PARALLEL FUNCTIONAL PROGRAMMING FOR MESSAGE-PASSING MULTIPROCESSORS

Gerald Ostheimer

Parallel Functional Programming for Message-Passing Multiprocessors

Thesis submitted for the degree of Doctor of Philosophy at the

University of St. Andrews

by

Gerald Ostheimer

Division of Computational Science

Department of Mathematical and Computational Sciences

University of St. Andrews

St. Andrews, Fife, KY16 9SS

March 1993

Th B352

Declaration

I, Gerald Kuno Ostheimer, hereby certify that this thesis has been composed by myself, that it is a record of my own work, and that it has not been accepted in partial or complete fulfilment of any other degree or professional qualification.

Signed Date 31 March 1993

I was admitted to the Faculty of Science of the University of St. Andrews under Ordinance General No 12 on 1st October, 1988 and as a candidate for the degree of Ph.D. on 1st October, 1989

Signed Date 31 March 1993

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate to the Degree of Ph.D.

Signature of Supervisor Date 31/3/93

Copyright

Abstract

We propose a framework for the evaluation of implicitly parallel functional programs on message passing multiprocessors with special emphasis on the issue of load bounding. The model is based on a new encoding of the λ-calculus in Milner's π-calculus and combines lazy evaluation and eager (parallel) evaluation in the same framework. The π-calculus encoding serves as the specification of a more concrete compilation scheme mapping a simple functional language into a message passing, parallel program. We show how and under which conditions we can guarantee successful load bounding based on this compilation scheme. Finally we discuss the architectural requirements for a machine to support our model efficiently and we present a simple RISC-style processor architecture which meets those criteria.

Acknowledgments

Many people have had profound influence on this thesis and I want to pay tribute to some of them here.

To my supervisor, Tony Davie, for his willingness to supervise what started out as a thesis on computer architecture, for always being available for a quick word, and for giving me the freedom to follow my own way, however twisted.

To Peter Burgess, for putting up with my quirks during two years of sharing an office, for sharing his experience on distributed systems, for his willingness to listen to half-baked ideas and for talking me out of the more silly ones.

To Ron Morrison, for his encouragement of my work when he could see it taking shape.

To Arvind, for instilling in me a fascination with non-strictness and parallel architectures.

To David Gifford, for introducing me to the beauty of functional programming, and for demonstrating that lectures can be at once fun and challenge.

To John Glauert, for pointing out the π-calculus to me at just the right moment.

To Bill Campbell, for guiding my life's journey to the beautiful town of St. Andrews.

To Kathy Hargreaves, whose competent editing advice on my MSc thesis helped me no end with this larger undertaking.

To my many friends, whose confidence in me always seemed greater than my own.

Finally, to my parents for their unquestioning love and support, and especially to my father, for his living example of the spirit behind that old saying, "a job worth doing is a job worth doing well."

Table of Contents

Chapter 1 — Introduction

1.1 Parallel computing and functional programming

Fifteen years after John Backus' clarion call to break the von Neumann bottleneck by adopting a functional programming style [Ba78], commercial multiprocessors are now available whose degree of parallelism is limited only by the size of a customer's wallet. Yet, there is little agreement on how best to program those machines and functional programming is not even among the leading contenders. While real speedups have been achieved for parallel functional programs on shared-memory multiprocessors with a limited degree of parallelism [AJ89], an effective *scalably* parallel solution is still outstanding. We hope that the work presented here will be an important step towards the goal of scalable parallel functional computing. We begin this introductory chapter by summarising our main results. We then state our perspective of the role and the historical development of parallel functional computing and show how our work ties in with previous developments. We conclude the chapter with a 'road map' to the rest of the thesis.

1.2 Contribution of this thesis

Central to this thesis is a framework for the evaluation of implicitly parallel functional programs on message-passing multicomputers. Our model is based on a new encoding of the λ-calculus in Milner's π-calculus. Milner has given two such encodings in [Mi92], a sequential one with call-by-name parameter passing semantics and a parallel one with call-by-value parameter passing semantics. Ours combines the advantages of both in the same framework. We can choose individually for each function application between eager and lazy evaluation based on the results of strictness analysis.

We are thus able to preserve non-strict semantics and at the same time exploit parallelism where safely possible. Our π-calculus encoding serves as the specification of a more concrete compilation scheme mapping a simple functional language into a message passing, parallel program by exploiting the implicit parallelism of functional programs. Since implicit parallelism gives rise to the problem of the *parallelism explosion* we give special emphasis to the issue of automatic load bounding. Without load bounding, the space requirements of some programs can change from linear to exponential. Our simple adaptive algorithm is integrated into the compilation scheme. We give an informal proof for the effectiveness of this algorithm, based on the structure of the compilation scheme, and discuss the limitations of our method. We discuss the architectural requirements for a machine to support our model efficiently and present the *STAR:DUST* architecture, short for 'St. Andrews RISC: Dataflow Using Sequential Threads', designed to meet those criteria. A STAR:DUST node is a simple RISC processor with two new instructions supporting message passing and task switching. We have simulated a multiprocessor STAR:DUST machine on a Meiko Computing Surface and have obtained experimental results based on this simulator.

1.3 Why functional programming matters

The virtues of programming in a functional style have been addressed often and eloquently enough for us not to dwell too long on this subject. In short, functional languages lend themselves to a declarative style of programming, often permitting a literal translation of a mathematical specification into a functional program. Many algorithms can be expressed in a succinct and intuitive manner unmatched by the prevalent imperative languages (see Hughes [Hu89]). This is due mostly to non-strict semantics which guarantees the termination of a larger class of programs. There is an 'algebra of functional programs' in the sense of Backus [Ba78] with a range of correctness-preserving transformations that facilitates correctness proofs and the derivation of efficient programs from specifications. The interested reader is referred to work by Darlington [Da82] and Bird [Bi89] for a more comprehensive exposition of these features. Of special interest for our work is the fact that functional languages do not force the programmer to impose a total order on program execution. Referential transparency ensures that only data dependencies constrain the order of evaluation. The resulting freedom can be exploited by systems like ours to extract parallelism from programs not necessarily written with parallel execution in mind. This is not to say, however, that writing parallel functional programs is a trivial task. While a compiler can easily extract parallelism from a *particular* functional program, different algorithms for the same problem may exhibit different amounts of parallelism, so good parallel programs will still require a special effort on the part of the programmer.

1.4 What is holding us back?

Only in relatively few cases—certainly when compared to the size of the present-day software industry—has functional programming had any impact on the practical solution of real world problems. This fact can not just be blamed on the inertia of practitioners who haven't been exposed to functional programming during the course of their training. Functional programming languages have long been suffering from a number of deficiencies. Chief among them is the difficulty of expressing non-trivial forms of I/O and, more generally, expressing many algorithms that are intuitively based on the notion of state. The nondeterminism of functional languages with respect to execution order, which is so beneficial to us for the purpose of parallelisation, also has its downside. Visualising the evaluation of a non-trivial functional program under a lazy or a parallel execution regime is exceedingly hard. Convential debugging techniques fail. In particular, the notion of 'stepping through a program' has little meaning in a functional context. For similar reasons, classical complexity analysis does not readily apply to functional programs. Finally, despite impressive progress that has been made with 'compiled graph reduction' originating with Johnsson [Jo84], functional programming systems are still behind their imperative cousins in terms of efficiency when measured on real-world algorithms. We will not enter into a discussion of the various reasons for this handicap. Let it suffice to say that one of the requirements for the design of Fortran, generally considered to be the first 'high-level' programming language, was to map to machine languages with minimal loss of efficiency. Recently this close relationship has been reinforced with the design of RISC microprocessors as 'C' machines, as described by Patterson and Sequin in [PS81].

1.5 'Functional' architectures

Since the earliest days of functional programming languages, their potential for parallelism has been recognised [Bu75]. At times, their inherent parallelism was presented as a means to catch up with imperative languages in the benchmark race. This expectation is, of course, to be taken with a large grain of salt. The competition for a parallel functional program is a *parallel* imperative one, thus moving the goalposts. However, there is some hope that parallel functional programming will indeed one day gather a larger following. The primary reason for this expectation is the enormous complexity of any but the most simple-minded imperative models for parallel programming. Reasoning about the correctness of a message-passing program, i.e., comprehending it, involves demonstrating its freedom of deadlock and starvation. See, for example Brookes and Roscoe [BR91]. Non-determinism is also a serious problem, especially when it comes to debugging. Functional programs, on the other hand, are deterministic by definition and have no need for special new 'parallel constructs' which could affect correctness.

Most of the initial work on parallel functional programming centered around various proposals for computer architectures specially designed to support functional programming, e.g., the FFP machine [Ma79], the Manchester dataflow machine [GW78], the MIT tagged token data flow machine [AK80] and ALICE [DR81]. A comprehensive survey can be found in [Ve84]. The reason for this focus on computer architecture was twofold. In the first place there simply were no parallel architectures available commercially at the time. The first research multiprocessor, the Illiac IV had only become fully operational in 1975 [Fa76], while the first commercial multiprocessor built according to scalable design principles, the Denelcor HEP, did not arrive until 1981 [Sm81, HB84]. But there was a second reason. Since imperative languages were seen as being wedded to von Neumann machines, the search was on for a 'non-von Neumann' machine which would execute functional programs in a kind of 'native mode'. With the benefit of hindsight, this search can safely be said to have failed. While some of the architectural projects did produce very interesting results (among them the hardware mechanism for load bounding by Ruggiero and Sargeant [RS87] inspiring our software solution, see Chapter 2), few of the proposed architectures were constructed and none of them was commercially successful. To our knowledge only the MIT project is still continuing as an architectural effort after undergoing a major transformation back towards a modified von Neumann model (see below). Several factors contributed to thwart the early hopes for 'functional architectures'. Principal among them is the problem of excessively fine granularity. The necessary overhead for managing and synchronizing a large number of small tasks often dwarfed any benefit that could be obtained from parallelism. Secondly, mainstream computer architecture did not stand still. Speed improvements for sequential microprocessors have continued at an enormous rate even to the present day. High-bandwidth, low latency networks were developed for connecting large numbers of conventional von Neumann microprocessors [DS87, Le92]. Interface technology has been advanced tying processors closely to networks, e.g., for the transputer [Wh85], the J-Machine [DC89], the CM-5 [Le92]. See also [HJ92] for a recent proposal. Finally, it has now been recognized that economies of scale will favour, for the forseeable future, parallel machines made out of large numbers of cheap von Neumann style microprocessors over special-purpose parallel processor designs.

1.6 Current work

Work on parallel functional programming has not stopped, however. Current activities can be broadly divided into two classes. On the architectural side efforts have been directed towards a rapprochement with von Neumann microprocessor technology. In the case of the MIT dataflow project this was a gradual process: from the fine-grain, tagged-token dataflow architecture Monsoon [PC90], via Iannucci's hybrid architecture [Ia88] and P-RISC [AN88], the MIT work has now progressed to the design of a coprocessor for the Motorola 88100, the *T (pronounced 'Start') project [NP92]. The GRIP project [PC87] at Glasgow ('Graph Reduction in Parallel') has been designed from the outset around a commercial microprocessor, the Motorola 68020.

The bulk of ongoing work, however, now concentrates its efforts on attempts to support the various forms of parallel architectures that have been developed for different programming models. Shared-memory machines are the target of work that has been done at Chalmers by

Augustsson and Johnsson [AJ89]. The FAST project at Imperial College and Southampton [GH90] supports networks of communicating processes such as those constructed with transputers. Hudak and his colleagues have implemented parallel graph reduction on an Intel hypercube as part of their work on the Alfalfa project [GH87]. Work by Cole at Glasgow [Co90] and Darlington at Imperial College on algorithmic skeletons is somewhat broader in scope. They propose writing parallel functional programs in terms of generic functions which would be implemented efficiently on a host of different architectures, spanning parallel models from SIMD to MIMD. For all their diversity, these projects share certain characteristics. In order to take advantage of parallelism, programmers need to use annotations of some sort to specify where parallelism is to be obtained. Consequently programmers are also responsible for bounding parallelism to sustainable levels. Furthermore, only coarse-grain parallelism can typically be exploited with any efficiency. The resulting programs often become non-portable and non-scalable. We will discuss some of these projects in more detail in Chapter 2.

1.7 Where we fit in

Our fundamental interest is in implicit parallelism for scalable computers. In this section we will state the problems we set out to address and discuss the motivations behind our research. We will indicate how those motivations arose from previous work in the field.

1.7.1 Concurrency in functional programming systems and the π-calculus

David Turner's popularisation of combinators [Tu79] provided the functional programming community with a beautiful theory for defining the semantics of program execution in terms of combinator reduction. At the same time his work resulted in an implementation technique which has since been refined to compiled graph reduction, now universally considered the most efficient method of evaluating functional programs on off-the-shelf sequential machines. Many researchers have now undertaken to apply the basic model of graph reduction to parallel evaluation as well. This is possible since the graph reduction model preserves the parallelism implicit in functional languages. Obviously the implicit parallelism needs be made explicit at some stage, either during compilation or during evaluation. This is typically done in an ad-hoc manner suitable to the kind of parallel machine supported by a particular project. It is our contention that it is both useful and possible to elevate the process of 'making parallelism explicit' to a higher level in the design of a parallel functional language system. What is needed is a proper theory of concurrency in parallel functional programming. In the π-calculus [Mi92] Robin Milner provides us with the necessary tools for such a theory, demonstrated by his two encodings of the λ-calculus in the π-calculus. We extend his work to modelling a functional programming system which exploits parallelism *and* preserves non-strictness, taking up a theme from [Mi92].

> *Thus, strategies which appear natural in the presence of textual substitution may not seem so natural in a model involving* ***autonomous agents****. The former have clearly been most deeply studied in research on the λ-calculus; one effect of providing π-calculus as a substrate may be to intensify the study of other strategies, such as those with* ***shared reductions****.* (Emphasis ours)

We would like to note that we approached the problem strictly from an implementation point of view and have been made aware of the π-calculus only at a late stage of the work described here. In particular, our compilation scheme for message-passing multicomputers predates our π-calculus encoding for the λ-calculus which is best viewed as a crystallisation of the main ideas.

1.7.2 Automatic load bounding

Consider the 'nfib' program, a program so trivial that it is now almost considered bad taste to publish performance figures based on it:

```
nfib 0  =  1

nfib 1  =  1

nfib n  =  1 + nfib (n-1) + nfib (n-2)
```

'nfib' is strict throughout, contains neither fancy data structures nor higher order functions and offers vast amounts of parallelism of a very simple kind. Any functional implementation, so the reasoning goes, should do well on it. Yet in the context of implicit parallelism 'nfib' poses a difficult problem. Sequential execution of 'nfib' corresponds to a depth-first traversal of the tree-structured program graph. Therefore sequential 'nfib' requires space proportional to the maximum depth of recursion, i.e., it runs in linear space. Maximally parallel evaluation of 'nfib', however, corresponds to breadth first traversal and worst case space requirements are proportional to the size of the program graph, i.e., we now require exponential space. Even relatively small instances of parallel 'nfib' can exceed machine resources. At the same time we obtain no benefit from exploiting parallelism far in excess of the hardware parallelism at our disposal. So clearly we are interested in *bounding* the amount of parallelism we will uncover. This problem was anticipated by Burton and Sleep in [BS81] where the basic idea for an adaptive solution is also outlined: follow a breadth-first evaluation strategy while the machine is underloaded, perform depth-first evaluation while the machine is saturated.

The 'parallelism explosion' was observed experimentally in the dataflow community and reported in [RS87, CA88]. Our work is closely related to that of Ruggiero and Sargeant who have implemented a hardware load *throttle* as part of their work on the Manchester dataflow machine. We are picking up an idea proposed in [RS87]

> *Another idea for dynamic software throttling is to plant two types of code for any parallel program: one serial and the other parallel. The machine switches from one style of code to the other at run time, according to how busy it is.*

However, they go on to add

> *Although this method could be useful in the future, achieving a complete solution in this way is well beyond the state of the art. The conclusion is that despite being useful, software techniques are not enough to implement a general and effective throttle. We need some help from the hardware.*

In this paper we present a software only solution with little overhead—in particular, we do not require expensive scheduling hardware. In addition, our work is novel in the following ways:

- We integrate our load bounding method into a simple compilation scheme for lazy functional languages.
- We present an informal proof of the effectiveness of our scheme based on the structure of the generated code.
- We identify sharing as a possible source of problems for load bounding.

1.7.3 Granularity

Consider the following simple functional program

```
        let  x = a*b;

             y = 4*c

        in   (x+y) * (x-y)
```

and its dataflow graph (Figure 1.1). This example has been taken (in slightly simplified form) from [AC86] where it is used to demonstrate the instruction level parallelism implicit in many functional programs (or dataflow programs). In the example, *x* and *y* can be computed concurrently, and so can the addition and the subtraction.

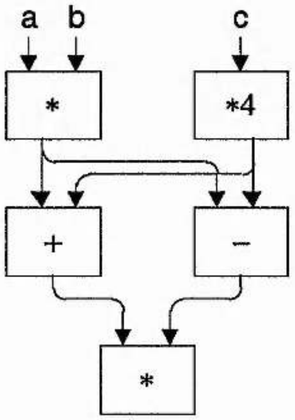

Figure 1.1: Instruction level parallelism

The original dataflow machine designs attempted to exploit instruction-level parallelism of this form as a special case of the general method. Experience with completed dataflow hardware has demonstrated, however, that the overheads of dispatching tiny parcels of work and synchronizing their results easily outweighs any benefits to be obtained from such parallelism. In fact, recent designs for superscalar (sequential) microprocessors have managed to exploit instruction-level parallelism much more effectively, see for example [DA92, MW92]. For this reason, much of the recent work on parallel functional programming has focussed on restricting the exploitation of functional parallelism to the coarse-grain type.

Since the 'right' level of granularity depends largely on architectural parameters, it is hard to identify while parallel computer architecture is a quickly moving target. We consider the trend to coarse-grain parallelism within the functional community to be an overcompensation for the previous focus on excessively fine-grain parallelism. Our working hypothesis is that it will be possible to support parallelism efficiently on the inter-function level (we do not attempt to exploit parallelism within function bodies). The trend towards closer integration of network interfaces and high-bandwidth networks [EC92, HJ92] permits more efficient synchronisation than previously considered feasible. In Chapter 6 we present a model RISC architecture with a tightly integrated network interface [Os91]. STAR:DUST was heavily influenced by Nikhil's P-RISC architecture [AN88] described in the next chapter.

We restrict ourselves to distributed memory, message passing machines as those now appear set to prevail among massively parallel architectures. This trend is exemplified by the development history of the Connection Machine, originally a pioneering SIMD architecture, which was recently reborn in MIMD form [Le92].

1.8 Thesis outline

In this chapter we have outlined the main contributions of this thesis and the developments which preceded and motivated our work. In Chapter 2 we will have a closer look at some previous work important to ours. Chapter 3 contains a simulation of the λ-calculus in Milner's π-calculus which serves as a specification for the compilation scheme presented in Chapters 4 and 5. The former contains the basic set of compilation rules required for lazy functional programming with alternative 'eager' rules for exploiting parallelism where safely possible. In the latter we adapt the eager rules to tie in with an efficient distributed load bounding algorithm in order to limit the resource requirements of the resulting programs and we give an informal proof of the effectiveness of the resulting scheme. In Chapter 6 we detail our proposal for STAR:DUST, a model RISC architecture for fine-grain parallel computing. In Chapter 7 we describe our simulation of this architecture on a commercial multiprocessor and present some experimental results obtained from the simulator. We conclude in Chapter 8 with a brief summary and suggestions for further work.

Chapter 2 — Previous and Related Work

2.1 Contents

In this chapter we will give a summary of previous work of which we make use, previous work that we improve on, as well as current related work moving in different directions. Parallel functional programming is a large field and we necessarily have to restrict our attention to those contributions that we deem most relevant to our work. In outline, we will give some details about Milner's π-calculus as well as his encodings of the λ-calculus. We will discuss some aspects of the Manchester dataflow project, in particular their achievements on load bounding. We review the development of the MIT dataflow work and discuss the design of the P-RISC architecture that grew out of it. We include a section on the first parallel graph reduction machine, ALICE, and its successor, Flagship. We discuss the design of the GRIP architecture, a major project in parallel functional programming which includes a strong architectural component. We conclude with a comparison of our load bounding approach to that of 'lazy task creation'.

2.2 Functional processes: simulating the λ-calculus in the π-calculus

The π-calculus is the result of a search for a algebraic framework which would capture the essence of the notion of concurrent processes. It is particularly suitable for the description of systems which can change their configuration dynamically. The π-calculus improves on Milner's previous work on CCS, the *calculus of communicating systems* [Mi80], in that the former needs no recourse to a universe of values outside its own scope. In terms of internal completeness and conciseness it is comparable to the λ-calculus which was a guiding paradigm in the design of the π-calculus. See Milner's Turing Award Lecture [Mi93] for an excellent discussion of the motivations that lead to the π-calculus. This section will be important to the rest of the thesis in two ways: we will require an understanding of the operational behaviour of the π-calculus in Chapter 3, where we use it to specify a non-strict parallel simulator for the λ-calculus. Secondly we will present a brief outline of two such simulations given by Milner in [Mi92], one of them non-strict but sequential, the other parallel but strict.

2.2.1 The π-calculus: a simple example

Rather than getting immediately bogged down in notation and detail, let us start by presenting a simple π-calculus term, the simplest term which admits reduction:

$$\bar{x}z \mid x(y)$$

This term is to be read as follows: write z to x and, in parallel, read y from x. Reduction is possible since there is a reader and a writer ready to communicate via the same channel, namely x. Communication takes place by cancelling read- and write-actions and substituting the value to be written (z in the example) for any free occurrence in the reading process of the variable to be read (y in the example). As y does not occur free anywhere, reduction yields

$$\mathbf{0} \mid \mathbf{0}$$

which is equivalent to $\mathbf{0}$, the empty process. A slightly more useful example is the term

$$\bar{x}z \mid x(y).\bar{x}y$$

Here the read-action $x(y)$ is followed by a write-action $\bar{x}y$. Reduction yields

$$\mathbf{0} \mid \bar{x}z$$

after cancelling matching read/write-actions and substituting z for the single free occurrence of y.

2.2.2 The π-calculus: syntax and reduction behaviour

We are now ready for a more systematic presentation of the π-calculus, adhering to the form given in [Mi92]. The π-calculus consists of a set of terms which intuitively stand for *processes*. The names of the π-calculus (x, y, z...) denote *channels* through which processes communicate. Channel names are also the only *subject* of communication. The syntax for π-calculus terms is summarised in Figure 2.1.

P	$::=$	$\overline{x}\,y.P$	write-action: write y to x and then P
P	$::=$	$x(y).P$	read-action: read y from x and then P
P	$::=$	$\mathbf{0}$	empty process
P	$::=$	$P_1 \mid P_2$	composition
P	$::=$	$(y)\,P$	restriction
P	$::=$	$!P$	replication

Figure 2.1: Syntax of the π-calculus

The terms of the π-calculus exhibit a simple block structure with two forms of name binding, namely read-action and restriction. In the following we will discuss each of the constructs in some more detail.

Write-action. A term of the form $P = \overline{x}\,z.Q$ represents a synchronous send-operation of a value z along a channel x. The process P cannot proceed until another process is ready to receive z on the same channel and will then continue with Q.

Read-action. Counterpart to a write-action. A process of the form $x(y).Q$ cannot proceed until another process is ready to send some value v along the channel x. The value v is then substituted for all free occurrences of y inside Q. The read-action is one of two ways of binding names, i.e., the scope of y is Q.

Empty process. We write $\mathbf{0}$ to represent the empty process which is necessary to ground our syntactic rules. We will always abbreviate action terms of the form $\bar{x}z.\mathbf{0}$ to $\bar{x}z$ (similarly for read-actions).

