PARALLEL EXECUTION OF FUNCTIONAL PROGRAMS

Jeffrey Nathan Kellman

January 1982
Report No. CSD-830114

UNIVERSITY OF CALIFORNIA

Los Angeles

Parallel Execution of Functional Programs

A thesis submitted in partial satisfaction of the

requirement for the degree Master of Science

in Computer Science

by

Jeffrey Nathan Kellman

1982

Table of Contents

Table of Contents

Table of Contents

List of Figures

List of Figures

ACKNOWLEDGMENTS

I would like to thank my M.S. thesis committee for their patience and encouragement during the time this report came into being. I would like to especially thank my thesis advisor, Miloš Ercegovac, for many insightful discussions about multiprocessing and functional programming systems. His enthusiasm for the investigation of functional programming systems has inspired the formation of a highly motivated group at U.C.L.A. to investigate such systems. I am very grateful for the opportunities I have had to participate in this group's debates and discussions. Last, but not least, I would like to express much gratitude to John Backus, who continues to put so eloquently and forcefully into words important ideas whose time has come.

ABSTRACT OF THE THESIS

Parallel Execution of Functional Programs

by

Jeffrey Nathan Kellman

Master of Science in Computer Science

University of California, Los Angeles, 1982

Professor Milos D. Ercegovac, Chair

This report proposes a multiprocessor system design to directly execute functional programs in massively parallel fashion. A maximally parallel reduction step in this system takes only $O(\log^s N)$ time on $O(N)$ processors (where $s \leq 4$). Furthermore, this system easily implements powerful operators as functional primitives, including such meta operators (functional forms) as apply-to-all and associative insert, and regular operators like matrix transpose and parallel sort.

1. Introduction

To achieve a high computational throughput in a wide range of computations, a computer system should execute with maximum parallelism wherever possible. In this pursuit we also want to enlist the aid of modern VLSI technology and use large numbers of microprocessors to achieve their maximum effect in any given computation. But how should we interconnect and organize all these processing elements? And just as important, how could we instruct such an assemblage to do our bidding in a reasonable and effective manner and still prevent the programming language from hindering parallelism?

This report investigates one approach which attempts to accomplish both of these objectives simultaneously. We shall see how functional programming languages[1] fulfill important necessary conditions for expressing maximum computational parallelism and how these languages specify parallel computations in a natural and straightforward manner. We shall also examine how the proper interconnection of many simple processing elements makes it possible to directly execute functional programs in a manner which can take full advantage of their expressed parallelism.

1.1 Background and Motivations

As the speeds of computer devices reach a limit, current large computer systems fail to keep up with the demands of many contemporary problems. Government, industry, and academic applications all continue to demand ever more massive computations at ever higher speeds. Computer designers, faced with this challenge, increasingly look to parallel processing as the proper approach for significantly increasing computing capacity. To perform ever more instructions in less time, a computer system simply must execute more than one instruction at a time whenever it can.

And yet, computer systems achieve parallel computation only with great difficulty, since computer parallelism can only happen if the program algorithms permit it, if the system hardware supports it, and if either the programmer or the system can find it quickly and cheaply enough for their efforts to pay off. Furthermore, to be useful in more than one application, any system organization for parallel computation needs to have upward scalability so that increasing the system size should also increase its capabilities. The degree to which a system accomplishes all four of these important prerequisites determines the degree of general parallelism which that system will achieve. To see where the difficulties lie, let us examine each of these prerequisites in more detail.

1.1.1 Computational Parallelism

Although parallel execution generally performs computations faster than sequential execution, not every computation has a parallel execution. For example, calculating x^{2^n} requires at least n steps (of repeated squaring) on any machine which supports multiplication but does not otherwise support exponentiation as a primitive operation[2]. Unless we use some parallel scheme to speed up multiplication, we simply cannot hope to compute x^{2^n} any faster through parallelism. Likewise, in any other such computation with very little inherent parallelism, parallel processing can completely fail to help.

On the other hand, a number of important applications show a great deal of inherent concurrency at various points of their computations. They include:

Image processing for:
- Scene interpretation (e.g., robot vision)
- Scene display (e.g., high quality graphics)

Control systems which must:
- Evaluate inputs from many sensors (e.g., industrial process monitoring)
- Send commands to many actuators (e.g., in a guidance system)

Simulation to solve complex problems
(e.g., weather forecasting)

Synthesizing complex signals
(e.g., high quality speech and music)

Notice how any of these applications would greatly benefit from using their massive inherent parallelism to achieve huge computational throughputs. For these cases and many others like them, parallelism is not only possible, it is desirable.

Fig. 1-1 shows a value-oriented view of the execution of such massively parallel computations.

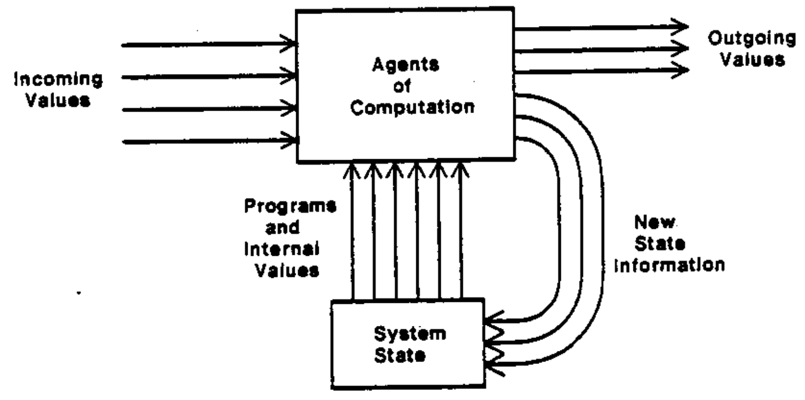


Fig. 1-1: Executing massively parallel computations

Possibly large numbers of incoming values enter the computation simultaneously to interact with each other and with values and representations (programs) already resident in some history-sensitive system state. This interaction produces possibly large numbers of new values, some of which might update the system state, and others which may leave the computation as outgoing results. If we focus just on the agents of computation, as in Fig. 1-2, we see that as a group, they could operate in a purely functional manner. Upon receiving their inputs in parallel (these inputs including the incoming values, the program representations, and the internal values), they might produce all their outputs (including outgoing values and possibly the new system state) in parallel, solely as a function of their inputs. Notice how this model factors out the history-sensitive system state from the agents of computation themselves (although actual implementations need not necessarily keep this system state physically separate).

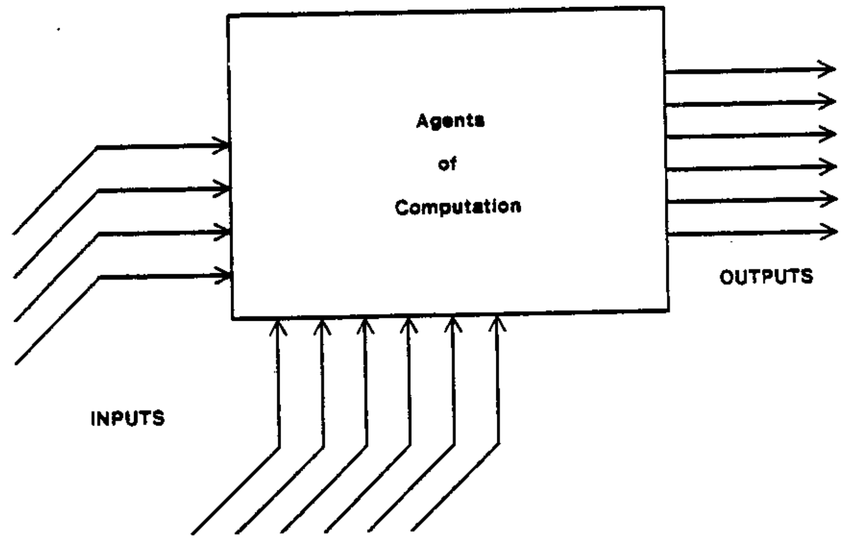


Fig. 1-2: The functional role of the agents of computation

For the rest of this report, these agents of computation will comprise the "system" we want to implement. Implementing the other parts of the system we leave to the application designer.

1.1.2 Programming Language Parallelism

Given a computation containing lots of parallelism, we still need to represent its program to the computer system in such a manner that the programmer or the system can easily find this parallelism. Furthermore, the time and cost of finding parallelism should not outweigh the advantages gained by using it. Failure to achieve this would restrict all future applications of parallelism to a small number of very special-purpose systems, --a not very exciting prospect.

Nonetheless, the current widely-used programming languages make the job of finding parallelism difficult, if not impossible. By hindering and confusing the detection of execution concurrency, usable parallelism might go undetected while at the same time we might falsely conclude parallelism for portions of computations where none actually exists. In this sense, the problems of programming parallelism relate to the more general problem of producing and maintaining correct and reliable software. Programming concurrent computations often leads to notoriously difficult-to-understand software which nobody wants to trust or modify. More and more investigators, however, suggest that functional or applicative programming languages can overcome these and other problems of the "software crisis".[3, 4, 5] These languages have mathematically tractable foundations[1] which make their programs easier to analyze. Furthermore, they express all their computations in expressions, and as Hehner[6] points out, we can always execute the subcomputations of pure expressions in concurrent fashion. In this report we discuss in more detail how and why the functional programming approach is well-suited for expressing parallel computation.

