general class of programming problems on shared-memory MIMD-class parallel computers. The incorporation of these resource management problems into the parallel language implementation domain is a topic of active research.

This dissertation discusses the design of a list processing engine for implementing parallel symbolic languages on stock multiprocessors. The design results from experience implementing an applicative language on the *BBN Butterfly* multiprocessor. The language and engine are based on *suspending construction*, a fine-grained, concurrent, non-strict computation model.

The contributions of the dissertation are:

1. A virtual machine architecture for fine-grained parallel list multiprocessing. This virtual machine design identifies key areas where hardware support would accelerate the execution of programs using computational models similar to suspending construction, such as Lisp with *futures*.

2. A microkernel providing memory management, process management and device management for high level parallel symbolic programs. The resource management techniques described represent new approaches to these problems for many parallel symbolic languages.

3. An analysis of *Daisy*, a language based on suspending construction and implemented on this architecture and microkernel.

ACKNOWLEDGEMENTS

I would like to thank my advisor and mentor, Steve Johnson, for introducing me to an interesting research project and for his guidance, support and warm friendship. I am also grateful to my research committee members: David Wise, Dennis Gannon and Jonathan Mills, for their help and suggestions. Assistant Dean Frank Hoole at the Graduate School funded my assistantship while I finished my dissertation. I am also indebted to the secretaries and staff of the IU Computer Science Department for innumerable favors and friendly, courteous service.

My family's support and encouragement has meant a great deal to me during my years at Indiana. Love and thanks to the Jeschke's: Marlin, Charmaine, David, Margaret, and Bart Miller.

I have made many wonderful friends during my "tenure" in Bloomington. The following people have directly contributed to my well being as a graduate student:

- Will Reilly, Charles Daffinger, Anand Deshpande and Terri Paullette;
- Tim Bridges and the rest of the crew at *Data Parallel Systems* for good camer-aderie and all the cheese crackers I could eat;
- Dear housemates Judy Augsburger and Susan Coleman;
- Good buddies Lisa Meeden, Gary McGraw, Amy Barley, Paul and Doug Steury, Ethan Goffman, Darrin Pratt and Shelly Barnard.

Extra special thanks to *Stash*, *Hoadley*, *Sophie*, and especially, **Amy Baum**, for the peace and happiness you have given me.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

Introduction

High-level programming languages for symbolic processing have been widely studied since the inception of Lisp in the late 1950's. The family of symbolic languages is now a diverse group, encompassing Lisp and its derivatives, purely functional languages like Haskell, logic programming (e.g. Prolog), Smalltalk, and many more. One of the attractive properties of these languages is that they unburden the programmer from storage management considerations such as allocation and reclamation of objects. By automating these low-level implementation details, programs written in these languages are rapidly developed and are usually simpler to understand and maintain.

Interest in symbolic processing has further increased in recent years as parallel computers have made inroads into the mainstream computer market. Small- to medium-sized RISC-based shared memory MIMD architectures will characterize the servers and workstations of the next decade. Symbolic languages are good candidates for programming these machines; their heap-orientation maps well to the shared memory architecture and their underlying computational models often facilitate parallelism. As with storage management, symbolic languages can subsume low-level concurrent programming details such as process creation, communication, synchronization, load balancing, scheduling and data distribution; detail that seriously complicates programs written in traditional imperative languages. Symbolic languages remove much of this "noise" from programs, improving readability, portability, and scalability of the code. Additional benefits are reduced software development cycles and maintenance costs. Removing these details from the language shifts the burden of memory and process management from the programmer to the language implementation. The incorporation of these difficult problems into the language implementation domain is a topic of active research.

The subject of this dissertation is the design and implementation of *DSI*; a virtual machine and microkernel for list-based, symbolic multiprocessing. I also describe and briefly analyze the operation of *Daisy*, an applicative language built on top of DSI. DSI and Daisy are designed around *suspending construction*, a *Lisp*-derived, demand-driven, parallel computation model developed at Indiana University. This dissertation provides insights into the problems, issues and solutions involved in a real parallel implementation of a language based on suspending construction. Previous incarnations of DSI/Daisy have been multi-threaded sequential implementations, designed primarily to explore the demand-driven, *lazy* aspect of suspending construction-based computation. The premise of suspending construction as a parallel processing vehicle, while much discussed, was not realized in an actual implementation. This dissertation work extends DSI's virtual machine architecture to handle the complexities of a shared-memory, MIMD parallel implementation. The DSI kernel, an embedded operating system that handles resource management for programs running on this architecture, is redesigned from the ground up to handle parallel task, memory and device management *in* parallel, using a loosely-coupled, distributed design. The algorithms presented here are based on an implementation of DSI/Daisy on the *BBN Butterfly*, a commercial multiprocessor.

In a practical sense, this work results in a platform for further exploration of parallel and system-level computing based on suspending construction, with a richer development environment and improved language, interfaces and tools. In a broader sense, the contributions herein can be categorized into three main areas. First, the virtual machine description highlights the key hardware needs of suspending construction and how that differentiates DSI's machine design from conventional architecture. This suggests specific enhancements to stock hardware that would greatly benefit execution of languages oriented toward fine-grained list processing. Secondly, we present the design of a low-level, general kernel for fine-grained parallel symbolic processing based on this architecture. The kernel is described in terms of memory, process and device management algorithms. My approaches to these problems suggest new, alternative strategies for resource management in parallel symbolic languages of all flavors. Finally, a discussion of suspending construction in the context of the Daisy language highlights the effectiveness and limitations of this computing model for parallel processing, and suggests ways in which the latent parallelism of the model might be better exploited in future designs of the language.

The topics discussed in this thesis are a cross-section of the areas of programming languages, computer architecture and parallel processing. As such, this dissertation will primarily be of interest to language implementors and, to a lesser degree, other researchers in the aforementioned areas. In particular, much of this work is applicable to other parallel symbolic language implementations, especially applicative languages.

1.1 Overview of the Dissertation

As with many symbolic languages, the architecture and resource management system (DSI) are strongly influenced by the surface language (Daisy) and computation model (suspending construction). Therefore it is important to provide a thorough background to explain the model that motivated our design and implementation decisions. The remainder of this chapter introduces symbolic processing and the suspending construction model and concludes with a brief history behind the Daisy/DSI project.

Chapter 2 provides general background and introduction to the topics and issues discussed in the rest of the thesis, particularly resource management problems for dynamic, fine-grained parallelism. Chapter 3 discusses the design of DSI's virtual machine. Chapter 4 describes the overall structure of DSI's kernel, which provides background for references to relevant implementation details in the following chapters. Chapters 5–7 present the design and implementation of the kernel's resource management algorithms along memory, process and device categories.

Chapter 8 discusses aspects of suspending construction in the Daisy language. The language itself is only tangentially relevant to this thesis as a vehicle for expressing suspending construction programs, and a brief introduction to the language and its primitives are all that is required. A more thorough introduction and tutorial to the Daisy language is the *Daisy Programming Manual* [Joh89b] and associated research papers (see bibliography).

Chapter 9 summarizes the work accomplished, indicates directions for future research and references related work in *Lisp*-based parallel symbolic processing (comparisons are sprinkled throughout the thesis).

1.2 Parallel Symbolic Processing

In the terminology of parallel processing, applications are sometimes characterized as falling into the domain of either *symbolic* or *numerical* computation [AG94, p. 37]. The distinction between these two categories is somewhat fuzzy; Almasi and Gottlieb [AG94] distinguish the two by the ratio of *calculation* to *data motion*. Numerical processing is arithmetic-intensive; example applications include particle calculations, fluid dynamics, and *CAD*. Symbolic processing emphasizes the rearrangement of data. Examples of applications in this category include searching, non-numerical sorting, lexical analysis, database management, artificial intelligence problems, etc. Traditionally, imperative languages such as *Fortran* and *C* have been used for numerical programming, while languages like *Lisp*, *Smalltalk*, and *Prolog* have been used to write symbolic programs.

Halstead [RHH85] notes that numerical computation generally has a relatively data-independent flow of control compared to symbolic computation. Compiler dependence analysis can turn this data independence into a fair degree of parallelism that is amenable to vector, pipelined and *SIMD* approaches. The parallelism of symbolic computation, on the other hand, can be very data-*dependent*; this suggests a *MIMD* approach in which parallelism is primarily generated and controlled dynamically. Although parallel symbolic languages are quite diverse and may differ substantially in many ways, they all share a core set of resource management problems associated with this dynamic control of parallelism: task scheduling and load balancing, dynamic memory management, etc. Chapter 2 provides an introduction to many of these problems from a *Lisp* perspective.

The number of *parallel* symbolic language implementations has grown considerably in the last ten years or so. Most of these languages fall into one of the following broad categories:

- parallel dialects and descendants of the *Lisp* family (we include *Daisy* here);

- parallel *PROLOG* and concurrent logic/constraint languages;
- purely functional languages (including *dataflow*), and
- concurrent object-oriented languages.

There are undoubtedly more than a few parallel symbolic languages that can't be comfortably lumped into one of these areas, but the bulk of working parallel implementations certainly do. The differences between these various languages are due to many factors, including:

1. an emphasis on particular data types or operations (e.g. list processing, tuples, objects, etc.);
2. an orientation toward a particular reduction or computation model (e.g. unification, graph reduction, data flow, etc.);
3. within the same language family, in syntax and semantics;
4. the type of architecture an implementation is targeting (e.g. distributed/message passing vs. shared memory);
5. specific approaches to parallelism and resource management (i.e. dynamic management of processes, memory and I/O).

In the course of this report we will elucidate in these terms how and where Daisy/DSI fits into the symbolic processing field. For now, suffice it to say that Daisy is a descendant of Lisp (although syntactically and semantically, it shouldn't really be called a Lisp *variant*). This heritage is reflected in its underlying architecture, DSI, which is designed as a list-multiprocessing engine.

1.3 Suspending Construction

Suspending construction is a non-strict, concurrent evaluation model for list-oriented, applicative languages. Suspending construction inspired the development of the *Daisy* language and the *DSI* platform on top of which Daisy is constructed.

1.3 Suspending Construction

The basis of suspending construction is a transparent decoupling of structure formation from determination of its content in a symbolic domain of lists and atoms. Under most Lisp variants, structure is built recursively by applicative-order evaluation of subexpressions, whose results form the template for initializing the structure (usually a *list*). For example, evaluating the *Lisp* expression

```
(list (fact 5)  (fact 7)  (fact 10) )
```

where `fact` is the *factorial* function, preforms three subexpression evaluations of `(fact 5)`, `(fact 7)` and `(fact 10)` *before* `cons`-ing the results into a list. Under suspending construction, *suspensions* are created in lieu of subexpression evaluation, and a structure is returned containing references to the suspensions.

A *suspension* is an independent, fine-grained, suspended process that contains any information necessary to compute the subexpression it represents. Normally a suspension remains dormant until an attempt is made to access it from a structure (a *probe*), which causes a kernel trap. At that point the system intervenes, *activating* the target suspension and temporarily blocking the probing process (which itself is a suspension). Once activated, a suspension attempts to compute its subexpression and replace its own reference in the structure with that of the result. When a suspension has *converged* in this manner, subsequent accesses (probes) to the same location fetch the expected *manifest* object. This cycle of probing—activation—convergence is known as *coercing* a suspension, and it is entirely transparent to programs, which are simply creating and manipulating data structures as usual. The system is responsible for detecting and scheduling suspensions as necessary.

We visualize suspensions as *clouds*; figure 1 shows the result of a suspending version of list construction applied to the previous example. If we attempt to access the second element of this list it results in two probes that coerce the tail of first cell and the head of the second cell into existence, as shown in figure 2.

Between activation and convergence a suspension will usually traverse some portion of its inputs, coercing any existing suspensions that it happens to probe, and it will also create its own data structures, forming new suspensions as a byproduct. Thus, a running program creates a dynamic web in the heap of many thousands of suspensions all linked together through data structures, as shown in figure 3. The locus of control shifts from one suspension to another as they probe each other. The

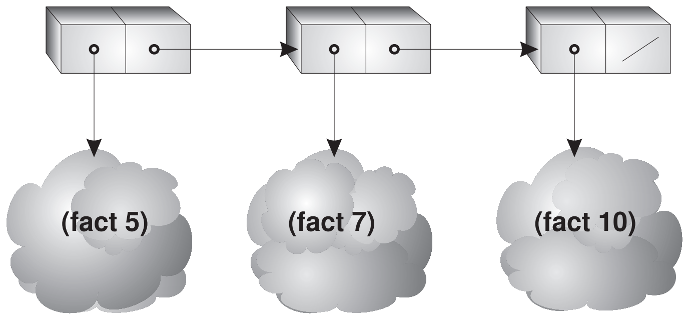


Figure 1: Suspending construction

web expands as new suspensions are added at the fringe of the computation and contracts as suspensions converge and dereferenced structure is garbage collected.

1.3.1 A Concurrent Computation Model

Suspending construction has several properties that make it an attractive basis for a parallel language implementation. First, it provides a natural, transparent decomposition of programs into networks of fine-grained processes, as described above. Second, it provides transparent communication and synchronization between processes. Processes communicate by producing and consuming shared data structures in the heap, without any global knowledge of their role in the network of processes. Synchronization between suspensions is implicitly handled by the system; there are no explicit synchronization constructs needed.

Thus, suspending construction provides a simple but complete automation of the important details of parallel programming: process decomposition, communication, and synchronization. Just as with automated storage management in Lisp programs, the programmer is not required to handle any of the complicated aspects of parallel task management and can concentrate on writing a suitably parallel *algorithm*. This leaves the "hard details" of parallel process management to the implementation, which is what this thesis describes.

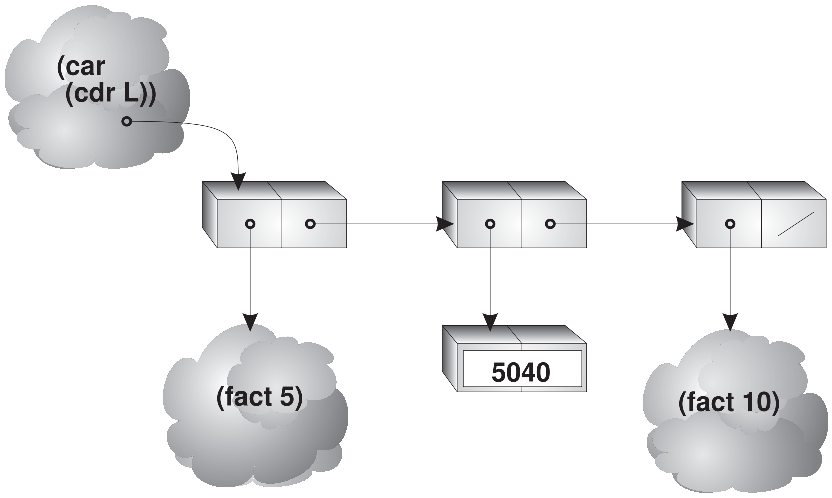


Figure 2: Coercing a Suspension

1.3.2 Suspensions and Evaluation Semantics

Suspending construction is based on the premise that many system primitives (notably `cons`) and user-defined functions are not fully *strict*, allowing some latitude in *if* and *when* their arguments are actually needed. *Lazy* or *non-strict* languages try to exploit this property of programs by explicitly avoiding evaluation of function arguments until they are needed. A language using suspending construction, on the other hand, derives its evaluation semantics implicitly from the underlying suspension scheduling policy rather than explicit laziness on the part of an interpreter or compiler.

To see this, consider two scheduling policies at the extreme ends of the scale:

- If a suspension is immediately activated upon creation, the result is *eager* evaluation of the associated subexpression[1].
- If activation is delayed until the suspension is probed, the result is *lazy* or

[1] The term *eager* does not have a universally agreed-upon definition in the programming languages community. In particular, it is occasionally confused in a semantic scope with the term *strict*. We use it here in the operational sense as described by [AG94, p.268] and others to mean early concurrent evaluation of the subexpression.

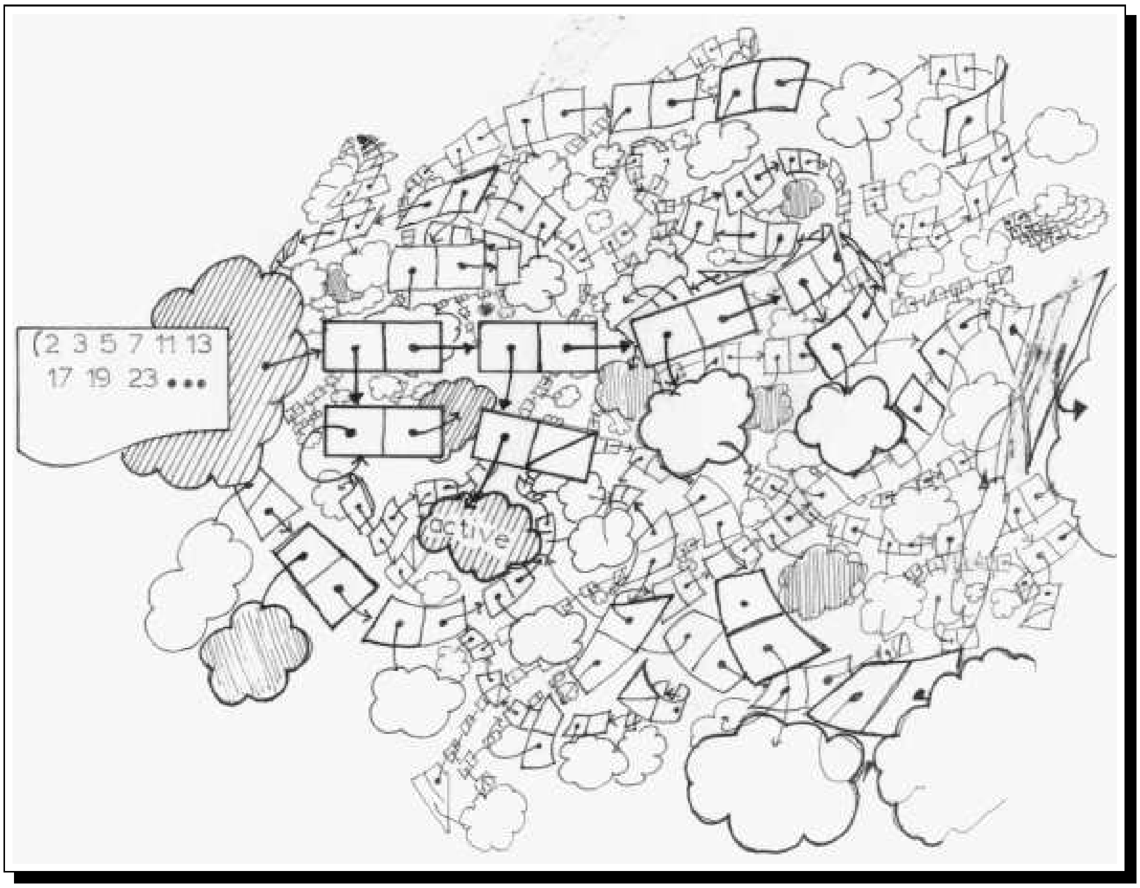


Figure 3: A Suspending Construction Data Space
Drawing by Steven D. Johnson, 1980. Reproduced with permission.

demand-driven evaluation of the subexpression.

By extension, an interpreter that uses a suspending constructor uniformly throughout, including the construction of *environments* from function arguments, will inherit its evaluation semantics from the scheduling policy. This renders the evaluation-order issue orthogonal to the language implementation; an interpreter can be made eager or lazy depending upon the scheduling policy used. The same remarks apply to compiled code. Daisy, which uses a lazy constructor by default, leverages a program's inherent laziness by deferring computations stored in structures until they are actually needed. This gives Daisy a *call-by-need* semantics.

The eager vs. demand-driven scheduling policies referred to above can really be viewed as two endpoints of a continuum [AG94, p.267], [Liv88]. By itself, demand-driven scheduling provides maximum laziness, computing no more than is absolutely necessary, but forgoes some opportunities for parallelism. Eager scheduling can generate *too much* parallelism. The ideal situation seems to lie somewhere between the endpoints of this scheduling continuum, and a number of ideas have been published in this area. Some of these, notably *data flow* reduction, start from an eager perspective and attempt to reduce parallelism. *Lenient evaluation* [Tra91] tries to take the middle ground, forgoing laziness but still attempting to achieve non-strict semantics where possible, making some sacrifices in expressiveness as a result. Still others, like DSI's approach, start from the demand-driven (lazy) perspective and relax it to allow safe eager parallelism and constrained speculative parallelism.

1.3.3 Side-effects and Parallelism

Parallel languages that include side-effects (e.g. assignment, destructive updating) usually adopt an intuitive (e.g. applicative) evaluation order so that programmers can reason about the ordering of effects in the program. The languages are then endowed with explicit constructs for parallelism or alternative (e.g. delayed) evaluation orders. Such languages must also supply explicit locks and other synchronization tools to allow the programmer to explicitly handle the extra dependences imposed by side-effects.

To extract implicit parallelism, an imperative language can employ dependence analysis and other compiler analytic techniques; this approach is popular for many "dusty-deck" Fortran applications, as well as for imperative symbolic programs [Lar91]. These methods are not always effective, however, primarily due to the increased number of dependences involved, as well as *aliasing* and other artifacts of imperative programming [AG94].

One alternative to these analytic techniques is to provide a history *roll-back* mechanism to be able to roll back the computation when conflicts occur due to side effects. At least one implementation of the *Scheme* language is based on this technique [TK88].

1.3 Suspending Construction

All of these techniques impose complications in various ways, either on the programmer to explicitly synchronize his program, on the compiler for analysis, or on the run-time system to roll back computation. Alternatively, a language can restrict or eliminate side-effects and not suffer these complications; such is the approach of the *declarative* family of languages, which includes "pure" functional languages, and also certain *quasi-functional* languages, like Daisy.

Daisy does not have side-effects in the traditional sense (it does provide top-level assignment of global function identifiers). There are two main reasons for this: laziness and implicit parallelism. Suspending construction does not *preclude* the use of side-effects, but their inclusion in the surface language would make it extremely difficult to reason about programs because demand-driven scheduling results in non-intuitive evaluation orders. A lack of side-effects also allows Daisy to take advantage of implicit parallelism opportunities. Removing side-effects removes all dependences from the program with the exception of *flow* dependences [AG94], which are handled by the probe/coercion mechanism described earlier. This makes suspensions mutually independent, and allows the system considerable flexibility in scheduling them. The consequences of this flexibility are particularly important for parallelism; suspensions can execute in any order, including lazily or concurrently, without affecting the result. This effectively reduces scheduling from a problem of efficiency *and* correctness and to one of efficiency only. That problem is determining which subset of suspensions to execute at a given time to achieve maximum system throughput, while minimizing the overhead of communicating scheduling information between processors. We want to avoid scheduling suspensions that are simply blocked awaiting the convergence of other suspensions.

Although Daisy lacks explicit side-effects, it is not *referentially transparent*, a property shared by many *declarative* languages. Many computing problems arise that are inherently nondeterministic, especially in the realm of systems programming. Daisy addresses these problems by providing an applicative nondeterministic construct. Daisy's *set* primitive returns a copy of its list argument, the copy permuted in an order determined by the convergence of the suspensions in the argument list. For example, the following *Daisy* expression

```
set:[ fact:10 fact:5 fact:7 ]
```

would likely return

```
[120 5040 3628800]
```

The order reflects the cost of computing the three subexpressions. The *set* primitive can be used to implement *fair merge*, for expressing explicit speculative computation and concurrency, and many asynchronous applications.

The rationale for including side-effects in parallel symbolic languages is elegantly laid out by Halstead in [RHH85], who appeals to arguments of expressiveness and efficiency. However, this approach negates many implicit opportunities for laziness or parallelism and leaves the problem of creating and synchronizing parallel processes in the programmer's domain. In this research we adopt the view that these details should be left to the underlying implementation as much as possible. Our concern is not with the expression of parallelism or concurrency in the language (see below), but rather with the inherent limitations on parallelism imposed by side-effects.

In regard to efficiency, a language with automated process management is unlikely to be as efficient as a more traditional imperative language such as Scheme or C. Just as there is a cost associated with doing automated storage management under Lisp (as opposed to explicit storage management under, say, C), there is a cost added with automating process management. The ever-increasing speed and capacity of hardware and gradual improvements in language implementation technology make this trade-off worthwhile in most situations.

The property of laziness can be considered as much a factor in language expressiveness as the inclusion of side-effects. Daisy's lack of side effects is a pragmatic approach to achieving laziness and parallelism rather than a means of upholding language purity. Similarly, the inclusion of indeterministic operators is a practical means to offsetting the constraints imposed by our choice to abandon side effects. Thus, we deemphasize equational reasoning in favor of expressiveness. This philosophy may help explain Daisy's seemingly odd position "straddling the fence" between imperative and pure functional languages.

1.3.4 Identifying Parallelism

One of the main issues relating to parallel symbolic processing is where the parallelism originates. There are three levels at which parallelism can be identified:

- Parallelism at the *system* level relies on the system being able to determine when two tasks can run in parallel. A good example of system-level parallelism is the *data flow* computation model.

- Parallelism at the *language* level uses the knowledge of the language to try to generate parallelism. This type of information includes whether side-effects are allowed or not, strictness and dependence information from analysis and the behavior of primitives. Functional languages fall into this area, as does Daisy.

- Parallelism at the *programmer* level relies on the programmer to specify the parallelism and may also require him to handle synchronization in the presence of side-effects.

These types of parallelism can also be considered "bottom-up" to "top-down": data flow corresponds to a bottom-up approach, while programmer-explicit parallelism is top-down.

Note that these distinctions are not always clear-cut; in actuality languages may overlap one or more of these areas. For example, Scheme with *futures* [RHH85] falls into the last category, but overlaps the second somewhat, since futures are not strictly semantically required to be parallel and thus may be optimized in certain ways [Moh90][2]. This is similar to the notion of *para-functional* programming [HS86, Hud86]; the idea that programmers *annotate* programs to provide clues for the system without actually changing the semantics of the program or getting involved in resource management issues (although the latter approach is also a possibility [Bur84]). This is a step backward from complete automated parallelism, but still does not cross the line into explicit parallel decomposition of tasks or synchronization for side-effects, etc. Daisy straddles the second and third categories, for reasons which are explained in chapter 8.

1.3.5 A Cost Analysis of Suspending Construction

There is a computational cost associated with the creation and coercion of suspensions over and above that of strict evaluation of subexpressions. After all, for each

[2]Eager parallelism is intrinsic in the notion of a future, however; see [RHH85] for details.

subexpression we are creating a new (albeit fine-grained) process, completely decoupled from the current one. The suspension will execute roughly the same code that it would if the expression were being strictly evaluated[3].

The cost of suspending construction for a given expression can be analyzed as follows:

- *Creating a suspension.* This is essentially the cost of allocating a suspension record from the heap and initializing it from processor registers; it is a constant-time operation, modulo garbage collection.

- *The space used by suspensions.* In addition to the space used by the suspension record itself, there is the space used by retaining references to environments and forms that would normally be discarded after strict evaluation. There is an implicit trade-off here against the space required for the result.

- *Checking for suspension references.* Except in certain optimized cases, the virtual machine must test the result of all probes that could return a suspension reference. This would not be a factor on specialized hardware, but must be taken into account on most stock architectures.

- *Context switching.* When a suspension is probed the system must suspend the current task and schedule the probed suspension for future execution. This scheduling may involve the manipulation of scheduling data structures and possibly interprocessor communication. At some point the suspension will be activated, requiring it to be loaded from memory into registers. When the suspension converges the probing task must be resumed.

These overheads are offset by any potential gains due to laziness and/or implicit parallelism in the program. All other things being equal, whether or not a program will run faster under suspending construction primarily depends on the degree of strictness and inherent parallelism in the algorithms used. Nevertheless, this overhead is significant, and we would like to minimize it.

[3]Not quite. Since it is running in a separate task, the suspended version usually does not have to save and restore as many registers on the stack. This is significant; depending on the amount of data it can be cheaper to allocate a suspension and garbage collect it than to save and restore a lot of registers on the stack.

To avoid it, we could revert to strict evaluation in cases where neither concurrency nor laziness is possible or desirable. Strictness analysis [Hal87, HW87] can reveal instances in a program where evaluation could be safely strict, but doing so may forgo opportunities for parallelism. Determining whether or not to evaluate an expression in parallel depends on the cost of the computation as well as dynamic factors such as the input data and current system load. All in all, determining when *not* to suspend is a difficult problem. Daisy manages some suspension avoidance by unrolling evaluation code to test for trivial cases before suspending them, allowing programming annotations for strictness, and under compilation, using strictness analysis [Hal87, HW87].

A more "brute force" approach to efficiency is to optimize suspension handling to make it more competitive with strict evaluation; more progress has been made at the DSI level in this regard. There are three main areas targeted for optimization:

- *Minimizing process state.* The smaller the task size, the faster it is to perform a context switch. There is a trade-off here: smaller state leads to more stack activity as registers are saved and restored. DSI uses a fined-grained suspension size of 32 bytes. This is all that needs to be saved or restored on a context switch.

- *Optimizing suspension manipulation primitives.* DSI implements suspension creation and detection in the virtual machine. This allows the most efficient mapping of these operations to the host's architecture and operating system capabilities.

- *Intelligent scheduling.* DSI's virtual register set uses *context windows* to hold multiple active processes. DSI's kernel takes advantage of this to accelerate context switching between processes.

1.4 History of the Daisy/DSI Project

The Daisy/DSI project had its genesis in research on purely functional Lisp in the mid-to-late 1970s at Indiana University. In their seminal paper "CONS Should Not Evaluate its Arguments" [FW76a] Friedman and Wise outline a simplified form of suspending construction (as described in section 1.3) with suspensions essentially implemented as special *thunks*[4] with convergence (i.e. they overwrite their respective references upon completion). In a sense they are like *memoized thunks* without the extra overhead of coercing the thunk on each invocation.

Friedman and Wise demonstrated that a pure applicative-order Lisp interpreter built using a suspending `cons` and suspension-aware versions of `car` and `cdr` has *call-by-need* semantics, resulting from suspended environment construction. They went on to describe how Landin's *streams* are naturally subsumed into such a system and how conditionals and other special forms can be implemented as simple functions due to the non-strict evaluation of function arguments [FW78e]. These revelations led to further observations on the ability to manipulate large or infinite data structures in constant space [FW78e].

Suspending construction naturally leads to *output-* or *demand-driven* computation [FW76c], because structure is coerced by an in-order traversal of the top-level data structures by the printer or other output devices. Printing a structure (and in the process, traversing it) propagates demand through suspension coercion to the *fringe* of the computation tree; in short, the output devices are the ultimate source of demand that drive the system.

In short order the connection was drawn between suspensions and multiprogramming [FW76b]. Since suspensions are mutually independent, self-contained objects, they can be coerced in parallel without mutual interference or synchronization. The problem is determining how to express parallelism in a demand-driven environment, which is naturally non-parallel. A few inherently parallel applicative constructs were developed, including *functional combination* [FW77, FW78c], a form of collateral function application; and *multisets* [FW78a, FW78b, FW79, FW81, FW80b], a non-determinate form of list constructor which was the precursor to Daisy's current *set* construct.