Composition. We write $P_1 \mid P_2$ to denote two processes P_1 and P_2 operating concurrently. Note that there is no separate construct for sequential composition. This can be modelled using parallel composition and suitable synchronisation via interaction.

Restriction. We write $(x)P$ to obtain a new channel name x which is private to P. The term $\bar{x}z \mid (x)\, x(y)$ has no reduction, as the sending process and the receiving process operate on different channels. Restriction is the second form of name binding, with P as the scope of x.

Replication. A term of the form $!P$ stands for the parallel composition of as many instances of the term P 'as necessary'. Replication can be 'unwound' according to the structural equivalence $!P \equiv P \mid !P$. Unwinding is unnecessary when there are existing copies of P which have not participated in any interactions. Note that unwinding does *not* represent reduction but is a structural equivalence relation like, for example, α-conversion.

In addition to these basic constructs we will also use a number of shorthands as defined below:

$\bar{x}(y).P$	$\overset{\text{def}}{=}$	$(y)\,\bar{x}y.P$	write a new channel y to x
$x(y)(z).P$	$\overset{\text{def}}{=}$	$x(y).x(z).P$	multiple reads from x
$\bar{x}yz.P$	$\overset{\text{def}}{=}$	$\bar{x}y.\bar{x}z.P$	multiple writes to x

Note in the first rule how private names (y above) can be exported out of the scope of restriction via explicit communication. Note, finally, that all occurrences of (x) signify *bindings* for the channel name x with simple static scoping rules.

The basic reduction rule, already hinted at in our explanation of read- and write-actions, is the following: for a pair of processes

$$Q_1 = \bar{x} z.P_1 \quad \text{and} \quad Q_2 = x(y).\, P_2$$

we get the reduction

$$Q_1 \mid Q_2 \;\rightarrow\; P_1 \mid P_2 \,\{z/y\}.$$

Put in words, if Q_1 is ready to send a name z along the channel x and Q_2 is ready to receive a name along the same channel then they can interact. Interaction results in cancelling each of the two send/receive actions and substituting z for all occurrences of y inside P_2. The complete set of reduction rules describes in detail the distribution of reduction over restriction and replication constructs, omitted here for sake of brevity. They can be found in [Mi92] along with a complete definition of structural equivalence for terms.

2.2.3 A comparison of the π-calculus to the λ-calculus

It will be useful to compare the π-calculus with the λ-calculus both to gain further understanding of the π-calculus itself and to appreciate the task before us of simulating the latter with the former. At the most elementary level, both calculi are term rewriting systems with a simple basic reduction rule. Both have simple static scoping rules for names. Neither provides any computational 'sugar' but both of them are computationally complete (for the π-calculus this will follow from the fact that it can simulate the λ-calculus). In the λ-calculus, the basic concept is that of a function: terms represent functions and names denote functions. Functions are first-class objects. In the π-calculus, there are two basic entities, processes and channels. Terms represent processes, but processes are not first-class in that the names of the π-calculus stand for channels only and channels are the only subject of of communication. There is a superficial correspondence between β-reduction in the λ-calculus and communication in the π-calculus in that both involve the substitution of an 'actual parameter' for a 'formal parameter'. However, while the term providing the actual parameter in a β-reduction becomes merged with the abstraction term, both reader and writer processes engaging in a π-calculus communication continue as independent agents after interacting. Studying the two reduction mechanisms closely we observe that β-reduction is the more complex of the two, involving as it does *terms* as actual parameters rather than 'atomic' channel *names*. On the other hand, identifying a redex in the λ-calculus is a simple 'local' syntactic operation whereas reduction in the π-calculus involves identifying two matching redexes, a reader and a writer, which can occur anywhere within a term (see the example below). A final and fundamental difference between the two calculi is that normal forms for λ-calculus terms are unique, whereas the π-calculus is nondeterministic. Consider for example the two reductions

$$x(y) \mid \bar{x}z_1 \mid \bar{x}z_2 \quad \rightarrow \quad \mathbf{0} \mid \mathbf{0} \mid \bar{x}z_2 \equiv \bar{x}z_2$$

$$x(y) \mid \bar{x}z_1 \mid \bar{x}z_2 \quad \rightarrow \quad \mathbf{0} \mid \bar{x}z_1 \mid \mathbf{0} \equiv \bar{x}z_1$$

Each yields a different normal form, depending on which send-action succeeds.

2.2.4 Simulating the normal-order λ-calculus

The motivation behind simulating the λ-calculus in the π-calculus is twofold. Firstly, it serves to demonstrates the power of the latter by relating it to its better-established cousin. Secondly, simulation of the λ-calculus is a useful application for the π-calculus, demonstrating its capability for expressing concurrency in λ-calculus reduction. In the first of two encodings for λ-calculus reduction, Milner presents a scheme for the normal-order (sequential) λ-calculus in [Mi92] from which the following is largely quoted. Note that even without any explicit sequencing combinator, the π-calculus is capable of expressing sequentiality by suitable synchronisation.

Each λ-calculus term M is encoded as $[[M]]$, a function which maps names to π-calculus terms. So $[[M]]\ u$ is a term of the π-calculus with the intuition that the name u is the link along which $[[M]]$ 'receives' its arguments. Now, suppose that M will itself be used in place of an argument represented by the variable x, i.e., x is bound to M. Each time M is 'called', via x, it must be told by the caller where to receive its own arguments. (In more familiar terminology, it must be given a *pointer* to its arguments). Thus the 'environment entry' binding x to M is a π-term defined as follows, with w representing the argument pointer(s).

$$[[x := M]] \quad \overset{\text{def}}{=} \quad !x(w).\ [[M]]\ w$$

Note the use of the replicator ! to allow for multiple references to the same environment entry*.

How does $[[\lambda x\ M]]\ u$ receive its arguments? Along u it receives (as x) the name of its first argument, and also the name of a link where the rest will be transmitted. This explains the first line of Milner's encoding, which we now give in full:

$$\begin{aligned} [[\lambda x\,.M]]\ u &\quad \overset{\text{def}}{=} \quad u(x)(v).\ [[M]]\ v \\ [[x]]\ u &\quad \overset{\text{def}}{=} \quad \bar{x}\,u \\ [[M\,N]]\ u &\quad \overset{\text{def}}{=} \quad (v)\,([[M]]\,v \mid \bar{v}(x).\ \bar{v}u.\ [[x := N]]\,) \end{aligned}$$

Let us look, with Milner, at an example. We assume x is not free in N.

$$[[(\lambda x.x)\ N]]\ u \quad \equiv \quad (v)\ (v(x)(w).\ [[x]]\ w \mid \bar{v}(x).\ \bar{v}u.\ [[x := N]]\,) \qquad (1)$$

* In using the notation $[[x := M]]$ for environment entries we follow [Mi92] but point out that it is distinct from the encoding function $[[M]]\ u$

$$\rightarrow \quad (v)(x)\,(v(w).\,[[x]]\,w \mid \bar{v}u.\,[[x:=N]]) \qquad (2)$$

$$\rightarrow \quad (x)\,([[x]]\,u \mid [[x:=N]]) \qquad (3)$$

$$\equiv \quad (x)\,(\bar{x}u \mid !x(w).\,[[N]]\,w) \qquad (4)$$

$$\rightarrow \quad [[N]]\,u \mid (x)\,[[x:=N]] \qquad (5)$$

$$\sim \quad [[N]]\,u \qquad (6)$$

The following remarks help us read the above calculation:

— In (1), we have expanded the definitions for lambda abstraction and function application

— In (2), we have communicated a new channel x along v. Since x now occurs in both parallel subprocesses, the restriction (x) has been moved outwards to cover both occurrences.

— In (3), the restriction (v) has been dropped because v no longer occurs

— The step to (4) represents the expansion of the definitions for identifier reference and environment entry

— In (5), (x) has been moved inwards as x now only occurs in the right subterm

— The last step, to (6), goes beyond simple equivalence and represents the garbage-collection of an environment entry $[[x := N]]$ which cannot be used further (since the subject x of the first action is restricted).

Milner also provides a proof that the reduction of $[[M]]$ in the π-calculus simulates that of M in the λ-calculus 'very closely'.

Reexamining the definition for environment entries $[[x := M]]$ we observe that M is evaluated completely for each new set of arguments w. So while the scheme outlined above faithfully simulates normal-order reduction in the λ-calculus, it takes no account of sharing and thus does not model graph reduction as first defined by Wadsworth in [Wa71].

2.2.5 Simulating the call-by-value λ-calculus

Since the π-calculus is suitable for expressing concurrency, we would expect it to be able to express the parallelism implicit in the λ-calculus. Milner does so in his second encoding which simulates the call-by-value λ-calculus. Under call-by-value semantics we completely reduce an argument term before passing its value to a function. In the new encoding $[[M]]\,p$, the name p will have a different significance. The reason is that two 'events' which coincided for the normal-order calculus must now be separated, namely

— the signal at p that M has reduced to a value (needed when M is the *argument* of an application);

— the receipt of arguments by an abstraction M (needed when M is applied).

Dealing with a call-by-value reduction strategy we will not need to pass 'apply nodes' as arguments, so our environment entries will now contain only values, i.e. abstractions and variables. So we begin by defining $[[y := V]]$ where V is either a lambda abstraction or a variable.

$$[[y := \lambda x.M]] \quad \overset{\text{def}}{=} \quad !y(v).\ v(x)(p).\ [[M]]\,p$$

$$[[y := x\,]] \quad \overset{\text{def}}{=} \quad !y(v).\ \overline{x}\,v$$

An environment entry for an abstraction keeps reading new sets of arguments v to which the abstraction is to be applied. For each set thus received, we pick up the first argument, x, and the rest of the arguments, p. We instantiate a new copy of M to which we pass the arguments p. The first argument x is bound to the free variable x in M implicitly. An environment entry for a variable simple passes on any sets of arguments to that variable.

The first action of a (translated) value, $[[V]]\,p$, is to signal its reduction to a value. The channel y representing the signal provides access to an 'environment entry'. Note that $[[y := V]]$ is here a *subterm* of $[[V]]\,p$, whereas the opposite was true in the normal-order encoding. In the most important difference, however, to the normal-order encoding, the new translation $[[M\ N]]\,p$ allows M and N to 'run' in parallel. The auxiliary definition $\mathbf{ap}(p, q, r)$ provides the necessary glue for relating functions to arguments: we pick up the value of M from q in the shape of an environment entry y and apply it to a new set of arguments v, which is constructed by extending the list of arguments p by the argument z computed by $[[N]]\,r$.

$$\begin{aligned} [[V]]\,p &\overset{\text{def}}{=} \overline{p}(y).\ [[y := V]] \qquad (y \text{ not free in } V) \\ [[M\,N]]\,p &\overset{\text{def}}{=} (q)(r)\,(\mathbf{ap}(p, q, r) \mid [[M]]\,q \mid [[N]]\,r) \\ \mathbf{ap}(p, q, r) &\overset{\text{def}}{=} q(y).\,\overline{y}(v).r(z).\,\overline{v}zp \end{aligned}$$

The example reduction sequence below for the λ-term $(\lambda x.M)V$ demonstrates the simulation of β-reduction.

$$\begin{aligned} [[(\lambda z.M)V]]\,p &\equiv (q)(r)\,(\mathbf{ap}(p, q, r) \mid \overline{q}(y).\ [[y{:=}\ \lambda z.M]] \mid \overline{r}(z).\ [[z{:=}\ V]]) \\ &\rightarrow (r)(y)\,(\overline{y}(v).r(z).\overline{v}zp \mid [[y{:=}\ \lambda z.M]] \mid \overline{r}(z).\ [[z{:=}\ V]]) && (1) \\ &\rightarrow (r)(y)(v)\,(r(z).\overline{v}zp \mid [[y{:=}\lambda z.M]] \mid v(z)(p).[[M]]\,p \mid \\ &\qquad \overline{r}(z).[[z{:=}V]]) && (2) \\ &\rightarrow (y)(v)(z)\,(\overline{v}zp \mid [[y{:=}\lambda z.M]] \mid v(z)(p).[[M]]\,p \mid [[z{:=}V]]) && (3) \\ &\rightarrow (y)(z)\,([[y{:=}\lambda z.M]] \mid [[M]]\,p \mid [[z{:=}V]]) && (4) \\ &\sim (z)\,[[M]]\,p \mid [[z{:=}V]]) && (5) \end{aligned}$$

Each of the four reduction steps consumes one of the actions defined by **ap**. The first step to (1) communicates the environment entry y for the functor $\lambda z.M$. Reduction (2) communicates the name v by which the functor can access its arguments. In step (3) we pick up the environment entry z for the argument V. Finally we make the extended argument set zp available via v.

The call-by-value nature of this scheme is apparent in the third and fourth actions of **ap**. The activation of a function call, performed by the action $\overline{v}zp$, cannot proceed until the argument to the function call signals its reduction to a value, detected by $r(z)$. While this simulation exhibits (some) parallelism in that for an application of the form $M\ N$ both subterms can be evaluated in parallel, the application itself cannot proceed until the argument has been computed. Thus the parallelism implicit in a non-strict constructor function like cons which could return a result even before its argument values are available is not exploited by this scheme. More seriously, Milner's call-by-value scheme obviously has strict semantics and will thus fail to terminate for many terms which have a normal form.

We shall give a third encoding, combining the advantages of parallelism and non-strictness, in the next chapter.

2.3 The Manchester dataflow project

Making somewhat of a conceptual jump, we move from the theoretical domain of the π-calculus to computer architecture related research. The connection will be established in Chapters 3 to 6, where we develop a π-calculus specification of parallel graph reduction into a practical compiler for a novel parallel architecture.

Work on the Manchester dataflow machine is interesting to us mainly for their results on load bounding. In order to put these results into context, however, it will be useful to present a brief overview of the architectural side of the Manchester dataflow research. The architecture in itself is interesting as a major early example of a non-von Neumann machine that has been successfully implemented in hardware. While the similarities of the Manchester dataflow machine to our STAR:DUST architecture may not be immediately obvious, the latter can trace back its origins to the former in a direct line. This lineage will be illustrated in detail in Section 2.4.

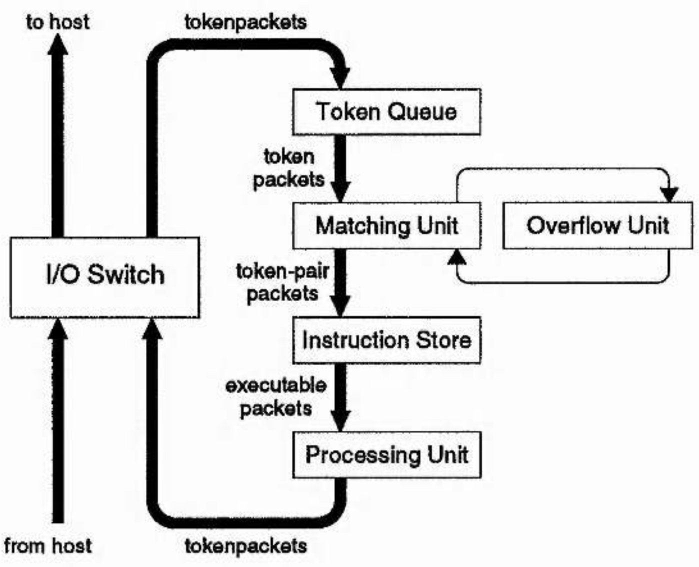

Figure 2.2: Structure of the Manchester dataflow system

2.3.1 Architectural overview

We will present the Manchester dataflow architecture by relating it to modern von Neumann processor designs. In Figure 2.2 we show a single ring-structured dataflow processor connected via an I/O switch to a host computer as implemented by Gurd and his colleagues [GK85]. The system can be extended to a multiprocessor computer by widening the I/O switch to accommodate more processor rings. A good way to start thinking about an individual ring is as a processor pipeline, such as is commonly found in modern microprocessors. An important task in the design of such pipelines is to prevent instructions from being scheduled for execution when their operands have yet to be computed by instructions still in the pipeline. The Manchester pipeline provides an extremely clever conflict resolution mechanism which guarantees that an instruction cannot enter the pipeline before all its operand values are available. Other than in von Neumann microprocessors, however, instruction scheduling is governed *solely* by data dependencies of this kind. There is no concept of a program counter. Since the result produced by one instruction can satisfy the dependencies of more than one successor instruction, multiple instructions can be ready for scheduling at the same time.

Let us follow the path of an imaginary token inserted by the host computer. The *I/O switch* which is responsible for directing the token to the appropriate dataflow processor (of which our system contains only one) sends it on to the *token queue*. This queue is a circular buffer that smoothes out uneven rates of generation and consumption of tokens in the ring. When reaching the head of the queue, the token is passed on to the *matching unit*, arguably the most unconventional component of the processor ring. The matching unit's prime responsibility is resolving instruction dependencies and scheduling instructions for execution. An incoming token carrying the first operand of a dyadic operation remains in the matching unit awaiting the arrival of its partner. The unit derives its name from this matching of partners. An incoming token that completes a match or is heading for a monadic operation is passed on to the *instruction store* which contains the program code. There the appropriate instruction is fetched and passed on along with the operands to the *processing unit*. Typically the processing unit is made up of several function units (akin to the multiple function units of a modern von Neumann processor), one of which will perform the required operation and generate one or two output tokens.

Note that the mechanism described above is readily expandable to multiple processing elements as the basic instruction scheduling mechanism is easily extended to work across processors. Note also that on our journey through the pipeline we did not encounter a stage which would obviously correspond to 'main memory access'. The matching unit implicitly provides the storage needed for data which would be kept in stack frames in a more conventional runtime model. For this reason the size of the matching unit exceeds 1 M-words and is unsuitable for complete VLSI integration even with present-day technology. Furthermore the structure of the matching unit is relatively complex as it requires associative access to waiting tokens. In the Manchester machine this is implemented via a hardware hash-function. Some of the data storage functions were later moved from the matching unit to a dedicated *structure store.*

2.3.2 Load bounding in the Manchester dataflow machine

When the Manchester project was started, it had not yet become clear that many typical programs would provide sufficient parallelism to keep a large parallel machine busy. So it came originally as a surprise when it was observed that the amount of parallelism exhibited by some programs could get so large that the design of a *throttle* mechanism was crucially important in order to limit the resource requirements of such programs (see Section 1.7.2). The throttle designed by Ruggiero and Sargeant and described in [RS87] is a hardware device which operates roughly according to the following principles. On receiving a request for starting a new process their throttle decides, based on the level of parallel activity in the machine, whether to grant a new activation name or not. Parallel activity is measured in terms of the length of the token queue. If the machine is too busy, the process is suspended and is reactivated only when the level of activity has dropped. In order to promote depth-first execution, the first child of a process is never suspended. Given several suspended subprocesses for one process, the leftmost one is the first candidate for unsuspension. This order is guaranteed by a queueing strategy. They impose a small delay between individual unsuspensions since processes take a while to start up and have no immediate effect on the length of the token queues.

The Manchester throttle is responsible not just for load bounding but generally for resource management. Its functions include allocation of activation names, suspending and unsuspending of processes, activity level reports and termination signals. From a hardware point of view, the throttle is a message processor with an attached store and was implemented using the same design as for the structure store, with different microcode.

2.4 The MIT dataflow project

While work on the Manchester machine has now ceased, research into dataflow machines and languages is still actively being pursued at the Computation Structures Group at MIT. Originating with Jack Dennis's early work on a static dataflow machine [De75] which appeared radically different from any architecture known at the time, the MIT work has progressed in several stages, as outlined below, to a system designed around a commercial microprocessor. One of the results of their long and productive work was the P-RISC architecture which became a major influence for the design of STAR:DUST (Chapter 6). One purpose of this section is to illustrate that our STAR:DUST architecture, and thus the computational model underlying our compilation scheme, is not just a 'wild stab in the dark' but rather a combination of the core features of dataflow machines with the sequential efficiency of modern von Neumann architectures.

2.4.1 From static dataflow to Monsoon

Jack Dennis early work on static dataflow machines modelled the flow of streams of data through a static dataflow graph. The nodes of the graph represented operations which could 'fire' as soon as a complete set of inputs was available. While providing ample opportunity for parallelism, static dataflow graphs do not support a very general programming model. In particular, they do not allow for recursion. Dennis' 1973 paper [De73] is generally regarded as the seminal work inspiring *dynamic dataflow* which is capable of modelling graphs that are expanding and contracting dynamically. Dynamic dataflow is thus suitable for modern recursive programming languages. The MIT work on dynamic dataflow machines was lead by Arvind, whose proposal for a dataflow architecture with tagged tokens [AK80] is substantially similar to the Manchester dataflow machine (but developed independently). Therefore we will not discuss it in detail here except for pointing out that it includes a waiting-matching unit requiring the same expensive fully associative matching capabilities for data sets. In contrast to the Manchester machine, the TTDA (tagged-token dataflow architecture) was never built in hardware.

The first dataflow hardware at MIT became operational in 1988 in the form of the Monsoon machine [Pa88] which represents a substantial redesign of the original TTDA proposals. In Figure 2.3 we show the structure of a Monsoon processing element as presented in [PC90], redrawn to highlight the key differences from the Manchester architecture.

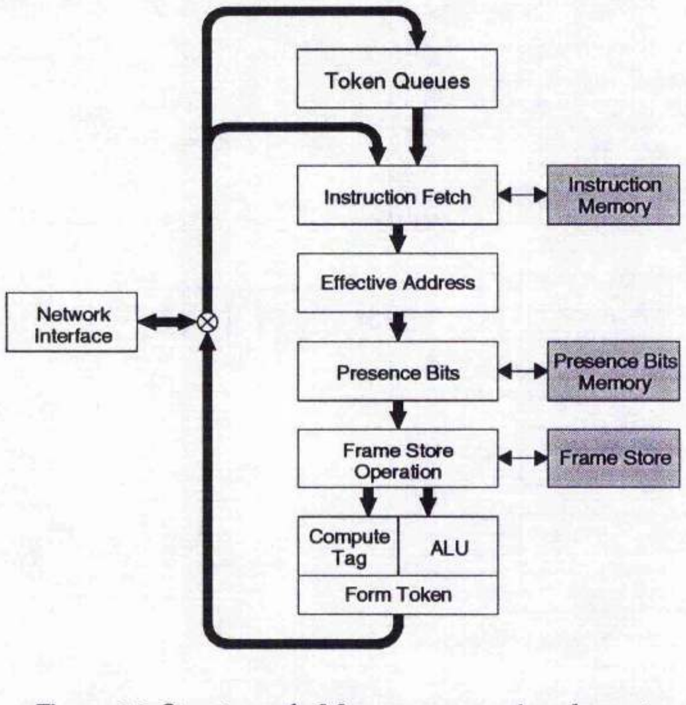

Figure 2.3: Structure of a Monsoon processing element

The fundamental improvement is the elimination of the waiting-matching unit which was achieved by making the token store *explicit*. Rather than relying on the storage of tokens to be performed implicitly and individually by hardware in the waiting-matching unit, storage allocation on Monsoon is performed by software and in units of procedure frames. An individual procedure frame is mapped wholly to a contiguous memory area within a *frame store*.

To understand the resulting changes in the operation of the processor pipeline consider the processing of a two-input operator (the following is largely quoted from [PC90]). The first token to be processed enters the pipeline and fetches the instruction specified in its tag field. During the effective address stage the location in the frame store where the match will take place is computed. The associated set of presence bits are examined and found to be in the 'empty' state. The presence state is thus set to 'full' and the incoming value is written into the frame store location during the frame store stage. Further processing of the token is suppressed because the other operand has yet to arrive. This 'bubbles' the pipeline for the remaining ALU stages; no tokens are produced during *form-token,* permitting a token to be removed from one of the token queues for processing. The second token to be processed enters the pipeline and fetches the same instruction. It therefore computes the same effective address. This time, however, the presence state is found to be 'full', so the frame store location (which now contains the value of the first token) is read and both values are processed by the ALU. Finally, one or two result tokens are created during the form-token stage.