1.1.3 System Hardware Parallelism

Directly supporting parallel computation in hardware implies the physical distribution of subcomputations among many processing elements. This strategy gets considerable encouragement from modern LSI and VLSI technology which can produce small, cheap microprocessors in large numbers. Intuition strongly supports the interconnection of many small (and perhaps even slow) microprocessors to obtain large fast systems for inherently parallel computations. If all these processors were identical instances of some standard universal building block (for example, a single chip microcomputer) then the diminished costs of mass production would make feasible systems with thousands of processors. Some current proposals already suggest 10^5 processors in a single system[7], and by putting many processors into each chip we may someday have far larger systems fitting into surprisingly reasonable amounts of space.

Naively, we might easily assume that interconnecting ever larger numbers of processing elements would always give us systems which could perform more parallel computations in less time. Yet, except for some extremely special-purpose applications (algorithms for mesh computations, for example), this has not often happened in practice. In fact, the opposite frequently happens as centralized shared busses and costly crossbars make the larger numbers of processors hinder each other more than they help each other. Obviously, the interconnection and organization of many processors impacts the overall system just as significantly as their actual number.

1.1.4 Upward Scalable Parallelism

When we apply the solution of one problem to solving an otherwise identical problem of larger size, we also need to worry about the solution's *upward scalability*, –that is, whether the solution solves the larger problem with as much effective power as it solved the smaller one. The upward scalability of a parallel processing system tells us how much increasing the size of the parallel system increases its computation rate per unit time. (Of course, we also assume that the amount of inherent computational parallelism increases to give the larger system more parallelism to use!) If we consider a system which executes its computations with maximum parallelism, then the maximum upward scalability one can conceivably have is proportional scalability, i.e., doubling the size of the system also doubles its throughput (assuming that the available parallelism in the computation also doubles). As we shall see throughout this report, many proposed systems for general-purpose parallel processing don't even come close to proportional scalability.

We concern ourselves with the upward scalability of parallel processing systems for several reasons. The most straightforward reason is expansibility. If a given system scales upward in a reasonable fashion, then we could feasibly expect to construct higher performing versions of this system. Upward scalability is also important for system generality. A parallel processing system which scales upward affordably would be, by implication, free of any organizational, technology-dependent, or application-dependent bottlenecks or any other "creeping hindrances" to parallelism which, although insignificant in small systems, might show up noticeably in large systems. For example, consider an organization, such as a binary N-cube[8], where the number of communication links per processor increase by one for every doubling of the number of processors in the system. For such a system the hardware costs would become quite unmanageable as soon as the processors had more than ten links apiece. But if we instead had a fixed number of links per processor, as in tree structure, then the number of links in the system would remain proportional to the number of processors (although communication delays between certain processors in the system might get worse).

We do not worry about *downward* scalability in this report because a special-purpose system well-tailored to its task will always handle a small scale parallel task far better than any alternative, and would also cost less. Hence, we find it difficult to justify constructing any general-purpose parallel system for only small scale uses. Large scale applications, on the other hand, require large systems anyway, whether special-purpose or general-purpose. For these, we certainly want to consider the asymptotic performances and hardware cost behaviors of competitive general-purpose systems.

1.2 Goals

In this report we investigate the design of a general-purpose computer system to achieve massively parallel execution in those computations which contain massive amounts of parallelism. Based on the forgoing discussion, we now set down some goals to fulfill in accomplishing this task.

1.2.1 Computation

The system should detect and take advantage of the maximum amount of program-expressed parallelism in an on-the-fly manner without programmer intervention. Furthermore, since many parallel computations also contain some non-parallel subcomputations, the system should also be able to execute these successfully without any explicit preparation or warning.

1.2.2 Programming Language

Because of its mathematical basis, and its clean and natural expression of parallelism, the system should execute programs written in a functional programming language.

1.2.3 System Hardware

To successfully execute as a large scale system with massive parallelism the system should contain many processing elements. Because of their possibly large number, these processing elements should be as simple and as identical as possible, but they should still support a general purpose overall system capability. Moreover, to prevent communication bottlenecks, the interconnections between processors should allow general patterns of fast parallel interprocessor communication with fully distributed control and synchronization (no global system clock).

1.2.4 Scalability

Because of our concern with system scalability, we shall evaluate a system's performance parameters and hardware costs asymptotically and in terms of the *problem size*, N, of any program executing in that particular system. For this purpose, *problem size* shall refer to the total number of primitive data values all the primitive operations use (where a primitive instruction is one that executes within a single processor and a primitive datum is one that a processor can hold in a single local storage cell or register). Using these conventions, then, we can say that sequential uniprocessors execute their programs in O(N) time, –meaning that as the problem size N becomes large, a sequential uniprocessor's execution time for that problem grows as fast as aN+lower order terms, for some fixed constant a.

We shall call a system's performance for a particular program *power-log fast* if that system can execute that program (with problem size N) in at most $O((\lg N)^s)$ time or less (where $\lg N = \log_2 N$, and s is some small fixed constant). Power-log fast performance implies one major asymptotic consequence: as N grows large, a power-log system will eventually outperform any other system with only polynomial-fast execution with respect to N. This holds even if the polynomial in N involves fractional exponents. Of course, not all computations permit power-log fast executions, and sometimes we can find both power-log fast and non-power-log fast executions for the same computation. For example, if we consider a binary adder (in TTL technology, say) as a network of simple combinational processors (gates) whose primitive data are single bits, then carry-propagating adders like that of Fig. 1-3 are not power-log fast because of their O(N) execution time, whereas carry-lookahead adders like the one in Fig. 1-4 are indeed power-log fast because they only require O(lg N) execution time.