These successive discoveries led to iterative refinements of the underlying suspending construction model, including the evolution of suspensions from simple thunks to processes and various proposals for architectural enhancements to support suspending construction on multiprocessors [FW78d, FW80a, Wis85a]. Much of the credit for this is due to Johnson [Joh81, Joh85, Joh89c, Joh89a, Joh89b] who was the primary

[4]A *thunk* is a *closure* with no arguments, capturing a *form* with an *environment* needed to evaluate the form when the thunk is *thawed*.

developer on the early Daisy implementations.

The motivation behind the Daisy/DSI project is to explore systems-level and parallel programming from an applicative perspective. The division into the Daisy and DSI components allows us to study facets of this research at both a high and low level of implementation. Several sequential implementations of Daisy (and later, DSI) were developed, combining various aspects of this research. Each release has provided increasing levels of performance and features. The underlying list processing engine, dubbed *DSI* [Joh77, JK81], has undergone four major revisions. The surface language implemented on DSI is Daisy [Koh81, Joh89b], a pure language internally similar to *Scheme*, that is the high-level vehicle for exploring various ideas borne out of this research.

Daisy has been actively used for hardware- and systems-level modeling [Joh83, Joh84b, Joh84a, Joh86, JBB87, JB88, O'D87], applicative debugging [O'D85, OH87, O'D] matrix algebra [Wis84b, Wis84a, Wis85b, Wis86a, Wis86b, Wis87, WF87] and experimental parallel applicative programming.

CHAPTER TWO

A Primer

This chapter is an introduction to the topics discussed in the rest of the thesis. The sections below mirror the organization of chapters. Each section provides a short introduction to the topics within and raises some pertinent issues which are addressed in the chapters that follow.

2.1 Virtual Machine Issues

In the late 1970's and 1980's, it was popular to design (and even to build) architectures designed around a specific language. Today, the construction of hardware tailored to non-traditional programming languages is out of vogue. Economics, coupled with the rapid pace of RISC technology in the mainstream microprocessor industry has discouraged the development of specialized hardware for symbolic processing (witness the slow demise of the Symbolics Lisp machine). The reasons outlined above are doubly true for multiprocessors; there have been very few symbolic multiprocessing machines ever built, and fewer still on the way. The prospects for parallel symbolic processing are still bright, however, because stock hardware turns out to be reasonably good at supporting the implementation of parallel symbolic languages[1].

In the absence of specialized hardware, the next best thing may be to use a *virtual machine*. This is a common approach to the implementation of symbolic languages. A virtual machine provides a low level abstraction of the actual hardware and provides a portable base for implementing a language (and more recently, entire operating

[1]This has also been a factor in the decline of alternative, language-specific architectures.

systems) across various target architectures. The level of abstraction provided by the virtual machine depends on the needs of the implementor, but generally speaking, the higher the level of abstraction, the less it remains a virtual machine and the more it becomes a native implementation of the language[2]. DSI's virtual machine is quite low level, with registers and instructions that would have close correspondence to those on the target machine, as opposed to intermediate-level instructions that would be further compiled, such as in [FM90].

The special needs of a language or its computation model are often distilled into specific features or instructions in the virtual machine to facilitate and accelerate execution. This aspect of the virtual machine approach can identify architecture support that would be useful for accelerating language execution on actual hardware [Tra84, Veg84]. For example, dynamic tag-checking has long been considered a prime candidate for hardware support of Lisp programs [SH91]. Such a feature would be a relatively minor enhancement to current microprocessor designs. By some estimates it could increase performance of Lisp programs by as much as 35 percent, at a cost of an additional 2 percent of CPU logic [Joh90]. By analyzing the design of DSI's virtual machine we can determine those architecture features that would accelerate the execution of fine-grained parallel symbolic programs; in particular, programs based on the suspending construction model of execution. We describe the virtual machine in terms of its components: the register model, memory interface, instruction set, exception handling mechanism, and processor interconnection network. This provides a basis for comparison against their counterparts in current architectures.

Some of the important issues surrounding the design of the virtual machine are:

- How is the orientation toward list and symbol processing reflected in the view of memory and registers?
- What is the importance of tags and how are they used?
- How is suspending construction supported at the hardware level?
- How does the process grain size influence the register model?
- How are exceptions handled in a fine-grained process context?

[2]Virtual machines are sometimes called *abstract machines* in the literature. This term may denote a higher level of abstraction than *virtual machine* implies.

An important issue in parallel architecture design is the processor-memory interconnection model. Designs fall into two categories: shared or distributed memory. Although there are certainly examples of distributed symbolic language implementations, they are the exception rather than the norm. The extensive sharing relationships of most heap-oriented reduction models make a strong argument in favor of true shared memory. Lazy languages promote even more sharing, which further tips the scales toward shared memory. Thus we can assume shared memory, but that still leaves a lot of room for design variations.

We distinguish between two shared memory designs: one in which accesses to memory from any processor have constant time (disregarding interprocessor contention and caching) and one that does not. Multiprocessor architectures using the latter type are called *Non-Uniform Memory Access* (NUMA) designs [AG94]. NUMA architectures are generally more scalable, since they are designed using a many processor-to-many memory network [LD88] as opposed to several processors attached to one or more memory banks on a bus, which limits scalability due to bandwidth limitations. DSI's kernel is designed to handle both types of architecture.

2.2 Kernel Issues

At the core of most parallel symbolic language implementations is a *kernel* that handles the low-level dynamic resource management for processes running on the system. This resource management typically includes the allocation and sharing of memory, processors and devices. This is also what an operating system provides; the distinction between a kernel and an operating system is that the latter also includes many services and administration tools for *users*, including a file system for managing disk space, quotas and user accounts, login sessions, etc.

Symbolic languages have to reimplement their own resource management kernel because operating systems oriented toward traditional languages do not adequately address the resource management needs of symbolic languages. The differences can be summed up along memory, process and device management lines:

- Traditional languages require the programmer to handle their own memory allocation and deallocation. Symbolic languages usually provide automatic storage allocation and reclamation of objects.

- The process granularity for traditional languages is much greater than for symbolic languages and is normally defined by separate address spaces. In contrast, parallel symbolic languages have much finer grained processes that normally operate in a shared memory space. Symbolic languages may also provide automation for other process handling such as implicit process creation, interprocess synchronization and even process reclamation.

- For lazy symbolic languages, the traditional imperative model of I/O is not compatible with the non-sequential scheduling orders imposed by the system. Symbolic languages may also provide automatic device reclamation (closing dereferenced file descriptors via garbage collection).

The traditional implementation of a kernel is a protected layer of code underlying every process that is accessed by a function call-type interface; data is passed between user process and kernel on the stack. Modern *microkernels* are a stripped down version of the kernel providing just the essential resource management described above, relegating all other services to external implementation (sort of a RISC-like minimalist philosophy applied to operating system design). In addition, these microkernels are often implemented as a separate process rather than in the traditional approach described above. Programs interact with the kernel through a form of interprocess communication.

Another issue in *multiprocessor* kernel design is whether a single kernel controls all the processors in a sort of *master-slave* design, or whether the kernel is a reentrant, distributed design that runs on each processor. In the latter case, some mechanism is required to communicate between kernels on each processor so that they can operate as a whole to manage the machine.

2.2.1 Memory Management

Memory management in symbolic systems is fundamentally different than in conventional languages and operating systems. In the latter, "memory management" is almost solely concerned with implementing demand-paged virtual memory. Processes are required to do their own memory allocation and reclamation, although the system is expected to be able to recover the physical memory used by a process when it terminates. In contrast, memory management for symbolic languages amounts to handling *automatic* storage allocation and reclamation. This may or may not be addressed in the context of virtual memory and protected address spaces, for reasons discussed in chapter 4. Allocation is an important issue for parallel symbolic languages because these languages allocate and recover memory in smaller granules and at a greater rate than in traditional languages. This is particularly true of pure applicative languages which discourage the notion of state.

Parallel memory management adds additional complexity to the problems of fine-grained allocation and reclamation. The most fundamental consideration in parallel memory management is the physical organization of memory; it affects all other implementation decisions. The distinction between NUMA and non-NUMA architectures is important because on NUMA architectures memory access patterns have a significant effect on performance[3]. These concerns constrain the way that the heap is allocated, how objects are allocated within the heap, and how processors cooperate to manage the heap. In this regard, there are two relevant considerations for NUMA architectures. One is the presence of network "hot spots" [PCYL87] where excess memory contention between processors can degrade performance (non-NUMA designs also suffer from memory contention, but it is mitigated by the use of heavy *coherent* caching). A second concern with NUMA architectures is *locality*. By this we refer to a processor's memory accesses in relation to its distance from other memories and position in the interconnection network. This concern is related to the hot-spot issue, because good locality minimizes contention, but it is also a more general concern about achieving good memory access times in a system where not all memory accesses are equal. For example, on the BBN Butterfly multiprocessor there is an average 4-to-1 difference in access times between a local memory access and a network memory access, even in the absence of contention. We would like to take advantage of locality where the opportunity presents itself, while still retaining the advantages of heap sharing. In a traditional system these concerns impact the programmer; in symbolic systems these concerns impact the implementation of automatic storage allocation and reclamation schemes.

Here is a partial listing of issues which are addressed in chapter 5:

- How do we structure our heap across processors?

[3] Strictly speaking, this is true for non-NUMA designs too, but for caching and virtual memory reasons, not network design.

- How can we perform distributed storage allocation (i.e. allocation occurring in parallel, with minimal synchronization)?
- How do we balance memory allocation, so that heavy allocation does not adversely affect any one processor?
- How do we allocate memory to avoid hot spots?
- How does allocation affect process scheduling?
- Where can we take advantage of locality?
- How can we perform distributed storage reclamation (parallel garbage collection)?
- How does garbage collection affect locality?

2.2.2 Process Management

The second major area of support that a kernel provides is process management. The kernel provides system calls for process scheduling and termination, and manages processes over the course of their life-cycles. Under *eager* parallel systems, process scheduling is implicitly associated with creation; in *lazy* languages like Daisy, process scheduling is deferred until the suspension is probed (demand-driven scheduling) or explicitly scheduled by a concurrent language primitive.

The issue of *process granularity* is related to process creation; if creation is explicit in the surface language (e.g. *futures* [RHH85]), the grain size is generally larger than if process creation is implicit, such as in Daisy; in either case process granularity is finer and less stateful than in traditional languages. This finer granularity leads to increased amounts of context switching, which must be handled in an efficient way.

A related issue is the problem of controlling excessive parallelism. Once the machine is fully utilized it is counterproductive to schedule additional tasks. The additional parallelism causes higher peak memory usage and increased context switching, reducing throughput. If the language has eager semantics, one solution to the problem is to revert to in-lining the computation [Moh90], thus increasing the grain size. If the language is lazy, there may be less choice in reducing granularity, but also less parallelism to contend with. In either case, the kernel should provide a way to automatically control the growth of parallelism in the system. Similarly, parallel symbolic systems are expected to *load balance* themselves, so that all processors are kept busy. The primary issue here is whether the system migrates processes and if so, how this affects locality.

Two concerns that eventually confront a parallel language implementor are how to handle interprocess communication and synchronization. Interprocess communication for shared-memory symbolic languages is implicitly handled by heap sharing. Languages with explicit side-effects need to provide *synchronization constructs* (locks, mutexes, etc.) to arbitrate read/write access between processes for mutually shared structures; pure applicative languages like Daisy do not. However, another type of *implicit* synchronization is required even for applicative languages to handle the dependences that arise between processes. When a running process probes a suspension it sets up a dynamic *dependence* between the probing process and the suspension. The probing process cannot make any headway until its demand is satisfied. The kernel must transparently schedule the target suspension and handle the execution synchronization between the two processes (see section 1.3). Two broad strategies for this synchronization are possible: *polling* and *blocking*. The method used can have a large impact on performance; blocking synchronization, while generally considered more efficient, has inherent problems (see below) that require workarounds elsewhere in the system.

An idea that has gained momentum in symbolic languages in recent years is the concept of *speculative computation* [Bur85a, Bur85b]. Speculative parallelism refers to scheduling parallel tasks that may not be needed (on the chance that they might) to increase available parallelism over that provided by regular *conservative* parallelism [Jon87]. For example, the two arms of a conditional might be scheduled in parallel with the evaluation of the predicate so that the system will have a head start on the overall result of the conditional regardless of the outcome of the predicate.

The presence of speculative processes complicates scheduling; it exacerbates the problem of controlling parallelism and introduces the problem of of removing speculative tasks that have become useless, both from the schedule and from the heap. There is also the question of establishing the priority of conservative tasks over speculative tasks for resource allocation decisions; otherwise speculative tasks can reduce throughput for needed processes. Distinguishing between conservative and speculative tasks is complicated by heap sharing, which can lead to multiple perspectives on a task's actual need, and the fact that a task's status can change dynamically over the course of its life. Finally, systems with side-effects must also deal with the decision of whether to abort or to "roll-back" side-effects that are the result of useless speculation [Osb90].

One solution to these problem (assuming that the speculative tasks can be distinguished) is to spawn a "termination process" to kill them [GP81, Hem85]. Another approach is to implement some kind of priority mechanism that dynamically upgrades or downgrades the status of a speculative process and its descendants [Osb90]. This is usually coupled with a suitably-modified garbage collector to remove pointers to the speculative tasks from the scheduling infrastructure if they have been unreferenced from the programs data structures [BH77, Mil87].

2.2.3 Device Management

The third major area of kernel support is for device management. A number of lazy languages use a *stream* model of I/O interface [Hud89] in which data is read or written from devices in streams. This model elegantly handles the interaction with demand driven computation, which would be difficult with an imperative, temporal I/O interface, such as that used in Lisp. In DSI this support is integrated into the kernel, so that a seamless stream I/O interface is available to any language implemented on the kernel without having to implement this in the language itself.

One of the problems with implementing the stream model is that most I/O is not inherently demand-driven, even though data may be accessed that way. I/O is, in fact, *event-driven* at the lowest level of implementation, and it is the responsibility of the kernel's *device drivers* to bridge the gap between the demand-driven access and the event-driven I/O. DSI's kernel uses the interrupt mechanism of the DSI virtual machine to handle event-driven I/O using a special type of device driver process. This process is unique in that it can be scheduled by a probe to release its data (demand-driven) or scheduled by an I/O event occurring on the system (event-driven) to service interrupts in a timely fashion.

DSI's device manager allows some types of I/O, like keyboard input, to be handled at a very low level. Keyboard input is implemented as a stream of raw characters. To share this stream among multiple processes it becomes necessary to split the stream on a demand-driven basis, and we describe how this can be accomplished, along with filtering and interleaving of prompts.

Another device problem that is common to symbolic languages is that of *dangling descriptors*. This refers to the problem of an I/O device that is left in an "open" state by a process that has become unreferenced. This problem may manifest itself in slightly different forms depending on the I/O model and constructs of the surface language, but it is a common issue to all lazy language implementations as well as imperative languages that allow speculative computation. A common solution to the problem is to endow the garbage collector with the capability to "close" the device, possibly through a generic *finalization* facility [Mil87]. This problem is an artifact of an underlying imperative model of I/O that could be avoided using a functional file system [HW92].

The stream model of computing, while elegant, has its limitations. One of these is the difficulty of handling interactive terminal I/O, which must be synchronized by artificial strictness measures [HS88]. This problem may be quickly becoming irrelevant with the advent of modern GUI windowing shells. The stream interface turns out to be very adept at handling a windowing interface; DSI's approach uses a client-server model to handle character-based interaction with multiple input windows.

2.3 Daisy

Although this thesis is primarily about how suspending construction-based parallelism is managed in our DSI implementation, it is helpful to understand how this parallelism arises in the surface language. In this section we introduce *Daisy* and describe how suspending list construction gives rise to parallel programs. This description is necessarily brief; for a fuller description of the language, see *The Daisy Programming Manual* [Joh89b]. Daisy is a descendant of pure Lisp and is a contemporary of *Scheme*. It is a statically-scoped, lazy, applicative language. It provides a few atomic objects, such as numbers and *literals* (atoms). Lists glue together atomic objects and other lists to create complex structures.

2.3.1 Syntax

Daisy's syntax is a departure from standard Lisp S-expressions, yet is still relatively simple. Brackets `[ ]` are used to denote list structure, and used with `!` (like Lisp's `.`) creates dotted pairs. The default interpretation of a bare list is evaluation/list construction, not application; application is expressed by infix colon. So, for example

```
mpy:[x y]
```

means to multiply the *values* of `x` and `y`. Expressions can be quoted with the *hat* character to suppress evaluation.

Lambda forms use a syntax reminiscent of the lambda calculus:

```
\ formals . body
```

where `formals` is a single identifier or an arbitrarily bushy formal argument list, and `body` is a Daisy expression. Assignment can occur at top level only, and is specified by the form

```
identifier = value
```

So, for example,

```
add5 = \ x. add:[x 5]
```

defines the function `add5`.

Numbers are expressed in standard decimal or floating point notation. *Literals* (identifiers and strings) are expressed as an alphanumeric sequence beginning with a letter; optionally, double quotes can be used to quote literals with embedded spaces and other characters.

Special Forms

Daisy has two main local binding forms that extend the environment for the scope of the form.

```
let: [formals actuals body]
```

extends the environment by locally binding the `formals` to `actuals`, both of which may be arbitrarily bushy structures. `body` is evaluated in the extended environment. The `rec` form is similar, but creates recursive bindings. It is somewhat similar to Scheme's `letrec`, but is fully general, allowing data recursions. For example,

```
        rec: [L [0 ! map:[inc L]] L]
```

specifies the list of integers.

An Example

Here is an example of the Quicksort program, written in Daisy:

```
quick = \L.
  rec:[loop
          \[dc L]. let:[[X ! Xs] L
             if:[ nil?:L  dc
                  nil?:Xs [X ! dc]
                     let:[[lo hi] partition:[X Xs]
                           loop:[[X ! loop:[dc hi]] lo]
                         ]
                ]]
       loop: [[] L]
      ]

partition = \[p L].
  rec:[part
         \[p L lo hi]. let:[[X ! Xs] L
            if:[ nil?:L      [lo hi]
                 le?:[X p]   part:[p Xs [X ! lo] hi]
                    part:[p Xs lo [X ! hi]]
               ]]
       part: [p L [] []]
      ]
```

2.3.2 Semantics

Daisy's laziness is a byproduct of list construction, not lazy interpretation or compilation [FW76a] (there are exceptions, such as the binding forms described above).

The evaluation of a list of expressions yields a list of suspensions for evaluating those expressions; in essence, lists are suspended-process builders. Application is strict; the application of a primitive to a list applies a *demand pattern* to the list, coercing some of the suspensions in it. The demand pattern of a primitive depends on the semantics and implementation of the primitive. For example,

```
seq:[exp1  exp2  ...  expN]
```

is a *sequencer*, which coerces suspensions sequentially from left to right.

Many demand-patterns allow parallelism. For example, the `add` operator referred to earlier is strict in both its arguments, which could be evaluated in parallel. In general, this is true of all strict Daisy primitives that take lists as arguments. This concurrency is possible because Daisy lacks side-effects, a property which allows suspensions to execute in any order. Daisy does not explicitly specify whether most of its primitives actually coerce arguments in parallel, since

1. the definition of these operators does not imply concurrency, even though the opportunity exists, and

2. it leaves the choice up to the implementation. On a sequential machine they would not be coerced in parallel, but on a parallel machine they might.

Some Daisy primitives *do* imply concurrency semantics. `seq`, from above, implies sequential coercion of its argument. The only Daisy primitive that directly implies concurrency is `set`:

```
set:[exp1  exp2  ...  expN]
```

`set` returns a new list of values corrosponding to the order that the suspensions converge in its argument list. This implies that the expressions will be concurrently evaluated, at least until the first one to converge. If the tenth element of the result is requested, one can assume that the expressions will be concurrently evaluated at least until ten of them have converged, and so forth.

Many Daisy primitives have *potential* concurrency. `add` is an example of *conservative parallelism* [Jon87]; Daisy has many more opportunites for *speculative parallelism* [Jon87]. For example, a speculative *OR*-parallelism can be specified by

```
any?:[exp1 exp2 ... expN]
```

which returns `true` if any of its arguments are non-*nil*. Similarly, a speculative *AND*-parallelism is

```
all?:[exp1 exp2 ... expN]
```

which returns *nil* if any of its arguments are *nil*.

Even `if` is potentially (speculatively) parallel:

```
if:[pred1 then1 pred2 then2 ... else]
```

`if` can speculatively schedule all its *then*, *else*, and subsequent (all but the first) *predicate* arguments, and evaluate the first predicate sequentially.

There is also abundant potential parallelism in all mapping primitives.

```
map:function
```

is an example of a *higher-order* primitive. It returns a closure that maps *function* over a list. That mapping, of course, can be entirely parallel.

Many other potential sources of concurrency exist in Daisy; it has vast amounts of *latent parallelism*. This section illustrates how the language affords abundant implicit parallelism. The DSI kernel is charged with managing this concurrency so that it does not overwhelm the machine. Daisy primitives can freely schedule their arguments without worrying about resource management issues. Chapter 6 explains how the DSI kernel throttles parallelism.

2.3.3 Annotation

As we have seen, Daisy has large amounts of latent parallelism. Nevertheless, there are cases where it cannot be exploited due to the limitations of laziness and demand-driven computation. Chapter 8 explains how these situations can occur.

For this reason, Daisy also includes parallelism annotations. Recall from chapter 1 that we are not philosophically opposed to parallel annotation, but rather the inclusion of side effects that limit opportunities for *implicit* parallelism.

CHAPTER THREE

Architecture of the DSI Machine

This chapter describes the design of DSI's virtual machine, the lowest level component of the DSI hierarchy (see figure 4).

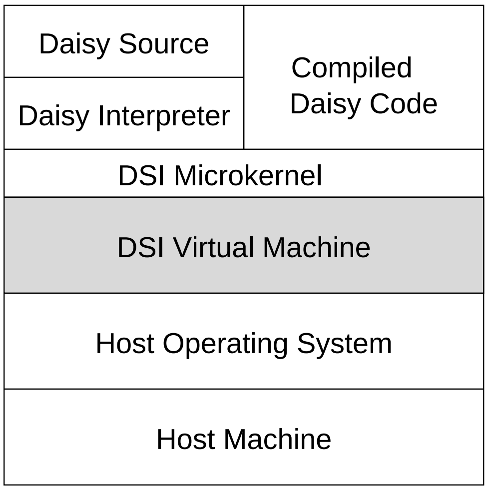


Figure 4: DSI Virtual Machine

DSI's virtual machine provides a low-level abstraction of the physical machine architecture (memory interface, processor state, instruction set, interrupt mechanism and processor interconnection topology) that forms a portable foundation upon which the rest of DSI and Daisy is implemented. The virtual machine approach provides several significant benefits:

- It provides a modular way to port DSI to new target architectures. The rest of DSI is implemented on top of the virtual machine, and Daisy on top of DSI, so the virtual machine isolates most of the machine-specific implementation dependences. For reasonably similar architectures, only the virtual machine layer needs to be ported. Substantially different architectures may require parts of the microkernel to be rewritten for performance optimization, however.

- It allows critical or missing functionality to be implemented as virtual instructions which can be mapped as efficiently as possible to the host architecture. This helps identify key areas in which hardware support could be helpful. For example, cell allocation is such a frequent occurrence during applicative language execution that it is supported by instructions in the virtual machine. This indicates that hardware support for allocation would be extremely helpful for efficient execution of applicative languages. We summarize these areas in section 3.6.

- It documents the minimum hardware capabilities and functionality required to implement the system, and delineates (potential) hardware components from software. This provides a sort of high-level blueprint for constructing a machine for a native, self-hosted DSI environment or for implementing DSI directly on a target machine, bypassing the host operating system.

3.1 Memory Interface

DSI's virtual machine specifies an architecture oriented toward list processing, and its memory interface reflects this orientation. The elemental addressable unit of memory is the *cell*, a 64-bit word that conforms to one of several fixed formats. Cells are classified at the storage management level by a particular configuration of data and pointer fields. Figure 5 shows three storage classifications for cells.

- *Unary* cells contain thirty-two bits of raw data and a reference. The data field can be accessed and interpreted in one of several formats, including signed integer, single-precision floating point or packed bytes.

- *Binary* cells have fields *head* and *tail*, each containing a reference.

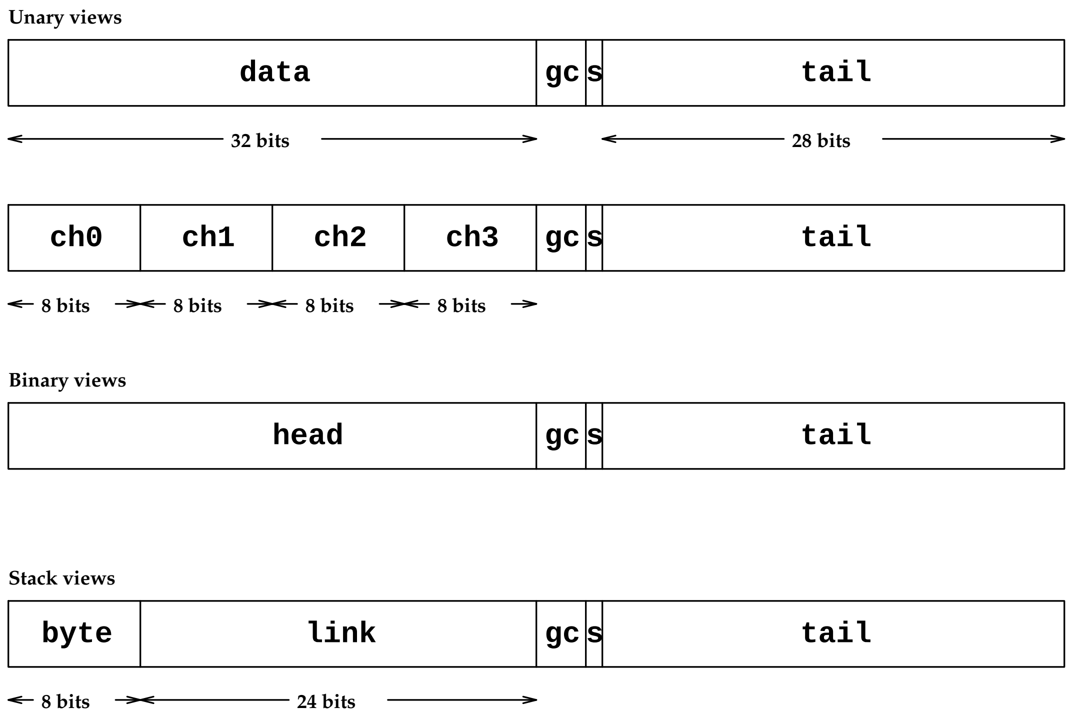


Figure 5: Cell Storage Types

- *Stack* cells contain a byte of data, a pointer and a reference.

Cells have additional small bit fields used for various purposes such as garbage collection and processor synchronization. These fields are described where applicable. The memory interface is facilitated by processor registers that conform to one or more of these cell views and machine instructions for accessing or allocating cells in those views.

3.1.1 Tags and Types

All scalar data types and compound data structures are represented by cells or linked graphs of cells. *Tagged pointers* are used to accelerate suspension detection and run-time type checking [Joh90, SH91] . We will hereafter refer to a tagged pointer as a *reference* or *citation*; a raw or untagged pointer is simply a *pointer*. A reference contains three bits of tag information and twenty-four bits of pointer information.

A reference's tag implies something about the storage class of the cell to which it is pointing, and vice versa, but tags and storage classes are different entities; the distinction is clarified in chapter 5. Briefly, a tag is a *logical* typing mechanism; it says that the object to which the reference points is a *list*, a *number*, etc. These objects have underlying representations using one or more cells conforming to a particular storage class; e.g. a *list* reference points to a *binary* cell. We may refer to a cell's *type* as either the cell's storage class or its logical (reference tag) type. In the former case the types are *unary*, *binary* and *stack*; in the latter case the types are *list*, *number*, and so forth.

3.2 Processor State

Figure 6 depicts the virtual machine state for a single processor node. The register file consists of a set of *context windows*. Each context window holds a single process (suspension) context. The context window pointer (*CWP*) always points to the currently running process. Control is transferred from one resident process to another by directly or indirectly changing the *CWP*. This method of context switching allows several processes to be multiplexed rapidly on the same node, and provides a mechanism for interrupt handling. The number of context windows (and consequently the maximum number of simultaneously register-resident processes) is implementation dependent[1] Many current RISC-type architectures implement some form of register windows (e.g. the *Sparc* architecture) or have a large register set with renaming capability, both of which are capable of implementing our context window model. Context windows can also be implemented in memory, using register-based pointer indirection for the *CWP*.

Each context contains four fixed storage class cell registers: *DMC*, of type *unary*, *STK*, of type *stack*, *ENV*, of type *binary* and *VAL*, type *unary*. These registers are further subdivided into smaller addressable fields reflecting the cell views as described above. Registers C, X, F, A and R are references, B is a byte, L is a pointer and D and K are 32-bit raw data words. A context window's state is identical to that of a suspension's, and the kernel swaps suspensions in and out of the context windows as necessary using the `ldctx` and `stctx` instructions.

[1] The current DSI microkernel requires at least four.

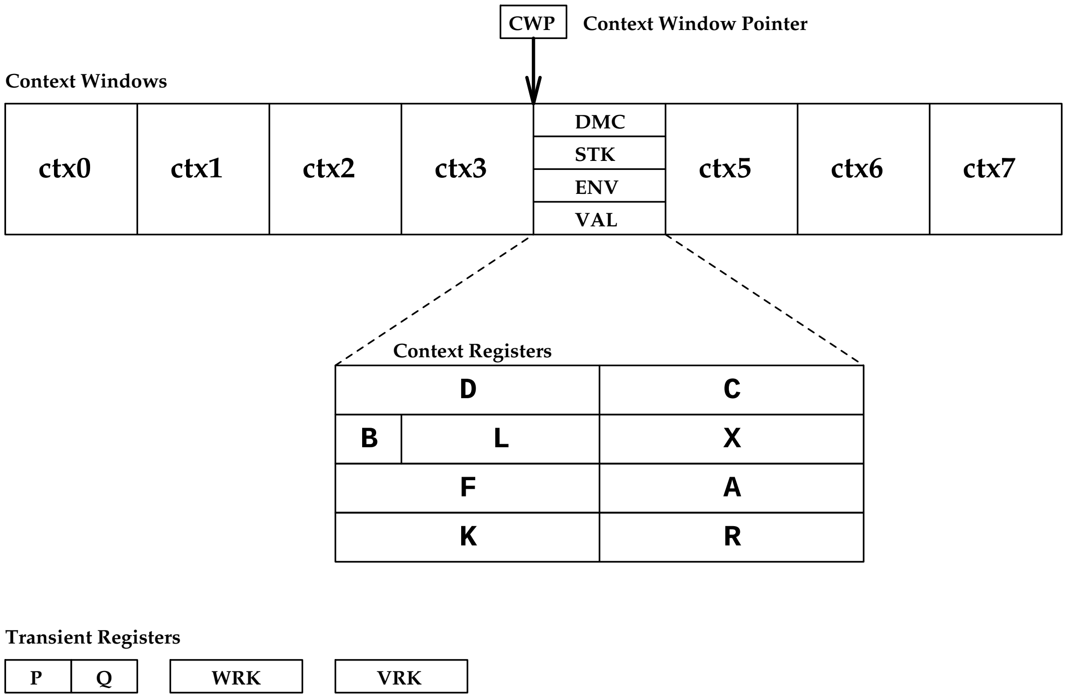


Figure 6: A Processor Node's Register State

Suspending construction provides the basis for the design of DSI's process structure. Suspension creation and coercion is the implicit behavior of any code that builds, traverses, or otherwise manipulates data structures. Process granularity is much finer than in conventional languages; accordingly, suspension manipulation is a critical performance issue. This is reflected in DSI's suspension design. Its small, fixed-size state (32 bytes) can be quickly allocated, initialized, loaded and stored. Conditional probes (e.g. `hdc`, `tlc`) provide implicit detection and dispatch. Because these and other suspension manipulation operations (e.g. `suspend`, `ldctx`, `stctx`) are implemented as DSI instructions, they can be optimized at the lowest level of implementation for best performance on the host architecture.