Note that the pipeline of a Monsoon processing element is rather more similar to a conventional processor pipeline than that of the Manchester machine. An instruction fetch stage is followed by the computation of an effective address, a load/store stage interacting with local memory, and an ALU stage. There is even direct support for sequential execution by allowing successor instructions to bypass the token queue and re-enter the pipeline immediately. The differences to conventional pipelines are still significant, however: instructions can have multiple successors, all but one of which are stored in a token queue. The resulting parallelism is used to good effect to avoid bubbles in the pipeline. And finally, the stage responsible for 'presence bits' provides efficient hardware support for 'join' synchronisation.

2.4.2 Dataflow/von Neumann hybrid processors

The analogies between Monsoon and conventional processors were not lost on the MIT team. In [Ia88], Iannucci proposed the idea of dataflow machines and von Neumann machines sitting at opposite ends of a *spectrum* of architectures. The hybrid processor proposed in his paper travels another step towards von Neumann architectures by re-introducing registers as a very efficient means of communication for sequentially related instructions. (By constrast, Monsoon instructions can only send a single value to their successor in the form of a token. Where this is not sufficient, data have to be deposited in the frame store, i.e., in local memory.) By providing registers and dedicated instructions for testing and manipulating presence bits, Iannucci's hybrid processor can dispense with the idea of 'tagged tokens' for communication within a procedure frame.

The next step in this process of evolution resulted from approaching the architectural spectrum from the opposite end by asking the question: how do we need to modify a von Neumann processor to make it suitable for efficiently executing dataflow programs? The design of the P-RISC architecture [AN88] which is shown in Figure 2.4 is an attempt to answer this question. P-RISC, short for 'parallel RISC', has at its core a plain sequential RISC processor, i.e., three-address instructions, a load/store architecture, simple instruction formats, a program counter, conventional jump and branch instructions, etc. Addressing of operands is relative to a *frame*, which is best viewed as a fixed-size window providing fast access to local memory. The frame pointer along with the program counter make up the current *continuation*. Each of the conventional instructions has a single successor continuation which shares the frame pointer of its predecessor. All continuations operating within the same frame are considered to belong to the same *thread*.

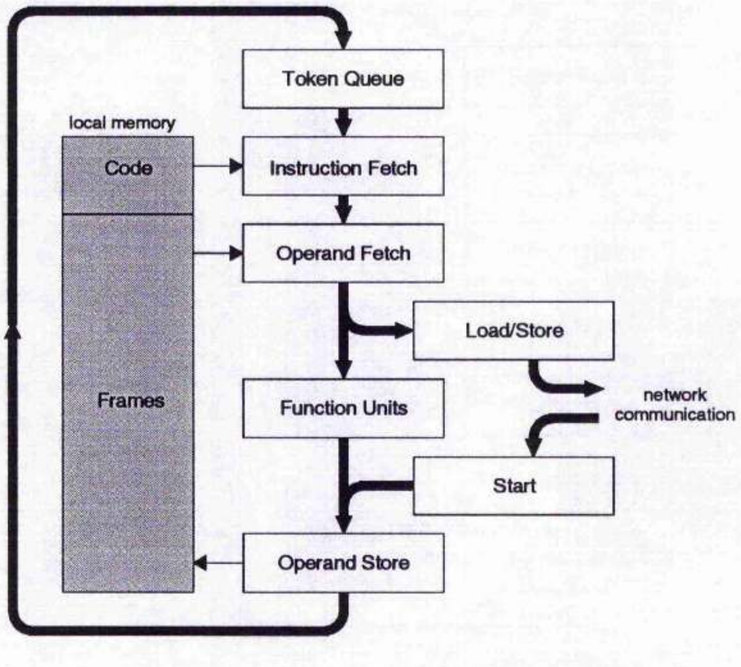

Figure 2.4: A P-RISC processing element

The parallel extensions to the sequential RISC paradigm come in the shape of several simple new instructions which create, control and terminate threads.

- `fork` produces an additional continuation which is placed in the token queue along with the natural successor

- `join` conditional termination of the current thread, i.e., depending on a condition, `join` has either one or zero continuations
- `load` reads a value from *global* memory; `load` has no immediate successor which allows other threads to keep the CPU busy during long-latency operations; the thread is restarted only on arrival of the value to be loaded
- `store` writes a value to global memory and continues with the next instruction; no synchronisation is performed
- `loadc` version of `load` with an implicit `fork`
- `start` sends a `start` message to a remote processing element; incoming `start` messages deposit a value into the local frame and place a continuation into the token queue (the arrival of values `load`ed from global memory is in the form of `start` messages)

A P-RISC system is made up of processing elements as discussed above as well as structure store elements which satisfy global read/write requests and perform low level synchronisation tasks and memory management functions. Comparing P-RISC with Monsoon we observe that the 'complex' dataflow instructions have been split into separate synchronisation, arithmetic and fork/control instructions, eliminating the necessity of presence bits in the frame memory.

The (so far) final step in the MIT effort towards finding the optimal balance between dataflow and von Neumann processors is the *T architecture [NP92], pronounced 'start', which is currently actively pursued in terms of hardware design (both Iannucci's hybrid architecture and P-RISC are 'paper architectures'). We show a block diagram of *T in Figure 2.5.

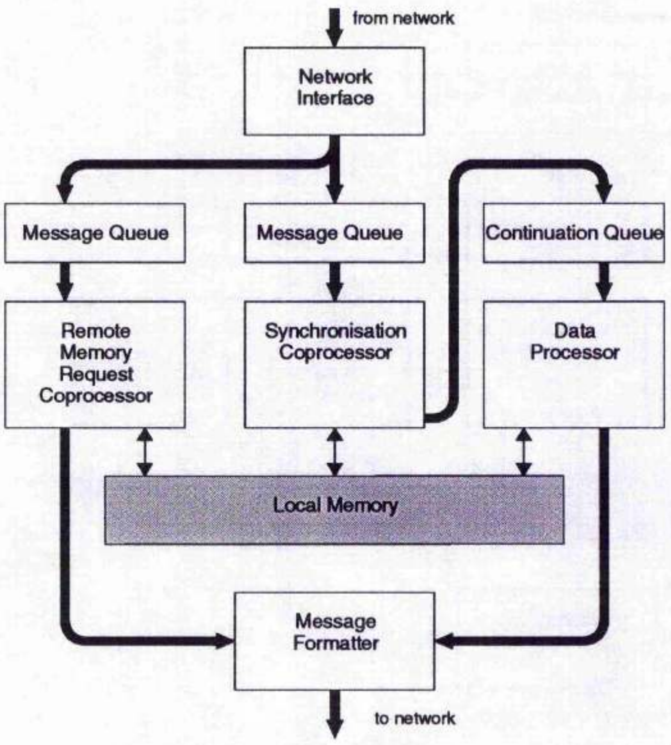

Figure 2.5: The *T architecture

The data processor in this diagram is a slightly enhanced version of the Motorola 88100, i.e., a commercial RISC microprocessor. The enhancements permit it to send messages to the network and to pick up new continuations (in the form of one word each for the program counter and a frame pointer). The data processor is responsible for the computational aspects of program execution. Two coprocessors sharing the same local memory are responsible for satisfying remote memory requests and synchronisation, respectively, without having to interfere with the data processor. Having a coprocessor for memory requests enables *T to avoid the complications of providing dedicated structure store units. The synchronisation coprocessor (SP) handles returning loads by storing any returning value into its destination location. If the original `load` instruction is followed by a `join` synchronisation, the SP executes it and, if successful, places its continuation on the continuation queue. Thus the data processor does not have to execute disruptive `join` instructions.

Being based on a standard microprocessor allows *T to provide competitive performance for existing sequential code (which would execute wholly on the data processor) as well as for sequential portions of parallel code. It also permits *T to 'ride the technology curve' by exploiting any advances in microprocessor design.

2.4.3 Load bounding for the MIT dataflow machines

Supporting an implicitly parallel programming model, the MIT dataflow work needs to address the problem of excessive parallelism. In [CA88] Arvind and Culler propose the technique of *loop bounding* which controls parallelism resulting from those portions of loop iterations which are not constrained by dependencies on previous iterations. Their approach is, quite simply, to restrict for each loop the number of concurrently active iterations to some constant k, i.e., iteration $n+k$ can begin only after iteration n has terminated. This solution appears to provide satisfactory results for many programs. Their compiler, as described in [TR86], generates parameterized code which allows the setting of loop bounds prior to execution, i.e., the choice of loop bounds is under user control and will typically depend on machine size, program type, and problem size.

Arvind and Culler critisise the Manchester approach of load bounding by deferring activations. They argue that loops can go on requesting activations, each of which would get deferred but would still require a small amount of resources. Therefore the Manchester technique is not a solution for the general case and they consider load bounding for general recursive programs an unsolved problem.

We would like to point out that the problem of loop parallelism is specific to dataflow programming languages. In pure functional languages, 'loops' are defined in terms of function application and do not behave in any special way. We would further point out that imposing loop bounds can in certain cases alter program semantics. It is possible to write programs where iteration *n+k* produces a value and makes it available to iteration *n* via side effect. In single-assignment languages computations block on unavailable values and therefore a loop bound of *k* will lead to deadlock in the case described. In addition, putting the responsibility for loop bounding on the user significantly weakens their claim to support implicit parallelism.

2.5 ALICE/Flagship

ALICE [DR81] was the first implementation of a parallel architecture dedicated to performing graph reduction in parallel. While the original design called for an implementation in customised VLSI, only a transputer-based hardware emulator was built. Predating the development of compiled graph reduction, ALICE suffered from severe interpretative overheads. While relative speedups were observed, absolute performance remained poor.

The Flagship project [WW88] grew out of and drew on the experiences with ALICE. While still a reduction architecture, it was designed to support a host of practical concerns such as distributed virtual addressing, a distributed I/O subsystem, a priority mechanism, caching, support for a multi-user environment and resilience to node failure. Like the architectural model underlying our work, Flagship is based on a distributed memory model with closely coupled processor-memory pairs. The much less ambitious STAR:DUST architecture presented in Chapter 6 represents a less radical departure from the von Neumann model and we expect to be better positioned to take advantage of progress in sequential microprocessor technology. Our implementation of parallel functional languages shares with that described by Watson and Watson in [WW87a] a relatively fine-grained outlook. While in their system 'packets' are the unit of parallelism and load distribution, this role is played by function activations in ours. However, our approach to load balancing and load bounding differs significantly. In both cases we take the 'RISC' approach of doing it in software. Whereas load balancing on Flagship is performed by a system service, our approach is incorporated into the compilation scheme of Chapter 4 by means of randomly distributing function applications. Whereas load bounding on Flagship is performed by hardware controlled scheduling of the 'active packet queue', our load bounding system is again integrated into the compilation scheme, supported by a cheap runtime system, as described in Chapter 5.

2.6 GRIP

The GRIP project at Glasgow, short for 'Graph Reduction In Parallel', is notable for its success in carrying over to parallel functional programming the significant advances that have been made for sequential implementations of functional languages. They achieve significant real speedups [HP90], as measured against sequential implementations, on largely conventional hardware: GRIP is based on up to 80 conventional Motorola 68020 microprocessors, each with a small local memory of 1 Mbyte, and up to 20 microprogrammable intelligent memory units (IMUs). The latter are the major architectural innovation, providing efficient hardware support for a global memory abstraction. GRIP is a bus-based design (which accounts for the limits on machine size) and was intended from the outset to provide cost-effective parallelism in the short term. Scalability was not a design criterion.

Their combination of efficient shared-memory support and conventional microprocessor technology provides the backdrop for any comparison of our work with that of GRIP. Their underlying architectural parameters imply that communication is cheap while task switching and, even more so, task creation is expensive. A crucial optimisation on GRIP is therefore to achieve long-running tasks. Their evaluation strategy is, in a nutshell, to make a single task responsible for evaluating the whole spine of the part of the program graph allocated to it. Other subtasks are put into a 'global task pool' from where they can be picked up by idle processors. On a heavily loaded system, a parent may return to a subtask to find that it is still unevaluated. In this case the parent will decide to evaluate it itself and the extra task is discarded. This technique, called the 'evaluate-and-die model' by Peyton Jones in [Pe89], was first described, to our knowledge, in [CP88] and has the effect of increasing the size of tasks still further. The technique is very similar to and predates that of Mohr et al. [MK90] described in Section 2.7. The inbuilt preference for depth-first execution also addresses the load bounding problem.

GRIP's non-scalability manifests itself in several aspects of its design, the choice of a bus as the main communications medium being the most obvious one. The decision to block the processor during long-latency access to global memory is entirely reasonable on a relatively small machine built from stock sequential components but would incur increasing performance penalties if attempts were made to build larger GRIPs.

In contrast to GRIP, the work reported in this thesis was performed with concerns of scalability in mind throughout. In order to be able to tolerate the high latencies of global communication we employ thread switching. Our approach to load-bounding (itself implemented in a scalable fashion) provides each PE with a sufficient number of threads for this purpose. We require no special hardware other than a general-purpose message-passing processor.

2.7 Lazy Task Creation

The problem of load bounding is closely related to that of granularity in the following way: rather than creating a large number of small tasks in excess of what can be exploited by machine parallelism, one could group them into larger tasks, limiting resource requirements at the same time as avoiding some overhead for task switching. On machines based on conventional sequential microprocessors where the cost of context switching is high, the latter effect is particularly desirable. In their paper on 'Lazy Task Creation' [MK90], Mohr et al. describe a technique for increasing the granularity of programs which performs load bounding as a side-effect. Their work was done in the context of implementing a strict imperative language with a pure functional subset on shared-memory architectures. Only one task per processor is active at one time. Here are the key ideas:

- programs identify sources of potential parallelism explicitly, using the 'future' construct
- a running task puts each into a 'lazy future queue'
- idle processors can 'steal' them and execute them, exploiting parallelism
- on returning to an 'unstolen' lazy future, tasks execute them directly, thus increasing granularity

Their method is virtually identical to the 'evaluate-and-die model' described in Section 2.6 on GRIP with similar implications for (lack of) scalability. By running only a single task per processor, high latencies cannot be masked using task switching. Mohr et al. do not describe any distribution strategy for their 'lazy future queue', so question marks must be put on the scalability of this concept as well. While their design efficiently supports conventional shared-memory machines unsuitable for fine-grain parallel computation, their 'task stealing' operation further *enforces* the need for large granularity even if an underlying machine were intrinsically fine-grained. This is due to the fact that processors are blocked while a new task is being fetched and the cost of this operation needs to be amortised. Even so, however, short tasks cannot be completely avoided as the amount of work implied by a 'stolen future' is not generally known.

2.8 Other previous work

In this chapter we have reviewed details of previous work which we considered most relevant to this thesis. In conclusion we would like to make mention of other related work not discussed here in detail. Traub investigates generating multithreaded code from non-strict functional programming languages in [TR91], focussing on producing long threads by analysing data dependencies. Nikhil outlines a compiler for Id to stock parallel hardware via dataflow graphs and P-RISC as an abstract machine [Ni89]. The FAST project, based on Kelly's work on process annotations for functional languages [Ke89] intends to exploit parallelism on a network of nodes each running a sequential implementations of Haskell. Darlington's work on 'skeletons' [Da91], higher-order functions for which efficient implementations exist on particular parallel machines, is an ambitious attempt of providing a unifying framework for exploiting parallelism on a wide range of parallel architectures. Dally's J-Machine [DC89] resembles our STAR:DUST architecture in that communication is data-driven, i.e., messages start handler threads, but their architecture is less RISC-like. Von Eicken et al. propose the concept of 'active messages' as a general purpose communications paradigm [EC92] suitable even for existing message passing multiprocessors.

Chapter 3 — A Simulation of the λ-Calculus in the π-Calculus

3.1 Contents

In this chapter we present a concise model for parallel graph reduction in the form of a translation scheme mapping a term of the λ-calculus to a π-calculus term. The latter is constructed in such a way that its reduction according to the rules of the π-calculus will result in a normal form which is equivalent to the weak head-normal form (WHNF) of the original λ-calculus term, provided there is such a normal form. We will use the π-calculus in the form presented in Section 2.2.

3.2 Motivation

The practical motivation for simulating the λ-calculus in the π-calculus is the capacity of the latter for expressing parallelism. As detailed in Section 2.2, Milner has given two different encodings for the λ-calculus. The first simulation follows normal order reduction rules and does not exhibit any concurrency—at any point in the π-calculus reduction sequence there is only one reduction that can take place. Furthermore, this solution takes no account of sharing. If the value of a function argument is required more than once, it is recomputed each time. His second simulation performs eager, call-by-value reduction. Both subterms of an application are reduced in parallel and the application is performed once the value of the argument term is available. This solution has two disadvantages. Firstly, it may not terminate for some λ-terms which have a normal form. Secondly, there is scope for additional parallelism by starting the reduction of a function body *before* the value of the argument is available. The encoding we give below has the following features:

— applications can be performed lazily to maintain termination properties

— applications can be performed eagerly where strictness analysis guarantees termination

— sharing is preserved, avoiding unnecessary duplication of reductions

— the body of a function can be reduced as soon as it is applied to an argument (and before the argument value is available)

Our solution also leaves scope for deciding dynamically in favour of lazy evaluation, even when termination is no problem, in order to limit parallelism to a desired level. We can easily extend our scheme by adding ground values other than functions, e.g., integers, and 'built-in' functions operating upon them.

3.3 Overview

While our scheme is concise, we consider it worthwhile to prepare the ground by giving an overview of the underlying principles. As in the two Milner schemes outlined in Section 2.2, we encode a λ-calculus term M as $[[M]]$, a map from π-calculus names to π-calculus terms, i.e., $[[M]]\ o$ is a term of the π-calculus. The intuition behind our scheme is fundamentally different, however. While in Milner's schemes the argument channel o was the link along which $[[M]]$ received its arguments, we will use it as a link where $[[M]]$ is to *deposit its value*. The difference was phrased poignantly by our colleague Peter Burgess: "Milner's scheme has no output—this scheme has no input".

Our intuition of a *value* is a scalar, e.g., an integer or a pointer. The following two rules will not be part of our λ-calculus scheme, but they will easily tie into it and are given here to provide a first intuition.

$$[[\,\text{const}\,]]\ o \overset{\text{def}}{=} \overline{o}\ \text{const}$$

$$[[\,M{+}N\,]]\ o \overset{\text{def}}{=} (m)(n)\,([[M]]\ m \mid [[N]]\ n \mid m(a).\ n(b).\ \overline{o}\,(a{+}b)\,)$$

The first rule is obvious. To deposit the value of a constant into a channel, we simple put it there via a send-operation. The rule for addition first introduces two new channels for the intermediate results for M and N, respectively. Of the three concurrent processes governed by the restriction, two are responsible for computing M and N. The third waits for value of M to arrive on m and reads it into a. It then waits for the value of N which will eventually appear in n and is read into b. Assuming a primitive addition operator, the result of the addition can now be written into the designated output channel.

The key to the complete translation scheme is an understanding of the two underlying protocols it adheres to. The first protocol governs the simulation of λ-calculus variables, i.e., the interaction of identifier reference and function application. The second governs function activations, i.e., the interaction of function application and abstraction.

Protocol 1 (Variables)

> A λ-calculus variable x is simulated by a π-calculus variable of the same name by interpreting it as a *request channel.* Each time a computation requires the value of x, it sends a request r (the destination channel) along the channel x. The computation at the other end, i.e., the process simulating function application, will then send back its value along r.

Protocol 2 (Abstractions)

> The value of a function expression $F = \lambda x.M$ is represented by a channel f in the following way: to apply F to an argument x, we send it a pair of channels along f consisting of a request channel for x, as described in Protocol 1, and a destination channel y where we require the result. We will call this pair an

activation. The simulation of an abstraction is responsible for handling activations of the form (x, y) by setting up an instance of M relating x and y.

3.4 The translation scheme

Having thus laid the foundations, we are now ready for the compilation scheme itself. We will provide five rules in total. Besides an auxiliary definition for environment entries there is one rule each for λ-abstraction and variable reference and two rules for function application. The reason for providing two rules in the latter case is our goal of representing both lazy (sequential) and eager (parallel) function application within the same framework. The obvious way for resolving the resulting ambiguity is to apply the lazy rule by default, reserving the eager rule for the application of functions that can be shown to be strict in their argument. Here is the complete set of translation rules, discussed individually below.

$$x \Leftarrow v \quad \overset{\text{def}}{=} \quad !x(r).\ \bar{r}\,v \qquad \text{(Environment entry)}$$

$$[[x]]\,o \quad \overset{\text{def}}{=} \quad \bar{x}\,o \qquad \text{(Name reference)}$$

$$[[\lambda x.M]]\,o \quad \overset{\text{def}}{=} \quad (f)\ \bar{o}f.\ !\,f(x,y).\ [[M]]\,y \qquad \text{(Abstraction)}$$

$$[[M\,N]]\,o \quad \overset{\text{def}}{=} \quad (m)(n)(x)\ ([[M]]\,m \mid [[N]]\,n \mid m(f).\ \bar{f}(x,o).\ n(v).\ x \Leftarrow v\ )$$

(Eager application)

$$[[M\,N]]\,o \quad \overset{\text{def}}{=} \quad (m)(n)(x)\ ([[M]]\,m \mid (m(f).\ \bar{f}(x,o).x(r).([[N]]\,n \mid n(v).\,\bar{r}\,v.x \Leftarrow v\ )))$$

(Lazy application)

3.4.1 Environment entry

We make a variable x refer to a value v by writing v onto any destination channel r that we can pick up on x. This is merely Protocol 1 codified in the π-calculus. We will repeat the complete rule again for future reference:

$$x \Leftarrow v \quad \overset{\text{def}}{=} \quad !x(r).\ \bar{r}\,v \qquad \text{(Environment entry)}$$

3.4.2 Name reference

The fact that this rule is identical to the corresponding one in Milner's lazy scheme is coincidental and the meaning here is substantially different as explained previously. Our rule is an abbreviation for

$$[[x]]\, o \quad \overset{\text{def}}{=} \quad (r)\ \overline{x}\, r.\ r(v).\ \overline{o}\, v$$

We define a channel r which forms the request that we send to x according to Protocol 1 which also provides for the return of the value v of x along r. We then send this value to the designated output channel o. Looking at this term more closely we find that the intermediary channel r is unnecessary. According to Protocol 1 we can send o directly to x and expect the value of x to be written to o, as desired, resulting in the slightly counterintuitive abbreviation given above,

$$[[x]]\, o \quad \overset{\text{def}}{=} \quad \overline{x}\, o \qquad \text{(Name reference)}$$

3.4.3 Abstraction

According to Protocol 2, the value of an abstraction is a channel which interfaces to an activation handler. The abstraction rule therefore defines a new channel f which is written to o, followed by such a handler. For any activation request (x, y) picked up on f, the handler provides a new instance of M. The complete rule again:

$$[[\lambda x.M]]\, o \quad \overset{\text{def}}{=} \quad (f)\ \overline{o} f.\ !f(x,y).\ [[M]]\, y \qquad \text{(Abstraction)}$$

The binding of the free variable x in M is mapped to the binding mechanism provided for the receive-action in the π-calculus. The transmission of pairs of channels transcends the pure π-calculus presented in Section 2.2, but is readily defined as in [Mi91]

$$f(x, y) \quad \overset{\text{def}}{=} \quad f(w).\ w(x).\ w(y)$$

$$\bar{f}(x, y) \quad \overset{\text{def}}{=} \quad (w)\ \bar{f} w.\ \overline{w} x.\ \overline{w} y$$

3.4.4 Eager application

As eager function application is slightly simpler than its lazy counterpart, we will discuss it first. Here, again, the rule given above:

$$[[M\,N]]\,o \quad \overset{\text{def}}{=} \quad (m)(n)(x)\,([[M]]\,m \mid [[N]]\,n \mid m(f).\,\bar{f}(x,o).\,n(v).\,x \Leftarrow v\,)$$

(Eager application)

When applying M to N, we require three new channels, m, n and x. The first two are set up to receive the values of M and N, respectively, the latter to serve as the request channel for the argument according to Protocol 1. The body of this rule consists of three concurrent processes. The first two compute M and N, respectively (since we are performing eager function application, the computation of N can proceed before its value is requested from within M). The third process, of the form

$$m(f).\,\bar{f}(x,o).\,n(v).\,x \Leftarrow v$$

is responsible for establishing the functional relationship between m, n and o according to Protocol 2. We pick up the function value, i.e., the activation handler f. We request a new activation (x, o) for the request channel x previously defined. After establishing this activation, we pick up the value v of N and bind it to x by an appropriate environment entry.