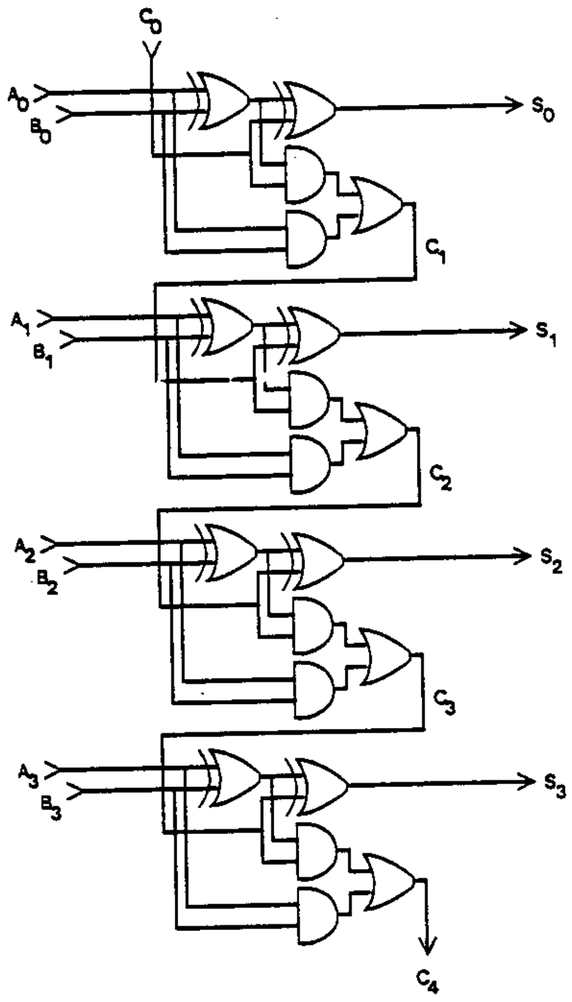


Fig. 1-3: Carry-propagating adder

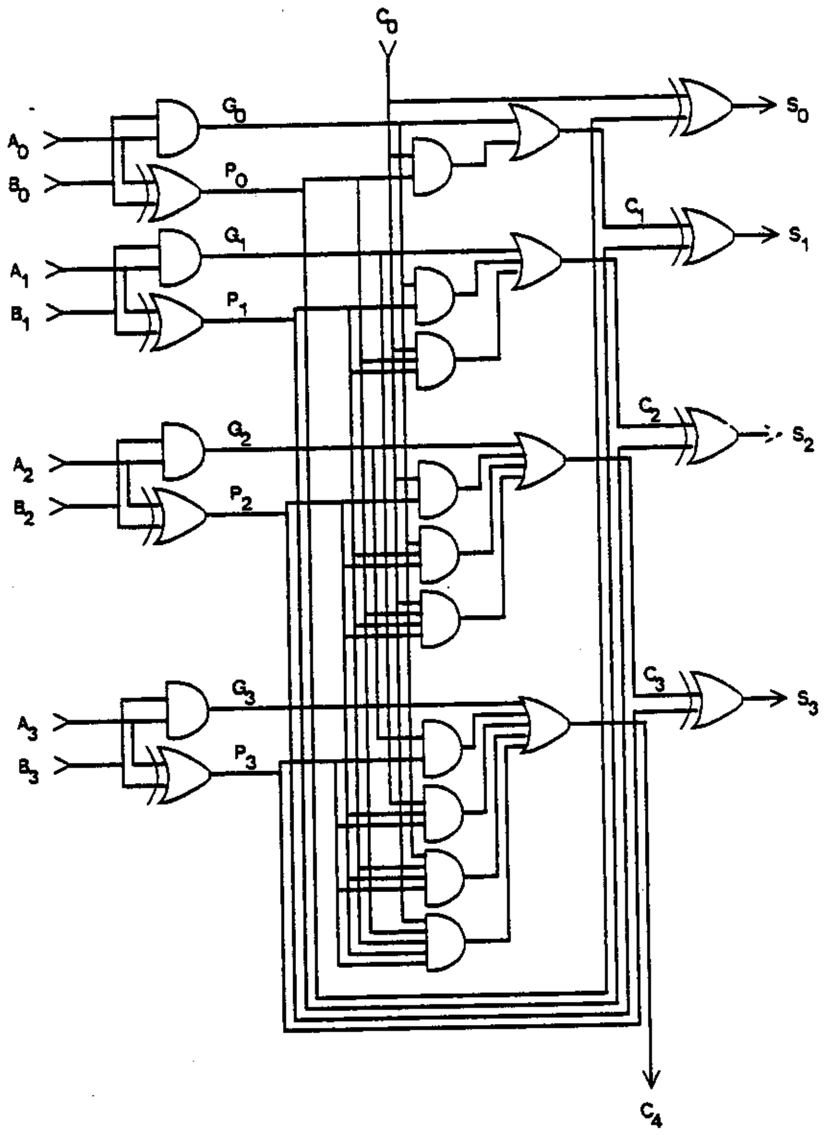


Fig. 1-4: Carry-lookahead adder

We shall call a system a *power-log fast system* only if that system has a power-log fast performance for every algorithm which has any power-log fast execution scheme. While this definition seems to belabor the obvious, we need it so that the existence of even a single slow algorithm for some computation won't force us to ignore all other possible power-log fast algorithms to obtain the same result. For example, some inefficient programmer might implement addition using repeated incrementation by 1 instead of using the fast adder built into most computer systems. Even a power-log fast system would fail to execute this slower addition with power-log speed. However, if any power-log fast system were programmed to add with the carry-lookahead algorithm, it should always execute this addition power-log fast.

A desire for a power-log fast system at any cost raises the following possibility: We could conceivably implement practically any computation by putting all its possible results into a large table ahead of time. "Executing" this computation would then merely involve performing a table lookup, –a power-log fast computation! Since we do not want to implement all the possible computations of a general-purpose computer in this manner, we need to impose some asymptotic limit on hardware as well as execution time.

We now state our system's two scalability goals: The system should behave as a power-log fast system for any computation which it can successfully execute to completion, and the system's hardware costs should scale no worse than $O(N(\lg N)^s)$, for a system containing $O(N)$ processors and some small fixed constant s.

One last remark on our scalability goals: When we compare system performances in this report, we shall insist on comparing their *worst case* speeds, rather than overall average speeds, for massively parallel computations. We do this for three important reasons: First, for time-critical system applications, always allowing for worst case execution time can prevent disasters caused by unanticipated executional latencies. Second, for any single algorithm the worst case execution time has less application-dependence than does the average execution time. And finally, as we shall see in chapter 4, the distributed synchronization requirements of our proposed design forces certain parameters (the length of a "reduction step", for example) to always have worst case values (albeit power-log fast ones).

1.3 Towards A Solution

The rest of this report develops and describes a multiprocessor design proposal attempting to fulfill all these goals. Chapter 2 describes the basic difficulties of current widely-used programming languages when they try to express parallel computation, and then establishes the special suitability of functional reduction languages for this same purpose. Chapter 3 examines the classical approaches to the direct concurrent execution of general-purpose functional programs, and also examines the tree machine proposed by Magó. Chapter 4 presents a new, more parallel approach to executing functional reduction languages and a multiprocessor design to accomplish this approach. Chapter 5 evaluates this new design and compares it against Magó's machine. And finally, chapter 6 draws some conclusions about our proposal and offers some thoughts toward the future of implementing such massively parallel computers.

2. Reduction Languages: Functional Programming with Parallelism

2.1 Conventional Limitations to Parallelism

Conventional computer systems impose two basic limitations which hinder parallelism in the programs they execute. First and foremost, as Backus[1] points out, they possess a conceptual communication bottleneck. The system CPU must communicate to a memory system through a channel usually only one word wide. Thus, an executing program can change its program state only a tiny step at a time. For programming languages like Fortran or Algol, these interactions involve complex protocols with a complex state (for example, Johnston's contour model for Algol[9]). This step-at-a-time constraint severely limits system speed. Furthermore, planning and managing data access overwhelms all algorithm construction and execution, since all operands, all results, their addresses, pointers, bounds, and so on must pass, one word per memory cycle, through the same bottleneck.

The architectures of conventional computer systems originated the communication bottleneck. The persistence of the programming community in using conventional step-at-a-time programming languages has perpetuated such architectures along with the bottleneck.[1] For example, even though most of the high performance computers of the late 1970s boasted simultaneous multiword fetches from multi-module memories, this only amounted to negligible widening of the memory-processor bottleneck in such systems, which still remained a fixed and microscopic size compared to their multi-megaword address spaces. Interference between accesses to the same memory module often defeated parallelism.[10] And besides, many of the users of these computers still continue to program them in Fortran. The communication bottleneck persists, therefore, in both the hardware and the software of these machines.

Conventional systems also suffer from a second limitation: their dependency entanglements. The data and control dependencies in conventional systems often get so complex, so obscure, and so tangled, that they can only safely execute their programs sequentially. Subroutines with side effects might unexpectedly alter the values of global program variables. Reusing memory locations (an old technique to conserve memory space) usually creates data dependencies not inherent to the logical operation of the program. Unstructured branching can completely bury any possibilities for program parallelism. A host of other common programming practices also produce such entanglements. Most of the trouble comes from the complete separation of a conventional program from its all-important, yet invisible program state. In fact, in order to find out what caused any certain program behavior the programmer must reconstruct the invisible program state by hand simulation or by forcing the machine to regurgitate a core dump. Programs usually fail to express all their usable parallelism because checking all their dependency entanglements requires too much expensive effort from both programmer and machine.

2.2 Overcoming these Limitations

To exploit maximum concurrency and parallelism at all levels we must have the following necessary conditions[11]:

NC1) All data dependencies should be completely deducible.

NC2) All sequencing constraints should be due only to data dependencies.

Ensuring all three of the following will fulfill these conditions[11]:

ATR1) Locality of effect: Keep the data dependencies of each instruction well-defined and explicitly limited in scope, so that independent uses of different data will not affect or interfere with each other.

ATR2) Freedom from side effects: All operations, functions, and subroutines should have read-only access to their operands and not be able to modify them (i.e., "call-by-value").

ATR3) Data driven execution: Every operation can proceed as soon as it has available all its operands.

ATR1 eliminates the unnecessary data dependencies which programmers might inadvertently introduce. ATR2 reduces all data dependencies to either the availability or unavailability of operands to an operator. ATR3 makes all sequencing constraints due only to these data dependencies.

2.3 Programming Language Approaches to Overcoming these Limitations

A few existing programming systems possess one or more of the attributes ATR1, ATR2, and ATR3 which we want for parallelism. For example: LISP 1.5 [12] has locality of effect (but not the other two attributes). Lazy Pure Lisp[13] and FGL[14] both have locality of effect and freedom from side effects, but neither have data driven execution. (These both depend on demand-driven evaluation, a strategy which conserves resources at the expense of speed and of parallelism.) And finally, single-assignment languages like Lucid[15], and data flow languages like VAL[16] and ID[17] have all three of these attributes. All of these examples have much in common with the paradigm which we call *functional programming*[5]. Functional programming systems use expression-oriented syntax to express computation via the application of functions to arguments.

In this report we investigate a purely functional programming approach to parallel execution. As we see in this chapter, *reduction languages* possess all three of the attributes we want for parallelism: They have locality of effect, freedom from side effects, and can execute via data driven execution. Furthermore, they have a simple execution semantics and express their parallelism in clear and natural fashion. Let's define reduction languages formally and justify all these claims.

2.4 Formal Reduction Languages

Reduction languages form a subclass of the class of closed applicative languages and closed applicative languages, in turn, form a subclass of the class of complete languages having a constructor syntax.[3] We shall clarify the meanings of these classifications so that we can examine more closely what properties of reduction languages make them suitable candidates for highly parallel systems.

In the following, we will use the conceptual notations and conventions of Backus[3] and of Berkling[4] to formally describe reduction languages.

2.4.1 Complete Languages

A *complete language* consists of a set E of expressions, a set C of constants, and a partial function μ from E onto C such that:

CL1) $C \subseteq E$

CL2) $\mu e = c$, for all $c \in C$

We call μ the *semantic function* of this complete language. For all $e \in E$, if μe is defined, then μe is the (unique) *meaning* of e. In this way, the definition of any complete language completely specifies its inherent computational semantics. There is no separate or invisible state. All computation consists of finding the meaning of an expression. Once we have this meaning or value we have the result of the computation.