In addition to the context windows, there are several global *transient* registers that are accessible from any context on the node. These are used as fixed operands by certain instructions and are also used as general-purpose registers for passing data between processes in different context windows. Registers P and Q are references.

WRK and *VRK* are *untyped* cell registers; they can be accessed in any storage format via the appropriate field name (e.g. `WRK.ch1` or `WRK.hd`). A process can only access the registers in its own context window and the transient registers; the kernel is the only process that has global access to all windows.

The memory subsystem has some private registers that are described in chapter 5. There are also some registers devoted to signal handling that are described below in section 3.4.

3.3 Instruction Set

DSI employs a load/store architecture that is similar in many respects to general-purpose RISC microprocessors. Its instruction set differs primarily in that it is very list (cell) oriented with dedicated instructions for allocation. Instructions operate on registers or immediate constants. There are explicit instructions for reading from and writing to memory in the cell format. Instructions fall into the following general categories:

- *Load* instructions read cells and cell fields from memory into registers. We use the terms *fetch* and *load* interchangeably. There are two kinds of loads: *conditional* loads generate a trap if a suspension reference would be returned to the destination operand, *unconditional* loads do not. Conditional loads are the basis of suspension transparency and efficient detection, supported right down to the machine level[2].

- *Store* instructions write register contents to memory. Like loads, there are conditional and unconditional versions of stores. The conditional versions will fail on any attempt to overwrite a non-suspension reference; this feature can be used to arbitrate interprocessor suspension scheduling (see [FW78d]), although DSI has more generalized primitives (e.g. atomic add/and/or) that can be used for more sophisticated coordination schemes.

- *Allocation* instructions allocate cells from memory and initialize them using register contents in a single operation. This could be considered a special type of

[2]At least for detection; suspension creation is still an explicit operation at this level.

store operation. Data creation (i.e. cell allocation) is such a frequent occurrence in applicative systems that it warrants special consideration at the machine instruction level. This is discussed in more detail in section 3.5.1 and chapter 5.

- *ALU* instructions provide simple arithmetic and logical operations on registers. These instructions are similar to those found on any traditional microprocessor.
- *Signal* handling instructions are used to control and manipulate the hardware exception state.

3.4 Signals

DSI exceptions and interrupts are handled under a common *signal* mechanism. Signals may be generated by hardware interrupts (e.g. timers, devices), instruction *traps* (e.g. a conditional load) or explicitly by the `signal` instruction. Some common uses of signals are:

- Conditional loads whose result is a suspension reference.
- Allocation instructions which invoke garbage collection.
- I/O activity associated with a device.
- Timer interrupts for context switching.
- Bad references or ALU errors.
- Tracing and machine-level debugging.
- Interprocessor synchronization.

There are 32 possible signals, enumerated and prioritized from lowest (0) to highest (31). Depending on its use, a signal can be sent to a single processor or multicast to all processors. An example of the former case is a localized conditional-load trap occurring on a processor/memory board; an example of the latter is a barrier synchronization signal such as used in garbage collection. The signals are assigned by the kernel and are listed in Table 1 in chapter 4.

3.4 Signals

DSI uses a process-oriented approach to signal handling, in contrast to the stack-oriented approach used in most imperative systems. The latter usually handle exceptions by pushing state on the currently executing process stack (or a system stack) and invoking an exception-handling procedure. Under DSI, when a signal is delivered to a processor node it causes a context switch via a change of the context-window pointer (`CWP`) to an alternative resident handler *process*. The handler might be the kernel or another process (subsystem) designated to handle that signal. The currently executing process is suspended by the context switch, but its state is not side-effected. Neither is the state of the handler process; that is, no continuation is forced upon the handler, it simply resumes at the point where it had left off previously. Thus, the only action of a signal is to context switch to a handler process, which must be designed as an iterative signal handling loop. DSI instructions are designed to be restarted in the event of a signal preemption[3], so the preempted process can be restarted (if necessary) by changing the `CWP` back to the appropriate context window. However, since process state is not side-effected by signals (and assuming none in the surface language), it is not necessary to maintain a stack or other structure to recover from the effects of some permutation of delivered signals. All that is required is a scheduling algorithm that results in the the proper processes being rescheduled, and this is provided by DSI's demand-driven scheduling[4]. This approach to exception handling fits well with the applicative framework that underpins the rest of DSI.

3.4.1 Signal Routing

Figure 7 shows DSI's signal handling architecture, which works like a state machine.

The signal lookup table maps incoming signals (on- or off-node) with the current value of the `CWP` to determine the new value of the `CWP`. This allows the same signal to be mapped to different handlers depending on which context window is active. Arbitration between multiple pending signals is handled by the fixed priorities of the individual signals; the highest numbered unblocked signal has priority. The kernel takes this into account when assigning signals to services.

Delivery of signals is controlled by a pair of 32-bit registers: `SIGLATCH` and

[3]Actually, in the current implementation preemption is allowed only on certain instructions, because transient registers are not saved across context switches.

[4]See chapter 6 for clarification.

SIGMASK. Each bit in SIGLATCH corresponds to one of the 32 signals; when a signal is delivered to the node it sets the corresponding bit. SIGMASK is used to block action on signals registered in SIGLATCH. If a bit is clear in SIGMASK it inhibits the processing of the corresponding signal in SIGLATCH; i.e. the signal is effectively blocked. This capability can be used to mask interrupts for critical sections, to implement polled device drivers, etc. The `signal`, `tstsig`, `setsigmask` and `clrsigmask` instructions are used for manipulating the bits in these registers.

3.4.2 Per-process Signals

The preceeding discussion is concerned with signal handling at the system level, on a *per-node* basis. Signals are a system phenomenon and transitions are specified in terms of context windows, not individual suspensions. It is the kernel's job to micro-manage the context windows to map the signals between the individual processes. In order to handle the signal mappings of thousands of suspensions the kernel has to handle some signals itself and swap suspensions between the finite set of context windows and the heap as necessary. Currently, this is done only for a few signals with fixed, implicit mappings, such as the the conditional load and timer signals. However, it should be possible to generalize this mechanism, and by overloading one or more unused system signals, to carry a variable number of *user-defined* "signals" on a *per-suspension* basis. This capability would provide the foundation for user-extensible signaling and exception handling. Assuming that a usable construct could be formulated for Daisy, such a mechanism could be used for error recovery, event-driven scheduling, port-based message passing and other systems-level applications.

It is unclear at present whether and how demand-driven system scheduling would conflict with event-driven user scheduling. The incorporation of this kind of primitive would add another source of non-determinism (besides multisets) into the system, but given the underlying implementation of signals it is likely to be "safe" non-determinism. This is an interesting area that deserves further consideration.

3.5 Interconnection Topology

DSI assumes a shared memory, MIMD architecture. Aside from that, the virtual machine does not specify a particular processor interconnection topology, primarily because that aspect of the architecture does not figure as prominently in the *programming* of the machine given those assumptions. However, the host platform must provide a certain core functionality to successfully implement DSI. This includes:

1. A globally-shared and addressable heap memory. DSI's reduction and communication models rely heavily on shared data structures. Those models are unlikely to be viably implemented on anything other than a physical shared memory machine (i.e. non-message passing). The shared memory *can* be organized as a *NUMA* architecture (see section 2.2.1), however. NUMA architectures require special considerations for locality that are not isolated within the virtual machine. These locality issues affect the process (chapter 6) and memory management (chapter 5) subsystems, and are discussed in those chapters.

2. *Symmetric MIMD multiprocessing.* DSI assumes that all processors have the same basic configuration and functionality, and are not organized hierarchically. The distributed algorithms used assume a logical enumeration of processor nodes, and a given node must be able to determine its index.

3. Efficient implementation of atomic 32-bit *add*, *or*, *load* and *store* operations. Atomic loads and stores are necessary to insure the integrity of references in the heap. The logical, bitwise atomic operations are used for interprocessor synchronization and updates on shared data structures.

4. The ability to interrupt any processor in a "timely" fashion. DSI's signals can be piggybacked on a single user-programmable host exception, provided the overhead of native exception handling is not significantly greater than DSI's own signal handling model implies. A polling approach can also be used, if necessary.

Any parallel architecture that shares these characteristics is a suitable candidate for hosting DSI. This type of architecture (MIMD shared-memory symmetric multiprocessor) is common in emerging multiprocessors.

3.5.1 LiMP

Although the virtual machine does not specify a particular network design, special architectures have been proposed. In a series of papers, Johnson ([Joh81, Joh85, Joh89c]) refines a buffered routing network as the basis for a DSI multiprocessor. Wise describes a layout scheme for implementing such a network in [Wis81]. The network described is a *Banyan* design (see figure 8) connecting a set of *List Processing Elements* (LPEs) to a corresponding set of *List Storage Elements* (LSE's). LPE's have roughly the same architecture described for DSI processors in section 3.2 above, but lack multiple context windows and signals. LSE's are intelligent memory banks, participating actively with the network in supporting the LPE's. Johnson [Joh81] classifies memory access instructions into fetch (*RSVP*), store (*Sting*) and allocation (*NEW*) requests, each having a distinct network transaction profile between LPE's and LSE's. *RSVPs* require a round-trip message from LPE to LSE and back to furnish the result. *Stings* are one-way requests requiring no acknowledgment. *NEWs* are handled in unit time by a cell routing arrangement called a *NEW-sink*, in which LSE's combine with the network switches to route the addresses of free list cells back towards the LPE's. Switches are endowed with buffers for the addresses; whenever a switch transfers an address out of its holding area to its outputs it requests a new one from its inputs. The switches are biased to transfer addresses directly across the network, but will transfer from the alternate input if demand is high. The idea is to preserve locality under normal allocation rates, minimizing contention at the switches for loads and stores, but to disperse high allocation rates across the machine so as to minimize hot spots. Johnson presents simulation results of LiMP in [Joh81].

The fundamental motivation behind the NEW-sink is to move resource management problems from software into hardware. If the goal of the NEW-sink is to handle data distribution then the goal of the *Process-sink* [Joh89c] is to manage load balancing. The Process-sink is a complementary operation in which LPE's route free suspension blocks through the network to LSE's. If LPE's only multi-task suspensions in their "local" space, then idle LPE's will be able to provide more free suspension blocks to the system than busy LPE's, effectively migrating new processes to idle nodes and balancing the load over the system. The behaviors of the NEW-sink and Process-sink are modeled in DSI by a load-based allocation strategy described in chapter 5.

3.5.2 The BBN Butterfly Architecture

DSI's has been implemented on a *BBN Butterfly* multiprocessor, selected because its shared-memory (NUMA) network architecture is similar to the *LiMP* architecture described above. The *Butterfly* differs from *LiMP* in that it uses a circuit-switched (*non-buffered*) enhanced banyan network. Each processing node contains a processor, a memory bank and network interface called the PNC (Processor Node Controller). Processors are tightly-coupled to their local memory bank, but all banks are accessible through the switch, as though the graph in figure 8 was folded laterally into a cylinder. Each processor actually has two paths for memory access: to its local memory over a local bus, or through the switch to any other node's memory bank. Each PNC arbitrates access to its memory bank between the processor and the network. Experiments indicate that *in the absence of contention* switch access to memory is about four times as slow as local access; with contention the switch access memory latency degrades even further. This leads to special considerations for memory management, which we generalize to all NUMA architectures.

3.6 Summary

Although DSI's instruction set is oriented towards list processing it otherwise provides much of the same functionality as a stock microprocessor, most of which are adequate for emulating a DSI virtual machine. It is encouraging that stock hardware performance is improving so rapidly and carrying with it the performance of symbolic languages. Nevertheless, we could do much better, because there are a number of areas in which relatively minor hardware enhancements would reap great rewards in the execution efficiency of these languages, and narrow the performance gap with traditional languages. The key areas in which DSI would benefit from hardware support are context windows, allocation instructions and DSI-style signals, especially signals for trapping on reference tags, which would benefit dynamic type checking and suspension detection. Wise has identified a number of other features that would be useful for hardware-assisted heap management, such as support for reference counts [Wis85a, DW94].

3.6 Summary

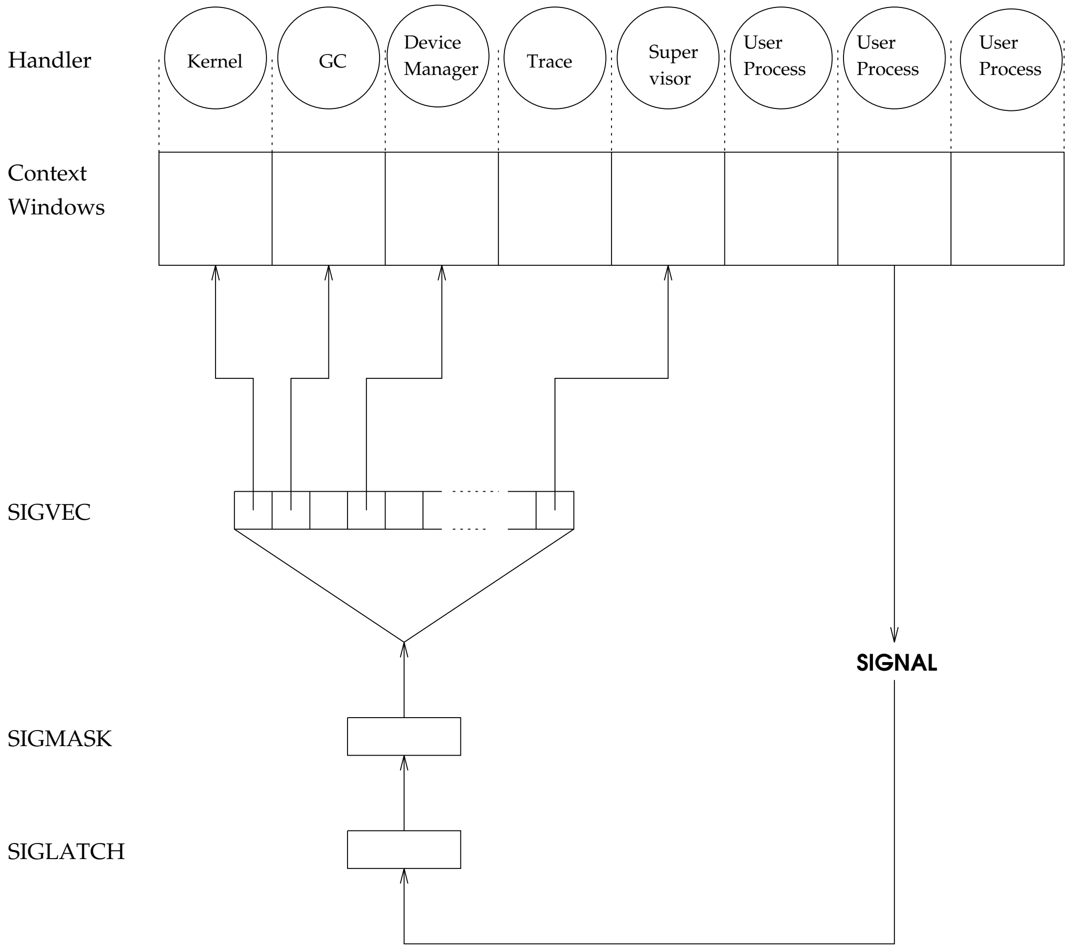


Figure 7: Signal Handling

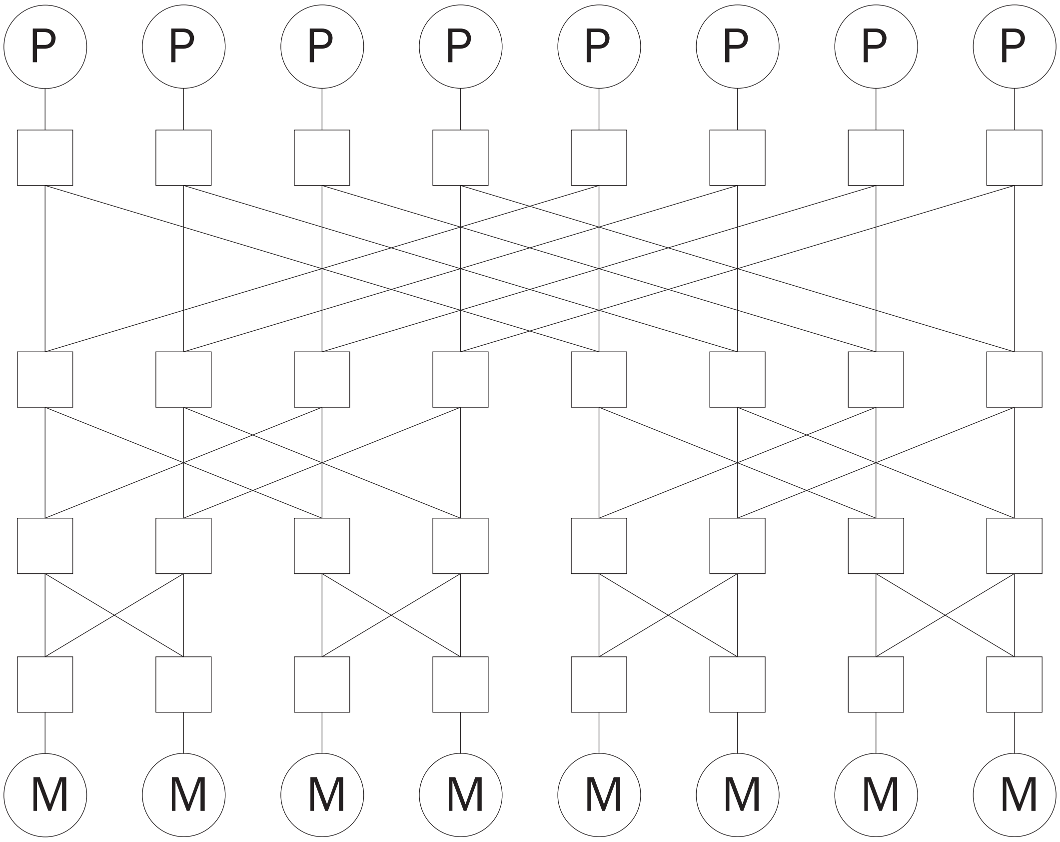


Figure 8: A *banyan* network

CHAPTER FOUR

The DSI Kernel

DSI refers not only to our virtual hardware architecture, but also to a *kernel* written for that architecture (see figure 4, p. 31). In this chapter we describe the role of the kernel and its overall organization and operation.

4.1 Resource Management in Symbolic Programs

DSI's kernel provides much of the same core functionality as an operating system. Its main purpose is to provide run-time resource management for programs running on the DSI architecture. We define *resource management* as:

- *Memory management*, in the form of distributed storage allocation and reclamation.
- *Process management*, in the form of distributed, demand-driven scheduling of processes, with system calls for parallel task scheduling. The kernel handles interprocess synchronization, excess parallelism throttling, system load balancing, and support for speculative parallelism.
- *Device management*, in the form of event-driven device scheduling, a character-stream I/O interface and garbage collection of open devices.

DSI's kernel differs substantially from conventional operating system and parallel language kernels due to its particular orientation toward fine-grained symbolic processing. The differences in processing requirements is reflected in the emphasis or deemphasis of certain features. For example, a finer process grain size leads to greatly increased emphasis on the efficiency of process manipulation. Suspension grain size is *much* smaller than in conventional operating systems; smaller than kernel-supported *threads* in conventional processes, and even smaller than that of many symbolic languages such as *Multilisp*, where grain size is specified by the programmer. Process manipulation overhead (creation, scheduling, synchronization, context switching) is therefore significantly more critical to system performance than in conventional systems, where it is typically a fraction of overall computation effort. This is reflected in DSI's use of context windows and demand driven scheduling traps in the virtual machine.

Another major difference is in virtual memory support, or lack thereof. Studies of the interaction between virtual memory and heap-based, symbolic languages [Moo84] reveal that the large memory requirements of symbolic languages plus the non-locality of heap allocation and garbage collection do not mesh well with conventional virtual memory systems which depend heavily on spatial locality. The development of *generational* garbage collection has reduced the problem somewhat [PRWM92], but the issue is still a topic of active research. Lazy languages such as Daisy may have even worse virtual-memory locality than applicative-order languages such as Scheme, which can make heavy use of stack frames. In any case, the task granularity of symbolic processes is much too fine to associate with virtual memory table flushes on every context switch.

Another reason for a deemphasis on virtual memory is that it is often used in conventional systems to support protected address spaces. In symbolic processing the dominant communication paradigm is shared memory; i.e. shared memory is the norm, rather than the exception. There has been some work in using *shared* virtual memory to support type-checking, memory barriers and other features needed by symbolic languages (e.g. [AL91]). This approach exploits the trapping aspect of memory management units on stock hardware for purposes that might also be handled by appropriate processor modifications.

Finally, symbolic languages do not have visible pointers as traditional languages do. Memory protection is accomplished at the language level by the simple fact that if you do not have a reference to an object you cannot modify it, even in systems with side-effects. Of more concern is the issue of *conflicting global name-spaces*. Most systems have some notion of a global namespace for identifiers. On systems with simple shallow binding or similar schemes, this leads to shared identifier references; on a concurrent system this is not always what is actually desired, especially for concurrent users. This problem might be addressed at the language level with a suitably designed *module* or *package* system, as opposed to the use of protected address spaces.

4.2 Kernel Structure

We distinguish between the design of the kernel's resource-management algorithms and the design of its structure and interface. DSI's design in the former sense is the subject of the following three chapters. The remainder of this chapter discusses the latter; the structure, interface and operation of DSI's kernel.

4.2.1 Monolithic vs. Micro Kernel

In traditional operating systems such as *Unix*, the majority of system resource management is encapsulated by a set of system services provided by a monolithic kernel. The kernel is implemented as a protected layer (or layers) of code underlying every process. System services are accessed by exceptions or special "function" calls, both of which are handled on the user or system stacks. These calls put the calling process in *supervisor* or *kernel* mode, which allows the called kernel routines to execute privileged operations on behalf of that process or other processes in the system. In other words, the kernel is a special mode (and code) that is executed by all processes and executes as a part of their respective address spaces.

Many modern operating systems are structured so that most system services are handled *outside* of the kernel by special user-mode processes. The resulting stripped-down *microkernel* provides only the absolute core supervisor-mode functionality required to manage the machine resources: processor scheduling, memory management, and interrupt handling. The microkernel itself is implemented as a separate process to which ordinary processes make requests for services, not as a special code layer in processes' own address spaces. This provides certain advantages over the traditional kernel implementation model, such as increased robustness, distributability, and concurrency in the operating system itself. A microkernel design results in a modular decomposition of operating system functionality into a few separate processes that operate interdependently to manage the system as a whole. This modularity greatly simplifies understanding, extending and debugging system-level code.

Many symbolic language kernels share aspects of the monolithic design; they may not be nearly as large and they may use very different resource management techniques, but they still implement the kernel as a low-level code layer interfaced through the stack of every process. In contrast, DSI borrows the microkernel design philosophy in the structure of its own kernel (we will hereafter not distinguish between the terms *kernel* and *microkernel* when referring to DSI's kernel). DSI's kernel is implemented as a set of special processes, not as layers of code accessed through a stack interface. Interfacing with the kernel is a kind of interprocess communication. This is not nearly as expensive as in conventional *IPC*, since

1. interprocess communication in DSI is through shared memory;
2. DSI's processes are very lightweight;
3. DSI's context window architecture allows multiple processes to be register resident.

These factors blur the distinction between traditional notions of interprocess communication and, say, shared-memory coroutining. However, from a programming perspective our approach offers the modularity and loose coupling of the former with the efficiency of the latter. The kernel interface is discussed in more detail below.

4.2.2 Distributed vs. Non-Distributed

Another issue in parallel kernel design is whether a single kernel controls all the processors in a sort of *master-slave* relationship, or whether the kernel is distributed and runs on each processor.

Master-slave arrangements are common in systems that do not use symmetric multiprocessing; i.e. in a processor arrangement in which one processor controls the others. In this case the "master" processor might run the kernel and distribute work to the slave processors. The advantage of this kind of design is that no locks or special synchronizations are needed for the kernel, since only one processor is running it. In symmetric multiprocessing systems all processors have the same functionality and capabilities. This kind of system encourages a distributed kernel design where all processors have kernel functionality, an approach that requires more care in determining how the processors interact, but pays off in greater parallelism in the kernel itself.

Although this issue is somewhat orthogonal to the issue of macro vs. microkernel, the two issues impact one another. If a distributed kernel is combined with a monolithic kernel design, the processors may need to use shared locks and other measures to arbitrate access to shared kernel structures. A microkernel design allows the kernels to use interprocess communication to inter-operate with each other, resulting in a more loosely-coupled design that is easier to scale and results in less bottlenecks. Many parallel symbolic processing kernels use a distributed design. The macrokernel approach of most of them is reflected in in the use of locks to control access to global allocation pointers, shared schedules, and other shared kernel structures. DSI uses a distributed design in which the kernel processes are distributed across all processor nodes. There are no shared kernel structures; the only synchronization required is to the queues of each processor's message area. The kernels communicate with each other through these queues to manage the machine as a whole.

4.2.3 Kernel Organization

The DSI kernel consists of four main processes. The *supervisor* handles process scheduling, context swapping, and miscellaneous other kernel duties. The *garbage collector* handles storage reclamation. The *device manager* handles device I/O, and the *tracer* handles machine level tracing and debugging. At system startup, each processor constructs this same configuration of processes.

4.3 The Kernel Interface

Kernel services are accessed in one of two ways: *traps* and *message requests*. Both types of communication are implemented with signals. Traps are associated with high-priority hardware interrupts or instruction exceptions. Examples of traps include device I/O signals, conditional load traps, garbage collection, etc. Message requests correspond to traditional "system call" type services such as scheduling and allocation requests. The term *message request* is *not* meant to imply copying data between processors, as is done in *message passing* systems; rather it refers to the loosely coupled, asynchronous communication that is required by our microkernel design.

4.3.1 Message Requests

Kernel "messages" are simply pointers to data (cells) in the heap that are passed to the kernel to act upon. Each message contains an indication of the type of request and some optional data. Common requests (e.g. scheduling requests) are encoded in the reference tag of the message itself; others are encoded in the head of the message. The data (tail) includes any further data being passed to the kernel. This may include a pointer to a location where data is to be returned to the sender; a sort of call-by-reference mechanism[1]. For example, this is how allocation requests are handled across processors (see chapter 5).

Message communication is handled with streams, a natural choice for communication in DSI. There are two parts to a message request: appending the message to the appropriate stream and (optionally) signaling the processor that a message of the appropriate priority has been sent. The signaling part may not be used for messages that are routinely handled, such as allocation requests. This approach is used both for local and remote kernel requests.

In normal stream communication under DSI there may be many readers but only one writer[2]; the only synchronization required is an atomic store operation, which is provided by the virtual machine. With message requests, there are multiple writers and only one reader; namely, the kernel handling the requests. Thus, some synchronization is required to arbitrate access between processors to the tail of the communication streams for the purposes of appending messages. Note that since message streams are distributed over the processors (each processor has several message streams), synchronization efficiency is not overly critical, and simple spin locks will suffice to arbitrate access among writers.

[1] Note that we are not talking about a side-effect visible in Daisy.

[2] Due to a locality scheduling constraint that is explained in chapters 5 and 6.

Implementation

The kernel's message streams are implemented as a set of queues. The queue pointers are stored as a contiguous vector of cells in the static data area of each processor's *heap segment*[3]. The vector resides at the same offset in each heap segment; a particular queue can be accessed by adding the offset and queue number to the heap segment base pointer for the target processor.

Each queue of a processor's set of message queues is associated with a signal (see table 1). After appending a message, the sender signals the processor with the signal associated with that queue. The priority of the signals assigned to queues establishes the priority of the queues themselves; messages in a higher priority queue will always be serviced before messages in the next lower priority queue, and so forth. This provides a way to prioritize various types of requests; see chapter 6 for an example of how this is used.

4.3.2 Traps

A special method is available for passing data between the kernel and a running process *on the same node*. The transient registers are globally accessible from any context window and allow small fixed amounts of data to be passed from resident process to resident process much faster than allocating storage in the heap. This method is used for implementing the conditional probe traps and other frequent kernel trap interruptions. Using these optimizations, a round-trip system trap in DSI is faster than issuing a message request, since no allocation operations are performed. The catch is that this technique can only be used with very high priority signals or with careful masking, because data in the transient registers is volatile. If a trap occurs when a higher priority signal is pending[4] then the transient registers may be overwritten before the kernel gets around to reading them. Message requests are relegated to the lower priority signals so as not to disrupt trap handling.

[3]See chapter 5.

[4]If possible, signals are delivered to a processor using the actual exception mechanism of the host (if this is too expensive simple polling is used). However, because of DSI's transient register design they can only be delivered in DSI code at certain virtual instruction boundaries. Therefore, they may get delivered in batches; see chapter 3 for further details.

4.3.3 Interruption

We have described how signals are used to prioritize the kernel's response; as discussed in chapter 3, DSI employs a number of optimizations to accelerate these transactions. Context switching speed between regular and kernel processes must be as efficient as possible to make this interface viable. The use of context windows allows both regular and kernel processes to be simultaneously register-resident[5]. The kernel maps signals to the context windows to allow direct transitions between programs and the handler processes; e.g. tracing signals directly invoke the tracer, etc. Signals have fixed priorities, which naturally provides a mechanism for arbitrating among them (this can be overridden by masking). The kernel assigns symbolic names to the values and exports the names (using the assembler module system) so that the actual values can be changed transparently without disrupting other modules. Table 1 lists some of the signals and their respective handlers used in the current implementation.

4.4 Summary

The development of DSI's kernel and virtual machine was motivated by the desire to support suspending construction at target levels; i.e. hardware and low-level software. Thus it is oriented toward applicative languages based on list processing, and using a fine-grained, largely demand-driven concurrency model. Nevertheless, we have attempted to draw clear boundaries between the language (chapter 8), the virtual hardware architecture (chapter 3) and the kernel (chapters 4–7). DSI's kernel is independent of the Daisy language and could easily support other applicative languages (see section 9.2). In this regard, the DSI kernel can be compared to other generalized parallel symbolic processing kernels such as *STING* [JP92b, JP92a], which was built around parallel Scheme, and *Chare* [KS88], which was motivated by concurrent Prolog. The DSI kernel differs from these systems, among other things, in that it is generally targeting a lower level of implementation. It also provides device management for lazy languages and uses different resource management algorithms.

[5] In some cases context windows cannot be fully mapped to registers in the host and are partially implemented in memory; this still speeds context switching by avoiding intermediate structure manipulation and suspension swapping.