3.4.5 Lazy application

Let us start again by restating the rule in question:

$$[[M\,N]]\,o \quad \overset{\text{def}}{=} \quad (m)(n)(x)\,([[M]]\,m \mid (m(f).\,\bar{f}(x,o).x(r).([[N]]\,n \mid n(v).\,\bar{r}\,v.x \Leftarrow v\,)))$$

(Lazy application)

As for eager function application, we define three new channels, m, n and x serving the same purpose as before. This time the body of the rule contains only two concurrent processes on the top level. The first computes the value of M, as before. The second has the following shape:

$$m(f).\,\bar{f}(x,o).\,x(r).\,([[N]]\,n \mid n(v).\,\bar{r}\,v.\,x \Leftarrow v\,)$$

This time we want to delay any evaluation of N until we know that its value will be required. We therefore pick up the function activation handler f and establish an activation of M between x and o. We do not proceed with any other reduction until we have received an initial request r for the value of x. Only after receiving r we start the evaluation of N. When its value arrives we read it into v and satisfy the original request by writing it to r. As there may be further requests, we then set up an environment entry as before.

3.5 Sample reductions

To get an intuition for the behaviour of the π-calculus terms produced by our translation scheme, let us step through a few simple examples. The first example will serve us to familiarise ourselves with the behaviour of (eager) function application and identifier reference. The second, slightly more involved, demonstrates sharing under in the context of the rule for eager function application. The third example will be identical to the second except for following lazy evaluation. In all three cases we will make use of the extended set of rules including those for integer constants and addition given in Section 3.3. We will unwind the definition of $[[\cdot]]$ only as far as necessary to make progress with the reduction. The transformation steps are marked by one of four symbols: '$=$' denotes equality by the definition of $[[\cdot]]$, '$\equiv$' denotes equivalence according to the rules of the π-calculus, '$\rightarrow$' denotes reduction in the π-calculus and '$\sim$' denotes a special case of 'strong bisimilarity' [Mi92] of which we will only need to know that it allows us to 'garbage collect' useless processes, i.e., those which can not participate in any further interactions.

3.5.1 Simple function application

For our first example, the reduction of $[[(\lambda x.x)\ 1]]\ o$, we will avoid any 'leaps' in the transformation process in order to aid understanding. Note, however, that in the resulting lengthy sequence only four steps represent π-calculus reductions. A discussion of each step follows below.

$$[[(\lambda x.x)\ 1]]\ o \tag{1}$$

$$= (m)(n)(x)\ ([[\lambda x.x]]\ m \mid [[1]]\ n \mid m(f).\ \bar{f}(x,o).\ n(v).\ x \Leftarrow v) \tag{2}$$

$$= (m)(n)(x)\ ((f)\ \overline{m}f.\ !f(x,y).\ [[x]]\ y \mid [[1]]\ n \mid m(f).\ \bar{f}(x,o).\ n(v).\ x \Leftarrow v) \tag{3}$$

$$\rightarrow (m)(n)(x)(f)\ (!f(x,y).\ [[x]]\ y \mid [[1]]\ n \mid \bar{f}(x,o).\ n(v).\ x \Leftarrow v) \tag{4}$$

$$\equiv (n)(x)(f)\ (!f(x,y).\ [[x]]\ y \mid [[1]]\ n \mid \bar{f}(x,o).\ n(v).\ x \Leftarrow v) \tag{5}$$

$$\equiv (n)(x)(f)\ ((f(x,y).\ [[x]]\ y \mid !f(x,y).\ [[x]]\ y) \mid [[1]]\ n \mid \bar{f}(x,o).\ n(v).\ x \Leftarrow v) \tag{6}$$

$$\rightarrow (n)(x)(f)\ (([[x]]\ o \mid !f(x,y).\ [[x]]\ y) \mid [[1]]\ n \mid n(v).\ x \Leftarrow v) \tag{7}$$

$$\sim (n)(x)(f)\ ([[x]]\ o \mid [[1]]\ n \mid n(v).\ x \Leftarrow v) \tag{8}$$

$$\equiv (n)(x)\ ([[x]]\ o \mid [[1]]\ n \mid n(v).\ x \Leftarrow v) \tag{9}$$

$$= (n)(x)\ ([[x]]\ o \mid \bar{n}1 \mid n(v).\ x \Leftarrow v) \tag{10}$$

$$\rightarrow (n)(x)\ ([[x]]\ o \mid x \Leftarrow 1) \tag{11}$$

$$\equiv (x)\ ([[x]]\ o \mid x \Leftarrow 1) \tag{12}$$

$$= (x)\ (\bar{x}o \mid x \Leftarrow 1) \tag{13}$$

$$= (x)\ (\bar{x}o \mid !x(r).\ \bar{r}1) \tag{14}$$

$$\equiv (x)\ (\bar{x}o \mid (x(r).\ \bar{r}1 \mid !x(r).\ \bar{r}1)) \tag{15}$$

$$\rightarrow (x)\ (\bar{o}1 \mid !x(r).\ \bar{r}1) \tag{16}$$

$$\sim (x)\ \bar{o}1 \tag{17}$$

$$\equiv \bar{o}1 \tag{18}$$

The whole transformation from (1) to (18) represents the application of the identity function to the constant 1 with output channel o. In the first step to (2) we apply the definition of the rule for eager function application. Since there is no obvious scope for any reduction yet we apply another translation rule, this time that for abstraction to obtain (3). The step to (4) represents the first π-calculus reduction, modelling the communication of the function value f along m. Since m now does not occur in the body of the resulting term, we can drop the restriction on m (5). Before applying the function we need to unwind an instance of its activation handler in (6). The second communication (of the pair (x, o) along f) represents the activation of the identity function, linking input x to output o (7). Since there is no possibility of requesting another activation of f, we can garbage collect the activation handler (8) and drop the restriction on f (9). In step (10) we apply the rule for integer constants. Communicating the argument value 1 along n results in (11) which can be simplified to (12) by dropping the now useless restriction on n. Applying the rule for referencing the value of x in the function body we obtain (13). The last instance of applying our translation rules, this time for the environment entry $x \Leftarrow 1$, results in (14). Before communicating a request for the value of x in step (16) we need to unwind an instance of the request handler (15). The request handler can not become active again and is garbage collected (17). Dropping the useless restriction on the non-existing name x we obtain the desired result, $\overline{o}\,1$.

3.5.2 Sharing of computation (1)

To gain a better understanding of the way sharing is modelled in our scheme, let us study the reduction of $[[(\lambda x.x+x)\ (1+1)]]\ o$. Although x is referenced twice inside the abstraction, we want to perform the evaluation of (1+1) only once. For the transformation sequence below, we will omit any steps involving dropping of unused names and garbage collection of handlers which cannot participate in any further interactions. We will further assume the reduction of $[[1+1]]\ o$ to $\bar{o}2$ which would make a useful exercise for the reader.

$$[[(\lambda x.x+x)\ (1+1)]]\ o \tag{1}$$

$$= \quad (m)(n)(x)\ ([[\lambda x.x+x]]\ m \mid [[1+1]]\ n \mid m(f).\ \bar{f}(x,o).\ n(v).\ x \Leftarrow v\ ) \tag{2}$$

$$\rightarrow \quad (m)(n)(x)\ ([[\lambda x.x+x]]\ m \mid \bar{n}2 \mid m(f).\ \bar{f}(x,o).\ n(v).\ x \Leftarrow v\ ) \tag{3}$$

$$= \quad (m)(n)(x)\ ((f)\ \bar{m}f.\ !f(x,y).\ [[x+x]]\ y \mid \bar{n}2 \mid m(f).\ \bar{f}(x,o).\ n(v).\ x \Leftarrow v\ ) \tag{4}$$

$$\rightarrow \quad (n)(x)(f)\ (!f(x,y).\ [[x+x]]\ y \mid \bar{n}2 \mid \bar{f}(x,o).\ n(v).\ x \Leftarrow v\ ) \tag{5}$$

$$\sim \quad (n)(x)(f)\ (f(x,y).\ [[x+x]]\ y \mid \bar{n}2 \mid \bar{f}(x,o).\ n(v).\ x \Leftarrow v\ ) \tag{6}$$

$$\rightarrow \quad (n)(x)\ ([[x+x]]\ o \mid \bar{n}2 \mid n(v).\ x \Leftarrow v\ ) \tag{7}$$

$$\rightarrow \quad (x)\ ([[x+x]]\ o \mid x \Leftarrow 2\ ) \tag{8}$$

$$= \quad (x)\ (\ (m)(n)\ ([[x]]\ m \mid [[x]]\ n \mid m(a).\ n(b).\ \bar{o}(a+b)\ ) \mid x \Leftarrow 2\ ) \tag{9}$$

$$= \quad (x)\ (\ (m)(n)\ (\bar{x}m \mid \bar{x}n \mid m(a).\ n(b).\ \bar{o}(a+b)\ ) \mid x \Leftarrow 2\ ) \tag{10}$$

$$\sim \quad (x)\ (\ (m)(n)\ (\bar{x}m \mid \bar{x}n \mid m(a).\ n(b).\ \bar{o}(a+b)\ ) \mid x(r).\ \bar{r}2 \mid x(r).\ \bar{r}2\ ) \tag{11}$$

$$\rightarrow \quad (m)(n)\ (m(a).\ n(b).\ \bar{o}(a+b\,) \mid \bar{m}2 \mid \bar{n}2\ ) \tag{12}$$

$$\rightarrow \quad (n)\ (n(b).\ \bar{o}(2+b) \mid \bar{n}2\ ) \tag{13}$$

$$\rightarrow \quad \bar{o}(2+2) \tag{14}$$

$$\rightarrow \quad \bar{o}4 \tag{15}$$

Let us again discuss each step in some more detail. Going from (1) to (2) represents expansion according to the rule for eager application, which enables us to perform reduction of the argument (1+1) immediately, obtaining (3). In step (4) we apply the abstraction rule to the term $(lx.x+x)$. We can now communicate the function value f along m to obtain (5), dropping the restriction on m. We unwind one instance of the activation handler and garbage collect all others, arriving at (6). The activation takes place in (7) followed by the communication of the argument value 2 along n in (8). Applying the translation rule for addition we get (9) which is transformed to (10) by applying the identifier reference rule. In (11) we have unwound two instances of the request handler for x and have garbage collected the rest. We now communicate the two requests, m and n, obtaining (12). We satisfy the first request in (13) and the second in (14). Performing the primitive addition we obtain the final result in (15).

3.5.3 Sharing of computation (2)

In the previous example we observed the behaviour of $[[(\lambda x.x+x)\ (1+1)]]\ o$ when translated according to the eager reduction rule. The reduction of the argument could be performed early on, before its value was requested from within the function body. Let us now study the reduction behaviour of the same term under the lazy scheme:

$$[[(\lambda x.x+x)\ (1+1)]]\ o \qquad (1)$$

$$= \quad (m)(n)(x)\ ([[\lambda x.x+x]]\ m \mid (m(f).\ \bar{f}(x,o).\ x(r).\ ([[1+1]]\ n \mid n(v).\ \bar{r}\,v.\ x \Leftarrow v\ ))) \qquad (2)$$

$$= \quad (m)(n)(x)\ ([[\lambda x.x+x]]\ m \mid m(f).\ \bar{f}(x,o).\ x(r).\ P) \qquad (3)$$

$$= \quad (m)(n)(x)\ ((f)\ \bar{m}f.\ !f(x,y).\ [[x+x]]\ y \mid m(f).\ \bar{f}(x,o).\ x(r).\ P) \qquad (4)$$

$$\rightarrow \quad (n)(x)(f)\ (!f(x,y).\ [[x+x]]\ y \mid \bar{f}(x,o).\ x(r).\ P) \qquad (5)$$

$$\sim \quad (n)(x)(f)\ (f(x,y).\ [[x+x]]\ y \mid \bar{f}(x,o).\ x(r).\ P) \qquad (6)$$

$$\rightarrow \quad (n)(x)\ ([[x+x]]\ o \mid x(r).\ P) \qquad (7)$$

$$= \quad (n)(x)\ (\ (p)(q)\ ([[x]]\ p \mid [[x]]\ q \mid p(a).\ q(b).\ \bar{o}\,(a+b)\ ) \mid x(r).\ P) \qquad (8)$$

$$= \quad (n)(x)(p)(q)\ (\ \bar{x}p \mid \bar{x}q \mid p(a).\ q(b).\ \bar{o}\,(a+b) \mid x(r).\ P) \qquad (9)$$

$$= \quad (n)(x)(p)(q)\ (\ \bar{x}p \mid \bar{x}q \mid p(a).q(b).\bar{o}\,(a+b) \mid x(r).([[1+1]]\ n \mid n(v).\ \bar{r}\,v.x \Leftarrow v\ )) \qquad (10)$$

$$\rightarrow \quad (n)(x)(p)(q)\ (\ \bar{x}q \mid p(a).\ q(b).\ \bar{o}\,(a+b) \mid [[1+1]]\ n \mid n(v).\ \bar{p}v.\ x \Leftarrow v\ )) \qquad (11)$$

$$\rightarrow \quad (n)(x)(p)(q)\ (\ \bar{x}q \mid p(a).\ q(b).\ \bar{o}\,(a+b) \mid \bar{n}\,2 \mid n(v).\ \bar{p}v.\ x \Leftarrow v\ )) \qquad (12)$$

$\rightarrow \quad (x)(p)(q)\,(\overline{x}q \mid p(a).\,q(b).\,\overline{o}(a{+}b) \mid \overline{p}2.\,x \Leftarrow 2\,))$ (13)

$\rightarrow \quad (x)(q)\,(\overline{x}q \mid q(b).\,\overline{o}(2{+}b) \mid x \Leftarrow 2\,))$ (14)

$\sim \quad (x)(q)\,(\overline{x}q \mid q(b).\,\overline{o}(2{+}b) \mid x(r).\,\overline{r}\,2\,))$ (15)

$\rightarrow \quad (q)\,(q(b).\,\overline{o}(2{+}b) \mid \overline{q}\,2\,))$ (16)

$\rightarrow \quad \overline{o}(2{+}2)$ (17)

$\rightarrow \quad \overline{o}4$ (18)

The rule for lazy application is applied immediately to obtain (2) from (1). This time, the computation of the argument (1+1) is guarded by a request handler in the form of $x(r)$. Since the argument computation ([[1+1]] n | $n(v).\overline{r}\,v.\,x \Leftarrow v$) will not proceed for several steps yet, we abbreviate it to P in step (3). In (4) we have applied the abstraction rule, followed by the communication of the function value (5), unwinding of a single activation handler (6) and function activation (7), as before. Expanding the definition of [[x+x]] o completely, we obtain (8) and (9), again in complete analogy to the eager case. Since we now have two requests for the value of x, in the form of $\overline{x}p$ and $\overline{x}q$, we re-expand the argument computation P in (10). We choose p as the first request to communicate along x, obtaining (11). The guard in front of the argument computation has now disappeared, and we can reduce the argument to obtain (12). The argument value 2 is communicated to the environment entry in (13). Before establishing the environment entry, however, we first need to satisfy the original request, which is performed in the step to (14). As we require the value of x only one more time, we unwind only a single instance of the request handler for x, garbage collecting the rest (15). After communicating the single request in (16), the rest of the computation proceeds as in the eager case.

3.6 Discussion

In this chapter we have presented a translation scheme mapping the λ-calculus into Milner's π-calculus. Reduction of the resulting term modelled the parallel graph reduction of the original λ-calculus term. Our scheme improves on two such schemes given by Milner in that we are able to preserve non-strict semantics and sharing while exploiting parallelism wherever safely possible. While we will not use our scheme directly for a distributed implementation of functional programming languages, its conciseness makes it extremely valuable for comprehending the more practical compilation scheme described in the next chapter and from which the π-calculus scheme was abstracted.

Chapter 4 — A Practical Compilation Scheme

4.1 Introduction

The π-calculus simulation of the λ-calculus which we presented in the previous chapter shows a clear path to a possible distributed implementation of modern functional programming languages. It is well-known that the latter can be viewed as 'sugared' versions of the λ-calculus and the techniques for translating them into the λ-calculus are textbook knowledge [Pe87, Da92]. The missing link for a complete distributed implementation of parallel functional languages is 'merely' an implementation of the π-calculus for a suitable distributed architecture. While such an approach is feasible, we do not advocate it. Since the π-calculus is not close to any specific parallel architecture, an implementation based directly on our π-calculus scheme would take roughly the following shape:

functional programming language

↓

λ-calculus

↓

π-calculus

↓

abstract parallel machine model

↓

concrete parallel machine

Such a multi-level implementation is liable to result in very inefficient code. Instead, we view our π-calculus model as a specification for a more direct compilation scheme which will be the subject of this chapter. Rather than going from the λ-calculus to the π-calculus, this compilation scheme will map a more conventional functional language directly to an abstract parallel machine model, resulting in a system with the following structure:

functional programming language

$\Downarrow$

abstract parallel machine model

$\downarrow$

concrete parallel machine

We will focus on the compilation scheme performing the transformation marked with '$\Downarrow$'. This scheme displays profound similarities to the π-calculus model of the previous chapter and the latter will greatly aid understanding of this chapter. We will point out correspondences along the way.

We begin our exposition by establishing a framework for the message-passing multi-threaded architectures underlying our work. Armed with this architectural background we then specify the runtime framework that will make up the abstract model representing the target of our compilation scheme. Following a section on the source language, the bulk of this chapter is concerned with the compilation rules.

4.2 Source language

The simple first order source language for which we will present our compilation rules is shown below. We omitted higher order functions for clarity. They can easily be added in the fashion described by Nikhil in [Ni89] without changing the reasoning underlying our arguments*. In the absence of higher order functions all our function applications are saturated supercombinator calls.

* Their method is based on the representation of a higher-order function (closure) as a pair consisting of a first-order function and a partial argument list. Function definitions are expanded into a series of functions of two arguments, each extending a partial argument list by an extra argument with the final one performing the application proper. Higher order applications are expanded into applying the first component of a closure value (the function) to two arguments: the second component of the closure (the partial argument list) and the argument to the higher-order application.

Prog	::=	$\text{Def}_1 \ldots \text{Def}_n$ Exp
Def	::=	f $\text{id}_1 \ldots \text{id}_n$ = Exp
Exp	::=	const
Exp	::=	id
Exp	::=	**apply** f $\text{Exp}_1 \ldots \text{Exp}_n$
Exp	::=	**let** $\text{id}_1 = \text{Exp}_1, \ldots, \text{id}_n = \text{Exp}_n$ **in** Exp
Exp	::=	**if** Exp_1 **then** Exp_2 **else** Exp_3
Exp	::=	**cons** Exp_1 Exp_2
Exp	::=	$\text{Exp}_1 + \text{Exp}_2$

The last two are representative of construction and primitive operations, respectively. Several types of expressions will have both lazy and eager rules associated with them, as was the case for function application in our π-calculus scheme. As we did for our π-calculus simulation, we will assume the presence of a strictness analyser to resolve any resulting ambiguities in the compilation scheme.

4.3 Architectural framework

The machine model underlying our compilation scheme is that of a message-passing multicomputer made up of multithreaded processing elements (PEs), each with their own local memory, such as the P-RISC model of Arvind and Nikhil [AN88] outlined in Section 2.4.2, or our own STAR:DUST architecture [Os91] described in Chapter 6 which we will briefly summarise here. We assume that of the multiple threads that may be able to proceed only one *thread* per CPU executes at any one time. Threads execute to completion and cannot preempt each other. A thread in our sense is best described as an ordinary von Neumann program, operating on registers, exchanging data with local memory, sending messages to other nodes and eventually terminating. Upon termination of a thread, the next incoming message is consumed, i.e., its contents are transferred to registers and the handler thread specified by the message is inititated. A similar model, 'active messages', was described by von Eicken and Culler et al. in [EC92]. Threads communicate in one of two ways: a thread can start other threads ('dataflow style') or it can exchange data with another via the local store. In the latter case both threads need to reside on the same PE. Several threads will often run within the context of a single *frame*, i.e., a range of memory in the local store of one PE. We will find it convenient to call such a collection of threads a *process*. While the STAR:DUST architecture provides direct support only for the simpler concept of threads, processes in this sense are easily modelled.

4.4 Runtime framework

Our framework for graph reduction on multithreaded architectures is based on the following principles:

- The whole program code resides on each node in the same relative location in order not to waste communications bandwidth on feeding the instruction units.
- There are three classes of memory objects, namely function frames, constructed cells and suspension cells. Each object resides on a single PE and can be referenced by a *global pointer* comprised of PE identifier and an address local to that PE.
- Each function invocation is associated with a unique function frame which holds the arguments plus space for any necessary temporary local data.
- Function invocations are allocated randomly to PEs and all of the code associated with the invocation executes on the PE which holds its frame.

In order to give some structure to our compilation scheme, we will distinguish the three types of message-send operations listed below. Each can be interpreted as a macro which will construct a message from a number of scalar arguments, inject it into the network and *continue execution asynchronously*.

- **apply** (f, x_1, ... , x_n, o)

  Send a message to a random PE to start the execution of the specified function. The function argument `f` is the address of the entry point of a function. By convention the function will allocate a frame in which to store the arguments x_i and a pointer `o` to the node in which to place the result. The x_i represent pointers to the nodes where the argument values can be obtained, if and when necessary, Before terminating, the function will cause the result node to be overwritten with a value in WHNF. An `apply` message corresponds very directly to the π-calculus action $\bar{f}(x,o)$ in the rule for function application (Section 3.4.4 and 3.4.5), extended to deal with multiple arguments.

- **eval** (x, label)

  Send a message to the PE holding the suspension node `x` to initiate its evaluation, if not previously initiated. Recall that suspension pointers are machine global and therefore `x` is sufficient to specify the destination PE. The handler for `eval` registers the current process as a consumer of the suspension's value when it becomes available, using a linked-list data structure. The current process is specified as a continuation consisting of a global pointer to the caller's frame (an implicit parameter to `eval`) and a label in the caller's code. An `eval` message corresponds to our π-calculus rule for name reference (Section 3.4.2), the major difference being that the target 'channel' is mapped into the caller's frame rather than being an independent object.

- **update** (o, value)

  Send a message to the PE holding the suspension node o. There is a single update handler on each PE which overwrites a suspension with its value and restarts any waiting processes. Restarting involves sending further messages (not explicitly described here) which carry the value to each requester. Update messages correspond to occurrences of $\bar{o}x$ in our π-calculus rules of the form [[M]] o, e.g., the rule for name reference (Section 3.4.2), and are used for depositing constant values, and the results of primitive computations like + and `cons`.

It is worth pointing out that the suspension nodes of our runtime model play the dual role of request channels along which consumers can request the value of a computation (via an `eval` message) and value channels along which producers deliver their results (via an `update` message).