Actually, we should find such a value-oriented view of computation quite familiar. Consider, for example, the simple complete language of arithmetic expressions without variables, where:

$$\mu\,((1+2) * (3+4)) = 21$$

The concept of "state" obviously has no meaning here. The value of an arithmetic expression is simply its value, nothing more. Of course, we haven't yet discussed how a value is found. The semantic function μ subsumes all the computation to find a value.

2.4.2 Constructor Syntax

A set E of expressions has what is called a *constructor syntax* iff:

CS1) It contains a subset, $A \subseteq E$, of elements called *atoms.*

CS2) It has a set K of partial functions called *constructors* which map E^n into E ($n \geq 0$), so that for all $e \in E$,
either $e \in A$,
or else $e = k(e_1,...,e_n)$, for some unique $k \in K$ and $e_1,...,e_n \in E$.

Thus, a set of expressions has a constructor syntax if all its expressions are either atoms or formed of constructed subexpressions. The familiar language of arithmetic expressions like

$$(((1*2)+3)*(((4+5)+6)+7))$$

with all operator associations made explicit using parentheses, has a constructor syntax. Here each matched pair of parentheses denotes a single-subexpression constructor. If we add to this language a vector notation to accommodate expressions like

$$(0 , 1 , (2,3) , 4 , ((5*(6+7))+8) , 9)$$

then we have a language with multi-subexpression constructors.

2.4.3 Closed Applicative Languages and Strict Realizations

We call a complete language with a constructor syntax a *closed applicative language* iff:

CAL1) All its atoms are constants, i.e., $A \subseteq C$.

CAL2) It has a two-place constructor **ap** $\in K$ such that:
$\mu\cdot\mathbf{ap}(e,f) = \mu\cdot\mathbf{ap}(\mu e,\mu f)$, for all $e,f \in E$.

CAL3) For all its other constructors $k_n \in K$:
$\mu\cdot k_n(e_1,...,e_n) = k_n(\mu e_1,...,\mu e_n)$, for all $e_1,...,e_n \in E$.

CAL4) We can find a total function ρ mapping $C \rightarrow (C \rightarrow E)$ such that
$\mu\cdot\mathbf{ap}(c,d) = \mu\cdot(\rho c)d$, and is defined for all $c,d \in C$.

Expressions constructed by **ap** we call *applications*. According to CAL1 above, in closed applicative languages atoms are their own meanings. We find the meaning of any constructed expression e by first replacing its components with their meanings. If e is not an application CAL3 yields its meaning completely. If e is an application CAL2 simplifies it without changing its meaning. The function ρ is the *representation function* associated with the language. Closed applicative languages represent functions by constants (either primitive or composite) and then use the **ap** constructor so that CAL4 will apply the represented function to its argument.

All closed applicative languages have two of the attributes we want for parallelism: locality of effect and freedom from side effects. CAL4 tells us that the result of an application depends only on the represented function's own argument. CAL2 and CAL3 together ensure that replacing an expression's components by their meanings cannot change the meaning of that expression. The question now arises: Can we use data driven execution to evaluate closed applicative expressions (the third attribute we wanted)? To answer this, let's clarify what we mean by "executing an evaluation".

We say a complete language has a *strict realization* iff:

SR1) We can find a function τ mapping $E \rightarrow E$ where:
μe is defined for $e \in E$ iff
there exists an integer n such that $\tau^n e \in C$,
in which case $\mu e = \tau^n e$.

We call τ the transition function for μ. We can always find μe, if it exists, by calculating $\tau(e)$, $\tau(\tau(e))$, $\tau(\tau(\tau(e)))$, ... ; If this sequence converges within C, then that value is μe. In fact, μe is undefined unless repeated τ on e converges. Such a τ-repetition sequence defines an *execution sequence* for a complete language.

Given any strict realization, we can immediately determine the complete language it realizes since we can easily derive μ from τ. The converse is not so easy, however, because given a complete language, even a closed applicative language, we have no general way to find a strict realization for it. At least one subclass of the closed applicative languages, the reduction languages, does not have this problem.

2.4.4 Reduction Languages

A *reduction language* is any closed applicative language containing the constructors:

Application:	**ap** , as defined above in CAL1 and CAL4
Sequence:	σ , where $\sigma_n(e_1,...,e_n) = (e_1,...,e_n)$
Abstract Error:	$\omega(\ )$, a nullary constructor

and where:

$\emptyset$, the empty sequence, is also an atom,

and where ρ, the representation function has the following definition:

RF1) $\rho\emptyset$ is the identity function, i.e., $(\rho\emptyset)e = e$ for all $e \in E$.

RF2) For all other $a \in A$, a particular reduction language must define ρa as a primitive function, or else the programmer must supply ρa in a *definition.* Otherwise $\rho a = \omega(\ )$.

RF3) For $\rho(c_1,\ldots,c_n)$, if c_1 is regular, then
$\rho(c_1,\ldots,c_n) \;=\; (\rho c_1)\cdot\mu\cdot\rho(c_2,\ldots,c_n)$.

RF4) For $\rho(c_1,\ldots,c_n)$, if c_1 is meta, then
$\rho(c_1,\ldots,c_n) \;=\; (\rho c_1)\cdot\pi(c_1,\ldots,c_n)$
(where πc is such that $(\pi c)d \;=\; (c,d)$ for all $c,d \in C$).

We call an expression *meta* if it is atomic and defined as meta, or if it is a sequence whose first (or only) element is Ø; otherwise we call the expression *regular*. As we see in Fig. 2-1, operators involving meta expressions evaluate in a different manner than those containing only regular expressions.

<u>Regular composition:</u>

ap (REVERSE,(A,B,C)) = *reverse* ((A,B,C))
= (C,B,A)

<u>Meta composition:</u>

ap ((META-REVERSE),(A,B,C)) = ap (META-REVERSE,((META-REVERSE),(A,B,C)))
= *meta-reverse* (((META-REVERSE),(A,B,C)))
= ((A,B,C),(META-REVERSE))

(where *reverse* and *meta-reverse* have identical meanings, except that REVERSE is regular and META-REVERSE is meta)

Fig. 2-1: Evaluation of expressions involving regular and meta composition

With *meta composition*, functions can operate on representations as well as values. As a result, meta expressions can specify higher-order functionals (functions which operate on functions). In fact, by manipulating a representation of its own self a meta operator can specify functional recursion. *Regular composition*, on the other hand, corresponds to the familiar notion of function composition in which functions only operate on values.

Now let's see how to obtain a strict realization for any given reduction language:

Theorem A: Every reduction language has a strict realization.

Proof[3]: We give τ.

τ: Find an innermost application, ap(c,d) and do one of:

TR1) If $c \in A$, then
ap(c,d) $\rightarrow$ $(\rho c)d$.

TR2) If $c = (c_1,\ldots,c_n)$ and c_1 is meta, then
ap(c,d) $\rightarrow$ ap(c_1, (c,d)).

TR3) If $c = (c_1,\ldots,c_n)$ and c_1 is not meta, then
ap(c,d) $\rightarrow$ ap(c_1, ap(($c_2,\ldots,c_n$), d)) if $n>1$, or
ap(c,d) $\rightarrow$ ap(c_1,d) if $n=1$.

TR4) If $c = \omega(\ )$, then
ap(c,d) $\rightarrow$ $\omega(\ )$.

The above proof gives us *reduction rules.* We call the application of τ to a reduction language expression a *reduction transition.* Note how every reduction transition operates only on explicitly present information in the expression. We never refer to an invisible state when finding the meaning of an expression.

The following remarkable theorem holds for all reduction languages:

Theorem B (The Extended Church-Rosser Property)[3, 18]:

ECR1) Every terminating sequence of reductions on an expression yields the same meaning for it, and

ECR2) If an expression has a meaning, then every sequence of reductions on it terminates.

This theorem finally gives us the third attribute we want for parallelism: data driven execution (ATR3). Reduction transitions can proceed in a concurrent data driven fashion because ATR1 and ATR2 guarantee the mutual data-independence of separate innermost applications, while the Extended Church-Rosser Property guarantees their order-independence. Thus, any sequence of executable reductions on an expression will always give us the same result, even if we reduce more than one innermost application at the same time (i.e., in parallel).

2.5 Necessary vs. Sufficient Conditions for Parallelism

We have just seen how the execution of reduction languages can possess all three of the attributes we want for parallelism: locality of effect, freedom from side effects, and data driven execution. These attributes fulfill two **necessary** conditions (NC1 and NC2 we saw in 2.2) to suggest that, given enough hardware, we can always execute all the expressible parallelism of any reduction language program. Any particular reduction language or program, however, might still lack **sufficient** conditions for expressing parallelism, and thus fail to achieve any parallel execution at all!

To solve this sufficiency problem we depend on both the language designers and the user programmers. The designers of any particular reduction language must provide easy ways of specifying concurrency in functional applications. For example: Fig. 2-2 shows how a single function might simultaneously act on a large number of arguments,

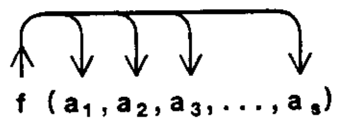


Fig. 2-2: A single function acting on many arguments

Fig. 2-3 shows how a large number of functions might act on a single argument,

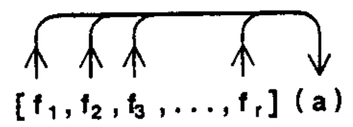


Fig. 2-3: Many functions acting on a single argument

and Fig. 2-4 suggests that many functions might act on many arguments.