4.4 Summary

Signal Name	Type	Description	Handler
SIG_EXIT	sw	Exit signal.	Supervisor
SIG_ABORT	sw	Abort signal.	Tracer
SIG_TRACE	sw	Tracing.	Tracer
SIG_PROBERR	hw	Invalid probe.	Supervisor
SIG_GC	sw	Garbage collection.	Garbage Collector
SIG_GC_DUMP	sw	Dump heap.	Garbage Collector
SIG_HDC	hw	Conditional head.	Supervisor
SIG_TLC	hw	Conditional tail.	Supervisor
SIG_RESET	hw/sw	Boot/reboot.	Supervisor
SIG_SYNC	sw	Synchronize nodes.	Supervisor
SIG_TIMER	hw	Interval timer.	Supervisor
SIG_IO	hw	An input event.	Device Manager
SIG_IOBLOCK	hw	I/O blocked.	Supervisor
SIG_DETACH	sw	Detach request.	Supervisor
SIG_QUEUE8	sw	Message in queue 8.	Supervisor
SIG_QUEUE7	sw	Message in queue 7.	Supervisor
SIG_QUEUE6	sw	Message in queue 6.	Supervisor
SIG_QUEUE5	sw	Message in queue 5.	Supervisor
SIG_QUEUE4	sw	Message in queue 4.	Supervisor
SIG_QUEUE3	sw	Message in queue 3.	Supervisor
SIG_QUEUE2	sw	Message in queue 2.	Supervisor
SIG_QUEUE1	sw	Message in queue 1.	Supervisor

Table 1: DSI Signals and Handlers.

CHAPTER FIVE

Memory Management

DSI's kernel provides complete memory management for processes. In this chapter we discuss the organization of DSI's memory space and the techniques used for distributed storage allocation and reclamation.

5.1 Memory Organization

DSI memory requirements are divided into three categories:

- a large, shared *heap* of untyped cells;
- a small, per-processor private data area;
- the DSI implementation code in host executable format.

If the host architecture supports *distributed* shared-memory (e.g. the *BBN Butterfly*), the heap is allocated as identically-sized segments distributed across all nodes, as shown in figure 9. These individual *heap segments* are mapped into the logical address space of all processors as one large contiguous heap. The DSI implementation code is replicated across processors for maximum instruction locality. If only non-NUMA shared memory is supported, a single, large heap is allocated in shared memory, and is artificially divided up into equal segments. Thus, regardless of the underlying shared memory implementation, the heap as a whole is globally shared, and each processor is assigned responsibility for a particular segment of the heap. Figure 10 shows the view from one processor.

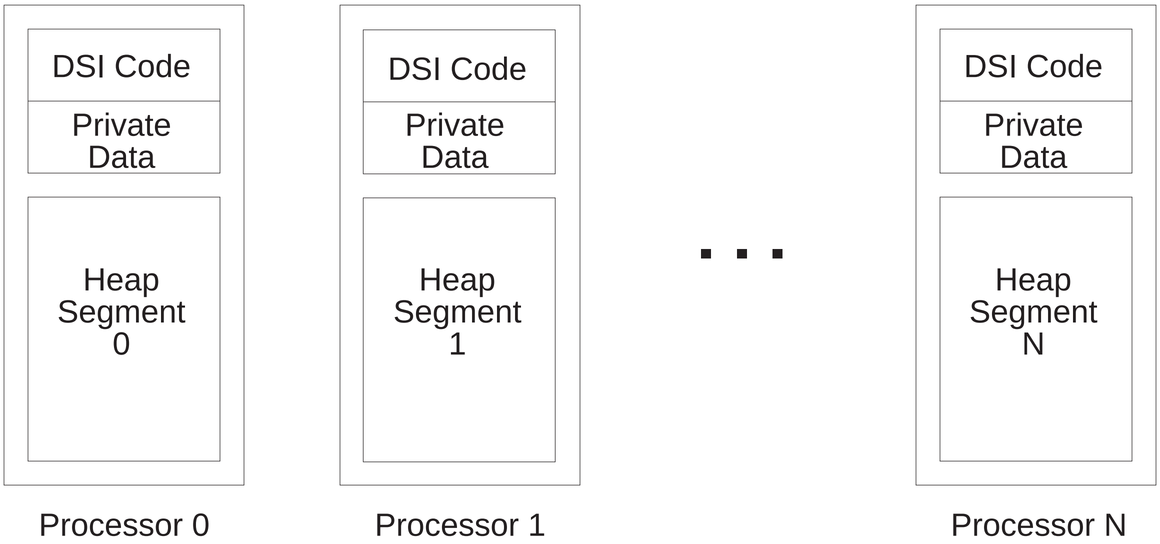


Figure 9: DSI Memory: *Physical Layout*

5.2 Storage Allocation

DSI uses a three-level distributed storage allocation scheme. Each level builds upon the level below it to provide its functionality.

At the lowest level, a processor manages its own heap segment, and only allocates from its own area. Each segment is organized as shown in figure 11. Ordinary cells are allocated in blocks from the low end of available space (`MEMLO`) upward, while suspensions are allocated in blocks from the high end of the segment (`MEMHI`) downward. The registers `AVLLST` and `AVLSPN` track the respective allocation regions; when they meet, available space is exhausted (for that node) and garbage collection must be performed to recover space. The reason for the bidirectional allocation has to do with garbage collection; its compaction algorithm is designed to relocate objects of the same size. This will become clearer in section 5.3; for now, suffice it to say that if DSI had a more sophisticated storage reclamation scheme this would not be necessary.

The second level handles the distributed aspect of allocation. Each heap segment maintains two *allocation vectors* stored in the *Static Data* area shown in figure 11. The allocation vectors contain pointers to reserved blocks of free space on other processors.

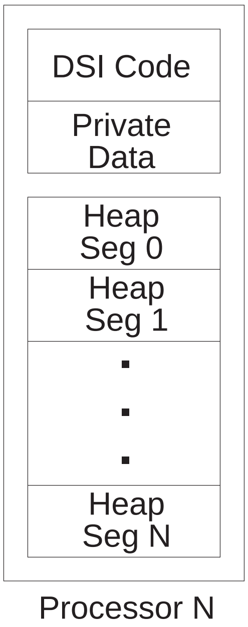


Figure 10: DSI Memory: *Logical Layout*

One vector is for free cell space, the other is for free suspension space[1]. Each vector is organized as a region of contiguous unary cells, indexed by logical processor number. Each element's tail points to a block of free space on the associated processor and the `data` field contains the size of the block (in cells). Figure 12 depicts a cell allocation vector.

The allocation vectors provide a general mechanism by which data can be allocated on any particular processor in the system. The technique of allocating larger blocks of free space and then suballocating out of it improves efficiency by reducing the frequency of communication costs between processors [MRSL88]. The free blocks pointed to by the allocation vector are *preallocated* and *reserved*; there are no mutual exclusion problems to contend with to arbitrate access to the blocks. The kernel can explicitly allocate objects on remote processors by manipulating the allocation vector directly, but most processes will typically use the allocation instructions `new` and `suspend` instead, the operation of which are described below.

[1] Again, with a different compaction scheme we would need only one vector.

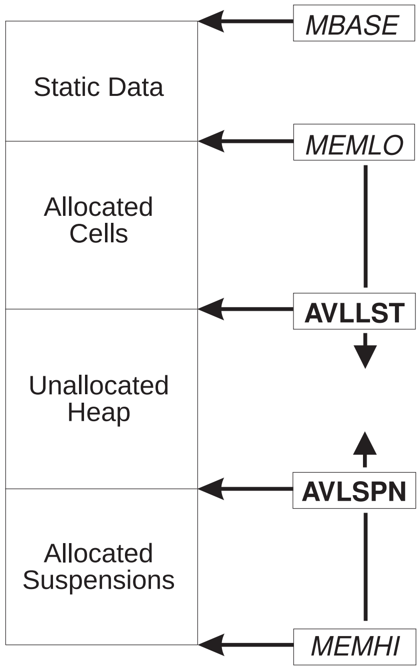


Figure 11: Heap Segment Organization

When a processor exhausts a block of free space in its allocation vector it sends a request to the remote processor in question to supply another free block. The requesting processor will be unable to allocate objects of that type (cells or suspensions) *on that particular remote processor* until it receives another free block from that processor. An *allocation server* on each processor handles these requests from other processors for blocks of free space. A request contains a pointer to the appropriate cell in the allocation vector of the requesting processor. The allocation server will allocate a block of cells or suspensions locally from its own `AVLLST` or `AVLSPN` (respectively) and store the location and size of the block into the remote allocation vector. This request/donate allocation mechanism stands in contrast to the shared heap state variables used in other systems [DAKM89, MRSL88]. Although the allocation block/suballocation technique can be (and is) used in these systems, processors must occasionally synchronize for access to shared global allocation pointers.

Two obvious factors affecting the availability of free space using our technique are

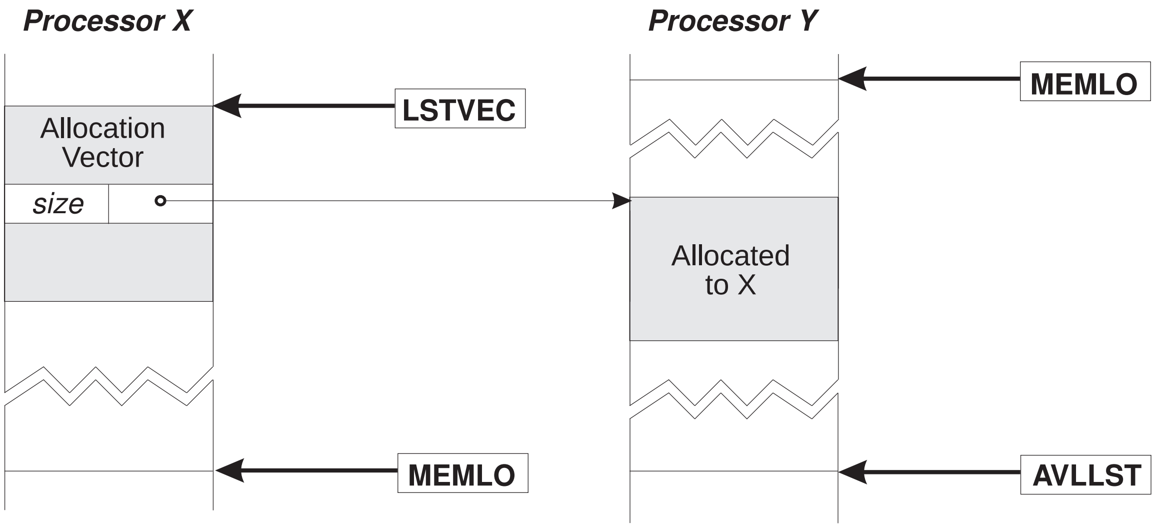


Figure 12: Cell Allocation Vector

the size of the free blocks exchanged and how responsive the allocation server is to requests. The larger the block requested, the more data can be remotely allocated before running out of space and the less often processors will have to exchange requests. However, allocating larger blocks to other nodes will result in a processor using up its own free space faster then if it doled it out in smaller chunks. Since garbage collection is a global phenomenon, the interval between collections is set by the first processor to run out of space; this would argue in favor of smaller blocks. As we shall see, there are other reasons that a processor should limit the amount of local space it is allocating to other processors. We will return to this topic in section 5.2.1.

If the server can't supply a block it simply ignores the request; in this situation it is very close to being out of space itself. The allocation vectors are not garbage collected; this is necessary so that any partially filled allocation blocks may be compacted and heap fragmentation eliminated. After a collection there is a flurry of allocation activity as processors request and resupply each other's allocation vectors.

The third level of storage allocation buffers cells from the allocation vector for use by the allocation *instructions* (`new` and `suspend`). The kernel culls free cell and suspension addresses from the allocation vector into two internal buffers. The `new` and `suspend` instructions implicitly access these buffers to obtain their allocation references. This culling action is activated by a signal, which is generated by the allocation instructions when the buffers begin to run low. The kernel scans the allocation vector, culling free references and building up the `new` and `suspend` buffers. When a free block for a particular processor is exhausted, the kernel sends a request to the allocation server on that processor for another free block, as described above.

5.2.1 Data Distribution

Under DSI's allocation scheme the data distribution pattern (across processors) generated by a series of allocations is determined by two main factors:

1. the culling (distribution) pattern used by the kernel (and thus `new` and `suspend`), *and*

2. the responsiveness of the allocation *servers* to requests from other processors for free allocation blocks.

The default distribution algorithm used in the Butterfly implementation culls free addresses in a round-robin fashion from the free blocks for each processor, skipping those for which the free blocks are empty. The rationale for this is that in the absence of any other useful metric we should distribute data as evenly as possible to prevent network hot spots [PCYL87, LD88, MRSL88]. The other logical benefit is simply to balance allocation across processors.

The responsiveness of the allocation servers is deliberately underplayed. The server is not signal-activated (event-driven) like other system processes. In fact, it runs at a very low priority compared to all other processes on the node.The effect is that the server on a given node runs with a frequency *inversely proportional* to the processing load on that processor. If the load is light, it will run frequently and quickly refresh the free blocks on other nodes; if the load is heavy, it will run infrequently or not at all, and supply few or no blocks to other nodes.

The end result of these design choices is that new allocations are distributed more readily to lightly loaded nodes in the system, because those processors are able to supply more free space in a timely manner. Recall the discussion of the *new-sink* in chapter 3, which effectively accomplishes the same result using a hypothetically constructed processor interconnection network (*LiMP*). Our load-sensitive allocation strategy models that behavior in software.

The success of this approach depends on maintaining the connection between processor load and allocation; i.e. allocation must be rendered *load-sensitive*. There are two significant factors at work here: proper scheduling of the allocation server (it should run less frequently when the load is heavier) and the size of the allocation blocks used. Using too large of a free block will render the allocation less sensitive to loading, because processors can suffer longer periods of allocation server latency before their allocation blocks are used up. Small-to-moderate block sizes seem best suited for this purpose. Smaller block sizes also increase the interval between garbage collections, as explained in section 5.2. Further experimentation is necessary to determine the precise effect of block sizes on system performance.

5.2.2 Load Balancing

The interaction between allocation servers and kernels described above results in a dynamic system that distributes allocation evenly under balanced load conditions and favors lightly used processors under imbalanced load conditions. This is particularly significant in regard to the allocation of new suspensions, because what this really implies is the distribution of new processes over processors; i.e. *load balancing*.

Our load-balancing allocation scheme is predicated on two factors:

1. The fine-grained nature of suspending construction coupled with an applicative environment forces data and processes to be recycled frequently.

2. A scheduling invariant in which suspensions can only execute *locally*; i.e. on the processor in which heap segment they are located.

Applicative languages enforce a stateless mode of programming in which new data and processes are constantly created. Most processes are short-lived; those that aren't are usually blocked awaiting the results of new processes. This is important because it lets our allocation strategy shift work to new processors as they become less busy. If our system consisted of relatively static processes then an unbalanced system would remain unbalanced, since work is not being reflected in the form of new processes. Without invariant 2, our load-sensitive allocation strategy would also be meaningless, since processors would be disregarding the load-based partitioning of suspensions. The garbage collector does not migrate storage between processors, so a given suspension lives out its existence on one processor. Since new suspensions tend to be created on lightly-loaded nodes, computation overall continually gravitates towards an equilibrium.

Our system of load-balancing stands in contrast to most other symbolic processing systems which actively move processes around to try to balance the system, or have processors *steal* tasks from other processor's ready queues[RHH85, DAKM89]. Our approach to load balancing has low overhead compared to these approaches:

- Process state does not have to be relocated (process migration techniques might not relocate either, but this adds cost for nonlocal context switching due to NUMA (see section 3.5) or cache locality.
- Processors do not have to spend time examining each other's queues, notifying each other of task availability, or handle intermediate routing of tasks to other processors.

Considerations for Load Balancing

DSI's load-balancing strategy doesn't distinguish between "important" suspensions and "trivial" ones. Under our allocation scheme, trivial suspensions are just as likely to be allocated remotely as suspensions representing significant computations, which we are more interested in offloading to other processors. This is not a consideration if the surface language is explicitly parallel, since the programmer will be indicating the parallelism of computations using annotations (or e.g., futures). It is possible under our allocation scheme to allocate on specific processors (provided a block is available) so the language implementor has the option of offloading important tasks should the language be able to distinguish them.

A potential weakness concerning the load-balancing aspect of our allocation scheme, is that under suspending construction, the time between suspension creation and coercion (scheduling) is potentially unrelated (this is the basis of the model). This would appear to undermine the load-sensitive strategy for allocating free suspension blocks, since it is when the suspension is *scheduled* that impacts the load and not when it is created[2]. Consider a situation in which a few idle processors are supplying blocks of free suspension space to other processors, which are creating significant suspended computations, but not coercing them. Then at some future point this group of significant suspensions is clustered on a few processors. It's worth pointing out that this anomalous condition would only arise in an unbalanced state, which our strategy is designed to prevent. Nevertheless, assuming that the system did get into this state, how is the situation handled? Here the small grain size of suspending construction may prove beneficial, since it tends to break up significant computations into many smaller processes which will be distributed.

5.2.3 Locality

Heap-based symbolic languages are notoriously non-local. Heap objects tend to be distributed randomly throughout the heap and are also extensively shared. Lazy languages compound the problem, because the control flow is not as amenable to locality optimizations for data allocation (e.g. stack allocation) and heap sharing is even more pronounced.

DSI's heap allocation strategy acknowledges this and attempts to make the best of the situation by dispersing data evenly over the machine. However, for data that is *not* going to be shared DSI makes an effort to maximize locality. The Butterfly implementation, for example, uses locality wherever feasible:

- The `push` operation allocates stack cells from the local heap segment, so stacks accesses are local and stack activity does not involve network traffic.
- Suspensions are scheduled for execution on the nodes on which they reside, so that context switching is localized to the node; entire process contexts are not transmitted through the network. This policy also serves the load-balancing strategy outlined above.
- Garbage collection does not move data across processor boundaries. This preserves existing locality conditions.

[2]Along these lines, our model bears consideration for eager parallel systems (e.g. using futures), which schedule processes as they are created.

- Compiled code is replicated across all nodes so that instruction stream access is localized.

The goal of these optimizations is to maximize locality, going through the processor network only when a nonlocal heap access is necessary. Tagged references help here as well, allowing typing of objects without the penalty of a network memory reference to examine a field within the object.

Finally, we should say a word about systems that have little or no sense of locality. On a uniform access shared memory system (e.g. a bus-based, cache-coherent multiprocessor) there is no sense of locality as regards *the heap*[3]. Cells and suspensions might as well be allocated anywhere; the access time is (supposedly) uniform. Nevertheless, our distributed allocation system provides benefits even in this environment. First, it provides locality for the allocation *instructions*. Once a series of addresses has been collected and buffered the individual processors are not competing to update shared allocation variables. Second, the system provides processor load balancing as described above if processors respect the (artificial) "ownership" of suspensions in other heap segments. There is an implicit trade-off here: on such systems it may make more sense not to enforce such artificial boundaries and instead allow processors to execute any suspension, anywhere. This would cut down on the amount of interprocessor communication required to transfer execution requests to other processors, but would also require a new method of load balancing to be devised. Task stealing [RHH85] might be appropriate in this case.

5.3 Storage Reclamation

Like many other heap-based symbolic processing systems, DSI uses garbage collection as its primary method of storage reclamation. DSI does employ ancillary methods, such as code optimization that recycles cells in non-sharing situations, but at best these methods simply increase the interval between garbage collections. Under normal circumstances allocation operations will eventually exhaust all avaliable free space in the heap and garbage collection must be performed to recover space.

[3] On the other hand, the *snoopy cache* architecture used by these systems may be just as sensitive to locality as a NUMA design [DAKM89].

5.3.1 Garbage Collection

DSI uses a distributed mark-sweep garbage collection algorithm. Garbage collection is triggered by a `SIG_GC` signal delivered to all processors. The kernel process *on each node* responds to this signal by invoking a local dormant garbage collection process. Garbage collection suspends all other processing activity for its duration; when the collection process terminates, the kernel resumes the system where it left off.

Figure 13 illustrates the major phases of garbage collection. All phases operate in parallel on each node; each phase is prefaced by a *synchronization barrier* to ensure that all processors are executing in the same phase.

Initialization

During the *initialization* phase the collector swaps out all active processes from the context windows to their respective suspensions so that all valid references in registers will be collected.

The Mark Phase

During the *mark* phase the collector process performs a pointer-reversal, depth-first traverse of the live data reachable from each node's root. Three bits in each cell are reserved for garbage collection (the `gc` field in figure 5): one for interprocessor synchronization and two for coloring. If, during the mark phase, two or more processors reach the same cell through different references, the synchronization bit determines which one will proceed into the cell and which one will back up and terminate that thread of its traverse.

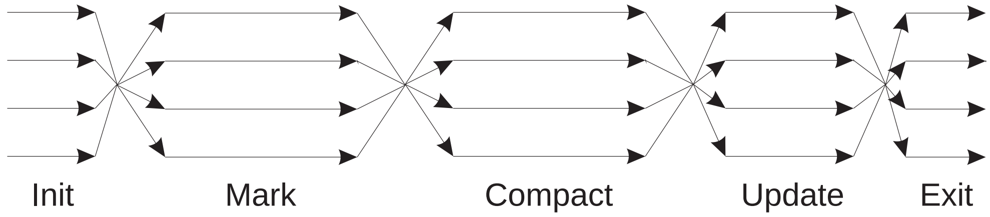


Figure 13: Garbage Collection Execution

5.3 Storage Reclamation

The mark phase leaves the `gc` coloring bits set according to the configuration of live collectible pointers found in the cell. At the end of the mark phase all cells are marked in one of the configurations shown in table 2.

The Compaction Phase

The *compaction* phase uses a "two-finger" algorithm to sweep through the heap segment, defragmenting and freeing up space for future allocation. The algorithm works as follows: Pointer A starts at the low end of allocatable memory (`MEMLO` in figure 11) and scans upward for unmarked cells, stopping when it finds one. Pointer B starts at the upper end of the allocation area (`AVLLST`) and scans downward for marked cells, stopping when it finds one. The cell pointed to by B is copied to the slot pointed to by A, thus freeing up a cell at the upper end of the allocation space. A *forwarding address* is placed in the *head* of the former B location noting the relocation address of the cell, and a special flag is left in the *tail* indicating a relocation. This process iterates until pointers A and B meet.

This compaction scheme relies on the fact that cells are of a uniform size. Although suspensions are fundamentally made up of cells, suspension cells must remain as an ordered, contiguous group. This restriction, coupled with our choice of compaction algorithm, accounts for the segregated allocation scheme discussed earlier in which suspensions are allocated at the upper end of the heap segment and cells at the lower end. This upper (suspension) area must be defragmented as well, so a similar compaction is performed at the high end of the heap segment between `AVLSPN` and `MEMHI` (see figure 11). After compaction each heap segment has been restored to the state pictured in figure 11, where free space is concentrated in the middle of the

Value	Configuration
`0x0`	Unmarked cell (garbage).
`0x1`	Unary cell.
`0x2`	Stack cell.
`0x3`	Binary cell.

Table 2: Possible `gc` Bit Configurations

segment.

Each processor compacts its own heap segment in parallel with the others. Unlike the mark phase, each processor's locality is perfect since the compaction does not require the processor to reference anything outside of its own heap segment.

The Update Phase

The *update* phase involves a second scan through the *compressed* live data in which the mark bits are cleared and any pointers to relocated cells or suspensions are updated from their forwarding addresses. The latter is handled by loading the tail of the cell pointed to by each valid reference encountered during the scan and checking it against the relocation pointer-flag. If the cell was relocated it is necessary to load the head of the cell to obtain the relocation address and then update the pointer in question. The `gc` bit configuration indicates which fields contain valid references during the scan. As with compaction, all heap segments are updated in parallel. Some switch contention is possible, because the relocation tests may require loads into other heap segments, but otherwise locality is good.

Cleaning Up

During the *exit* phase processors synchronize again to ensure that collection has finished system-wide, then reload the context windows with the active process set. Finally, the collector process relinquishes control back to the process that was interrupted in allocation.

5.3.2 Minor Phases

There are two minor phases devoted to collecting the system hash table and device list as a special entities.

DSI must provide a way to remove identifiers that are not referenced outside of the hash table and recover the space. Most systems facing this problem use *weak pointers* or similar mechanisms [Mil87]. DSI takes a slightly different approach. The hash table is maintained as a global entity that is not reachable from any garbage collection root. During collection, the normal mark phase marks any identifiers that are reachable from live data. A minor phase between the mark and compaction phases traverses the hash table and removes all unmarked entries and marks the table structure itself, so that it can be successfully scanned during the update phase.

A second minor phase devoted to collecting the device list is discussed in chapter 7. These two minor phases correspond to the *hash demon* and *files demon* described in [Mil87].

5.3.3 Garbage Collection: Observations

Our garbage collection algorithm is neither efficient or full-featured compared to others described in the literature (e.g. [AAL88]), nevertheless we are encouraged by recent studies on the efficiency of mark-sweep collection [Zor90, HJBS91]. Our collector has some useful properties that contribute to the overall storage management strategy in DSI, the most important being that storage is not migrated from one processor to another, preserving locality conditions that are the basis for a number of design decisions (see sections 5.2.2 and 5.2.3).

Our garbage collector's cost is proportional to the sum cost of the mark, compaction and update phases. Although our collector requires three main passes as opposed to one pass used in copy-collect, the heap segments are not divided into semi-spaces, and good locality is ensured for the compaction and update phases, which may have major caching benefits. Since the mark phase is fully parallelized, but allows individual processors to traverse the global heap, it is difficult to assess the total cost of marking. Optimally, it is proportional to

$$\textit{reachable cells}/\textit{number of processors} \tag{1}$$

although in practice it is unlikely to turn out that way, depending on the amount of live data reachable from each processor's root and contention in the network. The compaction phase sweeps the entire *allocatable* portion of a processor's heap segment (marked and unmarked), and so is proportional to that size. The update phase sweeps the static area and *compressed* portion of a processor's heap segment.

5.4 Summary

We have described a system of storage allocation and reclamation for a cell-oriented, distributed heap. Where feasible, the system allocates cells locally. When data must be allocated globally, the system resorts to a load-sensitive, distributed allocation scheme. The garbage collector does not migrate data or processes from one node to another, which helps maintain the locality of processes and data allocated locally. Our approach differs from related systems which attempt to improve locality by copying structure between nodes during garbage collection. Our system relies on the short temporal nature of data and processes to allow load-sensitive allocation to gravitate the system to a balanced state. The interaction between load-based allocation and process management is explored further in chapter 6.

CHAPTER SIX

Process Management

Like memory management, DSI provides complete process management. We begin this chapter with a description of the key events in a suspension's life-cycle: process creation, activation and termination. This provides a basis for a discussion of higher level process management issues: synchronization, scheduling, controlling parallelism, and speculative computation.

6.1 The Process Life Cycle

Traditionally, process creation is associated with both the allocation of process state and the manipulation of scheduling structures within the kernel. In DSI, process creation and activation are treated as two fundamentally separate events. Process creation is just another form of allocation that is nearly as fine-grained as cell-allocation (suspensions are, in fact, a sequence of four cells).

All suspensions are created with the `suspend` instruction, which initializes suspension records in the heap. The created suspension inherits some of its fields from the registers of the context window of the currently running process; the new suspension record can be swapped directly in and out of a context window to instantiate the process. Most suspensions require a *convergence context* to be useful: a cell in the heap containing the suspension reference that is the location for storing the manifest result of computing the suspended computation. A process executing a typical suspending construction code sequence will execute one or more `suspend` instructions interspersed with `new` instructions in building a data structure.

A suspension is activated in one of two ways:

- It can be *coerced* by a *probe*; in which case the system will schedule it implicitly, or
- it can be explicitly scheduled by a suspension-aware concurrency primitive in the language.

The first situation occurs when a running process traverses a structure containing suspensions. By default, structure accesses are ultimately handled at the DSI virtual machine level by conditional load instructions (e.g. `hdc` and `tlc`). These instructions are sensitive to suspension references (see chapter 3), generating a special signal if the result of the load would be a suspension reference. This signal causes a trap to the kernel, which suspends the probing suspension and schedules the target suspension for execution. The second situation occurs only for a special class of primitives which explicitly manipulate suspensions. This typically includes concurrency constructs in the surface language.

A scheduled suspension is considered an *active* process. The kernel resumes an active suspension by loading it into an open context window and transferring control to that window. The suspension resumes execution at the control point in the `K` register. A suspension in this state is a *running* process. Note that there may be any number of active processes on a given node, some of which are swapped into context windows. There is only one running process per node, namely, the one in a context window that has control of the processor.

The end result of a suspension's execution is almost invariably to compute a value which replaces its own reference in the heap, an operation known as *converging* to a value. If the suspension is not referenced from another location this causes the suspension to *un-reference* itself; its state will be recovered at the next garbage collection. Converging is normally accomplished by a *sting* (store) operation followed by a `converge` system call, which signals a process' intention to terminate. The kernel responds by swapping the suspension out of its context window and selecting another active suspension to run.

Sometimes, a suspension may store a value that *extends* the structure of the convergence context, and embeds itself farther down in the structure. For example, the code for an iterating *stream* process might append items to the tail of a list and then resuspend itself at the new end of the stream. This resuspend behavior is accomplished with the `detach` operation, a system call which is similar to `converge`, but has the effect of resetting the control register `K` such that if and when the suspension is reactivated it resumes execution at the specified control point. Thus, `detach` is essentially a call to relinquish the processor.

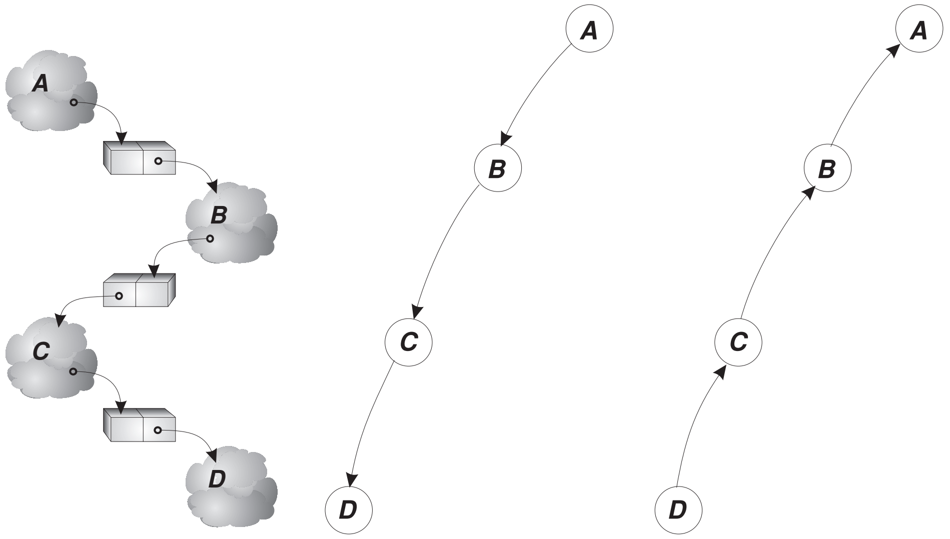
6.2 Interprocess Synchronization

We illustrate interprocess synchronization in DSI with an example. Consider the situation depicted in figure 14a. Suspension A is activated as a running process; during the course of its computation it probes suspension B. A dynamic *dependence* is established between A and B; process A needs suspension B's result. This causes suspension B to be activated, in accordance with demand-driven scheduling. When process B converges or detaches with a result, A's demand is satisfied and it can be restarted. Clearly, we need some method of *synchronizing* execution between A and B, since A cannot make progress until B converges with a result. Our suspending construction model requires that this synchronization be performed transparently by the system on behalf of processes.