4.5 Overview of the compilation scheme

There are three compilation functions, $\mathscr{P}$ for compiling programs, $\mathscr{D}$ for definitions and $\mathscr{E}$ for expressions, each mapping a piece of abstract syntax into our message passing abstract model. We will present the resulting code in a C-like notation that we hope will appeal better to the reader's intuition than the short sequence of RISC instructions to which each line corresponds. In this code, all communication is explicit and takes the form of one of the message types outlined in the previous section (`apply`, `eval` and `update`).

The $\mathscr{P}$ scheme applies to the main program only and provides the packaging for top-level definitions and the main program expression. The $\mathscr{D}$ scheme compiles top level function definitions (supercombinators) and is again mostly packaging for the code generated for the function body. All the interesting work is done by the $\mathscr{E}$ scheme which generates code for expressions. Entering the $\mathcal{C}$ scheme code for an expression amounts to initiating its evaluation. The $\mathcal{C}$ scheme takes as an additional parameter a (pointer to) the suspension node which is to be overwritten with the result of the expression. The resemblance to computation in continuation passing style is not coincidental, with the target suspension pointer taking the place of the 'rest of the program'. Our scheme is not equivalent to CPS computation, however, since we continue executing the thread into which the $\mathcal{C}$ scheme code is embedded, often in parallel with new threads that were spawned by this code.

Let us have a closer look, starting with the $\mathcal{P}$ scheme: We allocate a suspension node for the main program expression, initiate its evaluation via an `eval` message and terminate (denoted by a horizontal bar). The result will eventually arrive, restarting the main process at `L`. The resulting `value` is accessible via the message passing interface and does not involve any additional communication—it is mapped to a register in our STAR:DUST architecture. Upon arrival of the result we can print it (assuming the result is a printable scalar value for simplicity) and terminate the program. If the result of the main program is a list, further `eval` messages are required for the head and the tail, respectively, until the end of the list is reached. In the general case, the structure of the `print` operation depends on the type of the result. The main program code is followed by that for the supercombinator definitions and for the main program expression resulting in the following compilation rule:

```
𝒫 [[ Def₁ ... Defₙ Exp ]] =

        s = suspended_node (M)
        eval (s, L)
        ___________
    L:  print (msg.value)
        ___________
        𝒟 [[ Def₁ ]]
              ⋮
        𝒟 [[ Defₙ ]]
    M:  𝒞 [[ Exp ]] → s
        ___________
```

Note that since s has been allocated locally we could in this case replace the eval message by a construct not involving communication, an option not exploited here in order to maintain consistency of presentation with the compilation rules to follow.

The 𝒟 scheme translates a supercombinator into a handler which is activated by corresponding apply messages. For each activation the handler allocates sufficient space to hold (pointers to) the arguments, a pointer to the destination node for the result and any private temporary space required during evaluation. Before entering the code for the supercombinator body, the function handler needs to save the message parameters into the newly allocated frame.

```
𝒟 [[ f id₁ ... idₙ = Exp]] =

    f:  allocate_frame
        id₁ = msg.id₁
              ⋮
        idₙ = msg.idₙ
        result = msg.result
        𝒞 [[ Exp ]] → result
        ___________
```

The handler so produced corresponds directly to the handler in our π-calculus rule for abstraction (Section 3.4.3). In the latter case the handler had the form $!f(x,y).\ [[M]]\ y$, i.e., the handler was represented as the parallel composition of an unspecified number of processes, each responsible for a single activation. Note how the idea of 'infinite composition' is mapped into a passive handler which does not require any resources until activated. Note also that whereas the π-calculus scheme can rely on an abstract notion of environments we have to introduce explicit frames in which to deposit the values we want to bind.

4.6 Translating non-strict expressions

Having had a taste of the kind of code produced by our scheme, we are now ready for the $\mathfrak{C}$ scheme which handles expressions. In this section we will provide the rules for lazy function application, the lazy `let` construct and the lazy version of `cons`. We will start, however, with the rules for numeric constants and identifier reference which are transparent to issues of strictness. All our rules will be of the general form

```
𝕮 [[ Exp ]] → result =
          ... code ...
```

where `Exp` is the expression to be evaluated and `result` is the name of the suspension node which is to receive the result. In other words, $\mathfrak{C}$ is a function of two arguments, a piece of abstract syntax and a name. Note that the structure of this scheme is identical to that of our π-calculus scheme of Chapter 3. There we had a compilation function of the form $[[M]]\ o$ where M was a term of the λ-calculus and o was a name of the π-calculus.

4.6.1 Constants

The rule for constants is easy enough. We simply send the value of the constant to the result node. Note that the `update` message is not followed by termination, so we continue executing the code into which the code for the constant is embedded.

```
𝒞 [[ const ]] → result =
        update (result, const)
```

This rule corresponds directly to the rule for constants (Section 3..3) in our π-calculus scheme which had the form

$$[[\,\text{const}\,]]\; o \quad \stackrel{\text{def}}{=} \quad \overline{o}\;\text{const}$$

4.6.2 Identifier reference

Before we can update the result node for an identifier reference, we need to make sure via an `eval` message that the value is available in WHNF. We transmit the label `L` to specify where execution is to continue once the value arrives and then terminate. On arrival of the value, we update the result as before.

```
𝒞 [[ id ]] → result =
        eval (id, L)
        ________
    L:  update (result, msg.value)
```

At this point the reader is likely to notice that there is a substantial amount of communication going on even for the simplest kinds of expressions. We therefore hasten to point out that the compilation rules given here are only meant to cover the general case as concisely as possible. For many of the messages produced here source PE and destination PE coincide. More specialised rules can be given to avoid message-passing overhead altogether for important special cases.

Comparing the rule for identifier reference with the corresponding one in the π-calculus scheme (Section 3.4.2) we find that we have not made use of the shortcut described there which would cut out one communication step. Such a shorthand is perfectly possible and would involve the definition of a second kind of `eval` message which would take the address of a target suspension node as a parameter. We abstained from this optimisation to simplify our presentation.

4.6.3 Lazy function application

The code generated for a lazy function application starts by allocating suspension nodes for each of the parameters, marking them 'suspended' to indicate they are as yet unevaluated, and storing within it the continuations of the computations that can determine the parameter values if and when required. The code for the individual expressions is then bypassed and the function is activated, with pointers to the suspension and result nodes passed as parameters.

```
$\mathcal{C}$ [[ apply f Exp$_1$ ... Exp$_n$ ]] $\rightarrow$ result =
            s$_1$ = suspended_node (L$_1$)
                ⋮
            s$_n$ = suspended_node (L$_n$)
            goto L$_{n+1}$
    L$_1$:  $\mathcal{C}$ [[ Exp$_1$ ]] $\rightarrow$ s$_1$
            ___________
                ⋮
            ___________
    L$_n$:  $\mathcal{C}$ [[ Exp$_n$ ]] $\rightarrow$ s$_n$
            ___________

    L$_{n+1}$: apply (f, s$_1$,..., s$_n$, result)
```

This rule does not appear to correspond very closely to the rule for lazy application in our π-calculus scheme (Section 3.4.5) for two reasons. Firstly, the rule above only implements first-order function application, i.e., `f` is a code label rather than an expression. Secondly, as pointed out in Section 4.4

on the runtime model, suspension nodes play a dual rule as both request channels and value channels. Thus some of the functionality of the original π-calculus application rule is here performed by the handlers for suspension nodes, in particular the waiting for and servicing of requests.

4.6.4 Lazy `let`

This rule is structurally sufficiently similar the rule for lazy function application above for us to restrict ourselves to pointing out the two differences: instead of introducing anonymous suspension nodes for the arguments to a function application, we will use the names provided by the `let` construct. Instead of sending an `apply` message, we simply evaluate the body of the `let` expression in line.

```
𝒞 [[ let id₁ = Exp₁, … , idₙ = Expₙ in Exp ]] → result =
        id₁ = suspended_node (L₁)
           ⋮
        idₙ = suspended_node (Lₙ)
        goto Lₙ₊₁
   L₁:   𝒞 [[ Exp₁ ]] → id₁
        ____________
           ⋮
        ____________
   Lₙ:   𝒞 [[ Expₙ ]] → idₙ
        ____________
   Lₙ₊₁: 𝒞 [[ Exp ]] → result
```

4.6.5 Lazy `cons`

Again the similarities to lazy function application are striking. Obviously we only have two arguments to deal with. The pointers to suspension nodes allocated for them are stored into a locally allocated frame for the `cons`-cell. The result node is updated with a pointer to this cell.

```
𝒞 [[ cons Exp₁ Exp₂ ]] → result =
        s₁ = suspended_node (L₁)
        s₂ = suspended_node (L₂)
        goto L₃
   L₁:  𝒞 [[ Exp₁ ]] → s₁
        ____________
   L₂:  𝒞 [[ Exp₂ ]] → s₂
        ____________
   L₃:  c = cons (s₁, s₂)
        update (result, c)
```

4.6.6 Conditional

Given that we are dealing with a lazy functional language we could map the conditional construct to an appropriate built-in function

```
cond (test, then, else)
```

which will return the value of either its second or its third argument depending on the value of the first. Nonetheless we give a direct compilation rule here.

```
𝒞 [[ if Exp₁ then Exp₂ else Exp₃ ]] → result =
        s = busy_node ()
        𝒞 [[ Exp₁ ]] → s
        eval (s, L₁)
        ____________
   L₁:  if (!msg.value) goto L₂
        𝒞 [[ Exp₂ ]] → result
        goto L₃
   L₂:  𝒞 [[ Exp₃ ]] → result
   L₃:
```

While we do not suspend the computation for $\mathtt{Exp}_1$, we still allocate a graph node `s` for its value and mark it 'busy', i.e., currently undergoing evaluation. As evaluation of $\mathtt{Exp}_1$ has already begun by the time we send the eval message, this message here only serves the purpose of synchronisation. When the value of $\mathtt{Exp}_1$ becomes available in `s`, we restart the current process at $\mathtt{L}_1$, evaluating either $\mathtt{Exp}_2$ or $\mathtt{Exp}_3$ based on the result.

4.7 Translating strict expressions

While the rules presented in the previous section employ message-passing and distribute function applications across processing elements, they do not provide us with any opportunities for parallelism as any instance of thread creation is swiftly followed by the termination of the original thread. To verify this claim note that in the scheme presented so far all recursive invocations of $\mathcal{C}$ are immediately followed by a termination. In order to obtain parallelism we need to take advantage of the known strictness of built-in operations like addition, or we have to perform strictness analysis on the rest of the program and translate selected expressions according to the rules presented in this section. To simplify the presentation we will only show the code generated for expressions that are strict throughout, e.g., function applications that are strict in all the arguments. Hybrid rules, such as the one for the conditional construct, would have to be employed for a more complete implementation. The rules given below are 'naive' in that they exploit maximal parallelism. They will be adapted in the next chapter to guarantee load bounding.

4.7.1 Eager function application

As in the code for lazy function application we allocate a suspension node for each of the parameters. Rather than actually suspending the computations in question, we immediately enter the code for each of the arguments. Parallelism comes about if any of the argument expressions itself involves a function application which results in starting a new thread on a (generally) remote processing element.

```
𝒞 [[ apply f Exp1 ... Expn ]] → result =
        s1 = busy_node ()
        𝒞 [[ Exp1 ]] → s1
           ⋮
        sn = busy_node ()
        𝒞 [[ Expn ]] → sn
        apply (f, s1,..., sn, result)
```

Note that under a parallel execution regime strictness information can generally *not* be used to avoid 'boxing', i.e., the allocation of graph nodes for subexpressions. To avoid such graph nodes we would have to evaluate the argument expressions *before* entering the function code, thus limiting parallelism. Such a strategy may still be interesting in special cases, for example if the function invoked will immediately require the argument values.

4.7.2 Eager `let`

Again, as in the lazy case, the code generated for the `let` construct is almost identical to function application. Again the only difference is that we use the identifier names supplied by the programmer and we enter the body of the expression directly rather than sending an `apply` message.

```
𝒞 [[ let id₁ = Exp₁, ... , idₙ = Expₙ in Exp ]] → result =
        id₁ = busy_node ()
        𝒞 [[ Exp₁ ]] → id₁
            ⋮
        idₙ = busy_node ()
        𝒞 [[ Expₙ ]] → idₙ
        𝒞 [[ Exp ]] → result
```

4.7.3 Eager cons

The eager version of cons is constructed completely analogously and is given for the sake of completeness.

```
𝒞 [[ cons Exp₁ Exp₂ ]] → result =
        s₁ = busy_node ()
        𝒞 [[ Exp₁ ]] → s₁
        s₂ = busy_node ()
        𝒞 [[ Exp₂ ]] → s₂
        c = cons (s₁, s₂)
        update (result, c)
```

4.7.4 Addition

Finally we provide the compilation rule for addition.

```
𝒞 [[ Exp₁ + Exp₂ ]] → result =
        s₁ = busy_node ()
        𝒞 [[ Exp₁ ]] → s₁
        s₂ = busy_node ()
        𝒞 [[ Exp₂ ]] → s₂
        eval (s₁, L₁)
        ___________
    L₁: v₁ = msg.value
        eval (s₂, L₂)
        ___________
    L₂: v₂ = msg.value
        update (result, v₁+v₂)
```

This rule will require a short explanation. After initiating the evaluation of both arguments in parallel, like we did for any of the other strict constructs, this time we require their actual value for addition to complete its task. Therefore we perform synchronisation via `eval` messages to obtain the numeric values of the arguments. As the evaluation of both arguments has already started, the eval message will merely serve to request their values to be reported back to the current process. As given, the rule requests first the value of the left subexpression and then the value of the right subexpression. This does not impose any sequentiality on the order in which the arguments are *evaluated,* only on the order in which their values are *reported*. As both graph nodes are managed by the local processing element anyway, we have nothing to gain from eliminating this admittedly artifical order.

As the rule for addition has a counterpart in the π-calculus scheme of the previous chapter it will be useful to compare the two. The correspondence here is a very natural one. Our π-calculus addition rule (Section 3.3) started by introducing two names corresponding to the allocation of two graph nodes. We had one parallel process responsible for computing the value of either of the subexpressions. Like in the above rule, our π-calculus code reads first the value of the left subexpression, then the right, before writing the value of their sum to the output channel.

4.8 Summary

In this chapter we have made more concrete the abstract simulation in the π-calculus of graph reduction in the λ-calculus, as presented in Chapter 3. This was done in the form of a practical compilation scheme for message-passing multicomputers. Our rules were divided into two sets. The group of rules dealing with instances of language constructs not known to be strict did not create any parallelism. All parallelism to be obtained was provided by rules that applied to expressions that were strict throughout. We have pointed out along the way the multiple close correspondences between the compilation scheme and the π-calculus simulation of the previous chapter.

Chapter 5 — Load Bounding

5.1 Introduction

Compilation rules like those of the previous chapter allow us, in principle, to exploit the maximal parallelism implicit in a functional program, starting new threads whenever the opportunity arises. As pointed out in Section 1.7.2, however, maximal parallelism may be rather more than we desire. We argued there that without a prudent strategy for *bounding* parallelism many programs, in particular those of the divide-and-conquer type, can exhibit vastly increased resource requirements. In this chapter we will present a simple automatic technique for bounding the parallelism of executing programs dynamically and adaptively, based on the changing workload of the underlying parallel system. Our technique will be integrated in a cheap and simple manner into the compilation scheme of Chapter 4. We present an informal proof of the effectiveness of our method based on the structure of the compilation scheme. Determining the workload of a large parallel computer is not a trivial task, involving as it does a form of global synchronisation. We have to ensure that its computational cost does not have undue impact on the runtime of the user program. We present a simple and cheap load computation algorithm with the required properties. Finally we point out why the sharing of subcomputations can have a detrimental impact on the effectiveness of load bounding.

5.2 Pitfalls and dangers

Given that the compilation scheme presented in the previous chapter provides distinct rules for parallel and sequential execution of some of the same program constructs, the first idea that comes to mind is to apply the parallel rule when additional parallelism is required and the sequential rule when the machine is already saturated with work. More specifically, we could extend the definition for the eager case to revert to the lazy case if a dynamic test determines that the current workload already exceeds a fixed limit, as in the following scheme

```
    𝒞 [[ Exp ]] → result =

           if (workload > limit) goto P
           𝒞_lazy [[ Exp ]] → result

           goto E
    P:     𝒞_eager [[ Exp ]] → result

    E:
```

Provided we can implement the computation of the current workload cheaply (see Section 5.6), this scheme provides an effective means of bounding parallelism. It has, however, an obvious and severe drawback in that we generate duplicate (if slightly different) instances of code for the same expression. Both the code for the eager case and the code for the lazy case can in turn contain more strict subexpressions which would again result in code duplication, and so on. In the worst case the size of code generated by such a scheme would be exponential in the length of the original code—an unacceptable proposition. A second, less obvious disadvantage of this method is the fact that it is not adaptive. On entering the code for an expression Exp with subexpressions $\texttt{Exp}_1$ ... $\texttt{Exp}_n$ we may decide, based on the current workload, that the subexpressions ought to be computed serially. If the workload has dropped after computation of $\texttt{Exp}_1$ is complete and additional parallelism is desirable, there is no opportunity for evaluating the remaining subexpressions $\texttt{Exp}_2$ ... $\texttt{Exp}_n$ in parallel with each other. The scheme presented below avoids the code explosion as well as being more adaptive in situations of changing workload.

5.3 A workable solution

Before going into the details of our extended compilation scheme we will introduce the invariant on which our informal proof of the effectiveness of our load bounding method will be based:

While the workload is high, no thread will start more than one new thread. After creating a new thread, the current thread terminates immediately.

This invariant will apply, in particular, to the thread which starts the evaluation of a complex expression, i.e., it will only be able to start one successor thread during situations of high workload. The same applies to the thread thus started and its descendants. It is therefore not possible for complex expressions to enter two subexpressions simultaneously as long as the workload remains high, thus guaranteeing the effect of depth-first execution we desire (see Section 1.7.2).

When analysing a compilation rule for the preservation of the invariant we have to keep in mind that the code generated for most expressions consists of multiple threads. Each of these threads must be analysed separately. Furthermore, threads transcend the boundaries of the code generated by our compilation rules, i.e., one thread will go into the code generated for an expression and another will leave it at the end. In designing our scheme we have stuck to the convention that no thread creates other threads prior to entering the code generated by the $\mathcal{C}$ scheme. However, threads often use up their allotment of one extra thread just prior to leaving the code for an expression.

Let us consider the lazy compilation rules given in Section 4.6. We will repeat them here for easy reference, starting with the rule for constants.

```
𝒞 [[ const ]] → result =
        update (result, const)
```

This rule consists of a single thread sending a single message. By our convention, we will not have started a thread prior to entering this code, thus satisfying the invariant*. Next we will consider the rule for identifier reference, repeated below.

```
𝒞 [[ id ]] → result =
        eval (id, L)
        ____________

    L:  update (result, msg.value)
```

This code consists of two threads. The thread entering here has not yet started any new threads by our convention. It sends a single message and terminates immediately. The outgoing thread sends a single message, again preserving the invariant. Now for a slightly more involved example, the rule for lazy function application.

```
𝒞 [[ apply f Exp1 ... Expn ]] → result =
        s1 = suspended_node (L1)
           ⋮
        sn = suspended_node (Ln)
        goto Ln+1
    L1:  𝒞 [[ Exp1 ]] → s1
        ____________
            ⋮
        ____________
    Ln:  𝒞 [[ Expn ]] → sn
        ____________

    Ln+1: apply (f, s1,..., sn, result)
```

* but see the restriction on sharing detailed in Section 5.7

Here the main thread extends through the code generated (circumventing the suspended argument expressions by means of a goto), producing only a single message before leaving. Each of the threads responsible for computing the value of one of the suspended argument expressions directly enters the code for the corresponding subexpression. This code can generate only a single thread (by a simple inductive argument). In order to maintain the invariant we have to terminate immediately, and we do.

Identical arguments can be made for the rules applying to the lazy let construct and the lazy cons operator. Thus we can see that all our lazy rules satisfy the invariant irrespective of the current load situtation. This was of course only to be expected: as mentioned before, there is no parallelism in purely lazy evaluation. For the eager rules of Section 4.7, however, the inductive argument would quickly break down. In the following section we will therefore adapt those rules to comply with our invariant.

5.4 Eager compilation rules modified for load bounding

For the eager rules that follow, we will assume the existence of a boolean test operation parallel which will determine, at any instant during the execution of a program, whether the current load status of the machine permits the creation of new parallel threads. We will discuss in Section 5.6 how such a test can be implemented efficiently.

5.4.1 Eager function application

The adapted rule for eager function application which incorporates the core idea of our scheme is given below. We immediately observe that we generate code for each subexpression only once.

```
$\mathcal{C}$ [[ apply f Exp$_1$ ... Exp$_n$ ]] → result =
    L$_1$:  s$_1$ = busy_node ()
            $\mathcal{C}$ [[ Exp$_1$ ]] → s$_1$
            if (parallel) goto L$_2$
            eval (s$_1$, L$_2$)
            ____________
                 ⋮
            ____________

    L$_n$:  s$_n$ = busy_node ()
            $\mathcal{C}$ [[ Exp$_n$ ]] → s$_n$
            if (parallel) goto L$_{n+1}$
            eval (s$_n$, L$_{n+1}$)
            ____________

    L$_{n+1}$: apply (f, s$_1$, ... , s$_n$, result)
```

Let us first verify that the modified eager rule still performs the required operation. Assuming that each of the tests for additional parallelism succeeds, it is readily apparent that but for the testing overhead the sequence of operations is identical to that for the naive eager rule of Section 4.7.1. Now what happens if the first test for extra parallelism fails? Instead of immediately entering the code for the second subexpression, we send an `eval` message to the suspension node for the first subexpression and terminate. Since the evaluation of s_1 has already started (we have just initiated the code for Exp_1), the `eval` message will merely register the current process as a consumer of the resulting value, once available. Only when this value has been computed is the current process restarted at L_2. The `eval` message therefore merely has the effect of synchronising the start of the second subcomputation with the arrival of the result of the first, leaving the course of the computation otherwise unchanged. The same argument applies to each of the subcomputations in turn. The arrival of the result for the last subexpression is synchronised with the activation of the function to be called.

Knowing that the above rule implements function application correctly, independently of the load situation, what can we say about its effect on system load? In order to preserve the invariant of the previous section we again need to verify that each thread of the code generated by the modified rule has at most a single successor thread while the workload is high. So let us assume the system is already saturated with parallelism on entering the code generated by the above rule. We start evaluation of the first subexpression which will, in general, create a new thread. We now have to guarantee that more threads are not created. The test for more parallelism will fail so we send an `eval` message and terminate. At first sight, it would appear this `eval` message should initiate a new thread. However, as pointed out in the previous paragraph, this message has the sole effect of registering the current process with the suspension node $\mathtt{s}_1$. Since $\mathtt{s}_1$ has been allocated locally, the obvious way to implement this 'message' is to change the contents of the suspension node *without* starting an extra thread, thus preserving the invariant for the initial thread. Again the same argument applies to each of the subsequent threads. If the workload remains high throughout, each thread will initiate one more subcomputation and terminate after changing the contents of the corresponding suspension node. Finally the last thread will start a single new thread via the `apply` message, again complying with the invariant.

5.4.2 Eager `let`

After having dealt successfully with the rule for eager function application, the rest of our rules follow swiftly along the same lines. Here is the adapted eager rule for the `let` construct:

```
𝒞 [[ let id₁ = Exp₁, ... , idₙ = Expₙ in Exp ]] → result =
    L₁:  id₁ = busy_node()
         𝒞 [[ Exp₁ ]] → id₁
         if (parallel) goto L₂
         eval (id₁, L₂)
         ____________
              ⋮
         ____________
    Lₙ:  idₙ = busy_node()
         𝒞 [[ Expₙ ]] → idₙ
         if (parallel) goto Lₙ₊₁
         eval (idₙ, Lₙ₊₁)
         ____________
    Lₙ₊₁: 𝒞 [[ Exp ]] → result
```

The eager `let` construct differs from eager function application merely in the naming of the suspension nodes and in the inline expansion of the body of the `let`. The argument for the invariant is precisely the same as above.