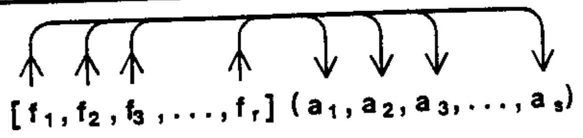


Fig. 2-4: Many functions acting on many arguments

In section 2.6 we define an example reduction language which neatly expresses all of these situations.

A parallel reduction language system should encourage programmers to write parallel programs. But no system can always enforce this. An inept or unsympathetic programmer might completely defeat the intent of the system, say, by using a sequential algorithm instead of an equivalent parallel one, for example. To discourage this, might require re-educating the entire programmer community which still thinks only in terms of sequential execution and the communication bottleneck. Backus provides some hope in this endeavor with his very attractive functional reduction language called FFP.[1]

2.6 An Example Reduction Language

We now define a small example reduction language which closely resembles one the languages described by Backus[3]. To do this, we specify its concrete syntax and its concrete execution semantics. This language serves only as an example for this report, and lacks many of the useful features we would want in a real programming language. We shall nevertheless find it powerful enough to demonstrate highly parallel execution.

2.6.1 Concrete Syntax

Our example reduction language uses strings of *symbols* to represent expressions. Each of our symbols is either a *bracket*, an entire atomic *name*, or an entire atomic *value*. We use two kinds of brackets to represent two of our constructors as follows:

Application:	ap(e,f) is represented by <e,f>
Sequence:	$\sigma_n(e_1,...,e_n)$ is represented by $(e_1,...,e_n)$

We use "bottom" to represent our third constructor:

Abstract Error: $\omega(\ )$ is represented by $\perp$

In this report we show atomic names as strings of characters, although a reduction machine might internally represent them in some other fashion (bit patterns, perhaps). Our atomic values include a set of integers (written as numerals) and the boolean values true (T) and false (F). These conventions, together with the definition we saw in 2.4.2 of a constructor syntax completely specify the syntax of our reduction language. Fig. 2-5 shows some sample expressions in our reduction language.

< + , (2,3) >	-adds the numbers 2 and 3
< [+ ,-, *,/] , (2,3) >	-simultaneously finds the sum, difference, product, and quotient of 2 and 3
< (AA,AA,*) , (((1,2),(3,4)),((5,6),(7,8))) >	-multiplies the innermost pairs of numbers together using the AA meta operator to "get inside" two levels of parentheses

Fig. 2-5: Some expressions in our reduction language

Our bracket notation for constructors provides another useful way of viewing reduction language expressions. Fig. 2-6 shows a reduction language expression and its equivalent representation as a tree structure.Notice how all the constructors constitute the connecting nodes of this tree, while all the atoms sit as its leaves. This tree structure represents each inner level of bracket nesting as a lower tree level. In fact, if we isolate each symbol, associate each with its level number, record them all in preorder fashion, and delete the now-redundant right brackets and commas, we obtain the equivalent (but more compact) cellular representation shown at the bottom of Fig. 2-6. Like Magó[19], we will find this preorder-with-level-number format very convenient for internal machine representations of expressions.

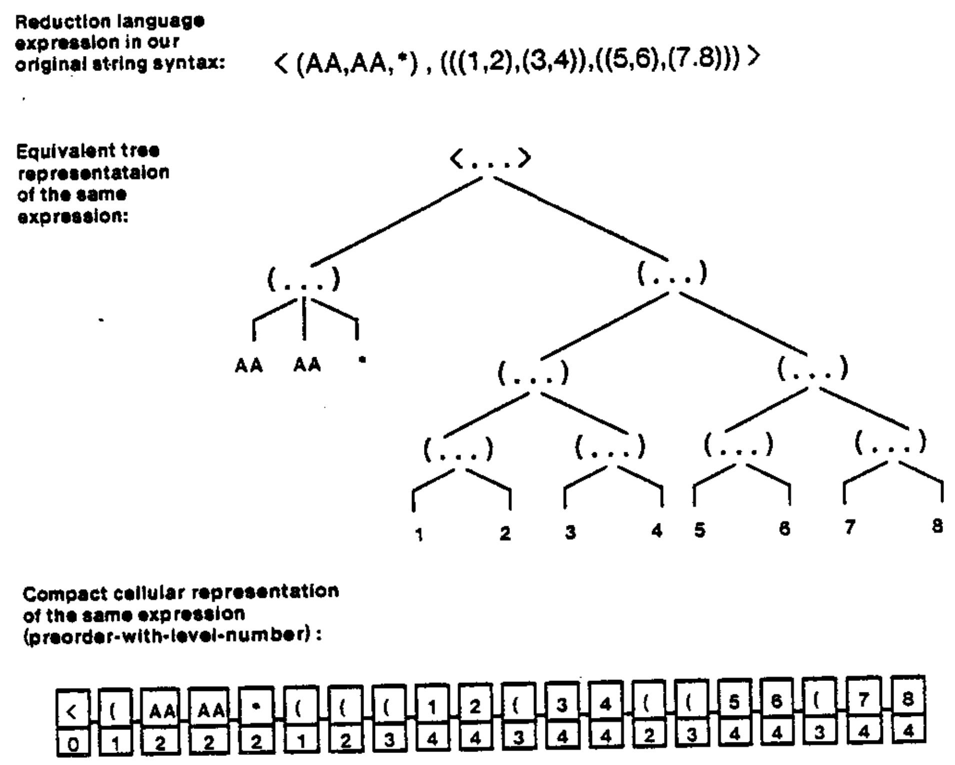

Reduction language expression in our original string syntax:
< (AA,AA,*) , (((1,2),(3,4)),((5,6),(7.8))) >
Equivalent tree representataion of the same expression:
<...>
(...)
AA
AA
*
1
2
3
4
5
6
7
8
Compact cellular representation of the same expression (preorder-with-level-number):
<
0
(
1
AA
2
AA
2
*
2
(
1
(
2
(
3
1
4
2
4
(
3
3
4
4
4
(
2
(
3
5
4
6
4
(
3
7
4
8
4

Fig. 2-6: Equivalent representations of an expression

2.6.2 Execution Semantics

To completely reduce an expression in our reduction language we must perform reduction transitions on innermost applications until we have no more applications left in the expression. Each reduction transition overwrites applications with their results. We call this style of execution *string rewriting semantics.* At the end of reducing an expression (assuming we reach an endpoint) we either have an expression containing no applications (i.e., a constant consisting of either an atom or a sequence) or else an error ($\perp$).

To completely describe the actions of any reduction transition, we must first fully specify all the primitive functions of our example reduction language. The following list of names and their corresponding meanings supply all the information we need to invoke reduction rule TR1 of Theorem A for any of our primitive functions:

Meta Operators

NF (N Functions)

$\langle(NF,f_1,f_2, \ldots ,f_m),x\rangle$ reduces to $(\langle f_1,x\rangle,\langle f_2,x\rangle, \ldots ,\langle f_m,x\rangle)$

Comment: NF (also called "construction" in many functional programming systems) enables many functions to simultaneously operate on the same argument(s), as shown in Fig.2-3 and Fig.2-4. Because of this usefulness we also use the square brackets, [...], to abbreviate (NF ...) in our examples. Thus we equivalently say that:

$\langle[f_1,f_2, \ldots ,f_m],x\rangle$ reduces to $(\langle f_1,x\rangle,\langle f_2,x\rangle, \ldots ,\langle f_m,x\rangle)$

CN (ConditioN)

$\langle(CN,f_1,f_2,f_3),x\rangle$ reduces to

$\langle(CN,\langle f_1,x\rangle,f_2,f_3),x\rangle$ if f_1 is not a constant function,

$\langle f_2,x\rangle$ if $f_1=T$,

$\langle f_3,x\rangle$ if $f_1=F$,

otherwise it reduces to $\perp$.

C (Constant)

$\langle(C,v),x\rangle$ reduces to v

Comment: We also use C(v) to represent (C,v). Thus we equivalently say that:

$\langle C(v),x\rangle$ reduces to v

AA (Apply to All)

$\langle(AA,f),(a_1,a_2, \ldots ,a_m)\rangle$ reduces to $(\langle f,a_1\rangle,\langle f,a_2\rangle, \ldots ,\langle f,a_m\rangle)$

Comment: AA enables the same function(s) to operate on many arguments as shown in Fig.2-2 and Fig.2-4.

INS (INSert)

$\langle(\mathrm{INS},f),x\rangle$ reduces to

$\langle f,(a1,\langle(\mathrm{INS},f),(a_2, \ldots ,a_m)\rangle)\rangle$ if $x=(a_1,a_2, \ldots ,a_m)$ and $m>1$,

a_1 if $x=(a_1)$,

$(\ )$ if $x=(\ )$.

Comment: Notice how INS produces recursion via meta composition.