How we implement this dependence synchronization can have a dramatic effect on performance. Our choices fall into two broad categories: *polling* and *blocking* synchronization. The difference between these two methods is primarily in where the responsibility lies for rescheduling A. In a polling implementation, the system either periodically wakes up A to reattempt its probe, or checks B's convergence cell on behalf of A at some interval, rescheduling A when a manifest value has appeared. With blocking synchronization, A is blocked indefinitely (removed from the schedule) and the act of rescheduling A is associated with B's convergence.

Both synchronization methods have their strengths and weaknesses:

1. Polling does not require any external information about the dependences between processes to be maintained, since that information is encoded in the heap sharing relationships and in the code of the various processes themselves. Blocking synchronization requires that dependences be dynamically recorded and associated with each process so that blocked suspensions can be explicitly rescheduled when their dependences are satisfied.



(a) Actual Dependences (b) Dependence Graph (c) Dependence Stack

Figure 14: Process Dependences

2. Polling does not require any special action or communication when a process converges; eventually the polling mechanism will succeed in restarting A. Blocking synchronization requires that the suspensions blocked on a process be explicitly rescheduled when that process converges.

3. For polling, the waiting interval is critical, since the act of polling wastes processor time that could be devoted to other useful processes (we do not want to busy-wait). The interval between polls should be spent executing any other ready processes on the node. In some cases it may be necessary for the processor to artificially extend its polling interval with idle periods, since flagrant busy waiting could have deleterious effects on the processor network [LD88, PCYL87]. This might be the case, for example, early on in the computation before the program has established many parallel tasks.

4. *Synchronization latency* refers to the lag between the time that a suspension has converged and the time that a process depending on it is actually resumed. This can occur under both synchronization methods, but is only a concern for polling.

If the polling processor is idle during the latency period (see item 3, above) then that time has been wasted. The concern here is for the *aggregate* time wasted over the course of a program's execution due to synchronization latency. Blocking synchronization does not suffer this problem because blocked tasks are immediately rescheduled, so an idle processor would take it up immediately.

Given 1 and 2, we would prefer to poll, provided we could select an ideal polling interval (see item 3, above) for each blocked process, such that negligible time and resources are spent on the actual polling. Unfortunately, that is a difficult problem, since it all depends on the length and termination of individual process' computations. The wide range of possible values makes even a heuristic choice difficult (even using a dynamically changing interval). Our experience has shown that explicit synchronization is very effective and worth the extra overhead involved in maintaining dependences. However, polling's advantage in (1) is important and we shall return to the synchronization issue later in section 6.7.

6.3 Tracking Dependences

Let us return for a moment to our example. Suppose that in the course of execution, process B probes suspension C; C probes D, and so on. The result is that a dynamic *chain* of dependences is established between suspensions. This dependence chain can be visualized as a *dependence graph* as shown in figure 14b. The dependence graph shows the temporal data dependences between a subset of suspensions in the heap at a given point in time.

Because of the lazy nature of suspending construction, dependence chains can grow hundreds or thousands of processes deep. Assuming that we are going to do explicit synchronization, we need to track these dependences with an actual structure that mirrors, but reverses, the dynamically unfolding chain of dependences. We depict our *reverse dependence graph* like a normal dependence graph, only with the arrows reversed, as in figure 14c. Using our dynamically constructed reverse dependence graph we can easily determine which processes should be resumed when a given process converges. The main problem with this approach is maintaining an accurate mapping between our reverse dependence graph (an actual structure that the system manipulates) with the dynamically changing state of actual dependences in the system. As we shall see, this assumption can be problematic, especially in the presence of speculative computation (section 6.7).

6.3.1 Distributed Demand-Driven Scheduling

Our first logical choice for implementing a reverse dependence graph is a *stack*, with elements pointing to suspensions in the dependence chain. The top of the stack points to the currently executing process. Every probe that results in a suspension forces a stack push, so that the *top* of the stack always points at the *fringe* of the expanding computation, and backward links record the dependences between suspensions resulting from a series of probes. When the current process converges this *dependence stack* is popped, and the suspension underneath is rescheduled.

Our dependence stack then, is a layer of scheduling infrastructure laid out over the computation as it unfolds. Probes result in pushes, and convergence results in pops; we assume that the process at the top of the stack is the next ready process in that dependence stack.

Dependence stacks are the basic scheduling handles used by the system. When a probe results in a suspension the kernel pushes the dependence stack (as described above) and then examines the suspension reference. If the suspension is local it can proceed to swap it in and execute it. If it is a *remote* suspension, the kernel sends a scheduling message to the remote processor containing a pointer to the dependence stack, and moves on to examine messages in its own scheduling queues (see section 4.3). When the remote processor gets around to processing this message, it loads the suspension reference from the top of the stack and proceeds to swap it in and execute it. If *that* suspension probes yet another suspension the same scenario recurs. When a process converges, the kernel pops the dependence stack and checks the suspension reference at the top. If it is a local reference, the suspension is swapped in and executed. If it is nonlocal, the dependence stack is sent to the remote processor and the current processor looks for other stacks in its message queues.

What we have just described is the basic mechanism for distributed demand-driven scheduling in DSI. The fringe of the computation corresponding to a particular dependence chain shifts from processor to processor to take advantage of locality of execution for whichever suspension is currently pushing the fringe. The cost of this scheduling communication is discussed in chapter 4; it is considerably less than the cost of context swapping over the network, which would involve the loading and storing of suspension contexts, perhaps multiple times, plus any additional overhead due to non-local stack manipulation, etc. For details refer to section 5.2.3.

6.4 Creating Parallelism

As we have seen, demand-driven scheduling leads to extending and collapsing *existing* dependence chains; at any given time only the suspension at the top of a dependence stack is active, the others are blocked on dependences. If there is only one dependence chain then only one processor would be busy at a time, executing the suspension at the top of the sole dependence stack. Thus, any parallelism in the system is due to *multiple* dependence chains (and correspondingly, dependence stacks). Each dependence stack represents a point on the fringe of the computation tree that is not currently dependent on any other process.

It might seem logical to create and assign a dependence stack to each processor; if each processor is busy handling a stack the entire machine is utilized. Actually, we want *multiple* pending stacks for each processor. The reason is due to our requirement that suspensions execute locally; if a processor passes its current dependence stack to another node we do not want it to be idle. Rather, it should take up any dependence stacks that have been passed to it from other processors. Given a balanced distribution of suspensions and a small backlog of waiting dependence stacks, any processor should always have work to do. Kranz [DAKM89] briefly discusses the merits this kind of task backlog.

6.5 Static vs. Dynamic Parallelism

We can classify parallelism in our system under two broad categories. *Static* parallelism is due to pre-existing multiple chains of demand. An example of static parallelism would be multiple initial output devices, arising from a system of multiple output windows or a multi-user/multi-terminal system. One advantage of static parallelism is that there is no possibility of overloading the machine with too much parallelism; the amount of parallelism never increases, and can be initially matched to the size and capabilities of the machine.

Dynamic parallelism is created in response to explicit scheduling requests by concurrent primitives. These requests are the source of *branching* in the dependence graph. Each branch forms a new dependence chain; the amount and type of branching is determined by the primitive, and possibly by the number of suspensions in its argument. For example, consider a hypothetical primitive, `pcoerce`, that implements a form of *parallel coercion*, scheduling each suspension in its argument list and returning when they have all converged.

`pcoerce:[`*exp1 exp1* `...` *expN*`]`

Concurrent primitives are the primary source of parallelism in Daisy. This is consistent with the underlying philosophy of the language; namely, that list construction produces suspended processes, and primitives coerce those processes is various ways (including in parallel). See section 2.3 for details.

6.6 Controlling Parallelism

Dynamic parallelism introduces the problem of controlling excessive parallelism. Once the machine is *saturated* with processes (i.e. is fully utilized, with a sufficient backlog) additional parallelism is not beneficial, and may even be harmful. Too many concurrent tasks result in greater peak memory usage then if the processes were coerced sequentially. Furthermore, additional processes result in more context switching, reducing overall throughput. Thus some method of constraining parallelism is required so as not to overwhelm the machine with active processes. This mechanism must be dynamic, since the amount and type of parallelism all depend on the program and data.

One technique to constraining parallelism is to tie process creation (i.e. granularity) to the load of the machine [GM84, Moh90]. For explicitly eager parallel languages (e.g. *futures*), this may be possible. For Daisy, the laziness of suspensions is a semantic requirement; there is no way to randomly "inline" a suspension without changing the semantics of the language[1]. Note that for lazy tasks, it is not the actual *creation*

[1]Although this could be done with strict arguments, revealed by strictness analysis [Hal87, HW87]; see chapter 8.

of tasks that leads to excess parallelism but rather the excess *scheduling* of tasks (i.e. by primitives). However, that there are other valid reasons to reduce granularity. This is discussed further in section 6.9.

With this in mind, we might consider tying process *scheduling* to current load conditions. For example, if the load is heavy, `pcoerce` might only schedule some of its suspensions in parallel, and coerce the others serially. However, since suspensions can only run locally, this would require `pcoerce` to access the load conditions of all the processors on which it might need to schedule suspensions. This would mean an unacceptable amount of overhead, in addition to placing the burden of these decisions on the surface language. Instead, DSI follows the philosophy of total process management. Language primitives can make any number of scheduling requests; the kernel is responsible for *throttling* parallelism to avoid the danger of excessive parallelism.

DSI's approach to controlling parallelism is based on our load-sensitive distributed allocation policy outlined in section 5.2.2. Since suspensions only execute locally, the load-sensitive allocation automatically limits the number of new processes created on a given processor by *other* processors. This in turn limits the possible number of scheduling requests made to a processor by other processors for those suspensions. Thus the only unlimited parallelism that a processor needs to worry about comes from *local* scheduling requests; i.e. made by local processes. This suggests a two-tiered approach for differentiating between local and remote scheduling requests. Each processor maintains one queue for incoming remote scheduling requests and a separate area for local scheduling requests. The two structures have distinct scheduling priorities. If a processor always favors its *incoming* scheduling queue (requests from other processors to execute local suspensions) then it automatically favors the *status quo* of current parallelism load on the machine, and thus discourages parallel expansion occurring from its local scheduling area. The allocation block size and allocation server responsiveness (see sections 5.2.1 and 5.2.2) indirectly determine the current load for the reasons outlined above. These factors affect the number of *possible* outstanding remote requests for a processor's local suspensions, thus controlling the backlog of multiple outstanding dependence stacks.

If there are no outstanding scheduling requests in a processor's incoming queue it obtains a scheduling request (dependence stack) from the local area. If this process should probe a remote suspension, it will be blocked on the dependence stack and a scheduling request will be sent to the remote processor. Note that this effectively raises the blocked task into a higher priority level, since it will be rescheduled in the sending processor's *incoming* queue after the probed suspension converges on the remote processor. Thus the absence of work in a processor's incoming queue results in that processor augmenting the level of parallelism in the overall system by expanding tasks in its local area.

The *type* of parallel expansion occurring in this way depends on whether we use a *stack* or a *queue* for the local scheduling area. A queue results in a breadth-first expansion; a stack results in a depth-first expansion. Superficially, a queue might seem to be the better choice. Breadth-first expansion expresses parallelism at higher levels in the process decomposition, which is generally preferable. Depth-first expansion limits the exponential growth of parallel decomposition. Note, however, that parallel expansion due to our distributed allocation/load-balancing scheme also affords breadth-first parallelism[2], and in fact, this is exactly the mechanism that is used to distribute parallelism evenly. Our remote queue represents the desired parallelism load; our local scheduling area is *excess* parallelism. With this in mind, we choose a LIFO approach for the local scheduling queue, which further throttles parallel expansion.

The priorities of the incoming queue and stack are easily handled by DSI's signal mechanism; these structures are implemented by the prioritized interprocessor message queues described in section 4.3. This mechanism insures a timely interruption of the processor should a remote scheduling request become available.

6.7 Conservative vs. Speculative Parallelism

Dynamic parallelism can be further classified into two types. Our hypothetical `pcoerce` primitive is a form of *conservative* parallelism [Jon87], also called *mandatory computation* [Osb90]. In conservative parallelism the results of the parallel computations spawned are known to be needed (in the case of `pcoerce` because that's what the primitive requires).

In contrast to conservative parallelism is *speculative* parallelism [Jon87] (also called

[2]At least potentially so. See section 5.2.2 and chapter 8.

speculative computation [Osb90]). The results of speculative processes are *not* known to be needed. Speculative processes are usually scheduled based on some *probability* of need. For example, consider a speculative conditional

```
if:[predicate then-part else-part]
```

that schedules its *then-part* and *else-part* in parallel with the evaluation of the *predicate*, on the idea that it will make some headway on the result regardless of the outcome of the test. There are a number of useful applications of speculative computation; section 2.3 contains several examples. Osborne [Osb90] provides a partial taxonomy for classifying speculative computation types.

6.7.1 Managing Speculative Computation

Speculative computation offers additional sources of parallelism over what may be available from conservative parallelism alone, but at the same time it introduces significant resource management problems:

1. Speculative processes should ideally only use *idle* resources; if there is relevant work to be done (mandatory computation), then those conservative processes should get priority.

2. Speculative computation can exacerbate the problem of excessive parallelism, outlined in section 6.5; thus constraining parallelism takes on added importance.

3. Speculative processes can become *useless* during the course of execution; it then becomes necessary to find a way to remove these tasks and their descendants from the system so that they don't continue to use up resources that could be devoted to other processes.

We will describe how these issues are addressed by iterative refinements to our scheduling model for conservative parallelism.

The first problem, that conservative processes take priority over speculative processes, is easily satisfied by having each processor maintain a separate, lower priority queue for speculative processes. If a speculative process probes or schedules any other suspensions they are also scheduled to the appropriate processor's speculative queue.

Thus, once a task has been scheduled speculatively, neither it nor any of its descendants should be scheduled to a conservative queue. At first, this approach does not seem to provide for *contagion* [Osb90]; namely, if sharing relationships are such that a conservative process comes to depend on a speculative task (e.g. by probing it after it has been scheduled speculatively elsewhere, like our speculative conditional) the speculative process should be upgraded to conservative, and any tasks it depends on should be upgraded as well. We will address this issue in our discussion on sharing in section 6.8.

Our second concern, that speculative parallelism be controlled, is satisfied by the same techniques we used in controlling conservative parallelism: using a stack to schedule local suspensions. Since we are segregating speculative and conservative processes, we add a local, speculative stack in addition to our speculative queue described above. The stack works in exactly the same way as the conservative stack described in section 6.6; local speculative suspensions are pushed on the stack, and remote speculative suspensions are appended to the speculative queue on the remote processor. The speculative stack has lower priority than the speculative queue, which favors existing parallelism over new parallelism. Thus, we have two queues and two stacks per processor, which are serviced in the following priority (highest to lowest):

1. Conservative queue.

2. Conservative stack.

3. Speculative queue.

4. Speculative stack.

The implementation of these structures are handled by DSI's multi-queue cascading-priority communication mechanism presented in chapter 4.

Our third concern, removing useless tasks, is more complicated. To illustrate the issue, consider a speculative conditional which produces a process decomposition as shown in figure 15. The *then-part* and *else-part* are speculative computations that have been scheduled in parallel with the evaluation of the predicate. Both speculative processes have started demand chains that are pushing the fringe of the computation forward; both have also initiated additional parallelism that schedules additional processes into the system. Once *predicate* has been evaluated, one of

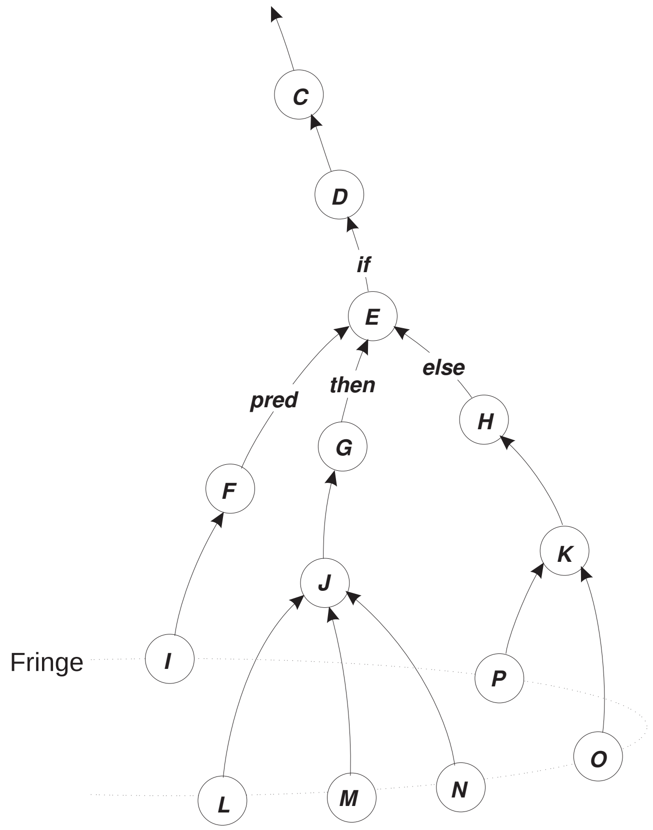


Figure 15: A Speculative Conditional

the dependence graphs emanating from the two speculative processes has suddenly become relevant (conservative) and the other irrelevant (useless). For example, if our predicate evaluates to false, we would like to remove processes G, J, L, M and N from the system as soon as possible. Presumably only L, M and N are actually scheduled, so their removal will suffice to remove any useless processes (G, J) blocked on them, under our dependence stack model.

Ideally, the system should spend as little effort as possible on useless task removal; any effort spent in doing so is chalked up to pure overhead that offsets any gains made in profitable speculation. The approaches used by current parallel symbolic systems generally fall into two categories: explicit killing of tasks [GP81, Hem85] and garbage collection of tasks [BH77, Mil87].

The first approach, explicit killing of tasks, relies on the kernel being able to determine when a task has become useless. At that point the kernel can spawn a termination task, which begins at the root of the speculation process subtree and recursively kills tasks. Alternatively, the kernel can cut the useless processes out of the schedule directly. This can be difficult to do if a distributed scheduling structure is used, as in DSI. Both methods require the kernel to be able to identify all descendants of the useless task and determine whether they are useless or not; because of sharing, some of them may not be useless and should be spared.

The garbage collection approach involves modifying the garbage collector to remove useless tasks from schedules, blocking queues, and other scheduling infrastructure. This approach assumes that useless tasks have become unreferenced from the computation graph. In order to keep them from being retained by scheduling references, *weak pointers* or *weak cons cells* [Mil87] must be used to build scheduling infrastructure containing pointers to tasks so that they aren't retained unnecessarily. A potential problem with this approach is that useless tasks continue to remain in the system until garbage collection [Osb90]. One way to prevent this is through the use of priorities, especially if priorities are already being used to give conservative tasks preference. When a task is discovered to be useless, priorities can be propagated down through the speculative process subgraph to downgrade the status of all descendant tasks. As with the killing approach, the kernel must be able to distinguish the useful from the useless as it descends the subtree; this may require a sophisticated priority combining scheme [Osb90].

6.7.2 Demand Coefficients

The methods described above are both rather *active*; that is, the kernel must proactively detect useless tasks and attempt to restructure the schedule to mirror the changing state of actual dependences occurring in the program. As we have discussed, this is difficult thing to do, and the sophistication of the approaches presented above attest to that fact. DSI uses a third, more passive, approach to speculative task removal; it controls the *extent* of speculative computation through the use of *bounded computation*. The idea is fairly simple; instead of scheduling speculative tasks like conservative tasks and then trying to remove them from a distributed schedule, only schedule them for a bounded amount of computation. This bound is given by a *demand coefficient* embedded in the suspension itself, which directly controls the amount of processing resources (time, memory allocation, etc.) it receives. If a process exhausts its demand coefficient before converging the owning processor removes it from the schedule. In order to keep it going, the scheduling process must *coax* it again; a process may need to be coaxed a number of times before it finally converges.

This approach supports a number of speculative models [Osb90]. Simple *precomputing* speculation is accomplished by a single coax. This type of speculation simply channels some effort into executing a suspension on the chance that it's result will be needed later. *Multiple-approach* speculation is more common, and refers to scheduling a number of tasks where not all of them are necessary. Constructs falling into this category are our speculative `if` (see above), *AND*-parallelism, *OR*-parallelism, and many others (see section 2.3). These primitives operate on the principle of *strobing*: coaxing a number of suspensions repeatedly until one converges. This may be enough for the scheduling task to produce a result; if not, the task may resume strobing until another converges, and so on. The kernel provides routines to handle this strobing on behalf of primitives[3] this simplifies the construction of speculative primitives, and allows the kernel to optimize the strobing behavior.

There are a number of advantages in this approach to managing speculative tasks:

- No effort is required on behalf of the kernel to identify and remove useless tasks. Speculative tasks are scheduled as such from the outset; from then on the process and any that it probes are scheduled in speculative scheduling queues. Speculative tasks will eventually drop out of the distributed schedule once the strobing process stops coaxing them.

- Demand coefficients provide a natural way to *prioritize* speculative tasks. A process's demand coefficient directly correlates to the amount of processing resources it receives; a higher coefficient will result in more attention.

In order for bounded computation to be effective, our system must be able to extend a process's bound to other processes that it probes or schedules indirectly. To do this we modify our demand-driven scheduling scheme to propagate demand

[3]The algorithms are loosely based on those used for *multisets* [FW79, FW80b]; an indeterminate construct appearing in earlier versions of Daisy.

coefficients and observe the limits of a process' computation bound. This essentially *quantifies* the demand propagating through the computation graph.

In implementation terms, this entails the transfer of coefficients along the fringe of the computation; i.e. at the top of the process dependence chains. When a dependence stack is extended, the kernel transfers some or all of the scheduling process' coefficient to the target suspension. If the target process converges, the kernel transfers the *remaining portion* of the coefficient back to the process being resumed. If a process exhausts its coefficient before converging, the kernel unwinds the dependence stack until it discovers a suspension with a positive demand coefficient, and schedules it on the owning processor.

The size and transfer characteristics of demand coefficients must be carefully determined, in much the same way as allocation blocksize is for distributed memory allocation. If set too small, then coaxed computation does not get very far, per coax, and coaxing overhead begins to overtake speculative processing; If set too large, useless tasks will remain in the system for longer periods, although they only compete with other speculative tasks. A strict requirement for the transfer function is that coefficients should monotonically decrease over the course of several transfers; otherwise a chain of speculative demand might live on indefinitely.

6.8 Sharing and Cycles

What happens if a suspension is probed by more than one process (e.g. due to list sharing)? Since we only execute suspensions locally, there is no contention due to multiple processors trying to execute the same suspension. Each processor wanting to schedule the suspension will send a scheduling request to the processor which "owns" the suspension. This can result in several references to the same suspension in a processor's scheduling area. Note that the references may be spread across all four of the scheduling structures (conservative queue, conservative stack, speculative queue, speculative stack), according to all the places *where* the suspension was scheduled and *how* it was scheduled. This handles the case where a suspension is scheduled speculatively by one process and later is probed by a conservative process. In this case the speculative process should be upgraded to conservative status, a situation Osborne calls *contagion* [Osb90]. The first time the process will be scheduled to a

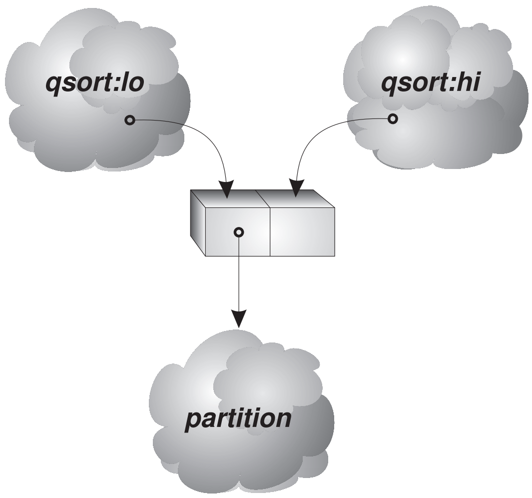


Figure 16: Heap Sharing

speculative queue (or stack), but the second probe will result in the process also being queued on a conservative queue (or stack). Since the conservative queue (stack) has priority over the speculative queue (stack) the process is automatically upgraded. Note that a suspension's demand coefficient reflects sharing relationships; if several processes coax the same suspension its demand coefficient will be larger than if a single process coaxed it. This also effectively upgrades its priority relative to other tasks.

6.8.1 Embedding Dependences

It might seem awkward to maintain dependence information outside of the process context. In other words, to handling sharing, why not simply add a link to each suspension on which you could enqueue other suspensions that are blocked on it. Then, when a process converges, the system could simply reschedule all suspensions in the wait queue. This is the approach used by Multilisp [RHH85] and most other implementations. There are several reasons why DSI does not do this. First, speculative computation renders all *dependences* to be potentially speculative. A given demand chain may be the result of a speculative computation. If we record all dependences in the process state we may retain pointers from needed processes to speculative processes that have since become useless. The problems with this approach and the solutions taken by other systems are outlined above.

The problem with speculative processes points out how dependences are a *temporal* phenomenon, which is the reason it is problematic for the system to accurately dynamically track the dependence relationships between processes. There are other reasons besides speculation that might result in a dependence changing without the system's awareness. For example, a suspension might be moved to a new location by another process, or a location containing a suspension might be overwritten by another process[4]. Therefore it is somewhat of a philosophical decision not to embed temporal dependence information in suspensions themselves, since it may later prove inaccurate. The actual state of dependences is recorded in the suspension state already, in the form of their actual pointer values and code that they are executing. Thus the truest form of demand-driven scheduling would be to poll repeatedly from the root of the entire computation tree, allowing the dependences to "shake out" naturally. Unfortunately, this approach is not feasible on stock hardware, and would result in many suspensions being swapped in repeatedly only to block on their dependences and be swapped out again. DSI's current approach using dependence stacks is a compromise; it prevents the most common form of polling inefficiency, while still providing a method (demand coefficients) for utilizing the programs actual dependences to handle speculation.

6.8.2 Cycles

Cycles can arise in our schedule due to circular dependences. A process that ultimately depends only on itself represents a logical programming error. DSI makes no attempt to recover these, but a solution might be possible using monotonically decreasing demand coefficients.

6.9 Increasing Granularity

The lazy nature of suspending construction coupled with a fine process granularity often combine to create potentially large dependence chains. These chains eat up

[4]This cannot happen under Daisy, but might under a language with side-effects.

valuable space and introduce stack overhead that cuts into useful processing time. One of the interesting open questions of this research is how to reduce the size of these chains.

One way to increase process granularity is by replacing invocations of `suspend` (especially for trivial suspensions) to inlined code or recursive function calls where appropriate. This helps, but not overly so because of the high level of optimization already performed for suspension processing. Informal tests indicate that a function call is only slightly faster than locally coercing a suspension. How much faster depends on how many registers need to be saved on the stack to perform a strict function call; because suspensions are so small, a context switch can be faster than a function call that has to perform several allocations to save and restore registers. Also, many suspensions cannot be eliminated or inlined and still retain proper laziness. Compilation, including strictness and dependence analysis can help to improve granularity. Demand coefficients may also provide a way to implement bounded eagerness; this idea is explored further in chapter 8.

CHAPTER SEVEN

Device Management

This chapter discusses the treatment of devices and I/O under DSI. We describe the stream I/O interface and its integration with DSI's process model. We then describe an implementation of input and output device drivers and their interaction with other system processes. We conclude with a discussion of the strengths and limitations of I/O in DSI.

7.1 The DSI I/O Model

DSI uses a character stream I/O interface; all data is presented to, or issued from, the device drivers using character streams. An *input* device driver appears as a producer, reading bytes from the device and injecting a corresponding stream of characters into the heap. An *output* device driver consumes a stream of characters and writes raw bytes to the device. An input-output, or *bidirectional* device driver both produces *and* consumes streams of characters. The drivers convert between raw bytes and DSI characters because the character representation is better suited for manipulation by programs dealing with lists and atoms.

A DSI character is represented by a singleton literal (an atom consisting of a single character), as shown in figure 17. The DSI `hash` module contains a table of statically preallocated characters. The input driver converts a byte to a character by indexing into this table with the byte. Thus, an input driver is really just generating a stream *spine* with the appropriate subreferences to existing identifiers.

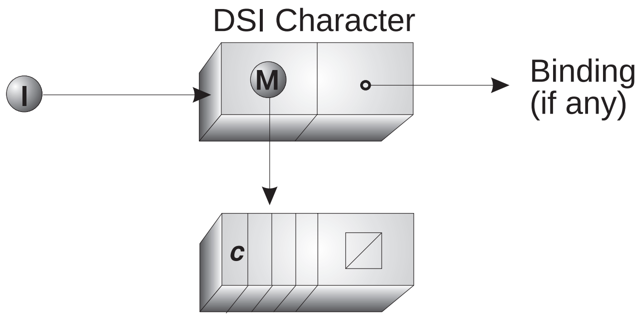


Figure 17: DSI Character Representation

Most DSI programs construct networks of streams that are considerably more sophisticated than this. These stream networks usually operate on higher level forms like expressions. The usual procedure is to parse the input stream into *atoms* and *forms*, process it through the network (evaluation) and convert the result back into a character stream for the output driver. Daisy provides four standard stream transducers for this purpose: a scanner, a parser, a deparser and a descanner. These filters were designed to support Daisy expressions, but can be used for general scanning of atoms, etc.

7.2 Device Drivers

Streams are naturally implemented in DSI using suspensions, so a system with drivers reading characters from a keyboard and echoing it to a terminal would look something like that depicted in figure 18. In the figure, DSI characters are shown as **c**'s; character substructure has been omitted for the sake of simplicity. The input and output drivers are implemented in standard fashion for DSI stream producers and consumers (respectively). Suspending construction provides the synchronization and blocking mechanisms required for transparently interfacing other processes with the drivers. Processes operating on an input stream are not aware of the embedded driver suspension nor any accompanying I/O activity. Similarly, output drivers blindly consume their argument streams without regard to the source of that data; all that is required is that the argument stream conform to a list of characters.

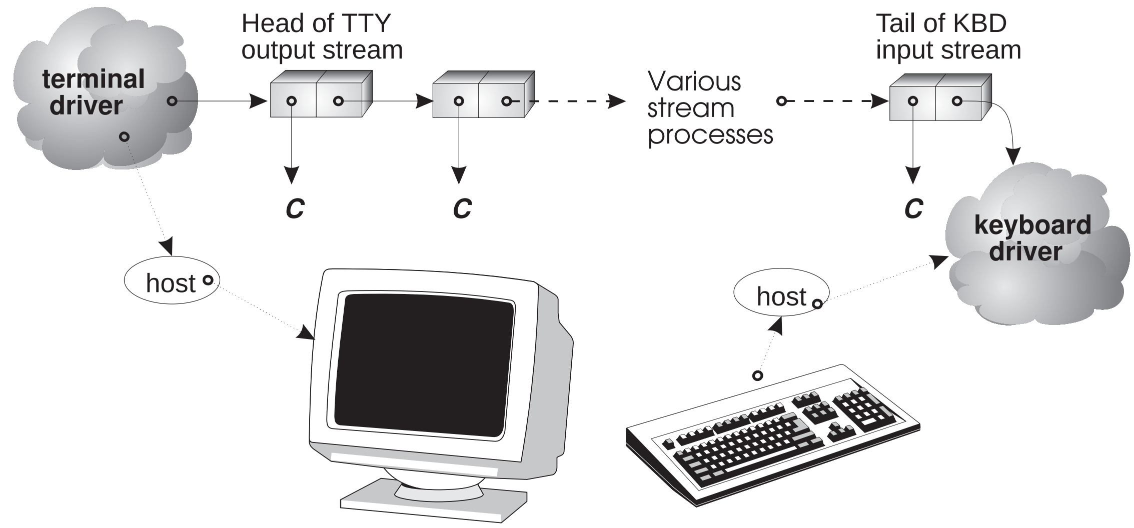


Figure 18: Character Stream I/O

DSI supports the keyboard, terminal, TCP/IP stream sockets and *Unix* pipes as I/O devices. In addition, any file or device that can be accessed through the host's filesystem and can be read or written as a serial byte stream can be used. Actual I/O to the various devices is handled by the host interface layer of the implementation, which provides a set of primitives for the DSI drivers. These primitives open and close devices, issue host I/O calls and buffer data, much like the *C* `stdio` library. The calls are listed in table 3.