5.4.3 Eager `cons`

Neither does the eager version of cons produce any difficulties, as the reader will immediately verify. Here is our modified rule:

```
𝒞 [[ cons Exp₁ Exp₂ ]] → result =
   L₁:  s₁ = busy_node ()
        𝒞 [[ Exp₁ ]] → s₁
        if (parallel) goto L₂
        eval (s₁, L₂)
        ___________
   L₂:  s₂ = busy_node ()
        𝒞 [[ Exp₂ ]] → s₂
        if (parallel) goto L₃
        eval (s₂, L₃)
        ___________
   L₃:  c = cons (s₁, s₂)
        update (result, c)
```

5.4.4 Eager addition

In conclusion, we present the modified rule for addition, explained below.

```
𝒞 [[ Exp₁ + Exp₂ ]] → result =
    L₁:  s₁ = busy_node ()
         𝒞 [[ Exp₁ ]] → s₁
         if (parallel) goto L₂
         eval (s₁, L₂)
         ___________
    L₂:  s₂ = busy_node ()
         𝒞 [[ Exp₂ ]] → s₂
         eval (s₁, L₃)
         ___________
    L₃:  v₁ = msg.value
         eval (s₂, L₄)
         ___________
    L₄:  v₂ = msg.value
         update (result, v₁+v₂)
```

Here we note that the original version already needed to synchronise the arrival of both argument values with the computation of their sum. Our modification to the original rule therefore consists of adding one additional synchronisation to make sure that evaluation of the second argument is not started until evaluation of the first has completed if the workload is high. The second `eval` message to s_1 in the load bounded case could be avoided at the expense of duplicating the code for $\mathtt{Exp}_2$.

5.5 Adapting to changes in the workload

The compilation rules provided in the previous section implement a load bounding regime which adaptively responds to changes in the workload. This adaptivity can come about in two ways. The first and more obvious results from the fact that we perform a test of the current workload on each occasion where we have the option of exploiting parallelism. Imagine for example an instance of an eager function application of the form

```
f exp1 exp2
```

where both of the subexpressions themselves provide opportunities for parallelism. Even if we decide to perform $\texttt{exp}_1$ and $\texttt{exp}_2$ in series, we will be able to exploit any *inner* parallelism inherent to either $\texttt{exp}_1$ or $\texttt{exp}_2$ should the system workload have dropped in the meantime.

Less obvious is the fact that even *after* choosing the sequential option for the function application, a drop in workload may result in evaluating some remaining parts of $\texttt{exp}_1$ and $\texttt{exp}_2$ in parallel. Consider for example an expression of the form

```
f (cons exp1 exp2) exp3
```

where both the application of `f` and the evaluation of `cons` can be performed eagerly in parallel. Consider the case where we decide, based on the current workload, to wait for the result of `cons` before evaluating the second argument to `f`. Imagine that immediately after taking this decision the workload drops to a level where we could again make effective use of additional parallelism. The code generated for `cons` (see previous section) would now opt for eager parallelism, start the evaluation of both $\texttt{exp}_1$ and $\texttt{exp}_2$ *and* return a newly-allocated cons-cell immediately. The computation of $\texttt{exp}_3$ could therefore start before the evaluation of the first argument has completed. This second form of adaptivity is restricted, however, to cases where the first expression in a series involves a non-strict constructor of which `cons` is out archetypical example.

5.6 Efficient load computation and load testing

The efficiency of the scheme presented in the previous section relies heavily on the ability to test the global workload quickly and cheaply. On a scalable parallel system, however, the local workload on each PE, and thus the average global workload is subject to sudden change. We cannot expect to be able to present an accurate and instantaneous picture of the average global workload to each PE. The situation is somewhat simplified by the fact that we are not ultimately interested in the precise figure for the global workload itself. We only need to know: is the global workload 'too high' or is it 'too low'? Put differently: should we attempt to *expand* parallelism or *reduce* it[*]. Two approaches came to our mind to allow PEs to make this decision quickly and on the fly.

5.6.1 Probabilistic load estimation

Our first intuition was to use the local workload as an estimate for the global workload. We adopt as a measure of the local workload the number of waiting messages at a PE which corresponds to the length of the token queue in a dataflow architecture. Due to our strategy of randomly allocating function applications to processors, we can expect decisions based on this estimate to be reliable with high probability in situations of high workloads and low workloads. However, even when the global workload is very high, there remains a finite probability of taking a wrong decision: an individual PE which is temporarily underutilised may start new parallel subcomputations and thus increase the global workload further. In a situation where all computations are very long-lived, i.e., when there are no matching reductions in parallelism in the form of completing threads, the global workload may exceed any given bound. This is the case, for example, for `nfib`. Simply replacing load computation by probabilistic load estimation is unsatisfactory.

5.6.2 Cyclic load determination

Our second attempt was based on the following idea: rather than relying on an inaccurate *estimate* that is *instantly* available everywhere, we could

[*] Strictly speaking we don't actively reduce parallelism but rather we wait for threads to die

periodically compute an *accurate value* for the global workload and rely on this value for the *whole period*. As long as the overhead of the load computation is small relative to the intervening user computation, the necessary global synchronisation can be efficient. Due to the multithreading nature of our underlying architecture, we can interleave the load computation with the user computation, which simplifies matters considerably. In particular, we have no need for any special hardware support. The load computation can be driven by a busily waiting timer process on a dedicated root node. At the start of each new period, the timer process initiates requests to every PE to report their local workloads. Requests are distributed down a logical tree which is superimposed on the system network. Local workloads are accumulated as they are reported back up the tree so that the value that reaches the root node will be the total workload. The root computes the average workload per PE and proceeds to distribute this value back down the tree. Upon receipt of acknowledgement from every PE, the root node restarts the original timer process. Experiments with this scheme showed that we successfully solved the problem of ever-increasing workloads. However, the combination of two problems makes this method impractical as well. Firstly, the period of the load determination cycle has to be fairly large (~10000 processor cycles) to dominate the fixed overhead per node (~100 processor cycles per load computation). Secondly, since load computation messages compete for bandwidth with user messages, the load computation can be seriously delayed when parallelism is expanding quickly within one phase. By the time one phase of the load computation is complete, parallelism can already be unacceptably high.

5.6.3 A hybrid solution and its cost

The solution we eventually adopted with some success, as evidenced in Chapter 6, is a hybrid of the two approaches above. We do cyclic load computation to be able to put a cap on parallelism, but we also decide in favour of sequential execution on any node whose local load is unacceptably high. Improvements to this scheme are possible by adding hardware support, for example in the form of a hierarchical control network which is separate from the data network and is suitable for fast global synchronisation. Such a control network has been implemented in the CM-5 [Le92] and is valuable for various other functions, e.g., for synchronising distributed garbage collection not discussed in this thesis.

The computational cost of our scheme has two components, the cost of the actual workload computation and the cost of the test operations that we need to integrate into our compilation scheme. Our implementation of the distributed workload computation has the structure described in the previous section. In Figure 5.1 we show the protocol followed by each node in the tree. The four handlers were implemented in STAR:DUST machine code (see Chapter 6 for details of STAR:DUST) and amount to a total of about 100 RISC instructions. While message latency has a significant effect on the time from initial request to final acknowledgement, it does not affect the computational cost in a well-designed multi-threaded architecture, as processors can quickly switch to user threads instead of blocking and waiting for an immediate response. The cost per node per load computation is therefore of the order of 100 processor cycles on a single-issue RISC processor.

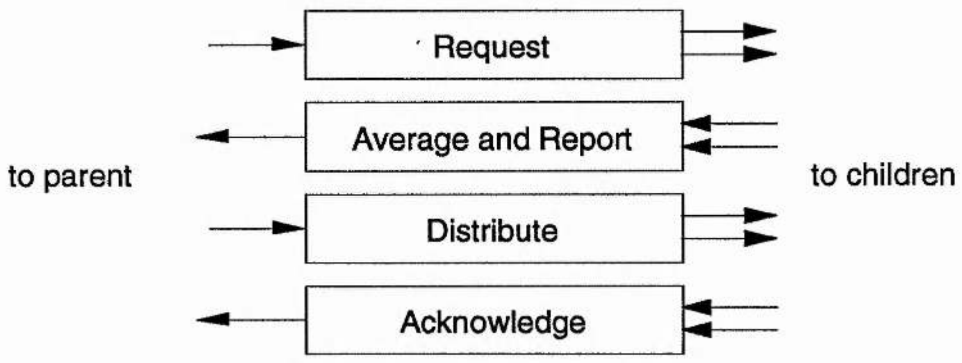

Figure 5.1: Node protocol of distributed load computation

If we set the timer process to initiate a load computation every 10000 cycles, the cost of load computation will amount to about 1% of total runtime.

The second part of the total load bounding cost is incurred by the test operations embedded in the scheme of Section 5.4. In our implementation on STAR:DUST, each such test is performed with extremely little overhead. Each test of the form

```
if (parallel) goto L
```

is mapped to a sequence of four RISC instructions as follows: we load the length of the local message queue into a register, add to that the the average global workload as determined by the last global load computation, compare the result to a constant, called the *target load value*, and branch to `L` if the sum was smaller.

5.7 The problem of sharing

There is one important exception for which our scheme is not guaranteed to succeed in load bounding. The source of this problem lies with the handler for `update` messages which overwrites nodes of the graph with values in WHNF. While an `update` message itself will only start a single new thread, namely the `update` handler, the handler in turn needs to restart any threads that have become suspended as a result of that value being previously unavailable, independently of the current system workload. Therefore, a single `update` message has the potential to restart an unbounded number of processes that were created while parallelism was low, thus defeating our load-bounding strategy. Multiple computations can only be waiting for the same suspension in the presence of sharing. Obviously sharing is an non-expendable part of functional programming languages, so this deficiency is somewhat disheartening. In practice many programs which exhibit sharing will be successfully load-bounded by our scheme as well, but we have no doubt that worst-case programs can be constructed that will provoke arbitrarily high workloads. We would not expect to be able to solve the sharing problem completely, since it is well-known that the space complexity of many lazy programs exceeds that of corresponding eager programs. Nonetheless we are investigating several approaches towards at least a partial solution and we will quickly sketch two of them.

In the case where the program is strict in a shared expression `E`, we can insist on reducing `E` to WHNF before initiating any computations which share `E`. In this way, no computation could become suspended as a result of waiting for `E` to complete. This solution has obviously limited scope and also restricts opportunities for parallelism.

Part of the sharing problem is the fact that processes suspended while waiting for an 'eval' to complete represent 'sleeping parallelism' which does not enter into our load computation. We could change the load computation method to count such processes as active. This approach will obviously restrict parallelism as well and there is the danger that pathological examples may exist which produce so much 'sleeping parallelism' that actual parallelism is starved out. Nonetheless we consider this idea sufficiently interesting to warrant further study and experimentation.

5.8 Summary

We have presented a simple and efficient software method of bounding the parallelism generated by parallel functional programs. This was achieved by modifying the naive compilation rules of the previous chapter to take into account the current workload. We have given an informal proof for the effectiveness of our algorithm based on the structure of the compilation scheme. The efficiency of our method was guaranteed by a hybrid strategy of computing the average global workload at fixed intervals and using the local workload as a rough estimate. Finally we have pointed out how sharing can disturb load bounding.

Chapter 6 — The STAR:DUST Architecture

6.1 Introduction

This chapter plays a dual role in the structure of our thesis on parallel functional programming. On the one hand we perceive a need for presenting the idea of multithreaded architectures in some more detail in order to give greater credibility to a compilation scheme and runtime system targeted at such architectures. To further this aim, our STAR:DUST architecture* will serve to illustrate that individual multithreaded microprocessors can be competitive with modern high-performance von Neumann microprocessors while at the same time providing the efficient support for communication and synchronisation which is necessary for practical parallel computing. Secondly we present STAR:DUST as an architectural contribution in its own right, interesting for the fact that its support for multithreading is reduced to two simple machine instructions, in line with the RISC principles underlying the rest of its design.

We begin this chapter by stating the architectural requirements that arise from the work presented in the earlier chapters. We will introduce STAR:DUST as an ordinary RISC microprocessor architecture before introducing the two simple extensions to the instruction set that are required to support a multithreaded model of parallel programming. After addressing some important questions regarding the resulting architecture, we conclude with a section comparing STAR:DUST to the P-RISC architecture from which it was derived.

* STAR:DUST is short for "St Andrews RISC: Dataflow Using Sequential Threads"

6.2 Architectural requirements for scalable, parallel programs

STAR:DUST is not an 'abstract' machine in the sense that it is not meant as an interface between a high-level language and concrete parallel hardware, such as the <v, G> machine of Augustsson and Johnsson [AJ89]. Rather, one of the major design goals was that it should be implementable in modern VLSI technology. Given that STAR:DUST exists only in form of a software simulation the reader may regard claims of 'efficient implementability' with caution. Our case rests largely on the fact that key aspects of our architecture have *already* been efficiently implemented individually. In particular, our architecture is only an evolutionary step from conventional RISC microprocessors and is well-positioned to take advantage of any advances in sequential RISC technology. In the following we will outline the architectural principles that guided our design.

6.2.1 Scalability

We have taken great pains not to introduce any ideas into either our compilation scheme or the load bounding system which would restrict our ability to scale up the size of the machine. In particular, we avoided introducing artifical bottlenecks not present in the user program or making assumptions about the underlying architecture which would not easily scale to large numbers of processors. Clearly scalability must now also be a major concern of the architecture. An unavoidable consequence of scaling up parallel machines is the fact that communication latencies between nodes will grow relative to access times for local memory and processor cycle time. An architecture must be able to tolerate such latencies without loss of efficiency by using what is called 'parallel slackness' by Valiant in [Va90], i.e., trading parallelism for communications latency. The key idea, which goes back at least to Sullivan and Bashkow [SB77], is to switch to another thread of control whenever the original thread would otherwise block while waiting for a communication to complete. See also Arvind and Iannucci [AI87] for an excellent discussion of this issue.

6.2.2 Topology independence

We require a complete connectivity abstraction, most likely implemented by packet routing in a relatively sparsely connected physical topology. A processor architecture should not make assumptions about particular network topologies so that the implementation of the network can proceed independently from that of the processing element and can be optimised for maximum bandwidth and minimum latency without interfering with architectural principles. The technology for building such networks has matured to commercial applicability and we will not discuss it here. See, for example, the discussion by Leiserson et al. of the network architecture of the CM-5 [Le92].

For us, topology independence also implies that we do not aim to exploit 'near-neighbour' properties among processing elements, distinguishing only between local and global data. In the terminology of Cole [Co90] we only take advantage of 'partial locality'.

6.2.3 Asynchronous, message passing communications

While continuous streams among concurrent sequential processes are a highly efficient and often conceptually simple means of communication and can be put to good use in parallel functional programming as described by Paul Kelly in [Ke89], they do not fit the bill for the compilation scheme presented in this thesis. A message passing model is more suitable for operations such as function invocation, parameter passing or dereferencing of global pointers. Due to our usage of message passing for basic language constructs we require a high sustained communications bandwidth.

6.2.4 Fast context switching and process synchronisation

Fast context switching and process synchronisation are the principal factors determining the granularity of parallelism which can be supported effectively. While excessively fine-grain parallelism has proven impractical, reducing hardware constraints on granularity remains not just a valid but a crucial goal. We need to be able to switch quickly among multiple quasi-concurrent threads on a single processing element to mask high-latency remote accesses.

6.2.5 Sequential efficiency

A suitable architecture must be able to execute strictly sequential programs on a single processing element with an efficiency comparable to that of a sequential processor. We do not want to pay the price for fine-grain synchronisation where it is not required. Locality is a crucial issue in this context. We require high-speed access to local data. This includes the relatively small but extremely fast on-processor state in the form of on-chip caches and registers on which modern RISC processor rely for their impressive performance.

6.3 STAR:DUST as a RISC processor

STAR:DUST is so close to a modern RISC design that it is best understood in terms of the basic RISC processor at its core, depicted in Figure 6.1. This processor is modelled after the Sun SPARC [Ga88] to facilitate comparison. Where we committed ourselves to concrete architectural parameters, like the number of directly addressable registers, we did so to comply with equivalent commitments in the SPARC. Like on the SPARC we have an ordinary sequential program counter, a set of 32 directly addressable registers and a status register. Instructions are executed sequentially. There are only two instruction formats compared to the SPARC's three: we fit the call instruction into the branch format. All memory access is via explicit load/store instructions. A memory manager can provide a cache to speed up memory access. The ALU operates on registers only (three-address operations). Sequential instruction streams are amenable to pipelined execution. By remaining close to a conventional architecture for the sequential part of our design we can justify our claim being able, in principle, to provide comparable sequential efficiency.

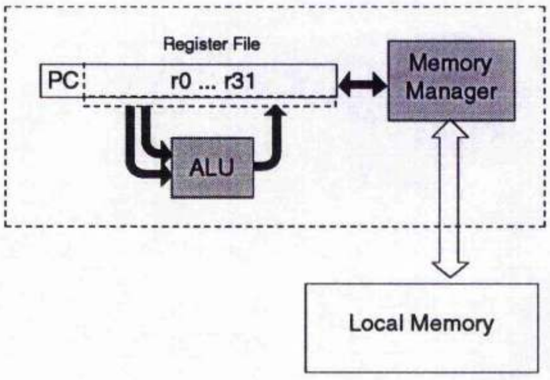

Figure 6.1: The RISC core of a STAR:DUST processor (data paths only)

6.4 STAR:DUST as a building block for multiprocessor systems

While at any time there is only a single active sequential thread per processing element, STAR:DUST supports multithreading by allowing for further dormant threads, i.e., threads ready for execution. Each dormant thread is represented by a *context* in the context store which is organized as a queue as shown in Figure 6.2. A context consists of a program counter value, plus a set of eight values for the *context registers* `r0-r7`. The program counter and the context registers together will be called the *active context*. Contexts also form the basis of our communications model. Our understanding of a *message* is a context in transit. Support for cross-processor parallelism is centered around two parallel control instructions:

- An instruction of the form `start pc` initiates a new thread, generally on a remote PE, via the following mechanism: the `start` instruction constructs a message from the `pc` operand and the values in registers `r24-r31` which we will also refer to as the *communications registers* `c0-c7`. Execution of the current thread continues immediately and asynchronously. A network manager ensures delivery of this message to the processor specified in `c0`, where the message contents are queued in the context store. In other words: the values in registers `c0-c7` of the sending processor end up in registers `r0-r7` of the receiving processor, representing a dormant thread.
- The `term` instruction terminates the active thread by taking the next context from the context store and making it the active context.

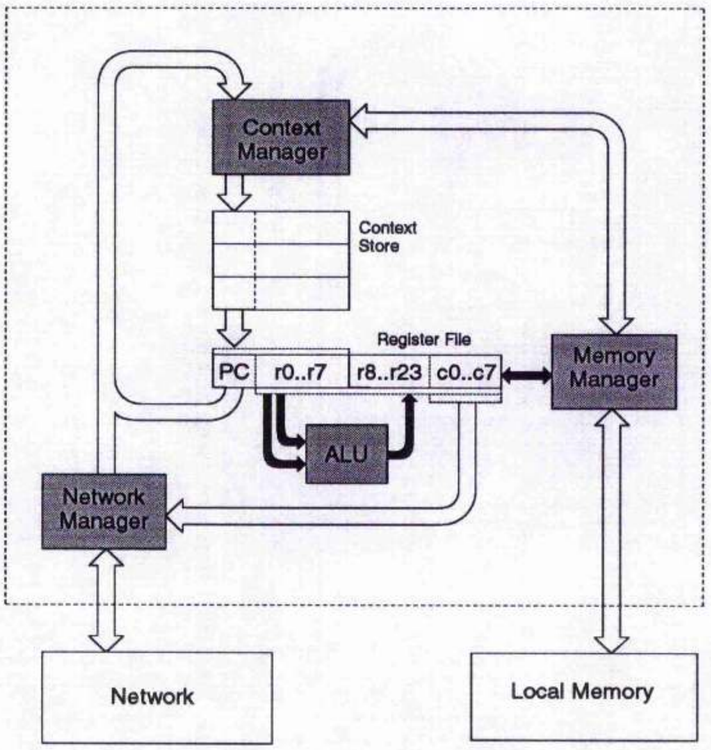

Figure 6.2: A STAR:DUST processing element (data paths only)

6.5 Discussion

Before demonstrating how one goes about writing parallel programs in terms of these primitive instructions, we will address a number of questions which are bound to arise.

6.5.1 Is STAR:DUST still a RISC architecture?

The two parallel control instructions might be seen to violate RISC principles by 'operating' on eight registers at a time. However, both instructions can be implemented using register windowing techniques comparable to those used in the SPARC for high-speed parameter passing to subroutines. The network manager can operate largely asynchronously from the RISC core.

6.5.2 Does STAR:DUST require infinite buffers?

As the context store is to reside in high-speed memory on-chip, its size is necessarily limited. In order to avoid the possibility of deadlock—or undue restrictions to the programming model that can be supported—STAR:DUST needs to provide an abstraction of an unlimited context store. The obvious solution is to overflow the context store into main memory when necessary. Again there is a correspondence to the SPARC register windowing mechanism which overflows the call stack (kept in register windows) into main memory. Since our context store is organized as a queue, however, the penalty for overflow in the form of thrashing behaviour is more severe. Therefore, the overflow mechanism is only to be seen as dealing with a 'worst case'. The programming model will have to avoid context buffer overflow to guarantee maximum performance. The load bounding scheme presented in the previous chapter provides the necessary guarantees for parallel functional programming.

6.5.3 What network bandwidth can we reasonably assume?

A crucial parameter in a VLSI implementation of STAR:DUST is the bandwidth of the network interface, the major factor limiting the minimum granularity we can support efficiently. For the following argument we will measure bandwidth relative to the processor cycle time. For comparison, the figure for the T800 transputer [Gr90] is 4 bit/cycle (4 channels of 1 bit/cycle each). Assuming similar specifications for a hardware implementation of STAR:DUST we would be able to sustain outgoing network traffic of the order of one message every 72 cycles (our messages contain 9 words of 32 bits each). Such a figure may not be completely sufficient to support the generous use of message passing in our simple compilation scheme of Chapter 4. Any effort to turn this scheme into a 'production quality' compiler would therefore have to include measures to eliminate unnecessary message passing.

6.6 Sample Programs

In the following we will present two STAR:DUST program fragments to illustrate our basic model of parallelism and, at the same time, to demonstrate that STAR:DUST is in no way specific to functional programming. The first fragment is a purely sequential subroutine which computes whether the point ($x+iy$) is in the Mandelbrot set and terminates after at most n iterations. It expects its parameters in the context registers (`r0-r7`) and is designed to return its result to a remote caller. The latter identifies itself by a 'continuation' consisting of a frame pointer `fp` and a code pointer `cp` specifying the return address.

```
define cp = r0, fp = r1, off = r2  ; return 'address'
define n = r3, x = r4, y = r5      ; parameters
define i = r6, t1 = r7, t2 = r8    ; temporary workspace

M       move    0 -> i             ; set up loop counter
N       fmult   x x -> t1          ; next iteration
        fmult   y y -> t2
        fmult   x y -> y
        fmult   y 2.0 -> y
        fsub    t1 t2 -> x
        fadd    t1 t2 -> t1
        fcmp    t1 4.0             ; termination condition
        bgt     T
        add     1 i -> i
        cmp     n i                ; max loop count reached?
        bgt     N
T       move    fp -> c0           ; prepare return
        move    off -> c1
        move    i -> c2            ; result value
        start   cp                 ; restart caller
        term
```

The main point of this example is to demonstrate the fact that sequential segments of STAR:DUST code can be executed with the efficiency of a purely sequential processor.