AI (Associative Insert)

$\langle(\mathrm{AI},f),x\rangle$ reduces to

$\langle f,(\langle(\mathrm{AI},f),(a_1, \ldots ,a_{\lceil m/2\rceil})\rangle,\langle(\mathrm{AI},f),(a_{\lceil m/2\rceil+1}, \ldots ,a_m)\rangle)\rangle$

if $x=(a_1,a_2, \ldots ,a_m)$ and $m>1$,

a_1 if $x=(a_1)$,

$(\ )$ if $x=(\ )$.

Comment: AI directly produces tree reduction involving a single implicitly repeated operator part ("f" above).

Regular Operators

ID (IDentity)

$\langle ID,x\rangle$ reduces to x

Comment: When the empty sequence, ( ), occurs in the operator part of an application, we can say: $(\ )^{\cdot} = ID$ and $ID = (\ )$, and thus, either one can serve as the concrete representation of $\varnothing$ (recall 2.4.4).

HD (HeaD)

$\langle HD,x\rangle$ reduces to

a_1 if $x=(a_1,a_2, \ldots ,a_m)$ and $m\geq 1$,

otherwise it reduces to $\perp$.

TL (TaiL)

$\langle TL,x\rangle$ reduces to

$(a_2, \ldots ,a_m)$ if $x=(a_1,a_2, \ldots ,a_m)$ and $m\geq 2$,

$(\ )$ if $x=(a_1)$,

otherwise it reduces to $\perp$.

AP (APply)

$\langle AP,x\rangle$ reduces to

$\langle x_1,x_2\rangle$ if $x=(x_1,x_2)$,

otherwise it reduces to $\perp$.

EQ (generalized EQuals predicate)

$\langle EQ,x\rangle$ reduces to

T if $x=(x_1,x_1)$,

F if $x=(x_1,y_1)$ and $x_1 \neq y_1$,

otherwise it reduces to $\perp$.

ATOM (ATOMic predicate)

$\langle ATOM,x\rangle$ reduces to

T if $x=(\ )$, or $x \neq \perp$ and x is atomic,

F if $x \neq \perp$ and x is not atomic,

otherwise it reduces to $\perp$.

NULL (NULL sequence predicate)

$\langle NULL,x\rangle$ reduces to

T if $x=(\ )$, or $x=ID$,

F if $x \neq \perp$ and $x \neq (\ )$ and $x \neq ID$,

otherwise it reduces to $\perp$.

UN (UNion)

$\langle UN,x\rangle$ reduces to

$(a_1,a_2, \ldots ,a_m)$ if $x=(a_1,(a_2, \ldots ,a_m))$ and $m>1$,

(a_1) if $x=(a_1,(\ ))$,

otherwise it reduces to $\perp$.

+,-,*,/ (arithmetic operations)

$\langle +,x\rangle$ reduces to the value a_1+a_2 if $x=(a_1,a_2)$ and a_1,a_2 are numerical values

$\langle -,x\rangle$ reduces to the value $a_1 - a_2$ if $x=(a_1,a_2)$ and a_1,a_2 are numerical values

$\langle *,x\rangle$ reduces to the value a_1*a_2 if $x=(a_1,a_2)$ and a_1,a_2 are numerical values

$\langle /,x\rangle$ reduces to the value a_1/a_2 if $x=(a_1,a_2)$ and a_1,a_2 are numerical values

INX (INdeX)

$\langle INX,x\rangle$ reduces to

$(1,2, \dots ,m)$ if x has the numerical value m,

otherwise it reduces to $\perp$.

REVERSE (REVERSE a sequence)

$\langle REVERSE,x\rangle$ reduces to

$(a_m,a_{m-1}, \dots ,a_2,a_1)$ if $x=(a_1,a_2, \dots ,a_m)$ and $m>1$,

a_1 if $x=(a_1)$,

$(\ )$ if $x=(\ )$,

otherwise it reduces to $\perp$.

TRANS (TRANSpose 2-dimensional array)

$\langle TRANS,x\rangle$ reduces to

$((a_{11},a_{21}, \dots ,a_{j1}),(a_{12},a_{22}, \dots ,a_{j2}), \dots ,(a_{1k},a_{2k}, \dots ,a_{jk}))$

if $x=((a_{11},a_{12}, \dots ,a_{1k}),(a_{21},a_{22}, \dots ,a_{2k}), \dots ,(a_{j1},a_{j2}, \dots ,a_{jk}))$ and $j,k>1$,

$((a_{11}))$ if $x=(\ (a_{11}))$,

otherwise it reduces to $\perp$.

In our example language, meta composition looks like:

$$\langle (c_1,\ldots,c_n)\ d\rangle \rightarrow \langle c_1\ (c_1,\ldots,c_n,d)\rangle\ ,\ \text{when } c_1 \text{ is meta}$$

and regular composition looks like:

$$\langle (c_1,\ldots,c_n)\ d\rangle \rightarrow \langle c_1\ \langle (c_2,\ldots,c_n)\ d\rangle\ \rangle\ ,\ \text{when } c_1 \text{ is regular}$$

while errors still propagate:

$$\langle \perp, d\rangle \rightarrow \perp\ ,\ \text{for any } d$$

In fact, any atomic name which does not represent a primitive function must represent $\perp$ unless the programmer supplies a definition for it. Let's look at how the programmer might do this.

2.6.3 Function Definitions

In our language we include one more primitive operator, the *definition facility.* It has a syntax which looks like:

<DEF (atomic_name, expression)>

Despite its functional appearance, it cannot appear in any expression. All definitions must appear outside of any expression and outside all other definitions. Reductions have no effect on the definitions themselves, but definitions can affect the reduction of program expressions. Whenever the reduction of an expression uncovers an atomic name which doesn't represent a primitive function, we must check if the programmer defined it in a definition. If so, we rewrite the atomic name with its corresponding expression. This *definition expansion* takes up one entire reduction transition.

Definitions allow programmers to represent chosen subexpressions with convenient atomic names. In this way, programmers can avoid tedious repetitions of long subexpressions that an expression may happen to use many times. This "hiding of information" encourages a modular programming style which usually improves the understandability of programs. Moreover, if a definition's own atomic name appears in its expression part, functional recursion results. Thus, the programmer can specify functional recursion without resorting to meta operators. Including a definition facility in our reduction language has no formal impact on its expressive power, however; because given any expression which contains definitions, we can always find an equivalent (but more cumbersome) expression which uses meta composition instead.[3]

3. Computer Execution of Reduction Language Programs

3.1 Requirements

No matter what kind of system does it, the execution of reduction language programs via reduction transitions always involves four basic activities: storing expressions, locating reducible applications, executing reducible applications, and overwriting reducible applications with their results. A conventional uniprocessor can only accomplish these activities one way: sequentially. The ideal multiprocessor system, on the other hand, would accomplish these activities with full parallelism. Let's look at these four activities and see what they entail.

3.1.1 Storage of Expressions

The reduction language program and the data it acts on together constitute an expression. The computer system must somehow store this expression's symbols and values in a readily accessible fashion. As we noted in section 2.1, conventional storage in a separate memory, or even a small group of memories would give us the familiar processor-memory bottleneck to hinder parallelism. A large group of (necessarily small) memories might widen this bottleneck, but unless these memories were "close" to the processing hardware, using them might dominate the cost of using the computer system. Having memories extremely close to the processing hardware means co-location of the storage space and the processing space ("processing in memory"). Such a system would distribute its entire working storage space among a large number of processors. Each processor in this system would contain its own few registers of working storage. Accessing any piece of storage from elsewhere in the system would require communication with the processor containing it. The rest of this report describes the enormous power of this system approach when we use it in multiprocessor reduction machines.

3.1.2 Location of Reducible Applications

In reduction language programs, the applications (i.e., all subexpressions immediately within a matched pair of brackets "<" and ">") denote all executable computation. Furthermore, data driven reduction, by definition (see sections 2.2 and 2.4), reduces the innermost applications first. Thus, for any reduction transition, the computer system must find these innermost applications. We call these innermost applications the *reducible applications* (RAs) of the expression. The syntax of a reduction language expression uniquely determines where all of its RAs reside: If we can find any subexpression bracketed with "<...>", and with neither "<" nor ">" within it, then we have found an RA. We call such a syntax-directed search for RAs *scanning.*

A conventional uniprocessor scans for RAs sequentially. It serially progresses along an expression and examines symbols one at a time, as in Fig. 3-1.

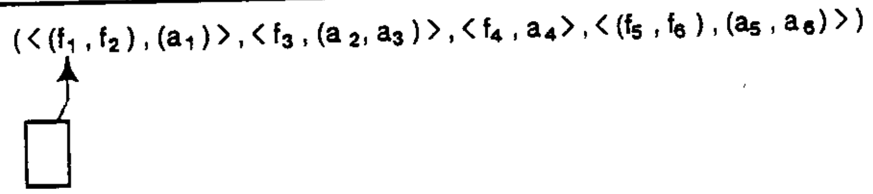

$(<(f_1, f_2), (a_1)>, <f_3, (a_2, a_3)>, <f_4, a_4>, <(f_5, f_6), (a_5, a_6)>)$

Fig. 3-1: Uniprocessor serially scanning an expression

A parallel multiprocessor might also scan an expression serially, but all of its processors would not need to scan the same place at the same time. Each processor element might simultaneously and independently scan at a different place along the expression, as in Fig. 3-2. In this fashion, we would hope to find many RAs at once.