DSI interfaces with these host I/O functions via *device descriptors*. A descriptor is represented by a unary cell with a host I/O handle stored in the non-collectible `data` field. The tail is used to store a prompt string (a `MSG`-list) if the device (e.g. a terminal) supports one. The host I/O handle points to a structure containing information about the device, such as its type, buffers, optimal blocksize, and current state. The host I/O routines in table 3 create and use this information to handle differences between devices and to perform efficient transfer and buffering for each device type.

The `dvcin` (input) and `dvcout` (output) modules define system primitives for instantiating a stream process (producer or consumer) of the appropriate device type.

Call	Description
`H_stdinp`	Open `stdin`.
`H_stdout`	Open `stdout`.
`H_stderr`	Open `stderr`.
`H_ttyin`	Open `/dev/tty` for input.
`H_ttyout`	Open `/dev/tty` for output.
`H_input`	Open a file for input.
`H_output`	Open a file for output.
`H_sockin`	Create a socket for incoming connections.
`H_sockout`	Create a socket for outgoing connections.
`H_accept`	Make an incoming socket connection.
`H_pipe`	Open a Unix pipe.
`H_ptypipe`	Open a Unix pipe/pseudo-terminal.
`H_close`	Close a file or device.
`H_poll`	Non-blocking check for input data.
`H_readreq`	Issue disk read request.
`H_get`	Get a character.
`H_put`	Put a character.
`H_iostat`	Check device or file status.
`H_prompt`	Issue a short prompt to console.

Table 3: Host Interface Layer I/O Calls

The corresponding *Daisy* functions are listed in table 4. The following sections describe the operation of the driver processes for input and output.

7.3 Input Devices

All input devices are handled by a single generic driver, the *input device manager*. When a DSI input primitive is invoked, it calls the appropriate host I/O routine to obtain a native I/O handle (e.g. `dski` calls `H_input`). It uses this handle to build a corresponding DSI descriptor. The primitive then calls a device manager subroutine to make the descriptor known to the device manager. The device manager subroutine instantiates the input stream and returns a reference to it. The input primitive may return this stream, or filter it, depending on the device (see 7.3.4 below).

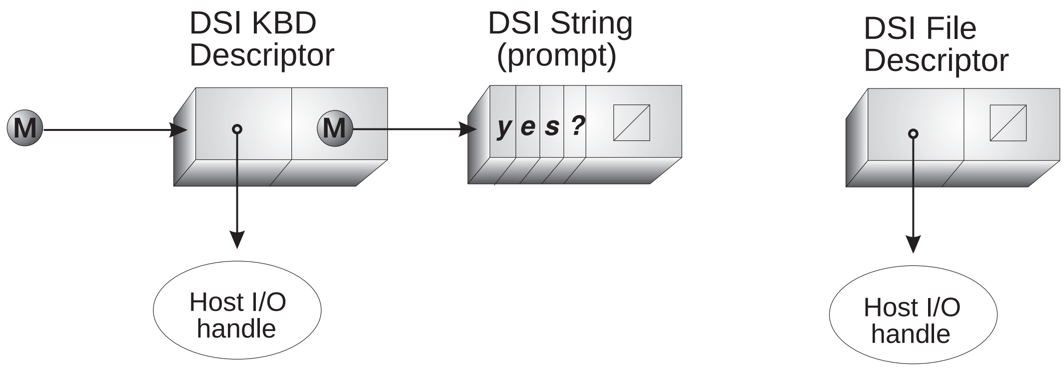


Figure 19: DSI Device Descriptors

7.3.1 The Device Manager

The device manager configuration is shown in figure 20. The device manager maintains a list of descriptor/stream associations on the *device list* (`DVCLST`). The stream part of the descriptor/stream pair is a reference to the *tail* of the input stream of characters associated with the device. When `DVISCAN` wakes up it scans the device list and polls each descriptor in turn:

- If there is data pending, `DVISCAN` will enter an input loop, reading raw bytes from the descriptor, converting them into DSI characters and appending them onto the associated input stream. When there is no more pending data to be read from a descriptor, `DVISCAN` updates the tail reference in the descriptor/stream pair and moves on to test the next descriptor in the device list.

- If the descriptor indicates an end-of-file or error condition `DVISCAN` closes the descriptor, terminates the input stream and deletes the device/stream pair from the device list.

After it has processed the entire device list `DVISCAN` suspends itself and awaits its next activation.

`DVISCAN` is an example of a multi-way shared suspension; the kernel retains a reference to it, and all low-level input streams are formed by using the same reference. `DVISCAN` can be activated in one of two ways: by the kernel in response to receiving a

Input	Description
`console`	Returns a buffered tty input stream.
`rawtty`	Returns a raw tty input stream.
`dski`	Returns a file input stream.
`exec`	Returns a pipe output stream.
Output	**Description**
`screen`	Writes a tty output stream.
`dsko`	Writes a file output stream.
Bidirectional	**Description**
`socki`	Reads/writes a socket stream.
`socko`	Reads/writes a socket stream.
`pipe`	Reads/writes a pipe stream.
`tpipe`	Reads/writes a pipe stream.

Table 4: Daisy I/O Primitives

`SIG_IO` signal, or by a consumer process probing it as an argument stream. In either case `DVISCAN` acts as described above, servicing all descriptors with pending input. This arrangement provides asynchronous I/O support, since any input activity results in all descriptors being serviced.

7.3.2 I/O Signals

There are three signals reserved for device handling. `SIG_IO` should be implemented by a hardware interrupt or host operating system exception that is asserted when there is data ready to be read from an input device. If the device is attached to a single processor, then only that processor should receive the `SIG_IO`. Otherwise, the signal may be multicast, and the processor which "owns" the device manager suspension will activate it. If there are multiple I/O channels that can be serviced concurrently (perhaps on a processor network), it might be desirable to have more than one instance of `DVISCAN` running. This is not conceptually difficult, and would simply require that each instance of `DVISCAN` manage a separate device list, or that they coordinate to avoid corrupting a single shared device list. The current implementation assumes a single instantiation of `DVISCAN`.

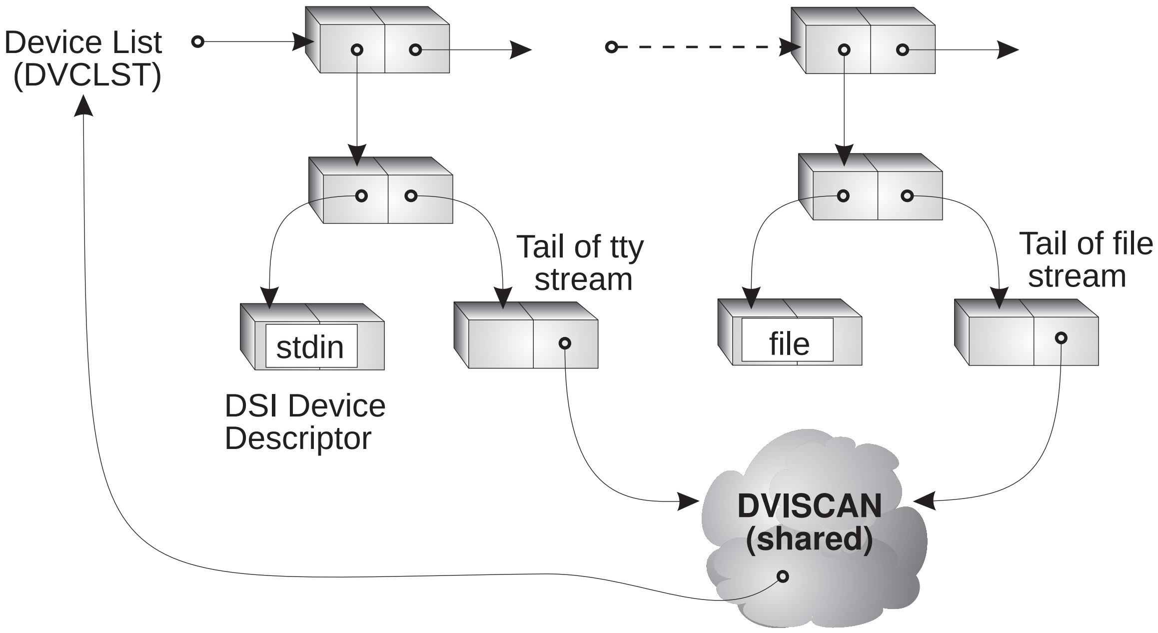


Figure 20: Input Device Handling

Both the `DVISCAN` and `DVCOUT` will assert `SIG_IOBLOCK` if there is no data to be read or the output device is not ready for writing, respectively. In either case the intent is to indicate that no headway is currently possible in the current process chain. Thus `SIG_IOBLOCK` is intended to signal a lateral scheduling change (preemption), and is, in fact, handled like a timer signal.

7.3.3 Garbage Collecting Devices

Occasionally an input stream may become unreferenced while the device or file is still open. The device manager operates in conjunction with the garbage collector to close these *dangling* file descriptors.

`DVISCAN` starts by loading the header cell of `DVCLST` from a fixed heap location, and it finishes by clearing any references to the device list before detaching itself. The result is that unless `DVISCAN` is running, there are no device list references in any suspensions in the system (although there *are* references to the descriptors and input streams). Thus, while descriptors, input streams, and the `DVISCAN` suspension are marked by the garbage collector, the *spine* and *ribs* of the device list are not marked[1]. After the normal mark phase, the garbage collector makes a special pass over the device list, checking to see if any spine/rib elements have corresponding unmarked descriptors. If so, it means that the corresponding input stream has been dereferenced. The collector closes the unmarked descriptors and splices the corresponding elements out of the device list. The device list and the hash table are the only two structures that are collected specially by the garbage collector. The implementation is different, but the effect is identical to that of the *files demon* in MultiScheme [Mil87].

7.3.4 Flavors of Input Devices

This section describes the differences between the various kinds of of input devices supported by DSI.

Disk Input

DSI relies on the host filesystem to manage files. Disk input is handled on a per-file basis. `dski` takes a host filesystem path as argument and creates a new descriptor and `DVISCAN` stream for each open file. It returns a stream of characters read from the file.

Keyboard Input

Keyboard input is differentiated from other kinds of input in that it is implemented at a lower level. Instead of relying on the host buffering to handle multiple instances of keyboard input, DSI requests a single stream of input and multiplexes the stream itself.

At the lowest level, a `DVISCAN` input process presents a single raw stream of characters issuing from the keyboard. This stream is consumed by a single splitter process (`SPLIT`), as shown in figure 21. `SPLIT`, like `DVISCAN`, is a multi-way shared suspension. The splitter splits its input stream, serving up characters on demand to whichever consumer happens to be coercing it at the moment. To add another

[1]Unless a collection occurs while `DVISCAN` is running, in which case part of the list is marked, and no dangling devices are collected on that occurrence.

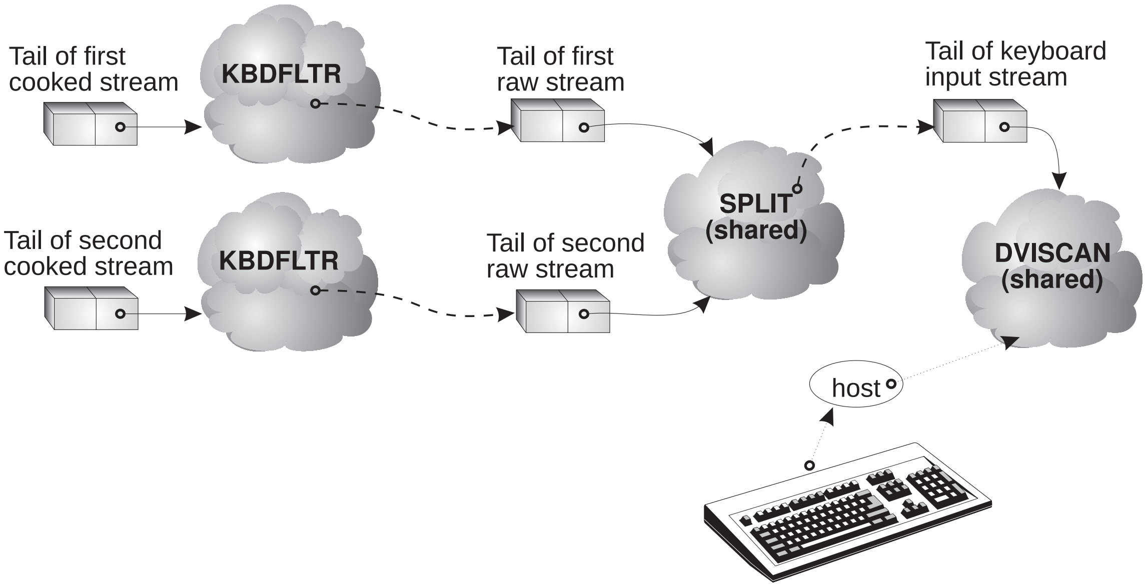


Figure 21: Input TTY Handling

process into the keyboard input mix it suffices to create a new trolly-car stream using the single `SPLIT` reference. The splitter guarantees that no two clients will receive the same character.

The streams produced by `SPLIT` are also all raw character streams. This is what you get using Daisy's `rawtty` primitive. Raw character streams are not particularly useful as keyboard input, however. Most operating systems will reserve certain characters for special interpretation, such as interrupting processes, job control or editing and buffering input. DSI filters the raw stream using a standard keyboard filter (`KBDFLTR` in figure 21). `KBDFLTR` provides a filtered ("cooked") character stream, which is what you get using Daisy's `console` primitive. The functionality provided by `KBDFLTR` is fairly limited. It issues a prompt and then attempts to read up to and including a newline before detaching; when it reawakens it will prompt again and iterate. This simple interleaving of prompts and inputs is sufficient to give programs the proper interactive feel, something can be tricky in a lazy environment. `KBDFLTR` also performs a few other character interpretations, summarized in table 5. You will note that their isn't very much in the way of editing keys; DSI relies on the host

Reference	Character	Action
`ChrNL`	newline	Detach; prompt when awakened.
`ChrEOT`	end of file	Terminate stream.
`ChrDLE`	??	Forced detach.
`ChrESC`	escape	Escape the next character.

Table 5: Standard Keyboard Filter Actions

operating system to handle that. It would not be difficult to do, however. Using the `rawtty` primitive it is possible to write more complex or special purpose filters in Daisy.

There also aren't any process termination interpretations such as the ubiquitous *Control-C*. Which process would this affect? DSI's process model is vastly different from conventional systems; there are many, many processes associated with any "job". The *Daisy Programmer's Manual* [Joh89c] has examples of how to do job control in Daisy using multisets.

7.4 Output Devices

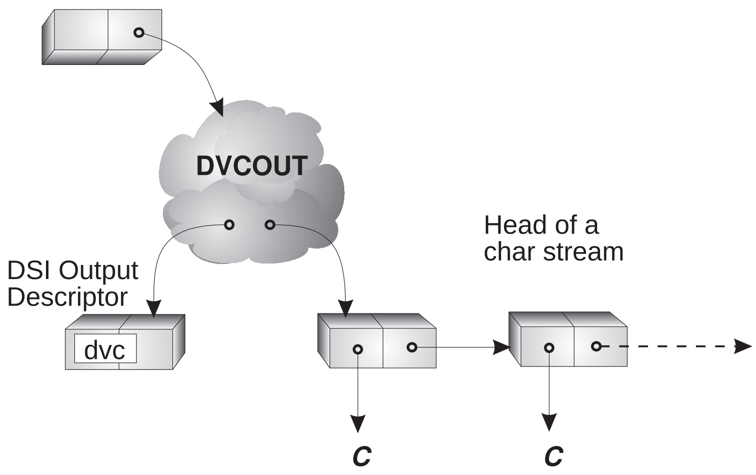


Figure 22: An Output Process

7.4 Output Devices

Output devices are symmetrically constructed to input devices. An output driver is a stream *consumer* of characters. Each character is converted to a raw byte and written to the output channel using the host I/O primitives which handle the proper buffering for the device type (e.g. disks are written in blocks, terminals are unbuffered. etc.).

As with input devices, output devices share a common generic output driver (`DVCOUT`), with differences between devices contained in the host I/O handle and handled at the host interface layer. *Unlike* the input manager, the output driver is multiply instantiated for each output occurrence; that is, a new process is created for each consumer. Output device primitives construct the appropriate descriptor, create a suspension for the `DVCOUT` process, and give it a converge context. That context is then probed to bootstrap the output process.

`DVCOUT` is a fairly simple loop. It probes its input stream for a character and converts it to a raw byte (see figure 17). It then checks the status of its output descriptor:

- If the descriptor is ready for writing the byte is emitted using `H_put` and the process iterates.
- If the device is not ready for writing, `DVCOUT` signals the kernel with a `SIG_IOBLOCK`, indicating that the process is blocked and a lateral context switch is in order.
- If the device indicates an error, `DVCOUT` converges to an erron.

If `DVCOUT` reaches the end of its input stream it converges to `NIL`. Note that `DVCOUT` may block, but never detaches; that is, it is infinitely strict in its argument.

7.4.1 Flavors of Output Devices

This section describes the differences between the various kinds of of output devices supported by DSI.

Disk Output

`dsko` takes a filename and a stream of characters as arguments. It creates or truncates the named file and writes the characters to it. It returns `NIL` after the last character is written.

Terminal Output

Terminal output is the complement to keyboard input; `screen` takes an input stream and writes it to the terminal, returning `NIL` after the last character has been written. Terminal output is not implemented at the same level of detail as keyboard input, however. If it were, multiple output streams would be merged by a `MERGE` process into a single output stream before being delivered to a single (tty) instantiation of `DVCOUT`. As it is, DSI lets the host handle the multiplexing of outputs. An upgrade to DSI's terminal output system is planned to make it complementary to keyboard input.

7.4.2 Output Driven Computation

The effect of output devices is more than just to output characters. Because the output device driver is infinitely strict, it presents a point source of continuous demand that propagates down through its input stream and through the data space to the fringes of the computation (and ultimately, to any input drivers). It is the output devices which ultimately provide the impetus for DSI's demand-driven behavior. In between the the output drivers and the inputs, the user program controls the *direction* of this demand flow.

Note that with the valid implementation of `SIG_IO`, DSI's input drivers are event-driven rather then demand-driven. This makes sense, given the transient nature of data on the I/O bus; most devices implement some form of buffering, but will experience overruns if their interrupts are not serviced in a timely manner.

7.5 Bidirectional Devices

In addition to input-only and output-only devices, DSI also supports *bidirectional* devices. A bidirectional device is one that can be modeled as having both input and output channels. For example, although the display and keyboard are actually two separate physical devices, under *Unix* they are integrated into a single logical device (e.g. `/dev/tty`) from which you can read and to which you can write.

There are two kinds of interfaces that can be used for bidirectional devices in DSI. One is to split the device into two logical devices corresponding to its input and output components; this is the basic approach used for terminals. The read and write terminal I/O streams are accessed by separate input (`rawtty`) and output (`screen`) primitives, which create distinct input and output driver suspensions, as described above. Another approach to bidirectional devices is to model the device like a stream *transducer*; i.e. a driver process that is both a producer *and* consumer of streams. Figure 23 shows how this is implemented using a single suspension. The driver multiplexes input and output functions; the input stream is written to the device and the data read from the device is appended to the output stream.

DSI provides two kinds of bidirectional "devices" in the transducer-type interface: *sockets* and *pipes*.

Sockets

DSI provides an interface to *BSD*-type *stream sockets*, a high-level TCP/IP networking standard on many operating systems, including most flavors of *Unix*. Sockets provide *full-duplex* (bidirectional) data streams between two points (sockets) over a network. Two Daisy primitives provide access to the *Unix* socket interface. Both `socki` and `socko` are *higher-order* primitives. They both return *functions* (closures) that map character streams to character streams; i.e. they create a producer-consumer process as described above. The input stream is the characters to write to the socket and the output stream is the characters read from the socket.

The difference between `socki` and `socko` is in how the socket connections are established; `socki` accepts incoming connections (server-style) and `socko` makes outgoing connections (client-style). The `socki` primitive takes a port number as argument, creates a socket, binds it to the port address on the host, and returns a function. When *that* function is applied to an input stream, DSI watches the port, establishing the bidirectional data flow when an incoming connection is made. This action corresponds to the *Unix* `accept` call, except that it does not block. `socko` takes a list argument containing two items: a host name and a port number on the network; it returns a function. When *that* function is applied to a stream it tries to `connect` to the named socket, and also establishes a a bidirectional data stream.

Unix Pipes

Another useful bidirectional device interface provided in DSI is a *Unix pipe*. Like the socket primitives, the `pipe` primitive is higher-order. It takes a flat list of literal arguments which are intended to be the elements of the `argv` argument vector passed to a forked *Unix* process, and returns a function. When *that* function is applied to a stream it creates a pipe, and `fork`s and `exec`s a *Unix* process. The forked process's I/O is redirected to the pipe, so that the input stream to the pipe interface suspension (see figure 23) is fed to the process's `stdin`, and the process's `stdout` (and `stderr`) are returned through the interface suspension as the output stream. The connections are illustrated by figure 23.

Two variations on `pipe` are provided. `tpipe` operates just like `pipe`, but opens a *pseudo-terminal* between the process and the I/O streams. This fools the forked process into thinking it is connected to an actual `tty` device, which some *Unix* programs require to work correctly. `exec` is a first-order function for creating processes that only write to `stdout`; its argument is in the same format as the function returned by `pipe`.

7.6 Limitations of DSI's I/O Model

DSI's I/O model reflects its roots in functional programming research. The stream representation provides a non-temporal, applicative interface to asynchronous device I/O that integrates well with DSI's process model. The stream interface is useful for a wide variety of devices, but some devices work better with stream model than others. In this section we will explore some of the limitations of stream I/O under DSI.

7.6.1 Interleaved Terminal I/O

This is an instance of more general problem involving non-strictness, but it becomes particularly evident with interactive stream I/O. Proper interleaving of input and output from a terminal is difficult when there is no intuitive sense of the demand patterns ordering execution of stream processes. The most typical manifestations of this problems are that input is requested before a prompt appears, prompts for

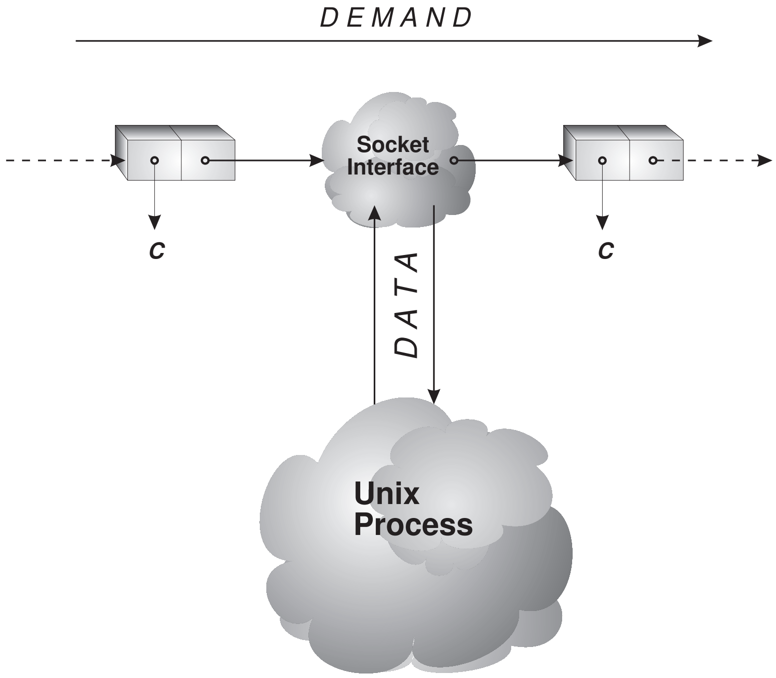


Figure 23: A Bidirectional *Unix* Pipe "Device"

various input channels are interleaved (so that one does not know which input channel he is typing to), and so on.

One solution to this problem is to insert explicit point-source coercion or laziness operations into the code to achieve the proper interleaving of prompts, input, and output [HS88, p.2]. At best this is a trial and error, ad-hoc procedure. Fortunately this problem is alleviated to a great extent by the ubiquitous use of modern windowing shells. The ability to direct I/O to individual windows usually does away with the sorts of machinations just described and the vagaries of multiplexing the console among many active I/O processes.

The *xdsi* interface allows DSI/Daisy programs to manipulate multiple character-mode I/O windows under the *X* window system. *xdsi* is implemented as a separate, non-integrated server that is bundled with the DSI source distribution. It relies on DSI's support for sockets (see above) to communicate with programs running on DSI. This arrangement nicely avoids the need to link in a number of large host *X* libraries into the DSI host executable; the only requirement is that the host operating system have TCP/IP networking support. It also makes possible a number of flexible connection arrangements. DSI programs can connect to multiple *xdsi* servers (usually on different workstations); conversely, an *xdsi* server can be connected to multiple DSI sessions (usually on different workstations or multiprocessors). This flexibility provides different ways to experiment with applicative operating systems, distributed computing, and other applications of networked user I/O.

DSI programs establish a *control* stream with the *xdsi* server by connecting to a specified socket address. Over this stream it can send commands to create and manipulate input and output windows on the display that *xdsi* is controlling; the model is very similar to that used by the X window system itself. Each window is associated with a DSI character stream. Input windows are stream producers in DSI; output windows are stream consumers. The streams interface in the normal way with user programs running on DSI.

One of the problems alluded to above in the description of terminal I/O is the difficulty of determining which process is requesting input. This is handled in *xdsi* by a special command on the control stream which inverts the background and foreground of an input window. The command is issued automatically by the *xdsi* interface in DSI on behalf of the input window when demand is applied to the input stream.

The effect allows the user to know which windows are requesting input. A screen snapshot of the *xdsi* system in action is shown in figure 24. The screen shows several input and output windows connected to a Daisy program which models a hardware blackjack machine. The black windows are quiescent; the white windows are currently expecting input.

As of this writing, *xdsi* only supports unidirectional text windows. An obvious extension to this approach is to allow other specialized types of windows, such as GUI widgets. The *xvi* system [Sin91] is a windowing interface for a lazy functional language that is also implemented using a server approach (although in a more tightly coupled way than *xdsi*). It allows widgets and even lower level drawing to be controlled from the language by interpreting text strings issued by the program to the standard output, and parsing them calls to the X graphics libraries. This approach requires a working knowledge of the widget and drawing interfaces in C. The *eXene* environment for CML [Rep91] is an example of a more general lower level use of streams to control a GUI interface. eXene models each window as a *collection* of streams: mouse, keyboard and control streams. Widget libraries can be created in CML using these stream interfaces.

7.6.2 Non-Character Mode Devices

Integrating non-character-oriented I/O devices is not superficially difficult—it just involves streaming data other than characters and converting it as appropriate in the driver. This would require a less generalized driver to interface these kinds of devices to the system. For example, a plotter might consume a stream of commands and integer coordinates.

7.6.3 Stateful Devices

For serial "stateless" devices, such as keyboards, mice, terminals, or network sockets, the stream interface is a reasonably accurate model of the underlying events. For "stateful" devices, such as disks, the stream interface provides only a limited subset of the device's capabilities. Some devices, such as bitmapped displays, seem very unlikely to be handled by a stream I/O interface under current technology, other than at a high level, such as the *xdsi* interface. Part of the problem is the large, stateful

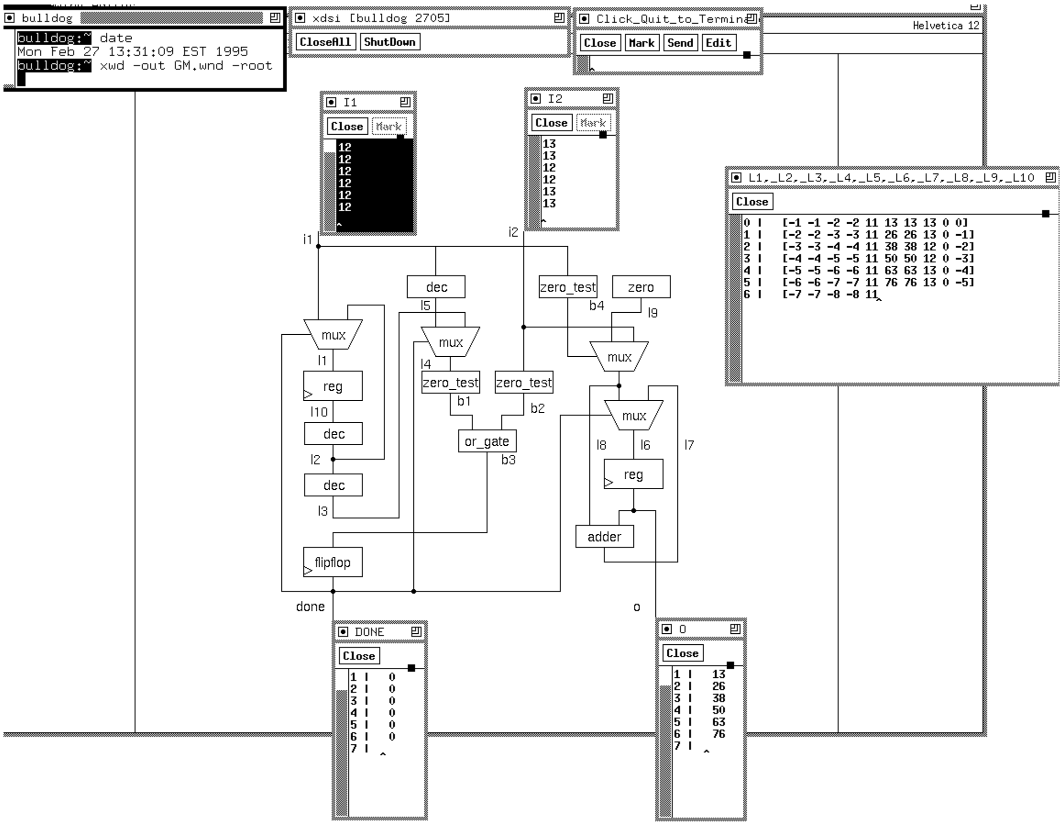


Figure 24: An *xdsi* Session

nature of a bitmapped display memory and part can be attributed to processor speed and bandwidth limitations of serializing a bitstream to the display.