The second program fragment shows how to invoke two Mandelbrot computations in parallel. Let us first outline the basic mechanism for achieving cross-processor parallelism, illustrated in Figure 6.3. The active thread on the first PE starts two new threads (asynchronously) on two remote PEs. Each subthread computes its result and returns it by restarting a thread in the caller. The two 'return threads' in the calling program synchronize through a location in a 'frame' in local memory. The caller resumes its computation when both results are available.

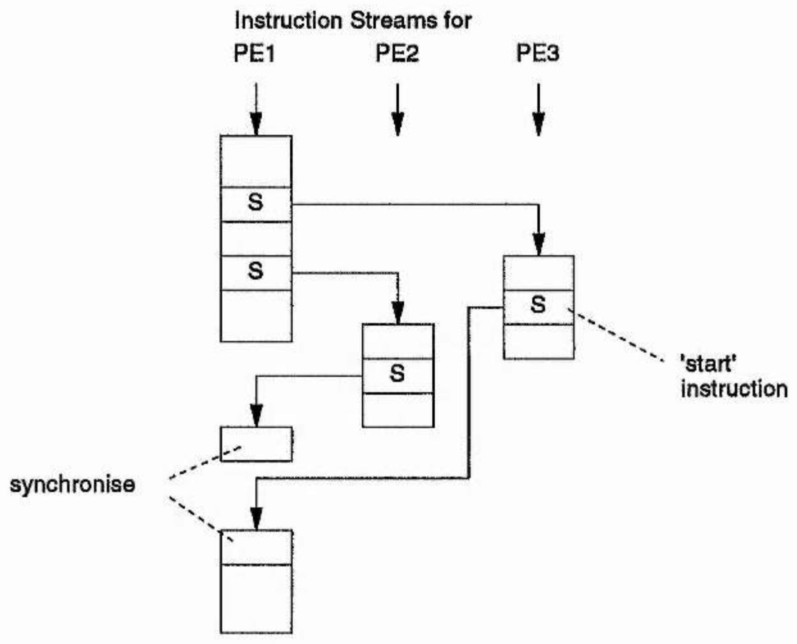

Figure 6.3: Basic scheme for cross-processor parallelism

Now for the program: We assume a frame pointer in register `r1` pointing to an activation frame in local memory. The two subthreads are synchronized through a location `sync` in this frame. 'Free variables' are to be treated as registers which have been preloaded with suitable values.

```
define fp = r0, off = r1

P       st #2 -> fp[sync]       ; prepare two threads
        move fp -> c1           ; pass frame pointer
        move off1 -> c2         ; pass offset for result
        move n -> c3            ; pass parameters
        move x1 -> c4
        move y1 -> c5
        move #J -> c0           ; pass return address
        sethi #1 -> c0          ; pick PE 1 for thread 1
        start M                 ; and start it
        sethi #2 -> c0          ; pick PE 2 for thread 2
        move off2 -> c2         ; pass different offset
        move x2 -> c4           ; pass parameters
        move y2 -> c5
        start M                 ; start second thread
        term                    ; and terminate
J       st val -> fp[off]       ; reentry point: store result
        ld fp[sync] -> r2       ; join the two threads
        sub r2 1 -> r2          ; by decrementing fp[sync]
        st r2 -> fp[sync]
        beq K
        term
K       ...                     ; process results
```

6.7 Origins of the STAR:DUST Architecture

Our design owes much to Nikhil's P-RISC [AN88], itself based on Iannucci's hybrid machine [Ian88]. These architectures show a successively stronger influence of modern von Neumann designs (see Figure 6.4). See Chapter 2 for a detailed discussion of this development.

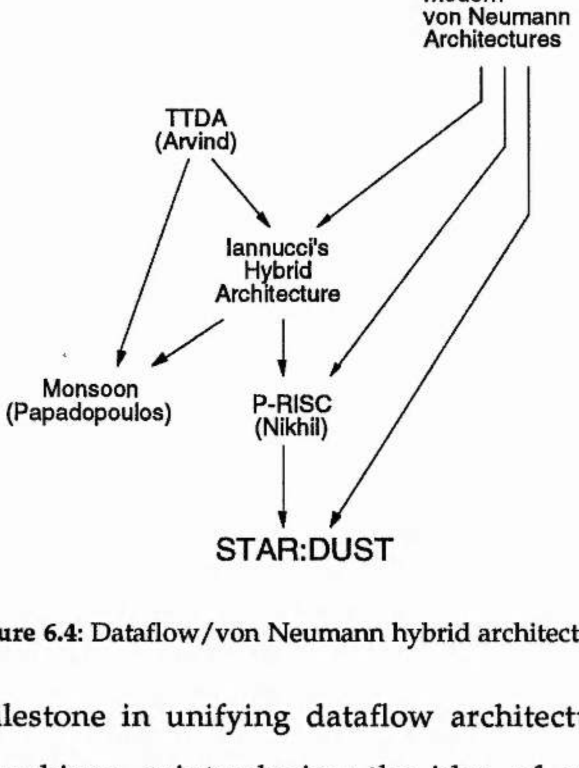
modern von Neumann Architectures
TTDA (Arvind)
Iannucci's Hybrid Architecture
Monsoon (Papadopoulos)
P-RISC (Nikhil)
STAR:DUST

Figure 6.4: Dataflow/von Neumann hybrid architectures

P-RISC was a milestone in unifying dataflow architectures with modern von Neumann machines, reintroducing the idea of efficient sequential threads into a dataflow context. By separating issues of synchronisation and instruction scheduling from those of local memory access, dataflow instructions were simplified to RISC level. STAR:DUST carries the move towards RISC architectures a step further by acknowledging a memory hierarchy, i.e., registers vs. local memory. P-RISC has a notion of 'frames' which combine the speed of one with the size of the other at an unclear cost. We subsumed all of P-RISC's communications instructions into the `start` instruction by widening tokens into 'contexts' (our context store corresponds to P-RISC's token queue), giving us more flexibility in the types of messages we can support. We further simplified the design by dispensing with independent memory elements. In the STAR:DUST architecture, all memory is local to some processing element, permitting us to perform memory management in software, again more in keeping with the RISC spirit. Finally, communication in STAR:DUST is register-to-register rather than memory to memory.

6.8 Summary

In this chapter we have outlined the architectural requirements resulting from the compilation scheme and the load bounding system of the previous chapters. We have presented the STAR:DUST architecture which extends the principles of RISC-based computing to communication and synchronisation in massively parallel architectures. An important design consideration for STAR:DUST was to enable it to make efficient use of sequential microprocessor technology. We have discussed some of the questions bound to arise regarding the extensions to the sequential core of STAR:DUST. We have given example programs illustrating the sequential and parallel aspects, respectively, of STAR:DUST programming. Finally we have compared our architecture to that of Nikhil's P–RISC from which it STAR:DUST was developed by following RISC-principles more faithfully, resulting in a much simplified design.

Chapter 7 — Distributed Implementation of a STAR:DUST Emulator

7.1 Introduction

In the previous chapter we have emphasised the fact that the STAR:DUST architecture is suitable for hardware implementation in VLSI. Obviously, such an implementation is beyond the scope of an individual PhD thesis. Instead we have chosen to prove the concept by means of a distributed implementation of a multiprocessor STAR:DUST emulator on a commercial multicomputer. The machine at our disposal was a Meiko Computing Surface with 24 transputers of type T800. This chapter begins by discussing some details of both the machine and the programming system used for the implementation. We give a structural overview of our distributed STAR:DUST emulator and discuss some issues of deadlock-free routing in a sparse network that we had to confront. Next we focus on the process structure of individual transputer nodes. Each transputer is responsible for emulating one STAR:DUST processing element as well as for through-routing incoming messages destined for other STAR:DUST nodes. We conclude with some experimental results obtained from running STAR:DUST implementations of two parallel functional programs, `nfib` and `quicksort`.

7.2 The Meiko Computing Surface and CSTools

The Meiko Computing Surface on which our emulator was implemented is a 24-node configuration of T800 transputers [MT90] on two VME boards hosted by a Sun 4/370, depicted schematically in Figure 7.1.

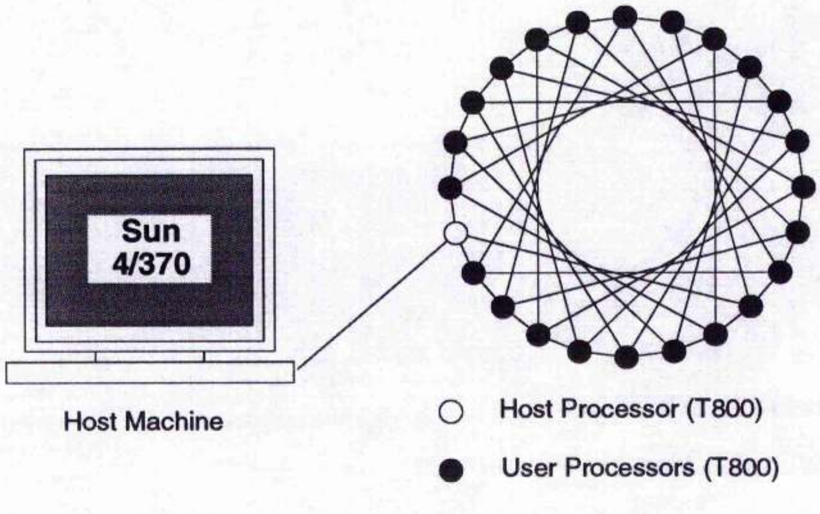

Figure 7.1: Schematic view of a 24-node Meiko Computing Surface

Each transputer has four links by which it can be connected to other transputers in the system. The Computing Surface provides a large degree of flexibility in the configuration of these connections by means of a programmable network. The network configuration depicted in Figure 7.1 is therefore only one of the many that are possible. Nonetheless, the fundamental limitation of transputers in the form of four hardware links per node must be respected. Complete connectivity via hardware links can only be achieved for system partitions made up of four transputers or fewer*.

For our programming environment we chose Meiko's CSTools system for its flexibility and its integration into a standard Unix environment. Under CSTools all program development is performed on the Sun host. CSTools programs run in two stages. In the first stage, the network description and process configuration is loaded onto the Computing Surface. In the second stage, the distributed processes are activated, communicating with each other and with the host. Program development reflects the two stages of program execution. The loader program is

* one link must be reserved for the connection to the host

compiled by a standard C compiler producing SPARC code, interfacing to the Computing Surface by means of calls to the CSBuild library. The code for the processes to be run on the Computing Surface itself is produced by Meiko's own C compiler. Processes communicate via calls to another Meiko supplied library which provides standard CSP style synchronous read- and write-operations to named channels. CSTools provides a convenient interface to most of the standard Unix system calls directly from transputers, including, in particular, the standard I/O operations.

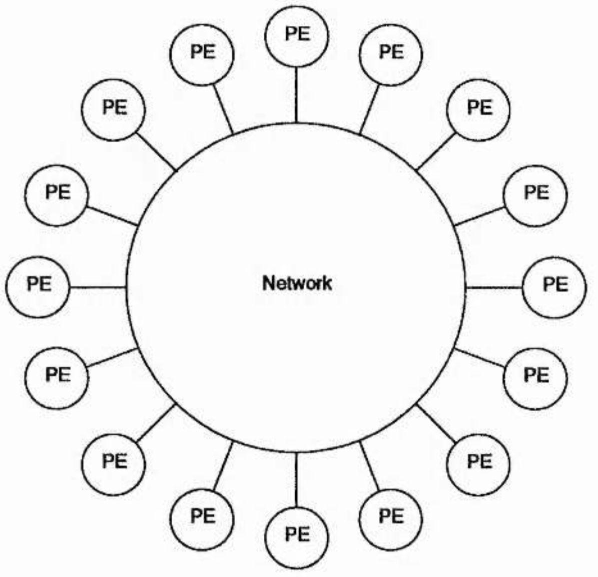

Figure 7.2: Emulator with generic network

7.3 System structure, routing and deadlock avoidance

The hardest problem we had to confront in the design of the distributed emulator is due to our need for a logically completely connected network as shown in Figure 7.2. Topology independence requires that any STAR:DUST node must be able to send messages to any other STAR:DUST node without regard to physical network topology. Given that the network of the Computing Surface is programmable, two design decisions have to be made in order to provide complete connectivity. Firstly, we need to decide upon a sparse physical network satisfying the constraints of the Computing Surface. Secondly, we need to decide upon a routing strategy within this network. The approach we chose in order to retain maximal flexibility was to *parameterise* the emulator by a routing table that simultaneously serves to define routing strategy and physical network topology.

Destination \ Source	0	1	2	3
0	0	0	0	2
1	1	1	3	1
2	2	0	2	2
3	2	3	3	3

Figure 7.3: Sample routing table

We show such a table in Figure 7.3, with entries to be interpreted as in the following example: to get from (source) node 0 to (destination) node 3, we go via the intermediate node 2. This obviously implies that there must be a physical link between nodes 0 and 2. Thus the routing table of Figure 7.3 defines an underlying physical network which is shown in Figure 7.4.

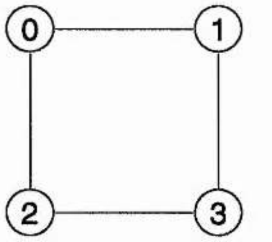

Figure 7.4: Corresponding underlying network

Valid routing tables are those where the degree of each node does not exceed four (due to the connectivity restrictions of individual transputers mentioned previously) and where all paths starting at node x with destination y eventually terminate at y. These properties, however, are not completely sufficient for guaranteeing successful routing. It is well-known that certain routing strategies can lead to deadlock for specific message patterns. See for example [NM93]. Efficient algorithms are also known for testing routing tables such as the above for the possibility of deadlock. For our implementation we restricted ourselves to hypercube networks and the well-known deadlock-free E-cube routing algorithm [SB77]. Hypercube networks were attractive to us also for their relatively short average path-lengths (logarithmic in the number of processors) and their inherent symmetry. Hypercubes up to dimension four, i.e., with up to 16 nodes, can be directly implemented on transputer systems. A four-dimensional hypercube topology is shown in Figure 7.5.

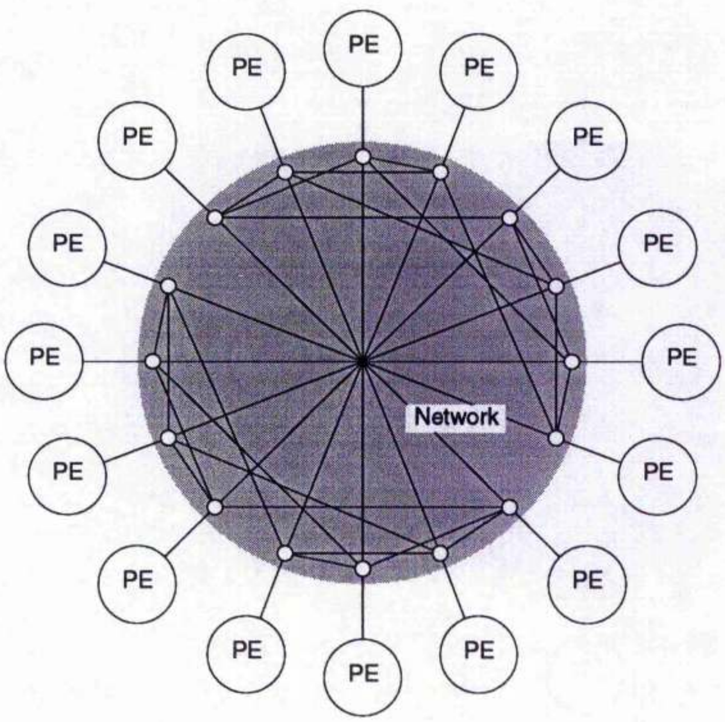

Figure 7.5: Emulator as a 4D hypercube

7.4 Node structure

From the above diagram we can now quickly derive the process structure of individual transputer nodes. Each transputer is responsible for one 'slice' of the network, as shown in Figure 7.6. Each such slice consists of a router process and a PE process which communicate via a transputer-internal link. The links to other router processes in the network represent the physical links of the transputer. The reader will quickly verify that for each node there are precisely four such links.

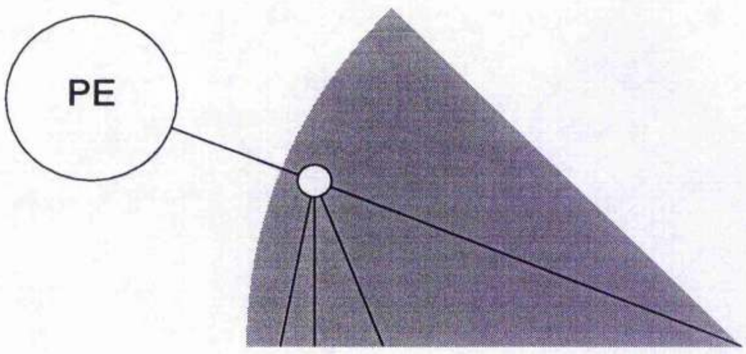

Figure 7.6: Process structure for a single transputer

The two processes in the above diagram can be further divided into sub-processes as shown in Figure 7.7. There we have split the bi-directional links of Figure 7.6 into two unidirectional channels each. The router process is implemented by the crossbar to the right. Each pair of input channels and output channels is served by associated receiver and sender processes and corresponds to one physical transputer link. The router is organised as a crossbar to avoid overlapping paths inside the router which might invalidate the deadlock freedom of the E-cube algorithm. The structure of our router is similar to that described by Burgess et al. in [BL93] to whom we are grateful for valuable suggestions.

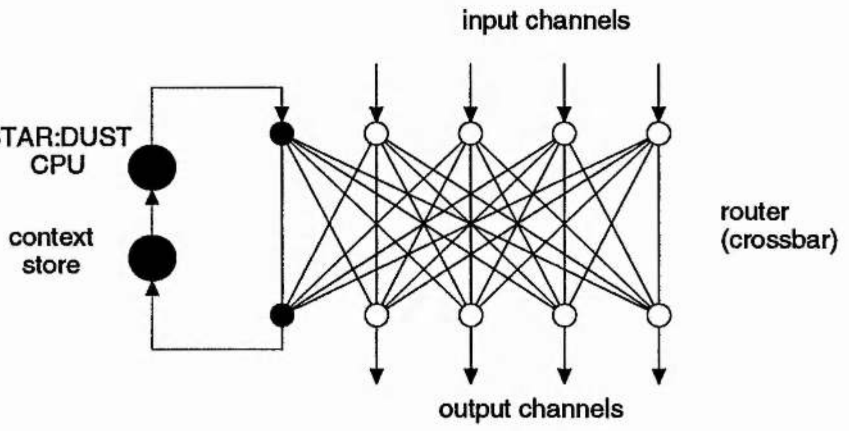

Figure 7.7: Detailed process structure for a single node

The pair of receiver and sender processes shown in black is provided for the link to the process which models the STAR:DUST processing element. This process in turn is split up into two subprocesses. One is responsible for the context store and is little more than a FIFO buffer, corresponding closely to the token queue of a dataflow machine. The other process models the activity of the STAR:DUST CPU, performing functions like switching to the next context, decoding instructions, reading and writing the local memory, changing register values and injecting messages into the network.

Note that the process structure on the nodes is independent of the structure of the network connecting them. This enables us to carry over the topology independence of the simulated STAR:DUST machine to the implementation of the emulator, evidenced by our ability to parametrise it with a network description.

7.5 Experimental results

We have performed several measurements on the emulator described above. Measurements were made running the emulator on a 16-processor STAR:DUST configuration. The measured load values are those reported by the global load computation scheme described in Section 5.6.3. Since we don't have a global clock, each processor keeps its own local time in the form of an instruction count. Messages carry the local time of the sender and advance the clock of the receiving PE, if necessary. Thus our measurements are slightly pessimistic as messages arriving out of order (with respect to local simulated time) can advance clocks further than necessary.

Our first test was the naive Fibonacci program which represents a kind of worst case for any load bounding scheme, as parallelism is growing extremely fast. The experiment was repeated with various target load values (Figure 7.8), i.e., target values for the average number of active threads per processor, as described in Section 5.6.3.

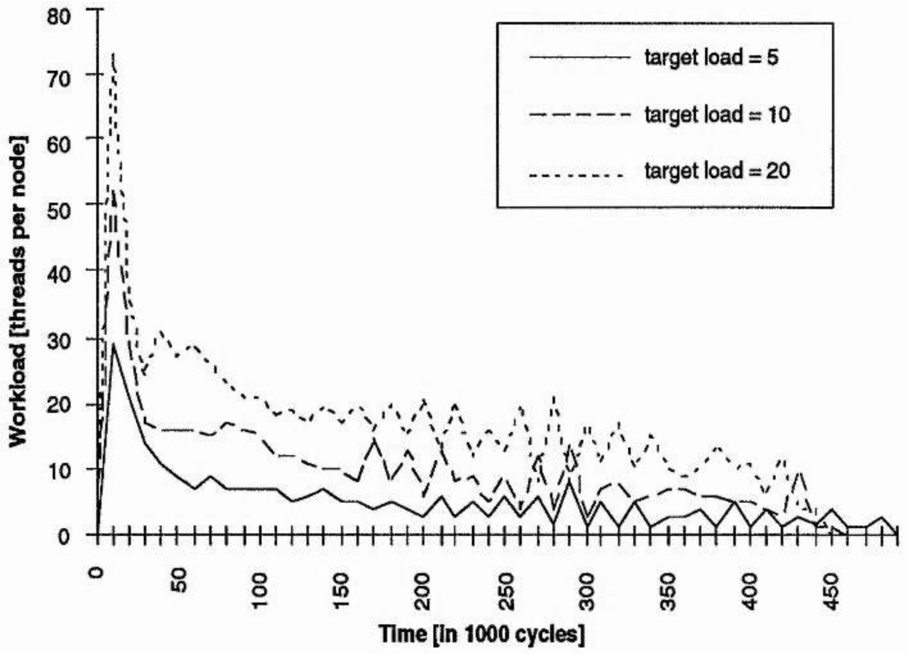

Fig. 7.8: Load bounding for parallel Fibonacci

In all cases we could observe an initial load spike, which is checked by the completion of the first load computation cycle, followed by a phase of more or less steady load reduction to the target value range. The computation then enters a phase where parallelism oscillates around the target value, dropping off gradually as opportunities for parallelism are steadily exhausted.

The `quicksort` experiment below shows the effects of the parallelism explosion in the absence of load bounding. Even though `quicksort` offers little opportunity for parallelism (see discussion below), we can observe two load spikes, one each at the start and at the end of the computation, as shown in Figure 7.9 (note the logarithmic scale).