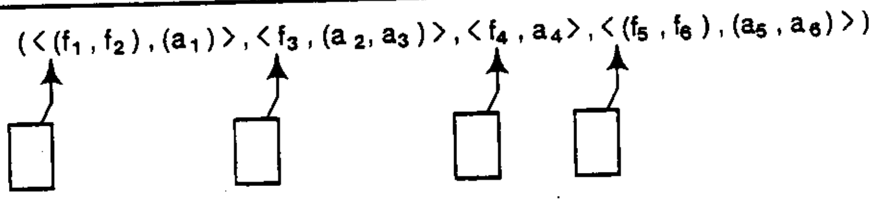

$(<(f_1, f_2), (a_1)>, <f_3, (a_2, a_3)>, <f_4, a_4>, <(f_5, f_6), (a_5, a_6)>)$

Fig. 3-2: Many processors simultaneously scanning an expression serially

For important cases, however, this particular multiprocessor approach would fail to locate all the RAs of an expression any faster than a single processor could. For example, Fig. 3-3 shows how the initial application of a composite function to a composite operand might give us an expression with only one RA: the entire expression itself!

$$\langle (f_1, f_2, f_3, [f_4, f_5], f_6, f_7, f_8, f_9, f_{10}), ((a_1, a_2, a_3, a_4), (a_5, (a_6))) \rangle$$

Fig. 3-3: A large composite expression containing only one all-inclusive reducible application

No matter how many processors the system devoted to this task, at least one serially scanning processor element would need to traverse the entire expression in order to find an RA. Fig. 3-4 shows how even later reduction transitions on this same expression still produce RAs of long length as we dissect the composite function into its regular and meta compositions.

$$\langle (f_1, f_2, f_3, [f_4, f_5], f_6, f_7, f_8, f_9, f_{10}), ((a_1, a_2, a_3, a_4), (a_5, (a_6))) \rangle$$
$$\downarrow$$
$$\langle f_1, \langle (f_2, f_3, [f_4, f_5], f_6, f_7, f_8, f_9, f_{10}), ((a_1, a_2, a_3, a_4), (a_5, (a_6))) \rangle \rangle$$
$$\downarrow$$
$$\langle f_1, \langle f_2, \langle (f_3, [f_4, f_5], f_6, f_7, f_8, f_9, f_{10}), ((a_1, a_2, a_3, a_4), (a_5, (a_6))) \rangle \rangle \rangle$$
$$\downarrow$$
$$\vdots$$

Fig. 3-4: Later reductions on this same expression still yielding single long reducible applications

At least one of the serially scanning processors would still need to traverse a majority of the entire expression during each of these transitions. In general, since a serially scanning processor must always traverse the entire length of an RA to locate it, the presence of long length RAs in any expression will significantly slow the serial scanning process, even for multiprocessors. Thus, we must take care when designing the RA scanning process, or else it alone can severely constrain parallelism.

3.1.3 Execution of Reducible Applications

Once it has found the RAs of an expression (or at least one RA, if operating sequentially), the computer system must apply the operators to their operands and compute their results. We call the computation required to reduce RAs *execution*. If an RA contains a composite function as an operator, several reduction transitions must first reduce the composite function via regular or meta compositions, as appropriate, until we eventually obtain our desired RAs, which only apply primitive functions to their data objects.

A conventional uniprocessor executes RAs sequentially, progressing serially along one RA at a time while it examines one symbol at a time. Likewise, a parallel multiprocessor might also execute each RA serially, but with many processors working on separate RAs, this multiprocessor could execute many RAs simultaneously. A functional program with many RAs could then take advantage of *tree reduction* to achieve high performance. Fig. 3-5 shows an example of tree reduction, whereby a parallel system evaluates a suitable expression of M innermost subexpressions in only ⌈lg M⌉ reduction transitions.

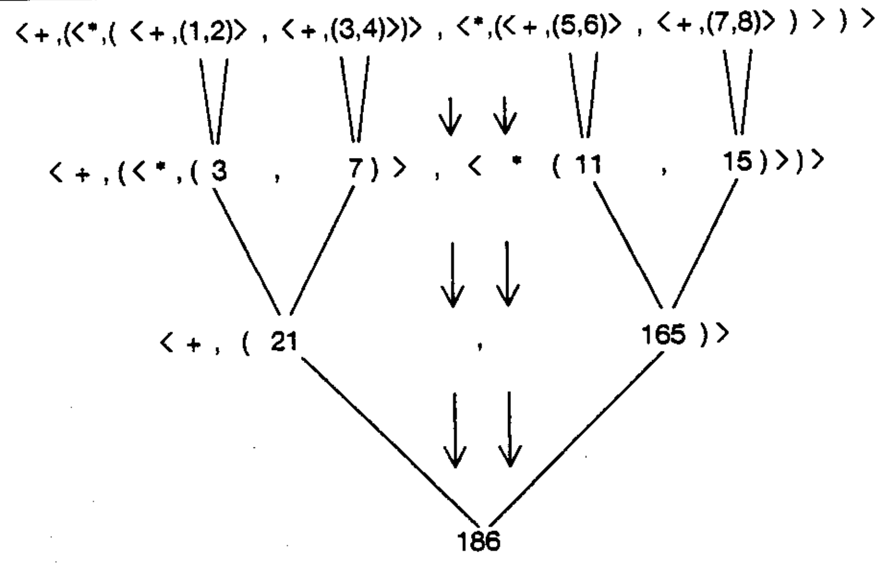


Fig. 3-5: An expression reduction taking advantage of tree reduction

A serially scanning multiprocessor approach, however, fails to execute a single RA any faster than a single serial processor could, because with certain important operators, we could get very long execution times for large data objects. For example, each of the primitive operators NF, AA, TRANS, and REVERSE rewrites its input data object with many changes distributed throughout it (see Appendix A). Even with many serially-executing processors working on such an RA at the same time, all of them would have to traverse the entire RA to first read the operator name and then find the correct spots for all these changes, all without mutual conflicts. Thus, we must take care when designing the RA execution scheme, or else it alone can severely constrain parallelism.

3.1.4 Overwriting Reducible Applications with their Results

After executing an RA the computer system must symbolically replace the RA with its result. (Recall the string-rewriting semantics described in 2.6.2.) During the rewrite of an RA, one of the following three things might occur:

RW1) the result might require the same amount of storage that the RA did,

RW2) the result might require more storage than the RA did, or

RW3) the result might require less storage than the RA did.

The first case causes no necessary storage management. The second case demands that the system find some idle storage space and make it available where we need it. The third case gives us some new idle storage space to possibly reclaim and use elsewhere in the system. If we assume that the system stores expressions in the processor-distributed memory space we described in 3.1.1, then many processors must co-operate in this storage management, —possibly all the processors in the system. The system should recognize all idle storage regions and move them to any RAs that need them. If this storage management happens only through nearest-neighbor communication, however, we find ourselves with a storage management bottleneck! Thus, we must take care when designing the system resource management scheme, or else it alone can severely constrain parallelism.

3.2 Previous Approaches

Let us now examine some previous work in the multiprocessor execution of reduction language programs. To date, the most execution-concurrent of these proposals fall into two categories: serial scanning and parallel scanning. All of these prior proposals use serial resource management techniques or have a major serializing bottleneck in their resource management.

3.2.1 Serial Scan / Serial Resource Management

Patel[20], and Treleaven and Mole[21] propose two similar multiprocessor reduction machines to execute functional programs. Fig. 3-6 diagrams Patel's machine,

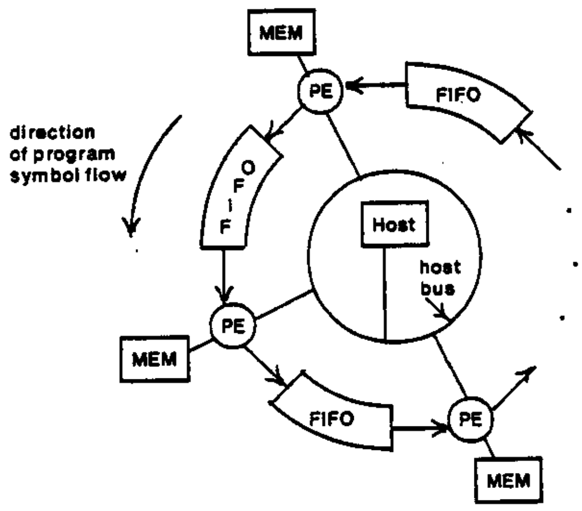


Fig. 3-6: Patel's machine

and Fig. 3-7 diagrams the machine of Treleaven and Mole.