CHAPTER EIGHT

An Analysis of Daisy

This chapter provides an analysis of Daisy programs in terms of parallelism and efficiency under suspending construction. A brief introduction to the language was given in section 2.3; for a full description of the language, see the *Daisy Programming Manual* [Joh89b].

8.1 Demand Driven Computation

It is instructive to understand the operation of the output-driven pipeline that is the conduit of demand in Daisy programs. Most Daisy programs are pipelined compositions of streams, terminated at either end by input and output devices. A good example of this is the main interpretation pipeline built by the bootstrap process, which is roughly analogous to the *init* process in Unix. The interpretation pipeline corresponds to the following Daisy program:

```
screen: scnos: prsos: evlst: prsis: scnis: console: "&"
```

Application is denoted by infix colon (`:`), and associates to the right. This program builds a stream pipeline as shown in figure 25. `console` returns a stream of characters from the keyboard, interleaving lines of input with ampersand prompts. This stream is fed to a scanner (`scnis`) which converts it to a stream of tokens. The tokens are passed through a parser (`prsis`) which creates a stream of expressions. The expressions are read by a mapped evaluator (`evlst`), which interprets the expressions and returns a stream of results. The results are piped through a deparser (`prsos`)

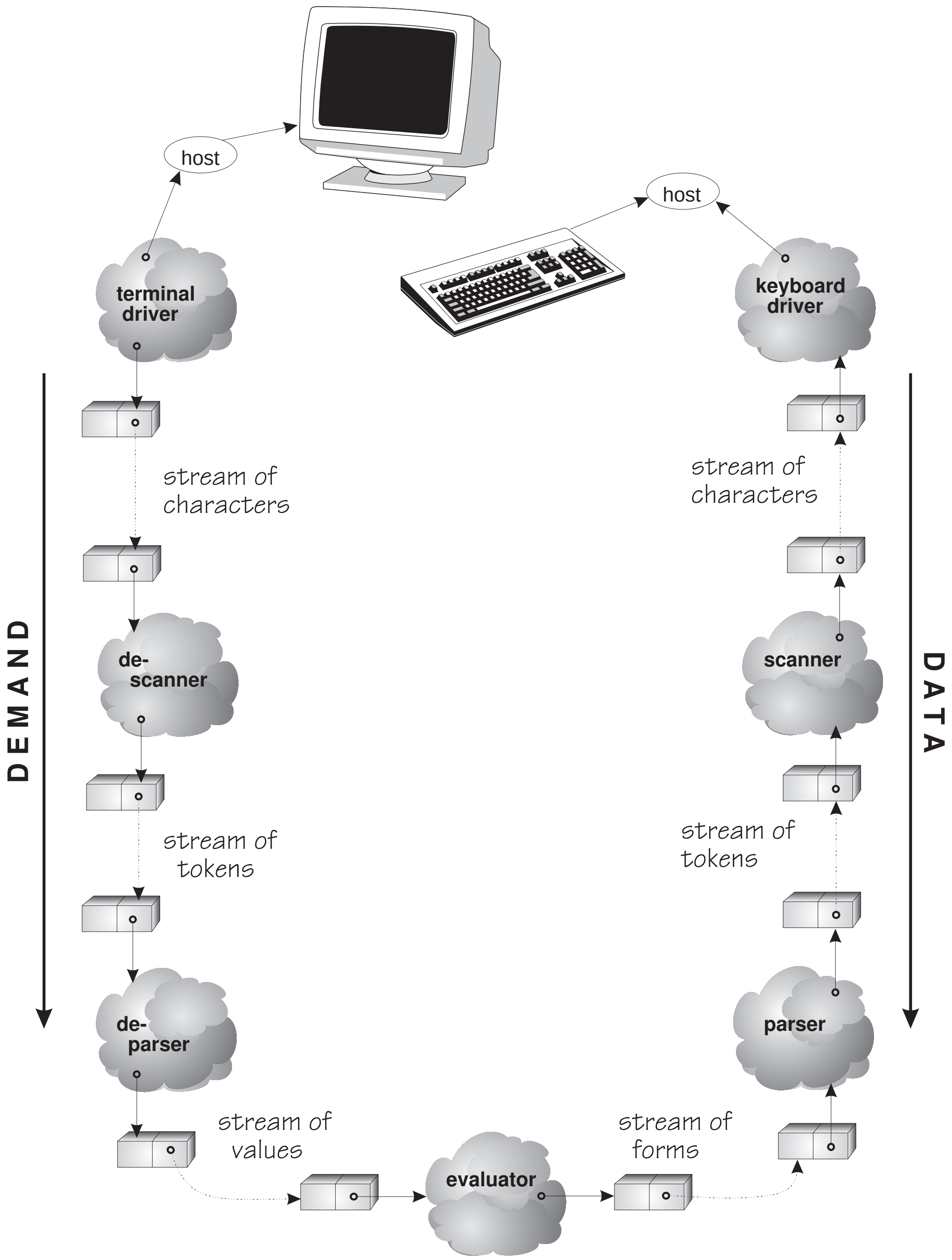


Figure 25: The Daisy Interpretation Pipeline

and a de-scanner (`scnos`), respectively, before finally being consumed by the terminal output driver (`screen`).

Demand emanating from the printer (`screen`) ripples down through the pipeline to the input device, and data flows in the opposite direction. In the middle (`evlst`), programs are evaluated. Programs will typically create their own network of streams, which plug in transparently to this top-level stream pipeline. The demand rippling down through the output pipeline is what drives program computation.

8.2 Limitations of Suspending Construction

Suspending construction offers many possibilities for implicit parallelism, as described in section 2.3. However, the model also has deficiencies that undermine its effective parallelism. We explore some of these here, and suggest solutions.

8.3 Excessive Laziness

While we have have touted the virtues of laziness, it can also have the unintended effect of squelching parallelism. This is well illustrated using a naive version of the quicksort program given in chapter 2, the code for which is shown in figure 26 (this example is adapted from [OK92]). `quick` operates in the usual way, partitioning its argument and then calling two recursive quicksorts on the result. This kind of divide-and-conquor parallelism is ideal for parallel applicative processing.

Consider the application of `quick` to a list of numbers. The result of `quick:[...]` eventually reduces to a sequence of nested, suspended `append`s. This can be *visualized* as:

```
append:[ append:[ append:[ ...
```

(this represents what a string reduction model might actually do). Under normal output-driven evaluation, the printer, attempting to coerce the *head* of this structure, causes the inner suspended appends to unfold until the innermost append performs a suspending *cons* (line 21 of figure 26). This satisfies the demand of the next outer append for `X`, so it also performs a lazy *cons* and converges. In this manner, computation ripples back up to the topmost append as the dependence chain is satisfied.

```
quick = \L.
    let:[[X ! Xs] L
      if:[nil?:L  L
          nil?:Xs L
            let:[[lo hi] partition:[X Xs]
                 append:[quick:lo [X ! quick:hi]]
                ]
         ]]

partition = \[p L]. part:[p L [] []]
part = \[p L lo hi].
  let:[[X ! Xs] L
    if:[nil?:L      [lo hi]
        le?:[X p]  part:[p Xs [X ! lo] hi]
                   part:[p Xs lo [X ! hi]]
       ]]

append = \[L1 L2].
  let:[[X ! Xs] L
       if:[nil?:L1 L2
           [X ! append:[Xs L2]] ]
      ]
```

Figure 26: The Quicksort Algorithm in Daisy

The printer then issues the first number of the result and iterates, probing the second element, which repeats the entire scenario just described.

There are two problems with this behavior. First, `append` is lazy in its second argument, which means the suspended recursive call `quick:hi` will not be coerced until the printer has output the entire result of `quick:lo`. Thus, no parallelism has been achieved. Secondly, the lazy `cons` occurring on line 21 results in N complete traversals of the suspended computation graph for outputting N total elements. Ideally, if we know that the entire structure is needed, we would like to propagate *exhaustive demand* [OK92], so that when demand ripples down through the inner `append`s, their result is completely coerced before converging. This would result in a single traversal of the computation graph, resulting in *much* less context switching. Neither of these problems are due to the kernel's default process management. Given a scheduling request, the kernel will execute that suspension and any suspensions that it coerces until such time as it converges. Rather, there are two problems are at the language level which are responsible:

1. the serialized demand of the printer, and
2. the laziness of suspending construction, which "cuts off" eagerness at each suspension creation.

Suppose we were to address the first problem by using a concurrent primitive at top level instead of the serial printer. For example, suppose we have a primitive `pcoerce` (see chapter 6) that schedules its argument suspensions in parallel and returns after they have all converged. We could apply it to our top-level quicksort like so:

```
pcoerce: quick: [...]
```

This would cause the suspension for `quick:hi` to be coerced as well as `quick:lo`. This example works because our result is not bushy; if it were, `pcoerce` would likely only coerce the top-level elements down to their first suspended cons, at which point they would have satisfied the top-level convergance requirements of `pcoerce`. This is the problem referred to by item no. 2 above; the suspending cons "cuts off" even our explicit attempt to be parallel.

8.3.1 Bounded Eagerness

We might consider addressing the excessive laziness problem in the kernel through the *demand coefficient* mechanism; having the system speculate on suspensions as they are created. Demand coefficients would apply not only to probes, but to *allocation* as well; i.e. when you create a cell containing a suspension the system automatically siphons off some of the running process' demand coefficient into the suspension and schedules it immediately. This could be implemented by modifying the `new` instruction to trap after the allocation if one of its source registers contained a suspension. This would implement a form of *bounded eagerness*.

The problem with this approach is that it is naive system-level speculative parallelism (see section 2.3). There is not enough useful context information for intelligent scheduling choices. Does the entire coefficient of the running process get transferred to the first suspension created? If only part of it, how much, and to which created suspensions? The former policy would essentially default the system to an inefficient form of strictness; every suspension would be immediately scheduled after creation and the creating process would be suspended. The latter case results in massively speculative parallelism; all suspensions are scheduled with a little bit of energy. This is far too general, and undermines the purpose of laziness. Suppose the demand is for a particular element (say the 10th) of the result of our quicksort. Under this "shotgun" approach we expend effort across the entire structure even through the exhaustive demand in this case is "lateral" and then "down".

The limitation of the demand coefficient approach is that while it can indicate the relative *overall* demand for a suspension's result, it does not provide specific information about how to *distribute* that demand through probes or suspension creations. If the demand coefficient was not just a number, but instead a bushy list structure containing numbers, it might be a start towards conveying the demand information we need, but such a structure would be expensive to update externally, and would require more effort to interpret by the kernel. For these reasons it seems preferable to avoid general system-level speculation on new suspensions and relegate scheduling decisions to the language level (except for demand-driven scheduling).

8.3.2 Granularity Revisited

The fine granularity of process decomposition under suspending construction is also a potential problem. Suspending construction suspends *every* computation embedded in a list, no matter how trivial. Under our allocation scheme, trivial suspensions are just as likely to be allocated remotely as suspensions representing significant computations, whuch we are more interested in offloading to other processors. We would like trivial suspensions to be eliminated or at least allocated locally and suspensions representing important parallelism to be allocated remotely, so that parallelism is migrated properly.

8.3.3 Strictness Analysis

One way to address these problems at the language level is through the use of *strictness analysis* [Hal87, HW87]. The information from strictness analysis can be used to eliminate many "trivial" suspensions that do not need to be created. Unfortunately, strictness analysis does not discriminate between trivial suspensions and "important" suspensions that we want to preserve for parallelism's sake.

Strictness analysis can also indicate instances where the system can be safely eager, but this bypasses some parallel opportunities where the system is being properly lazy. For example, strictness analysis does not increase parallelism in our quicksort program above, since append is not strict in its second argument. The language, being properly lazy, will suspend the recursive quicksort calls; the system, being properly demand-driven, will not schedule them until they are probed.

8.4 Program Annotations

We have discovered some limitations of naive suspending construction. We can either settle for what the system provides or annotate our program to provide clues for the system about how to handle these situations.

Consider our example; suppose we annotate the recursive calls to quicksort like so:

```
...
    let:[[lo hi] @partition:[X Xs]
```

```
                append:[quick:lo [X ! @quick:hi]]
              ]
      ...
```

Here, the `@` annotation could suggest suspensions that should be allocated remotely, thus insuring that important computations are offloaded. This kind of annotation might be most effective if coupled with a policy of allocating all non-annotated suspensions locally.

We might address the second problem, inefficient demand-driven, computation, by annotation as well. Suppose we used the following definition of `append`:

```
append = \[L1 L2].
  let:[[X ! Xs] L
       if:[nil?:L1 L2
            [$X ! $append:[Xs L2]] ]
      ]
```

Here, the `$` annotations would force the recursive call to append to be performed before returning, but still leaves append non-strict in its second argument.

Thus, it seems that we would like both an "important parallelism" annotation and a demand or strictness annotation. With this in mind, I suggest three kinds of annotations:

- *Strictness annotations* allow the programmer, parser or compiler control over process granularity.
- *Eagerness annotations* allow the programmer to specify "eager beaver" demand; this would help address the inefficiencies of demand propagation described earlier.
- *Parallel annotations* would convey the distinction between trivial suspensions and important suspensions, for making allocation decisions.

The use of annotations represents a step back from the goal of completely transparent resource management. However, the eager and parallel annotations we describe here can be considered guidelines, and the system is free to disregard them. Strictness annotations are not so, but can be considered a programmer or compiler optomization.

8.4 Program Annotations

The annotations we are suggesting are not overly intrusive, and deserve consideration as a viable approach to the problems presented in this chapter.

CHAPTER NINE

Conclusion

Section 9.1 outlines the main results and contributions of the work described in this dissertation. Section 9.2 describes possibilities for future research and development efforts stemming from this work, and section 9.3 examines other symbolic language implementations most relevant to my work.

9.1 Summary of Results

This dissertation discusses the design of a list processing engine for implementing parallel symbolic languages on stock multiprocessors. The design results from experience implementing an applicative language on the *BBN Butterfly* multiprocessor. The language and engine are based on *suspending construction*, a fine-grained, concurrent, non-strict computation model for *Lisp*-type languages.

The contributions of the dissertation are:

1. A virtual machine architecture for fine-grained parallel list multiprocessing. This virtual machine design identifies key areas where hardware support would accelerate the execution of programs using computational models similar to suspending construction, such as Lisp with *futures*.

2. A microkernel providing memory management, process management and device management for high level parallel symbolic programs. The resource management techniques described represent new approaches to these problems for many parallel symbolic languages.

3. An analysis of suspending construction in *Daisy*, a language implemented on this architecture and microkernel.

9.1.1 Implementation Notes

The virtual machine, kernel and language described in this thesis (DSI V4.1) has been implemented in slightly simpler forms (DSI V2.x and V3.x) on the BBN Butterfly multiprocessor.

The notable differences between that implementation and the design described herein can be summarized as follows:

- The Butterfly virtual machine has two identical parallel sets of registers: one for the kernel and one for regular processes; this was later generalized into the context window model.
- Signals are used, but in a more limited way.
- The memory management is essentially the same as described here, but uses a slightly different representation for allocation vectors.
- Process management is limited to conservative parallelism based on annotations in the user program; demand coefficients and speculative computation are not supported.
- Each processor maintains a separate, single circular queue for incoming scheduling and allocation requests. Scheduling requests consist of distributed dependence stacks, as described here.

This implementation served as a prototype for the current design.

The current system as described in this thesis (DSI V4.1) has been implemented sequentially. It includes a virtual machine assembler [Jes] implemented in Yacc/C, a host interface library (in C) and a set of DSI assembly modules comprising the DSI kernel and Daisy interpreter. A Daisy compiler, written in Daisy, translates Daisy code to DSI code.

I plan to fold the improvements described in this report back into the Butterfly implementation. Another possible target for a near-term port is a Silicon Graphics Challenge multiprocessor.

9.2 Future Work

Specific experiments aimed at analyzing the characteristics and performance of DSI's resource management algorithms are a top priority. These experiments would target:

- Load-balancing effectiveness and performance compared to other systems. I am particularly interested in comparing my allocation-based system with the task-stealing mechanism used in Multilisp [RHH85] and its offshoots.
- An analysis of the effects of allocation block size on parallelism and load balancing.
- Our local task execution model should be analytically compared to systems which use task migration strategies or non-local process execution. The basic question to be answered is whether locality considerations, such as the ones used in DSI, are beneficial enough to justify their use with such a small process granularity.
- The effectiveness of our priority scheduling structure on constraining parallelism and excessive speculation.

These are just a few of the immediate possibilities for analytical comparison and experimentation.

Some crude instrumentation is provided in the Butterfly implementation for *event logging* using the BBN *gist* utility. This has provided some limited preliminary quantitative data. Better instrumentation is needed for more accurate performance and analytic measurements for the experiments described above. Support for host-specific tools (e.g. gist) is useful, but a portable, cross-platform mechanism would be better.

9.2.1 Incremental Enhancements

In addition to the experimentation outlined above, there are a number of incremental enhancements that would improve DSI's utility and capability for supporting surface languages.

Daisy has a similar design philosophy as Scheme: a simple, but expressive core language, without a lot of bells and whistles. The suggestions for improvements in this section reflect that spirit and are meant to address limitations based on experience using Daisy and not to match feature for feature in other languages. Solving these problems would also address some of the issues faced in implementing other languages on DSI (see section 9.2.2).

Storage Management

There are two main limitations of DSI's storage management system that should be addressed. First, the current garbage collector only compacts two different sized objects: cells and suspensions. This rules out dynamically allocated arrays (static arrays are allowed) which are standard in modern applicative languages; their exclusion is a noticeable omission in Daisy. This could be rectified by implementing a more sophisticated compaction scheme.

A second limitation I would like to address is the system-wide pauses caused by garbage collection. System-wide garbage collection pauses become unacceptable as heap sizes increase, making interactive response time suffer and may cause data overruns in a device handler if it occurs during pending I/O and the host cannot provide sufficient buffering. Our system has the advantage of parallel collection in all phases, so that collection pauses increase according to the heap size divided by the total number of processors. Nevertheless, this still allows for significant pauses. I would like to incorporate an *incremental* garbage collector [AAL88, Moo84, Kie86] to avoid the latency penalties incurred by the current collector. The incremental collector should preserve locality of data as in the current scheme and not migrate data or processes to other processor nodes.

Generational collectors are becoming commonplace in many strict applicative systems, but are likely to be ineffective for a lazy language. Generational collectors are designed around the principle that there are few references from older generations to newer generations; stores into old generations have to be monitored so that the references can be updated when newer generations are collected. This is generally true of strict languages, but lazy languages have many updates of old structures pointing to new structures. Indeed, suspending construction is predicated on this fact. Therefore, generational collection may not be a viable method for incremental collection on DSI.

9.2 Future Work

Global Namespaces

Lexical scoping solves most identifier conflict issues, but does not solve the problem of conflicting *global* identifiers. While the current implementation is satisfactory for smaller single user sessions, a larger layered software architecture or multiuser environment will require the shared global namespace issue (see chapter 4) to be resolved. Some Lisp-based systems address this issue through per-task storage based on fluid-binding [Mil87], first class environments, or extending deep binding to top-level environments [DAKM89]. A flexible *module* or *package* system for Daisy would address this problem in a more general way that might have benefits for other languages implemented on DSI.

User-Defined Exceptions

Daisy's treatment of errors is woefully inadequate for serious programming. Errors are handled by propagating *errons* (error values) back up the computation tree to top level, which essentially provides the equivalent of a stack dump. Explicitly testing for errors is not a viable solution. In addition to introducing overhead, it is a disincentive to programmers to consistently check for errors in every conceivable spot in the code where they could occur. For this reason, many languages provide configurable exception handling. This often takes the form of user-specified procedures or continuations for exception conditions, like errors. These kind of control mechanisms are not a viable solution for Daisy, given its applicative model.

DSI does provide a low-level exception mechanism (signals) that is compatible with an applicative solution. Signals establish an asynchronous event dependence between two processes without imposing state or control changes *within* those processes. A signal occurring in one process simply causes another process to be activated, presumably to handle that exception. An exception mechanism based on specifying exception *dependences* between expressions might fit naturally into this model.

The major obstacle to implementing this in DSI is that signals are a component of the architecture, not processes; i.e. signals map between context windows (running processes), not suspensions. An example of how this is adapted to arbitrary processes is given by the use of signals in supporting demand-driven scheduling (see chapter 4 and 6). It might be possible to further generalize the mechanisms used for this purpose to support user-defined signals. Ideally, exception support could be introduced into the language without requiring additional state fields to be added to suspensions. However, some kernel support is probably necessary, so this can also be considered as an enhancement to the DSI system.

9.2.2 Implementing Other Languages

In addition to enhancing Daisy, I would like to implement additional languages on top of DSI. This would validate the usefulness of DSI as a symbolic language implementation platform and indicate what further enhancements are necessary in the virtual machine and kernel to support compatible models of computation, such as *futures*. I don't expect DSI to become a completely general-purpose language implementation platform; by nature that leads to inefficiencies for certain classes of languages. Rather, DSI will remain optimized for fine-grained symbolic languages. Support for richer data types will be determined by progress in enhancing storage management (see remarks in section 9.2.1 above).

Scheme

I am particularly interested in implementing a strict, parallel Scheme-dialect on DSI, primarily for comparison of DSI's resource management algorithms with those of Multilisp-type languages. DSI's resource management algorithms (especially load-sensitive allocation) seem well-suited for an eagerly parallel language. Scheme is similar internally to Daisy, but semantically quite different (applicative order, eager, explicit parallelism) and would thus also be a good starting point for language comparisons. The dialect would support *future* and *delay* constructs that would be implemented with suspensions, the difference being that *futures* are scheduled immediately after creation.

Some potential issues to be resolved in this proposed language are whether the language should provide side-effects and what sort of I/O interfaces should be provided. If side-effects are included, some general-purpose visible synchronization constructs will need to be provided in the language. Also, the interaction of continuations across process boundaries is troublesome [KW90]. If side-effects are not provided, the language is free to be parallel wherever possible in its primitives (including speculatively) just like Daisy. As for I/O, the language could easily provide stream I/O based on the kernel's device interface, but standard Scheme also provides imperative I/O; providing a proper superset of the Scheme standard [CJR91] might be beneficial.

Communicating Sequential Processes

Another language that would be particularly well suited to DSI's task and communication model would be a CSP-like language. A CSP language would be naturally implemented in DSI using streams for communication and signals for synchronization between processes. This kind of language would allow the safe integration of imperative programming with applicative programming, by encapsulating side-effects within the scope of a suspension, which communicates with other processes through a port-type interface to obtain its values. The integration of the two styles would be very similar to that described for *Rediflow* [RMKT84].

The major hurdle for implementing this type of language on DSI is the same one described above for providing exception handling in Daisy; namely, generalizing the signal mechanism from the architecture to arbitrary suspensions and user-defined signals.

9.3 Related work

In this section I compare several parallel symbolic language implementations to the Daisy/DSI work described in this thesis. The number of such projects is fairly large and a thorough comparison of all of them is outside the scope of this report. My comparisons are limited to those parallel Lisp dialects and "general-purpose" parallel symbolic kernels that are most similar to the Daisy/DSI project in terms of the characteristics listed in section 1.2 (p. 4). DSI *is* a list-processing engine, and the architecture and kernel are oriented toward Lisp-like languages; parallel Lisp systems therefore invite the most direct comparison. Nevertheless, the resource management solutions described herein may be applicable, in various forms, in wider circles. I reserve some general remarks for the class of functional languages, with whom Daisy also shares a fair number of traits.

9.3 Related work

9.3.1 Parallel Functional Languages

Although DSI and Daisy have a strong heritage in Lisp, superficially Daisy appears to have much in common with functional languages. Like functional languages, Daisy lacks side-effects; however, Daisy is not referentially transparent due to its indeterministic *set* operator.

Daisy's similarity to functional languages extends beyond laziness and lack of side-effects to include stream I/O and the use of error values. Lazy, side-effect-free programming also leads to similar styles of expressing data recursion, functional mappings and other functional programming hallmarks. At the same time, Daisy does not have many of the trappings of "modern" functional languages [Hud89] such as pattern matching, type inference, and algebraic or abstract data type constructors. In these areas Daisy remains very Lisp like, using lambda forms, dynamic typing, and standard Lisp data types.

Internally, Daisy's evaluation model is also Lisp like, using an environment-based closure model rather then the ubiquitous graph reduction, combinator or dataflow approaches common in functional languages. This point is significant, because the reduction model strongly influences the underlying view of hardware; DSI's virtual machine is fairly different from published descriptions of abstract machines for graph reduction.

This dissimilarity extends to the task model. Under graph reduction, a task is simply an available redex; this is just a pointer to a subgraph that can be reduced in parallel. The reduction machine will "spark" a parallel task [Jon87] by communicating this pointer to some other processor. In contrast, DSI's process decomposition occurs in list construction, not in reducing application redexes. Suspensions are implemented as a fine-grained process control records rather than as graph redexes.

In summary, Daisy shares many semantic traits with functional languages due to its lazy semantics and lack of side-effects, but differs substantially in most other respects. To avoid confusion, I refer to Daisy as a *quasi-functional* or *applicative* language.

9.3.2 General-Purpose Symbolic Processing Kernels

This category refers to kernels that are presented in the literature as general (symbolic) language implementation vehicles. Virtual machines for specific languages are not included here. Kernel issues relating to specific languages or implementations are described in the section 9.2.2.

STING

The STING project [JP92b, JP92a] lays claim to being a general purpose high-level parallel "substrate" for implementing parallel symbolic languages. The STING system offers a high-level programming environment (the system is based on a Scheme compiler) with a "concurrency toolkit" approach for implementing other languages. Under STING, users can dynamically *create* virtual machines, each containing an arbitrary number of *virtual processors*. Each virtual machine provides a separate address space for the *threads* executing within it and the user can define a scheduling policy for threads on each virtual processor. Their system provides explicit creation, synchronization, blocking and task killing operations. So, for example, users can implement speculative computation, but must manage the detection, killing and removal of useless tasks themselves.

In contrast to STING, DSI is at the same time both lower-level and higher-level. It is lower-level in the sense that programming the DSI virtual machine amounts roughly to assembly programming, with registers and instructions that are very close to the actual machine; STING's virtual machines, virtual processors, and so forth are all first-class objects that can be manipulated from within Scheme (in fact the system was written mostly in Scheme). DSI is higher-level in the sense that its kernel provides a more packaged solution to process control, with the kernel handling synchronization, demand-driven scheduling, useless task removal, and so forth.

Another very notable difference between STING and DSI is that DSI's processes are much finer-grained than STING's threads. Each STING thread contains a stack *and* a heap[1], and is associated (through virtual machines) with a protected address space. Thus STING's threads, while conceptually smaller than traditional operating system processes, are still fairly large-grained tasks. It is highly unlikely that one

[1]Although heap and stack segments are not allocated until a process runs for the first time.

could implement suspending construction on STING, using threads for suspensions, in any efficient way, other than by re-implementing a finer-grained process package on top of it, which would obviate the purpose of using it.

DSI's low-level implementation is due to one of the original motivations of the project: to explore fine-grained list-multiprocessing at target levels. DSI's virtual machine *is* "the machine" and its kernel *is* the operating system on that machine. In contrast, STING is a relatively high-level environment that is implemented on stock multiprocessing operating systems in *Scheme*. DSI's kernel reflects the motivation to provide as much system control of parallelism as possible, so that the language implementor is simply responsible for identifying parallelism. STING's approach is to allow the programmer full control (and thus full responsibility) over process decomposition, mapping, etc.

The Chare Kernel

The *Chare* kernel [KS88] is closer in spirit to DSI than the STING system described above. The Chare kernel was developed to support the distributed execution of Prolog on message-passing multiprocessors, although it is touted as a general-purpose parallel programming environment. Like DSI, the Chare kernel handles all aspects of resource allocation for processes (*chares* in their terminology). Programming on the Chare kernel is fairly low-level (an improper superset of *C*; i.e. not all C features are supported), and chare granularity seems to be similar to that of suspensions, perhaps finer.

That is about where the similarities end. The chare computation model is inherently distributed (although there is an implementation for shared memory machines). Chare programming consists of creating chares and sending messages between them. Shared data is handled on distributed processors by having the user program provide a callback routine to *package* any data that the kernel is about to pass to another processor, where it is unpacked. Pointers must be eliminated by the packaging procedure. On shared memory machines this is not required. The kernel can be configured to use one of several load-balancing procedures that migrate chares to neighboring processors and also to use one of several queueing strategies for messages.

It is not clear how speculative processing could be handled in the Chare kernel. Chares can be created with specific priorities, but termination is explicit (a chare kills itself). It is unclear from [KS88] whether or how chares are garbage collected (or for that matter, how storage management in general is handled).

In summary, the Chare system bears a resemblance to DSI in that the process granularity is similar and the goals of the two systems are total process resource management. Otherwise, the systems are fairly different not only in computation model and style, but in the algorithms used for resource management.

9.3.3 Parallel Lisp

In the family of symbolic processing languages, the work done on parallel Lisp is perhaps the most relevant to the work presented here and vice versa. That in itself is not surprising, since Lisp is the grandfather of symbolic processing, and the seminal research leading to the work presented here was conducted in Lisp. This is particularly true regarding the body of work centered around the `future` construct used in *Multilisp* [RHH85, RHH86, Osb90], and its offspring, *MultiScheme* [Mil87] and *Mul-T* [DAKM89]. There seems to have been a cross-fertilization of ideas in the genesis of suspensions and futures [BH77, RHH85], although the work on suspensions seems to predate that of futures [FW76a, FW76b]. A number of other *Scheme* constructs like *delay/force*, *continuations* and *engines* share some characteristics with suspensions in various ways, but only futures share the inherent notion of concurrent process entities. A high-level, denotational analysis of these constructs vs. suspensions can be found in [Joh89c].

Conceptually, both suspensions and futures represent fine-grained tasks. However, most implementation descriptions of a future's state indicate a larger granularity than that of suspensions. This is probably due to the historical fact that futures were designed primarily as an explicit eager construct, and thus can represent an arbitrarily large granularity, even though they are generally used in a fine-grained way. Suspensions, on the other hand, were intended from the outset to be a lazy object and to be created implicitly during list construction. Thus the suspension's design attempts to minimize the state profile. It is instructive to consider whether an efficient suspending construction type language could be built using *delays*, Multilisp's counterpart to suspensions.

Aside from granularity and scheduling semantics, futures and suspensions also differ operationally. Although they are mostly transparent in Scheme (most strict primitives will implicitly *touch* them into existence), futures can be directly shared in a way that suspensions cannot. Non-strict operations are free to pass future pointers around as arguments, insert them into new structures, etc. This is because when the future is *determined* (their terminology) the value is stored in the future itself, until the garbage collector comes along and "shorts out" the value. Suspensions, on the other hand, overwrite their references upon converging. This means that suspension checking occurs earlier than future detection, which is generally tied in to run-time type checking. Some implementations, such as MultiScheme, further differentiate between a *placeholder* (future) and *task*, actually separating the two entities into different objects, both of which may be manipulated within Scheme.

This more explicit manipulation of futures in *Scheme* is indicative of the language differences between *Scheme* and Daisy. Multilisp is a strict applicative order language (Scheme) with side-effects and explicit parallelism using futures. It also includes explicit synchronization constructs such as as atomic *replace*, locks and semaphores. These features allow explicit process granularity, low-level heap mutation and process synchronization to be performed in Scheme, which aids the parallel Lisp implementor, since more of the system code can be written in Scheme itself. It also provides a great deal of flexibility for implementing other parallel languages on top of it, as in [JP92b], although one implicitly inherits Scheme's control structures, memory management and other artifacts with this approach.