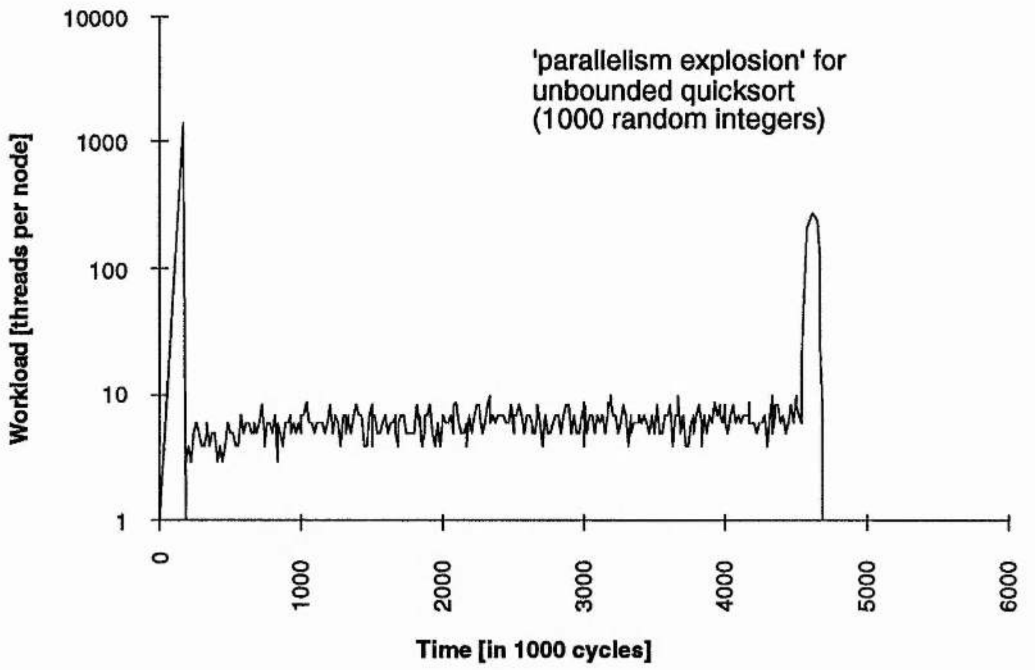

Figure 7.9: Unbounded parallel Quicksort

The initial load spike represents the divide-and-conquer style computation of a random 1000 element list. The final spike came initially as a surprise to us but is readily explained. While Quicksort is a typical divide-and-conquer algorithm, its parallelism is limited because of the bottleneck represented by the traversal of the original list. When this list is completely traversed, the bottleneck is replaced by a pair of bottlenecks, one each for the partitions of the original list. At this stage, the bottlenecks are successively broken up to give a final burst of divide-and-conquer parallelism. While this burst comes too late to have any significant effect on the runtime of parallel Quicksort, it may well result in crashing the machine at the very instant it is about to deliver its result. So load bounding can be important even for problems with low average parallelism. This example also clearly demonstrates the need for an adaptive solution: programs will often enter different phases during their execution, requiring different responses from the load bounding system.

7.6 Summary

In this chapter we have outlined our distributed emulation of a multiprocessor STAR:DUST system on a commercial multicomputer. We have reviewed those details of both the multicomputer and the accompanying programming environment that were relevant to our implementation. Due to restrictions in the connectivity of the underlying machine an important aspect of the resulting implementation was the provision of an abstraction for complete connectivity. The emulator we have implemented can be parameterised by both the underlying physical network and the routing algorithm within this network. The networks we have adopted for our experiments were hypercubes up to dimension four. We have shown the process structure both of the overall emulator system and of individual nodes. Finally we have presented and discussed the results of two experiments which show the effectiveness of the load bounding scheme of Chapter 5 and the consequences of unbounded parallelism, respectively.

Chapter 8 — Conclusion

8.1 What we have accomplished

We have attempted a compact treatment of the process of implementing parallel functional programming languages, ranging from a concise formal specification of parallel graph reduction to the presentation of a simple RISC-style multithreaded architecture and its simulation on a commercial multicomputer. Due to the breadth of the subject covered we have encountered many opportunies to probe more deeply and we will devote this chapter to pointing out some of the more obvious ones. The structure of this chapter resembles the structure of the thesis as a whole. For each of the main chapters we will summarise our main results before pointing out opportunities for improvement and further work.

8.2 Specifying parallel graph reduction in the π-calculus

We have given a new encoding of the λ-calculus in the π-calculus which models parallel graph reduction with shared reductions. Improving on two encodings by Robin Milner, ours preserves non-strictness by permitting a choice between lazy evaluation and eager parallel evaluation on a case-by-case basis for each static instance of function application.

To round off the work presented in Chapter 3 it would be highly desirable to provide a correctness proof for our encoding. Milner gives such proofs for each of his encodings in [Mi92]. In order to provide a proof of faithful simulation we need to define a suitable notion of equivalence between terms of the λ-calculus and the π-calculus terms produced by our scheme. Ideally we might expect for a term M of the λ-calculus with normal form M_0 the relation

$$[[M]]\,o \;\rightarrow_{\pi}^{*}\; [[M_0]]\,o$$

to hold for their π-calculus encodings. Unfortunately this simple relationship cannot be established, as reductions of encoded terms preserve sharing information that may have been lost in computing the normal form M_0 in the λ-calculus. For example, the π-calculus term obtained by reducing $[[(\lambda x.xx)(yy)]]\ o$ is different from $[[(yy)(yy)]]\ o$. Both of these terms will 'compute the same value' when embedded in the same context, differing only in their reduction behaviour. In particular, the second term will perform the application (yy) twice, after suitable instantiation of y, while the first only needs to compute (yy) once. Similar problems arise in Milner's encoding of the call-by-value λ-calculus and Milner's proof for his encoding may provide useful intuitions for a proof of our scheme.

8.3 Compilation scheme

Using our π-calculus model of parallel graph reduction as a guide, we have given a more easily applicable set of compilation rules for a simplified functional language. The target of the compilation scheme was multi-threaded parallel code which required no special mechanisms for communication and synchronisation other than thread creation and thread termination. As in the π-calculus encoding, we could decide for static instances of certain programming language constructs whether to evaluate them eagerly or lazily, thus combining the advantages of lazy evaluation with those of safe parallelism.

The compilation scheme presented in Chapter 4, while being concise, is also relatively simple-minded. In order to obtain an efficient compiler from this scheme, much work needs to be invested into optimisations. One important goal, certainly, is to achieve longer threads. Given that we do not attempt to exploit parallelism within function bodies, it is sometimes possible to combine several threads into one, producing longer threads with less communication overhead. A useful starting point for such analysis is work by Ken Traub on 'compilation by partitioning' [Tr91]. Another obvious source of improvement is direct support for higher order functions. Further optimisations can be considered to remove some of the 'boxing', i.e., the construction of graph nodes, which is being performed for every value in the program. See, for example, work by Peyton Jones and Launchbury [PL91]. Note, however, that strictness information alone is not sufficient to avoid 'boxing'. Consider for example the simple expression

```
nfib (n-1) + nfib (n-2)
```

Even though addition is obviously strict, we will generally want to allocate graph nodes for the two subterms for purposes of synchronisation. On the other hand, we can benefit from passing the argument to `nfib` as an unboxed integer without giving up any useful parallelism.

A final suggestion for further work, motivated by the work presented in Chapter 4, is a formal specification of an abstract model for the kind of multithreaded code produced by our compilation scheme. At the time of writing we have a fairly clear idea of how to specify such a model very succinctly and we hope to present it soon.

8.4 Load bounding and runtime system

The simplicity of the compilation scheme of Chapter 4 bore fruit in our work on automatic load bounding. We have presented a simple and efficient adaptive load bounding method which was integrated into the compilation scheme. We have given an informal proof for the effectiveness of our method, based on the structure of the compilation scheme. The method was based on considering an estimate of the current global workload before exploiting any new parallelism.

The main weakness of our approach was the fact that load bounding could not be guaranteed in the presence of sharing, as the reduction of a shared value to weak head-normal form may require a large number of waiting computations to be restarted. In Section 5.7 we suggested two partial solutions that warrant further investigation: reduction of shared expressions before sharing is established and inclusion of waiting computations in the workload computation.

Crucially important for the success of our approach is also an efficient solution to the problem of distributed garbage collection, not covered in this thesis. An often favoured technique for distributed architectures, due to its ability to proceed concurrently with the computation, is weighted reference counting, first suggested by Weng in [We79]. Simple reference counting suffers, however, from not being able to reclaim cyclic structures. While extensions have been proposed to address this issue [Hu84] they have not been, to our knowledge, adopted in practice due to their computational cost. A common approach, then, is to use a two-level garbage collector, a real-time reference counting scheme backed up by a mark-scan collector. See, for example, Watson and Watson [WW87b]. As a backup collector will be required in any case, we have turned our thoughts to a distributed copying collector first. Sequential copying collectors work by traversing all active data starting from the evaluation stack, copying any structures encountered from a 'from-space' to a 'to-space'. After traversal is complete, the whole 'from-space' can be reclaimed. In the next round of garbage collection, 'from-space' and 'to-space' switch their roles. It is not clear at first sight how a similar technique could be applied in a distributed environment, as the constantly changing pattern of messages in transit would seem to rule out any concept of a 'root' of reachable structures. We envisage a solution roughly along the following lines: global garbage collection is initiated by any PE which runs out of local space. The 'root' of reachable structures is represented precisely by the messages in transit. The compilation scheme could prefix every message handler by a test for garbage collection. A handler discovering that garbage collection is in progress would use its knowledge about the structure of the target context to initiate copying of active data referred to from within this context. The message handler would then be suspended until global garbage collection is complete. Copying of sub-structures can be performed in parallel or sequentially, again using the load bounding scheme of Chapter 5. Further work is clearly required.

8.5 The STAR:DUST machine

In Chapter 6 we have presented a simple concrete architecture, STAR:DUST, which provides efficient support for the compilation scheme of the previous two chapters. Rather than being designed as a dedicated architecture for functional programming, STAR:DUST follows RISC design principles. Relative to a conventional RISC processor, our architecture adds multi-threading capabilities in the form of two instructions, one each for thread creation and termination, respectively.

We have claimed that STAR:DUST can be implemented efficiently using modern VLSI technology and we have put forward strong arguments in support of our case. A convincing proof in the form of an implementation in silicon could in itself form the basis of a PhD level research project. We intend to conduct further work in this area in the form of a more detailed simulation which can observe the performance of a STAR:DUST system taking into account system parameters such as the following

- size of the context store
- cost of context store overflow
- cost of a context switch (`term` instruction)
- (local) caching behaviour of multithreaded programs
- message latency

— communications bandwith

Obtaining meaningful results for the effects of such design parameters on system performance also requires the implementation of substantial test programs.

8.6 Implementation and experimentation

We have described the implementation of a distributed emulator of a multiprocessor STAR:DUST system on a Meiko Computing Surface. We have focussed in particular on the issue of deadlock-free routing. While this work was crucial for us to obtain a truly distributed implementation on which to perform experiments, network-related issues are not at the heart of our work and had to be confronted only because of the limitations of the system available to us. The routing overhead imposed on us by the Meiko Computing Surface practically rules out competitive speedups for our programs on large networks.

An interesting line of further research would be to implement STAR:DUST more directly on a system which performs message routing in hardware, such as a CM-5, a J-Machine, or a system based on a more modern version of the transputer. Such an implementation could be prototyped on a small, completely connected T800 system, i.e., one of four nodes or fewer.

8.7 Epilogue

The success of parallel functional programming as a useful tool to the 'working programmer' is dependent on concurrent progress in diverse areas such as computer architecture, runtime systems, compiler technology, language design, and software engineering, with sound theoretical underpinnings required to relate them all. We have attempted in this thesis to preserve a holistic view of the field, relating many of the subject areas relevant to it. If we had to favour, on occasion, completeness of the whole over completeness of the parts, we hope the reader will agree with the wisdom of our choice.

Bibliography

For brevity we will make use of the following acronyms:

ASPLOS:	International Conference on Architectural Support for Programming Languages and Operating Systems
CACM:	Communications of the ACM
FPCA:	International Conference on Functional Programming and Computer Architecture
ISCA:	Annual International Symposium on Computer Architecture
LFP:	ACM Conference on Lisp and Functional Programming
LNCS:	Lecture Notes in Computer Science, Springer Verlag
PARLE:	Parallel Architectures and Languages, Europe
PIFL:	Workshop on the Parallel Implementation of Functional Languages
POPL:	Principles of Programming Languages

[AC86] Arvind, David E. Culler — *Dataflow Architectures* — MIT/LCS/TM-294, 1986

[AI87] Arvind, Iannucci, R.A. — *Two Fundamental Issues in Multiprocessing* — Proc. DFVLR Conf. in Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, 1987

[AJ89] Lennart Augustsson, Thomas Johnsson — *Parallel Graph Reduction with the <v, G>-Machine* — Proc. FPCA 1989, Imperial College, London, pp. 202-213

[AK80] Arvind, Vinod Kathail, Keshav K. Pingali — *A Dataflow Architecture with Tagged Tokens* — MIT LCS TM-174, 1980

[AN88] Arvind, Rishiyur S. Nikhil — *Can Dataflow Subsume von Neumann Computing?* — MIT CSG Memo 292, November 15 1988

[Ba78] John W. Backus — *Can Programming Be Liberated from the von Neumann Style?* — CACM, August 1978

[Bi89] Richard S. Bird — *Algebraic Identities for Program Calculation* — Computer Journal, Special Issue on Lazy Functional Programming, 32(2), pp. 122-126

[BL93] Peter Burgess, Mike Livesey, Colin Allison — *An Execution Harness for Transputer-Based Embedded Systems* — Proc. WoTUG-16, 28-31 March 1993, Sheffield (to appear)

[BR91] S. D. Brookes, A. W. Roscoe — *Deadlock Analysis in Networks of Communicating Processes* — Distributed Computing, vol. 4, 1991, pp. 209-230

[BS81] F. Warren Burton, M. Ronan Sleep — *Executing Functional Programs on a Virtual Tree of Processors* — Proc. FPCA 1981, Portsmouth, New Hampshire, pp. 187-194

[Bu75] W. H. Burge — *Recursive Programming Techniques* — Addison-Wesley, 1975

[CA88] David E. Culler, Arvind — *Resource Requirements of Dataflow Programs* — Proc. 15th ISCA, Honolulu, Hawaii, 1988, p. 141

[Co90] Murray Cole — *Towards Fully Local Multicomputer Implementations of Functional Programs* — Technical Report CSC 90/R7, University of Glasgow, January 1990

[CP88] Chris Clack, Simon L. Peyton Jones, Jon Salkild — *Efficient Parallel Graph Reduction on GRIP* — University College London, Research Note RN/88/29

[Da82] John Darlington — *Program Transformation* — in "Functional Programming and its Applications", eds. J. Darlington, P. Henderson, D. A. Turner, Cambridge University Press, 1982

[Da91] John Darlington et al. — *Structured Parallel Functional Programming* — Proc. PIFL 1991, Southampton, appeared as technical report, CSTR 91-07, University of Southampton, 1991, pp. 31-52

[Da92] Antony J. T. Davie — *An Introduction to Functional Programming Systems Using Haskell* — Cambridge Texts in Computer Science, Cambridge University Press 1992

[DA92] Keith Diefendorff, Michael Allen — *Organization of the Motorola 88110 Superscalar RISC Microprocessor* — IEEE Micro, April 1992, pp. 40-63

[DC89] William J. Dally, Paul Carrick, et al. — *The J-Machine: A Fine-Grain Concurrent Computer* — Information Processing 89, G. X. Ritter (ed), Elsevier Science Publishers B.V. (North-Holland), 1989

[De73] Jack B. Dennis — *First Version of a Data Flow Procedure Language* — MIT CSG Memo 93, 1973

[De75] Jack B. Dennis, D. P. Misunas — *A Preliminary Architecture for a Basic Dataflow Processor* — Proc. 2nd ISCA, 1975, p. 126

[DR81] John Darlington, Mike Reeve — *ALICE—A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages* — Proc. FPCA 1981, pp. 65-75

[DS87] William J. Dally, Charles L. Seitz — *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks* — IEEE Trans. on Computers, Vol. C-36, No. 5, May 1987, pp. 547-553

[EC92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser — *Active Messages: a Mechanism for Integrated Communication and Computation* — Proc. 19th ISCA, 1992, pp. 256-266

[Fa76] Howard Falk — *Reaching for a Gigaflop* — IEEE Spectrum, October 1976, pp. 65-69

[Ga88] R. B. Garner, et al. — *The Scalable Processor Architecture (SPARC)* — Proc. Compcon 88, IEEE CS Press, 1988, p. 278

[GH87] Benjamin Goldberg, Paul Hudak — *Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor* — Proc. of the Workshop on Graph Reduction, Santa Fe, 1987, LNCS 279, pp. 94-113

[GH90] Hugh Glaser, Pieter Hartel, John Wild — *A Pragmatic Approach to the Analysis and Compilation of Lazy Functional Languages* — Proc. PIFL 1990, pp. 203-221

[GK85] John R. Gurd, C. C. Kirkham, Ian Watson — *The Manchester Prototype Dataflow Computer* — CACM, January 1985, pp. 34-52

[Gr90] Graham, I., King, T. — *The Transputer Handbook* — Prentice Hall, 1990

[GW78] John R. Gurd, Ian Watson, John Glauert — *A Multilayered Data Flow Computer Architecture* — Internal Report, Dept. of Computer Science, University of Manchester, 1978.

[HB84] Kai Hwang, Fayé A. Briggs — *Computer Architecture and Parallel Processing* — McGraw-Hill Computer Science Series, 1984

[HJ92] Dana S. Henry, Christopher F. Joerg — *A Tightly-Coupled Processor-Network Interface* — Proc. ASPLOS-V, Boston, Massachusetts, pp. 111-122, 1992.

[HP90] Kevin Hammond, Simon L. Peyton Jones — *Some Early Experiments on the GRIP Parallel Reducer* — Proc. PIFL 1990, pp. 51-71

[Hu84] R. J. M. Hughes — *Reference Counting with Circular Structures in Virtual Memory Applicative Systems* — Programming Research Group, Oxford University, 1984

[Hu89] R. J. M. Hughes — *Why Functional Programming Matters* — The Computer Journal, Vol. 32, No. 2, 1989, p. 98

[Ia88] Robert A. Iannucci — *Toward a Dataflow/von Neumann Hybrid Architecture* — Proc. 15th ISCA, Honolulu, Hawaii, 1988, p. 131

[Jo84] Thomas Johnsson — *Efficient Compilation of Lazy Evaluation* — 1984 ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices, June 1984, pp. 58-69

[Ke89] Paul Kelly — *Functional Programming for Loosely-coupled Multiprocessors* — Research Monographs in Parallel and Distributed Computing, Pitman/The MIT Press, 1989

[Le92] Charles E. Leiserson, et al. — *The Network Architecture of the Connection Machine CM-5* — Proc. 1992 ACM Symposium on Parallel Algorithms and Architectures, pp. 272-85

[Ma79] Gyulà Mago — *A Network of Microprocessors to Execute Reduction Languages* — Int. Journal of Computer and Information Sciences, 1979, part 1 in (8) 5, pp. 349-385, part 2 in (8) 6, pp. 435-471

[Mi80] Robin Milner — *A Calculus of Communicating Systems* — LCNS 92, Springer Verlag, New York, 1980.

[Mi91] Robin Milner — *The Polyadic π–Calculus: A Tutorial* — University of Edinburgh Technical Report, ECS-LFCS-91-180, October 1991

[Mi92] Robin Milner — *Functions as Processes* — Mathematical Structures of Computer Science, vol. 2, pp. 119-141, 1992

[Mi93] Robin Milner — *Elements of Interaction* — CACM, January 1993, pp. 78-89

[MK90] Eric Mohr, David A. Kranz, Robert H. Halstead — *Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs* — Proc. LFP 1990, Nice, France, pp. 185-197

[MT90] D. A. P. Mitchell, J. A. Thompson, G. A. Manson, G. R. Brookes — *Inside the Transputer* — Blackwell Scientific Publications, Computer Science Texts, 1990

[MW92] Sunil Mirapuri, Michael Woodacre, Nader Vasseghi — *The Mips R4000 Processor* — IEEE Micro, April 1992, pp. 10-22

[Ni89] Rishiyur S. Nikhil — *The Parallel Programming Language Id and its Compilation for Parallel Machines* — Proc. Workshop on Massive Parallelism: Hardware, Programming and Applications, Amalfi, Italy, October 1989, Academic Press

[NM93] Lionel M. Ni, Philip K. McKinley — *A Survey of Wormhole Routing Techniques in Direct Networks* — IEEE Computer, February 1993, pp. 62-76

[NP92] Rishiyur S. Nikhil, Gregory M. Papadopoulos, Arvind — **T: A Multithreaded Massively Parallel Architecture* — 19th ISCA, 1992, pp. 156-167

[Os91] Gerald Ostheimer — *Parallel Functional Computation on STAR:DUST* — Proc. PIFL 1991, Southampton, appeared as technical report CSTR 91-07, University of Southampton, 1991, pp. 393-408.

[Pa88] Gregory M. Papadopoulos — *Implementation of a General Purpose Dataflow Multiprocessor* — MIT LCS TR-432, August 1988 (PhD Thesis)

[PC87] Simon L. Peyton Jones, Chris Clack et al. — *GRIP—A High-Performance Architecture for Parallel Graph Reduction* — Proc. FPCA 1987, LCNS 274, p. 98

[PC90] Gregory M. Papadopoulos, David E. Culler — *Monsoon: An Explicit Token-Store Architecture* — Proc. 17th ISCA, 1990, pp. 82-91

[Pe87] Simon L. Peyton Jones — *The Implementation of Functional Programming Languages* — Prentice-Hall International Series in Computer Science, Prentice-Hall 1987

[Pe89] Simon L. Peyton Jones — *Parallel Implementations of Functional Programming Languages* — The Computer Journal, Vol. 32, No. 2, 1989, p. 175

[PL91] Simon L. Peyton Jones, John Launchbury — *Unboxed Values as First Class Citizens in a Non-Strict Functional Language* — Proc. FPCA '91, Cambridge, Massachusetts, 1991, pp. 636-666

[PS81] David A. Patterson, Carlo H. Sequin — *RISC I: A Reduced Instruction Set Computer* — Proc. 8th ISCA, pp. 443-457, 1981

[RS87] Carlos A. Ruggiero, John Sargeant — *Control of Parallelism in the Manchester Dataflow Machine* — LCNS 274, p. 1, Proc. FPCA 1987

[SB77] Herbert Sullivan, T. R. Bashkow — *A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I* — Proc. 4th ISCA, 1977, pp. 105-117

[Sm81] B. J. Smith — *Architecture and Applications of the HEP Multiprocessor Computer System* — Real Time Signal Processing IV, vol. 298, August 1981

[Tr86] Kenneth R. Traub — *A Compiler for the MIT Tagged-Token Dataflow Architecture* — MIT LCS TR-370, 1986 (Master's Thesis)

[Tr91] Kenneth R. Traub — *Implementation of Non-Strict Functional Programming Languages* — Research Monographs in Parallel and Distributed Computing, Pitman, London, 1991

[Tu79] David A. Turner — *A New Implementation Technique for Applicative Languages* — Software—Practice and Experience, 1979, p. 31

[Va90] Leslie G. Valiant — *A Bridging Model for Parallel Computation* — Communications of the ACM (CACM), Vol. 33, Nr. 8, August 1990, p. 103

[Ve84] Steven Vegdahl — *A Survey of Proposed Architectures for the Execution of Functional Languages* — IEEE Trans. on Computers, Dec. 1984, pp. 1050-1071

[Wa71] C. P. Wadsworth — *Semantics and Pragmatics of the Lambda Calculus* — Oxford University D.Phil. Thesis, 1971

[We79] K-S. Weng — *An Abstract Implementation for a Generalized Dataflow Language* — MIT Laboratory for Computer Science, MIT/LCS/TR-228

[Wh85] Colin Whitby-Strevens — *The Transputer* — Proc. 12th ISCA, pp. 292-300, June 1985

[WW87a] Paul Watson, Ian Watson — *Evaluating Functional Programs on the FLAGSHIP Machine* — Proc. FPCA 1987, LCNS 274, pp. 80-97

[WW87b] Paul Watson, Ian Watson — An Efficient Garbage Collection Scheme for Parallel Computer Architectures — Proc. PARLE '87, LCNS 259, pp. 432-443

[WW88] Ian Watson, Viv Woods, Paul Watson, Richard Banach, Mark Greenberg, John Sargeant — *Flagship: A Parallel Architecture for Declarative Programming* — Proc. 15th ISCA, Honolulu, Hawaii, 1988, p. 124