Since these low-level hooks into the implementation exist, it is tempting to make them visible to the programmer, on the principle that it affords the programmer greater flexibility and capabilities. In fact, this is the hallmark of Lisp systems. However, this also can work to the language's detriment. It drags the language down toward the level of the implementation, exposing the gritty details of implementation and requiring the programmer to take more active control in resource management. In contrast, Daisy remains a fairly high-level language, without low-level hooks, and in doing so has much flexibility in parallel scheduling at the language implementation level. DSI remains at a very-low level, for fine control of the machine. In this light, parallel Scheme appears to be more of a middle-level language between DSI and Daisy.

This concludes my general comparison of parallel Scheme and Daisy/DSI. In the next few sections I highlight the differences in resource management between the various flavors of parallel Scheme and DSI. The information presented here is gleaned from various papers in the bibliography, cited here where appropriate.

Multilisp

Multilisp is a version of Scheme with futures implemented on the experimental *Concert* multiprocessor at MIT [RHH85, RHH86]. Multilisp was instrumental in starting the parallel Lisp revolution, providing one of the first working implementations of futures, and bringing many fine-grained parallel resource management issues to light.

Multilisp's memory management is based on a copying, incremental collector. Each processor has an *oldspace* and *newspace*; objects are allocated out of the processor's own newspace, which means there is no allocation synchronization and very high data locality for local processes. During garbage collection, the first processor to reach an object copies it to it's newspace, regardless of whose oldspace it was originally in. The author claims this may improve locality of reference because if the containing processor has dereferenced the object it will be entirely migrated to the copying node, which presumably still needs it. However, if there is more than one reference to an object there is no guarantee that the processor copying it is the one that will access it the most, so this may not be a valid assumption. Unlike Multilisp, DSI's allocation is distributed

Multilisp employs an "unfair" scheduler to constrain parallelism. Processes are placed on a local stack, oldest first; executing a future results in the parent task being stacked while the child continues to execute. When a processor runs out of tasks it looks around at other processor's stacks for a task to "steal" off the top. This arrangement limits task queue growth to the same order of magnitude as if the futures executed on the stack. Note, however, that this also encourages depth-first expansion of the process decomposition, since the processes higher in the process tree are buried on the stacks; these processes may be more likely to represent significant computations that should be migrated rather than leaf tasks. A good example of this is the quicksort program and other divide-and-conquer strategies. It is interesting to note that the Mul-T implementation (Multilisp's heir apparent) uses a queue for this structure. Halstead notes that under the LIFO scheduling strategy, once the system is saturated, futures complete before their parent task ever gets resumed; in essence, the future expression could have been evaluated on the stack. This may have motivated the work on *lazy future creation* [Moh90] used in Mul-T (see below).

MultiScheme

MultiScheme [Mil87, FM90] is a parallel dialect of MIT-Scheme, extended to support futures.

Miller's dissertation [Mil87] concentrates on the details of extending MIT Scheme to support futures and the other features of MultiScheme. The MultiScheme scheduler, written in Scheme, and presented in the thesis, uses a set of primitive functions to pass tasks to the underlying multiprocessing engine for distribution. There are no details about the underlying resource management algorithms used in the implementation, making it difficult to draw comparisons to DSI in this report, which essentially describes a different level of implementation and/or set of problems, although some issues such as task blocking and resumption are described.

Miller does describe the basis for speculative task recovery in MultiScheme. The system provides a special data type, *weak cons cells*, which have the property that the garbage collector will remove the `car` reference if no other references point to the same object. Thus, weak cons cells can be used to build hash tables, device lists, and other structures that must be specially collected in systems like DSI (see sections 5.3.2 and 7.3.3).

Weak cons cells are used to build the waiting (blocking) queues used for tasks waiting on the value of a future (*placeholder* in MultiScheme), so that any speculative task that has been dereferenced elsewhere will not be retained following a garbage collection. Osborne [Osb90] notes that the success of this approach depends on the frequency of garbage collection, although priority propagation was later added to downgrade speculative tasks between collections.

MultiScheme is similar to DSI in its use of global interrupts (visible in scheme) and barrier synchronization to organize garbage collection, although the garbage collection itself is encapsulated by a primitive function at the lower implementation level, and is not described. This example typifies the difference between Multilisp and Daisy, regarding low level hooks into the implementation, described earlier.

9.3 Related work

Mul-T

Mul-T is a extended version of Scheme augmented with futures and implemented on the Encore Multimax multiprocessor using a modified form of the *ORBIT* Scheme compiler [DAKM89]. For language and computation model comparisons with DSI, see the general remarks on parallel Lisp, above.

Mul-T's run-time system, like most other implementations (except ours, it seems) queues blocked tasks into the task state of processes directly. No mention is made of whether weak pointers are used or speculative tasks are garbage collected, as in MultiScheme. The language does provide a *grouping* mechanism for debugging, and users can suspend entire groups of tasks and kill the group. Mul-T provides support for process-specific variables (i.e. global name spaces) by converting T's normal shallow binding to deep binding.

Mul-T uses a memory management scheme very similar to that described for Butterfly Portable Standard Lisp (see below). Chunks of memory are allocated from a global heap and then objects are suballocated out of the chunks. This reduces contention for the shared global pointer managing allocation in the heap as a whole, although large objects are allocated out of the heap directly. Since the Encore is a non-NUMA architecture (see sections 2.1 and 2.2.1, chapters 3 and 5), no locality considerations are made for heap allocation. Mul-T uses a parallel stop-and-copy garbage collector.

The most significant difference between DSI and Mul-T regarding allocation is that Mul-T always allocates futures locally, whereas DSI distributes suspension allocation, subject to the availability of remote suspensions. This primarily reflects the difference in load-balancing strategies (see below). For further comparison remarks to DSI's memory allocation scheme, see section 9.3.3 below.

Mul-T scheduling uses two queues per processor, one for new tasks and one for resumed blocked tasks. Blocked tasks are restarted on the processor on which they last executed, in an attempt to improve snoopy cache locality, but tasks may migrate around the system due to stealing [RHH85]. The scheduling priority is as follows:

1. Run a task from the processor's own suspended task queue.

2. Run a task from the processor's own *new* task queue.

3. Steal a task from the new task queue of another processor.

4. Steal a task from the suspended task queue of another processor.

No priorities are provided within a queue.

DSI, like Mul-T, employs a multi-tier task scheduling algorithm, but the queuing strategies used in the two systems are quite different. DSI uses a load-sensitive allocation strategy for load-balancing rather than migrating tasks via stealing. An experiment is needed to determine whether this allocation strategy results in less overhead than task stealing, because processors do not have to spend time examining each other's task queues, or whether the overhead of our allocation strategy and inter-processor scheduling requests outweighs the cost of non-local context switching. DSI tasks *always* run locally and do not migrate, so cache locality should be good for DSI suspensions on a bus-based multiprocessor like the Encore.

Mut-T's per-processor task queues correspond closely with DSI's conservative task queue and conservative task stack (see chapter 6). DSI's use of a stack for local suspensions rather than a queue plus the lack of stealing may provide even more parallelism throttling than is available under Mul-T's queuing scheme. Note, however, that Mul-T uses a technique called *lazy task creation* [Moh90] to increase granularity of programs dynamically by *inlining*, and this probably reduces the need for kernel-based throttling. This technique is unlikely to be applicable to a lazy language like Daisy, which *relies* on suspension creation for laziness in addition to parallelism. DSI's suspensions are also likely smaller than futures.

In addition, DSI provides an additional speculative task queue and stack with corresponding lower priorities to keep speculative processes from disturbing conservative ones. DSI removes speculative tasks from the system implicitly. Kranz [DAKM89] provides no indication of how speculative computation is handled, if at all, in Mul-T. The techniques used in [Osb90] may be a possible direction for Mul-T in this regard.

Mul-T's I/O also shares similarities with DSI's device management. Mul-T uses a "distinguished task" dedicated to terminal I/O to avoid the problems of shared, parallel I/O; this corresponds roughly with DSI's *input device manager* (see chapter 7). Mul-T likely uses the imperative I/O model of Scheme, however, compared to DSI's stream I/O model; Mul-T's I/O efforts are motivated by integration with it's task group debugging environment (see above). Mul-T also uses a distinguished "exception handler task" for each processor in the system devoted to coordinating task groups. In DSI, *all* exception handling is accomplished via separate processes using *signals* (see chapter 3); our "distinguished exception handler task" is the kernel. DSI signals are also used to coordinate I/O, tracing and debugging, garbage collection, suspension detection, etc.

Finally, Kranz [DAKM89] makes very good arguments for hardware support for tag-checking to accelerate future detection on stock hardware, validating similar remarks I made in section 2.1 and chapter 3.

QLisp

QLisp [GM84, GG88] is a eagerly parallel version of Lisp based on a shared-queuing discipline. The language is based on a version of Lucid Common Lisp and compiles Lisp code to machine instructions. The language provides several explicitly parallel constructs:

- A *qlet* construct for evaluating arguments to a *let* form in parallel.
- A *qlambda* form, for creating a separate process which is accessed by passing arguments to the closure. The process executes the closure body on the arguments and returns the result to the caller.
- Futures, a la Multilisp.

An interesting feature of QLisp is that the process granularity of *qlet* and *qlambda* processes is directly controlled by a predicate expression in the form. If the predicate evaluates to false the form reverts to its normal (non-parallel) counterpart.

The philosophy of the QLisp implementation seems to be oriented toward providing full programmer control over parallelism. QLisp provides functions to create, acquire, release and test locks. It uses *catch* and *throw* to explicitly kill other tasks; that is, when a processor executes a `throw`, all other tasks created within the corresponding `catch` are killed. This feature can be used to manage *OR*-parallel speculation.

Published details [GM84, GG88] are sketchy on the memory and process management used in QLisp. The language is implemented on an Alliant FX/8 multiprocessor (a non-NUMA design), which means QLisp probably does not micro-manage for locality purposes. The report states that processors contend for a shared lock on the "free memory pool" and that whichever processor exhausts the memory pool first performs a garbage collection while the other processors wait. The comparison remarks for DSI's memory management scheme verses QLisp are basically the same as for PSL, below.

A comparison of process management is difficult, given the lack of details on QLisp's design. However, the authors [GM84] describe the use of a single shared queue for distributing work among processors, hence the name, *QLisp*.

Butterfly Portable Standard Lisp

A group at the University of Utah has also implemented Lisp on the BBN Butterfly [MRSL88]. PSL is a standard Lisp augmented with futures and fluid binding to support concurrent name spaces.

The memory management described in [MRSL88] has some resemblance to DSI's. It divides the heap into segments across all processors. Distributed allocation is accomplished by allocating *chunks* out of the global heap; suballocation occurs out of the chunk until it is exhausted, at which point another global chunk is allocated. Their scheme allocates a chunk at a time from shared global heap state variables. DSI also allocates in blocks, but each processor's allocation vector contains blocks *for each processor* so that suballocation is more distributed. Also, in our scheme blocks are requested directly from the remote processor and can only be *donated* by that processor; they are not allocated on demand from global shared heap variables. This supports our load-balancing scheme in which suspensions only execute locally (see section 5.2.2); under PSL, blocked tasks can be rescheduled on any processor. An interesting result in [MRSL88] supports our choice of distributed allocation of cells for the BBN Butterfly. They first tried a local allocation scheme, which fared poorly due to contention; a subsequent refinement to distributed round-robin allocation improved performance (see chapter 5).

The garbage collector described in [MRSL88] is a monolithic and sequential (i.e. it runs on one processor) stop-and-copy design; DSI's mark-sweep collector is distributed and fully parallel in all phases. PSL uses a single global scheduling queue, this also handles load balancing; DSI uses distributed queues, executes suspensions locally, and relies on distributed allocation for load balancing. Butterfly PSL is very similar to Multilisp and similar comparison remarks apply (see section 9.3.3).

9.3.4 Parallel Lisp: Summary

DSI's virtual machine architecture and parallel kernel differ from the other Lisp implementations described above in several ways. The most significant and notable differences are summarized below.

Machine Design

DSI's virtual machine differs in two significant ways from most other parallel Lisp VMs. First, it uses a *context window* design in which several processes may be register resident on the same processor at the same time, as opposed to context swapping into a single set of registers. Secondly, in the other systems exceptions are handled in the traditional way using a stack, and exception handlers are defined using procedures or continuations; in DSI, exceptions (signals) are handled by separate processes.

Kernel Design

Many parallel Lisps use a tightly-coupled kernel design in which interrupts and kernel services are handled on the user or system stack. The DSI kernel is implemented as a special group of processes, distributed across all processor nodes; kernel communication is asynchronous, implemented by message streams.

Distributed Allocation

The parallel Lisp systems we have reviewed use one of two allocation methods:

1. Processors contend for access to global shared heap allocation pointers. Synchronization (usually in the form of locks) must be used to arbitrate access. Some systems optimize this by allocating large chunks at a time and then sub-allocating out of them to reduce contention.

2. Processors allocate only out of their own local space and either rely on a copying collector to migrate data, or they don't worry about locality of data at all.

9.3 Related work

In DSI processors asynchronously *request* and receive allocation blocks directly from remote processors. They can only suballocate from blocks that have been provided by these processors; allocation is distributed evenly across processors, subject to availability of allocation blocks.

Load Balancing

The parallel Lisp systems we have reviewed use *task stealing* or other task migration methods to balance the workload across the set of processors. In DSI, tasks can only execute locally; since the garbage collection does *not* move data between processors, suspensions execute on the processor on which they were allocated. The distributed allocation mechanism is indirectly tied to the processor load in an attempt to self-balance the system.

Process Management

Most parallel Lisps require the programmer to specify parallel decomposition and explicitly manage synchronization, at least in the case of side effects. Daisy/DSI attempts to provide automated process management in addition to automated memory management. The user should no more care about process decomposition, synchronization, scheduling or process reclamation than he does about memory management. However, annotations may be necessary to achieve maximal parallelism where excessive laziness interferes.

Speculative Computation

The Lisp systems we have reviewed either require the programmer to explicitly kill useless tasks, or rely on garbage collection of processes coupled with priority propagation to prevent them from running. DSI uses *bounded computation* to control the growth of new speculation and to remove useless tasks.

Parallel I/O

Imperative I/O to a single device (e.g. a terminal) presents problems for any parallel system; like side-effects, it requires explicit synchronization to control access and achieve the proper sequencing of events. At least one parallel Lisp system (Mul-T) uses a specialized process and grouping mechanism to control access to the terminal or controlling window. DSI uses a stream model of I/O designed to work with the nonintuitive scheduling orders imposed by lazy evaluation; for the same reasons this model works extremely well for parallel I/O. Non-temporal character streams can be merged, directed to separate windows, or combined in any fashion by the user.

9.4 Conclusion

Parallel processing technology has been maturing for a number of years, to the point where shared-memory multiprocessors are poised to become commodity items on the desktop. High performance and numeric computing is likely to continue to be dominated by traditional imperative languages, but high level symbolic languages have the opportunity to carve a niche in a more general class of computing problems on these machines. These languages have much to offer programmers, and indirectly, end-users. The ease of programming in these languages is in a large part due to their automated resource management as much as any other attractive qualities they might possess. The success of these languages will depend on the performance and effectiveness of this resource management as well as the co-evolvement of hardware to better support the execution of symbolic language data types and fine-grained processes. I hope that the work presented herein will in some way aid in this effort.

REFERENCES

[AAL88] John Ellis Andrew Appel and Kai Lai. Real-time concurrent collection on stock multiprocessors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

[AG94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, 1994.

[AL91] Andrew Appel and Kai Lai. Virtual memory primitives for user programs. In *ACM Architectural Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.

[BH77] Henry C. Baker and Carl Hewitt. The incremental garbage collection of processes. *ACM SIGPLAN Notices*, 12(8):55–59, August 1977.

[Bur84] F. Warren Burton. Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Transactions on Programming Languages and Systems*, 6(2):159–174, April 1984.

[Bur85a] F. Warren Burton. Controlling speculative computation in a parallel functional programming language. In *IEEE International Conference on Distributed Computing Systems*, pages 453–457, May 1985.

[Bur85b] F. Warren Burton. Speculative computation, parallelism and functional programming. *ACM Transactions on Programming Languages and Systems*, c-34(12):1190–1193, December 1985.

[CJR91] William Clinger and Eds. Jonathan Rees. Revised$\hat{4}$ report on the algorithmic language scheme. Technical report, Indiana University, November 1991. Technical Report No. 341.

[DAKM89] Jr. David A. Kranz, Robert H. Halstead and Eric Mohr. Mul-t: A high-performance parallel lisp. In *ACM Symposium on Programming Language Design and Implementation*, pages 81–90, 1989.

[DW94] et. al. D.S. Wise. Uniprocessor performance of reference-counting hardware heap. Technical Report 401, Indiana University Computer Science Department, Bloomington, Indiana, June 1994.

[FM90] Marc Feeley and James S. Miller. A parallel virtual scheme machine for efficient scheme compilation. In *ACM Conference on Lisp and Functional Programming*, pages 119–130, 1990.

[FW76a] Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257–284. Edinburgh University Press, Edinburgh, 1976.

[FW76b] Daniel P. Friedman and David S. Wise. The impact of applicative programming on multiprocessing. In *International Conference on Parallel Processing*, pages 263–272. IEEE, 1976. IEEE Cat. No. 76CH1127-OC.

[FW76c] Daniel P. Friedman and David S. Wise. Output driven interpretation of recursive programs, or writing creates and destroys data structures. *Information Processing Letters* **5**, pages 155–160, December 1976.

[FW77] Daniel P. Friedman and David S. Wise. An environment for multiple-valued recursive procedures. In B. Robinet, editor, *Programmation*, pages 182–200. Dunod Informatique, Paris, 1977.

[FW78a] Daniel P. Friedman and David S. Wise. Applicative multiprogramming. Technical Report 72, Indiana University Computer Science Department, Bloomington, Indiana, 1978. Revised: December, 1978.

[FW78b] Daniel P. Friedman and David S. Wise. Aspects of applicative programming for parallel processing. *IEEE Trans. Comput.* **C-27**, 4, pages 289–296, April 1978.

[FW78c] Daniel P. Friedman and David S. Wise. Functional combination. *Computer Languages* **3**, 1, pages 31–35, 1978.

[FW78d] Daniel P. Friedman and David S. Wise. Sting-unless: a conditional, interlock-free store instruction. In M. B. Pursley and Jr. J. B. Cruz, editors, *16th Annual Allerton Conf. on Communication, Control, and Computing,* University of Illinois (Urbana-Champaign), pages 578–584, 1978.

[FW78e] Daniel P. Friedman and David S. Wise. Unbounded computational structures. *Software-Practice and Experience* **8**, 4, pages 407–416, July-August 1978.

[FW79] Daniel P. Friedman and David S. Wise. An approach to fair applicative multiprogramming. In G. Kahn and R. Milner, editors, *Semantics of Concurrent Computation*, pages 203–225. Berlin, Springer, 1979.

[FW80a] Daniel P. Friedman and David S. Wise. A conditional, interlock-free store instruction. Technical Report 74, Indiana University Computer Science Department, Bloomington, Indiana, 1980. (revised 1980).

[FW80b] Daniel P. Friedman and David S. Wise. An indeterminate constructor for applicative multiprogramming. In *Record 7th ACM Symp. on Principles of Programming Languages* (January, 1980), pages 245–250, 1980.

[FW81] Daniel P. Friedman and David S. Wise. Fancy ferns require little care. In S. Holmstrom, B. Nordstrom, and A. Wikstrom, editors, *Symp. on Functional Languages and Computer Architecture,* Lab for Programming Methodology, Goteborg, Sweden, 1981.

[GG88] Ron Goldman and Richard P. Gabriel. Preliminary results with the initial implementation of qlisp. In *ACM Symposium on Lisp and Functional Programming*, pages 143–152, July 1988.

[GM84] Richard P. Gabriel and John McCarthy. Queue-based multi-processing lisp. In *ACM Symposium on Lisp and Functional Programming*, pages 25–43, August 1984.

[GP81] Dale H. Grit and Rex L. Page. Deleting irrelevant tasks in an expression-oriented multiprocessor system. *ACM Transactions on Programming Languages and Systems*, 3(1):49–59, January 1981.

[Hal87] Cordelia Hall. *Strictness analysis applied to programs with lazy list constructors*. PhD thesis, Indiana University Computer Science Department, 1987.

[Hem85] David Hemmendinger. Lazy evaluation and cancellation of computations. In *IEEE International Conference on Parallel Processing*, pages 840–842, August 1985.

[HJBS91] Alan J. Demers Hans-J. Boehm and Scott Shenker. Mostly parallel garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–175, June 1991.

[HS86] Paul Hudak and L. Smith. Para-functional programming: a paradigm for programming multiprocessor systems. In *ACM Symposium on Principles of Programming Languages*, pages 243–254, January 1986.

[HS88] Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely functional i/o systems. Technical report, Yale, December 1988. Yale Technical Report YALEU/DCS/RR-665.

[Hud86] Paul Hudak. Para-functional programming. *Computer*, 19(8):60–71, August 1986.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[HW87] Cordelia Hall and David S. Wise. Compiling strictness into streams. In Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Munich, West Germany, January 21–23*, pages 132–143, 1987.

[HW92] Brian C. Heck and David S. Wise. An implementation of an applicative file system. In Y. Bekkers and J. Cohen, editors, *Memory Management*, pages 248–263. Lecture Notes in Computer Science 637, Springer-Verlag, 1992.

[JB88] Steven D. Johnson and C. David Boyer. Modeling transistors applicatively. In Jr. George J. Milne, editor, *Fusion of Hardware Design and Verification*. North-Holland, 1988. (Proceedings of the IFIP WG10.2 working conference on formal aspects of VLSI, Strathclyde University, Glasgow, July, 1988).

[JBB87] Steven D. Johnson, Bhaskar Bose, and C. David Boyer. A tactical framework for digital design. In Graham Birtwistle and P. A. Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–384. Kluwer Academic Publishers, 1987. Proceedings of the 1987 Calgary Hardware Verification Workshop.

[Jes] Eric R. Jeschke. Dsi assembler reference manual. (draft, unpublished).

[JK81] Steven D. Johnson and Anne T. Kohlstaedt. Dsi program description. Technical Report 120, Indiana University Computer Science Department, Bloomington, Indiana, 1981.

[Joh77] Steven D. Johnson. An interpretive model for a language based on suspended construction. Master's thesis, Indiana University Computer Science Department, 1977.

[Joh81] Steven D. Johnson. Connection networks for output-driven list multiprocessing. Technical Report 114, Indiana University Computer Science Department, Bloomington, Indiana, 1981.

[Joh83] Steven D. Johnson. Circuits and systems: Implementing communication with streams. *IMACS Transactions on Scientific Computation, Vol. II*, pages 311–319, 1983.

[Joh84a] Steven D. Johnson. Applicative programming and digital design. In *Eleventh Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 218–227, January 1984.

[Joh84b] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. The ACM Distinguished Dissertation Series, The MIT Press, Cambridge, MA, 1984.

[Joh85] Steven D. Johnson. Storage allocation for list multiprocessing. Technical Report 168, Indiana University Computer Science Department, Bloomington, Indiana, March 1985.

[Joh86] Steven D. Johnson. Digital design in a functional calculus. In G. J. Milne and P. A. Subrahmanyam, editors, *Workshop on Formal Aspects of VLSI Design (Proceedings of the Workshop on VLSI, Edinburgh)*, 1985. North-Holland, Amsterdam, 1986.

[Joh89a] Steven D. Johnson. Daisy, dsi, and limp: architectural implications of suspending construction. Technical report, Indiana University Computer Science Department, Bloomington, Indiana, 1989.

[Joh89b] Steven D. Johnson. *Daisy Programming Manual*. Indiana University Computer Science Department, Bloomington, Indiana, second edition, 1989. (draft in progress, available by request).

[Joh89c] Steven D. Johnson. How daisy is lazy. Technical report, Indiana University Computer Science Department, Bloomington, Indiana, 1989.

[Joh90] Douglas Johnson. Trap architectures for lisp systems. In *ACM Conference on Lisp and Functional Programming*, pages 79–86, 1990.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[JP92a] Suresh Jagannathan and Jim Philbin. A customizable substrate for concurrent languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 55–67, 1992.

[JP92b] Suresh Jagannathan and Jim Philbin. A foundation for an efficient multi-threaded scheme system. In *ACM Conference on Lisp and Functional Programming*, pages 345–355, 1992.

[Kie86] Richard B. Kieburtz. Incremental collection of dynamic, list-structure memories. Technical report, Oregon Graduate Center, January 1986. Technical Report CS/E-85-008.

[Koh81] Anne T. Kohlstaedt. Daisy 1.0 reference manual. Technical Report 119, Indiana University Computer Science Department, Bloomington, Indiana, 1981. ([Joh89a] should be delivered implicitly).

[KS88] L.V. Kale and Wennie Shu. The chare-kernel language for parallel programming: A perspective. Technical report, University of Illinois, August 1988.

[KW90] Morry Katz and Daniel Weisse. Continuing into the future: On the interaction of futures and first-class continuations. In *ACM Conference on Lisp and Functional Programming*, pages 176–184, 1990.

[Lar91] James R. Larus. Compiling lisp programs for parallel execution. *Lisp and Symbolic Computation*, 4:29–99, 1991.

[LD88] Yue-Sheng Liu and Susan Dickey. Simulation and analysis of different switch architectures for interconnection networks in mimd shared memory machines. Technical report, Courant Institute of Mathematical Sciences, June 1988. Ultracomputer Note #141.

[Liv88] Brian K. Livesey. The aspen distributed stream processing environment. Technical report, UCLA, December 1988. UCLA Technical Report CSD-880102.

[Mil87] James S. Miller. *Multischeme: A Parallel Processing System based on MIT Scheme*. PhD thesis, September 1987. Also available as MIT Technical Report MIT/LCS/TR-402.

[Moh90] Eric et. al. Mohr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *ACM Conference on Lisp and Functional Programming*, pages 185–197, 1990.

[Moo84] David Moon. Garbage collection in a large lisp system. In *ACM Conference on Lisp and Functional Programming*, pages 235–346, August 1984.

[MRSL88] Robert R. Kessler Mark R. Swanson and Gary Lindstrom. An implementation of portable standard lisp on the bbn butterfly. In *ACM Symposium on Lisp and Functional Programming*, pages 132–141, August 1988.

[O'D] John T. O'Donnell. An applicative programming environment. (draft, unpublished).

[O'D85] John T. O'Donnell. Dialogues: a basis for constructing programming environments. In *1985 ACM SIGPLAN Symposium on Programming Languages and Programming Environments*, in *ACM SIGPLAN Notices*, Vol. 20, No. 7, July 1985.

[O'D87] John T. O'Donnell. Hardware description with recursion equations. In *Proc IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications [CHDL]*, 1987.

[OH87] John T. O'Donnell and Cordelia Hall. Debugging in applicative languages. Technical Report 223, Indiana University Computer Science Department, Bloomington, Indiana, June 1987. To appear in the *International Journal on Lisp and Symbolic Computation.*

[OK92] et. al. O. Kaser. Fast parallel implementations of lazy languages: The equals experience. In *ACM Conference on Lisp and Functional Programming*, pages 335–344, 1992.

[Osb90] Randy B. Osborne. Speculative computation in multilisp–an overview. In *ACM Conference on Lisp and Functional Programming*, pages 198–208, 1990.

[PCYL87] Nian-Feng Tzeng Pen-Chung Yew and Duncan H. Lawrie. Distributed hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, c-36(4):388–395, April 1987.

[PRWM92] Michael S. Lam Paul R. Wilson and Thomas G. Moher. Caching considerations for generational garbage collection. In *ACM Conference on Lisp and Functional Programming*, pages 32–42, June 1992.

[Rep91] John H. Reppy. Cml: A higher-order concurrent language. In *ACM Symposium on Programming Language Design and Implementation*, pages 293–305, 1991.

[RHH85] Jr. Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[RHH86] Jr. Robert H. Halstead. An assessment of multilisp: Lessons from experience. *International Journal of Parallel Programming*, 15(6):459–500, 1986.

[RMKT84] Frank C.H. Lin Robert M. Keller and Jiro Tanaka. Rediflow multiprocessing. In *IEEE COMPCON*, pages 410–417, 1984.

[SH91] Peter Steenkiste and John Hennessy. Tags and type checking in lisp: Hardware and software approaches. In *ACM Conference on Architectural Support for Programming Languages and Systems*, pages 50–59, April 1991.

[Sin91] Satnam Singh. Using xview/x11 from miranda. In *Workshops in Computing, Functional Programming*, pages 353–363, 1991.

[TK88] Pete Tinker and Morry Katz. Parallel execution of sequential scheme with paratran. In *ACM Symposium on Lisp and Functional Programming*, pages 28–39, January 1988.

[Tra84] Kenneth R. Traub. An abstract architecture for parallel graph reduction. Technical report, MIT, September 1984. Also available as MIT Technical Report MIT/LCS/TR-317.

[Tra91] Kenneth R. Traub. *Implementation of Non-Strict Functional Programming Languages.* MIT Press, 1991.

[Veg84] Steven R. Vegdahl. A survey of proposed architectures for the execution of functional languages. *IEEE Transactions on Computers*, c-33(12):1050–1071, December 1984.

[WF87] David S. Wise and John Franco. Costs of quadtree representation of non-dense matrices. Technical Report 229, Indiana University Computer Science Department, Bloomington, Indiana, October 1987.

[Wis81] David S. Wise. Compact layout of banyan/fft networks. In H. Kung, B. Sproull, and G. Steele, editors, *VLSI Systems and Computations*, pages 186–195. Computer Science Press, Rockville, MD, 1981.

[Wis84a] David S. Wise. Parallel decomposition of matrix inversion using quadtrees. In *Proc. 1984 International Conference on Parallel Processing*, pages 92–99, 1984. (available as IEEE Cat. No. 86CH2355-6).

[Wis84b] David S. Wise. Representing matrices as quadtrees for parallel processors. *ACM SIGSAM Bulletin* **18**, 3, pages 24–25, August 1984. (extended abstract).

[Wis85a] David S. Wise. Design for a multiprocessing heap with on-board reference counting. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 289–304. Springer-Berlin, 1985.

[Wis85b] David S. Wise. Representing matrices as quadtrees for parallel processors. *Information Processing Letters* **20**, pages 195–199, May 1985.

[Wis86a] David S. Wise. An applicative programmer's approach to matrix algebra, lessons for hardware, and software. In *Workshop on Future Directions in Computer Architecture and Software,* Army Research Office, 1986.

[Wis86b] David S. Wise. Parallel decomposition of matrix inversion using quadtrees. In *Proc. 1986 IEEE Intl. Conf. on Parallel Processing.* IEEE, 1986.

[Wis87] David S. Wise. Matrix algebra and applicative programming. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **274**, pages 134–153. Springer-Berlin, 1987.

[Zor90] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *ACM Conference on Lisp and Functional Programming*, pages 87–98, 1990.

INDEX

Curriculum Vitae

Eric Jeschke was born on December 8, 1962 and grew up in Goshen, Indiana. He attended Indiana University in Bloomington, Indiana, receiving three degrees in Computer Science; a B.S. in 1984, a M.S. in 1986 and a Ph.D. in 1995.