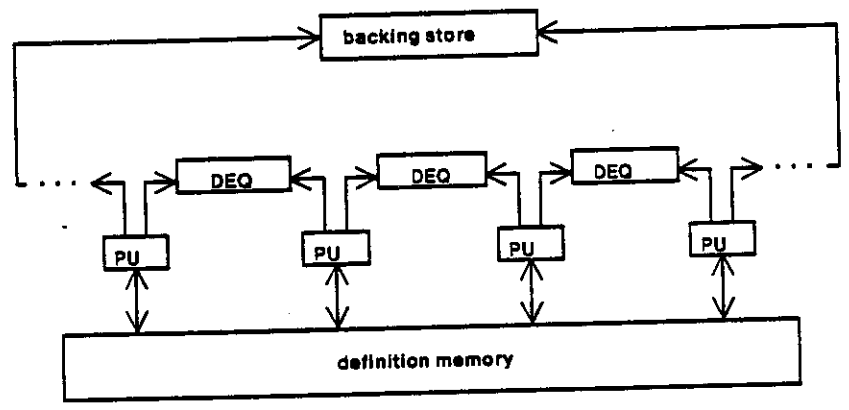


Fig. 3-7: The machine of Treleaven and Mole

Patel's machine consists of a number of identical processing elements arranged in a ring structure interconnected by queues (FIFOs). Each processing element has its own local memory and also connects to the common system-wide bus of a supervisory host. The functional program expression travels around the ring symbol-serially and each idle processor of the ring scans the symbol stream coming into it. When a processor finds it has scanned an RA it wants to execute it removes the RA from the ring and leaves a tagged placeholder to continue around the ring. This processor, after completing the execution of the RA, waits for the proper placeholder to come around again and replaces this placeholder with the RA's result. When any processor requires the definition of a defined function it puts the name of the function on the system bus and the host broadcasts the corresponding definition to all the processors that need it.

Similarly, the machine of Treleaven and Mole consists of a number of identical processing units arranged in a ring structure, but interconnected by double-ended queues (DEQs). Each processing unit contains some register memory and also connects to a system-wide definition memory. A backing store also resides on the ring between two DEQs to hold surplus parts of the expression. After an initial distribution of the functional program expression among the DEQs, each processor reads symbols from either its left or right DEQ and starts its serial scan for an RA. According to a prearranged distributed protocol, each processor might transfer symbols to its other DEQ, retain the symbols internally, or return them to its first DEQ and change its scanning direction. The distributed protocol has to prevent deadlocks and starvation while ensuring that processors will eventually locate RAs. Once a processor contains an RA, it executes it, outputs the result to one of its DEQs, and then continues scanning. Any processor needing the definition of a defined function gets it from the system-wide definition memory.

Since both of these reduction machines scan serially to find RAs, they both share the disadvantages we discussed in 3.1.2, because at least one processing element might have to scan the entire expression. Furthermore, since both machines require any of their processing units to hold an entire RA in its local memory before executing it, every processing unit must have enough memory to hold the largest possible RA (i.e., the entire initial expression if using our reduction language!). Because of this, Treleaven and Mole simplify the language of their machine to limit the maximum size of possible RAs. This destroys much of the expressive power of their reduction language by making impossible such useful primitive operators as NF, AA, TRANS, and REVERSE (as we defined in chapter 2).

Serial communication also takes its toll on the performance of these machines. In either of these two systems a processor always takes twice as long to output a result containing 2m symbols than to output an m-symbol result. And even then, this output goes first only to immediate neighbors. Furthermore, neither of the two designs mentions an efficient way for its system-wide central source to look up function definitions and still satisfy many simultaneous, yet different requests. To get rid of such system bottlenecks we need good parallel ways to communicate results and look up definitions.

Both of these serially scanning reduction machines also manage their resources in serial fashion. In Patel's machine, an RA travelling toward an idle processor for execution must pass though all the intervening busy processors on its way. Moreover, when an idle processor finally removes the RA from the symbol stream it must always wait through at least one complete revolution of the expression through the system so it can put the result back in the proper spot. The machine of Treleaven and Mole, on the other hand, forces any RAs within a DEQ to wait idly when the processors on both sides of that DEQ are busy. These idle RAs remain trapped inside the DEQ even if processors elsewhere in the system have nothing to do. When one of the busy processors finally finishes, only one of the trapped RAs leaves the DEQ to make that processor busy again, leaving the other RAs still trapped. Hence, both machines demonstrate how serial resource management serializes RA execution.

3.2.2 Parallel Scan / Serial Resource Management

Magó[19, 22] proposes a network of microprocessors to directly execute reduction languages. Magó's machine consists of a linearly-connected row of processors ("L-cells") which also sit as the leaves of a tree of processors ("T-cells"). Fig. 3-8 shows an instance of this machine with 8 L-cells.

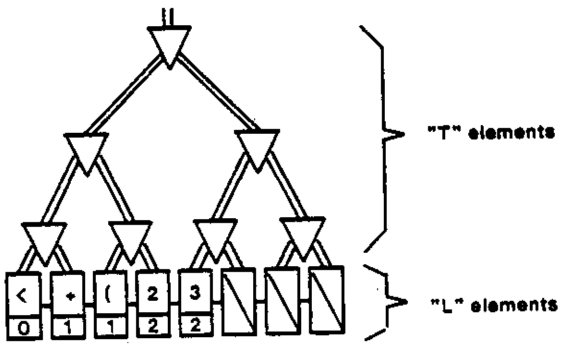


Fig. 3-8: Magó's machine

The functional program expression resides, one symbol per cell, in the L-cells (where a "symbol" is a name, a primitive value, or a bracket). Fig. 3-8 also shows how this machine efficiently represents the reduction language expressions using only the left brackets of the "< . . . >" and "(. . .)" constructors. (This suffices because each L-cell with a symbol also contains a level number.) In this reduction machine the T-cells locate and execute the RAs, while the L-cells perform the necessary storage management for the results. Notice how two links run between each connected pair of T-cells. Fig. 3-9 indicates how each T-cell actually contains four interconnected processors within it.

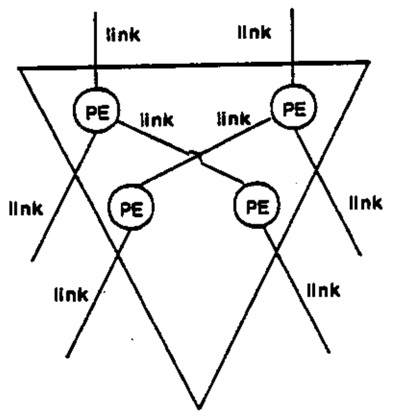


Fig. 3-9: Actual internals of a T-element in Magő's machine

Unlike the reduction machines we discussed in 3.2.1, Magó's machine scans for RAs in parallel. It accomplishes this feat by a process called *partitioning*: the processors within the T-cells connect their links in such a fashion that each RA in the L-cells sees its own private binary tree of T-cell processors above it. Fig. 3-10 shows an example of such a partitioned system.

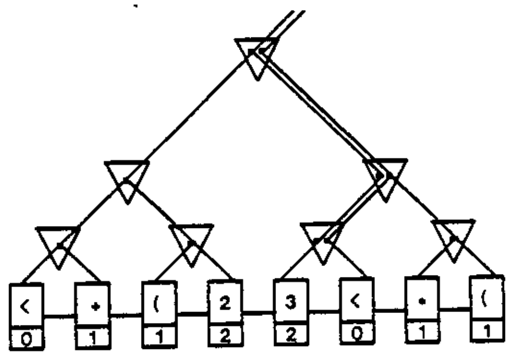
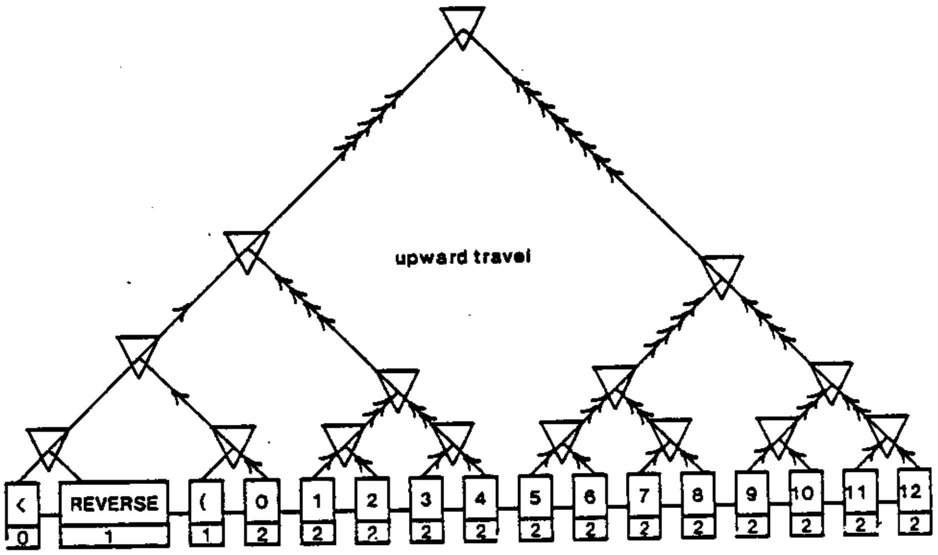


Fig. 3-10: Fragment of a partitioned Magő machine

Magó[19] gives a fast algorithm which completely partitions a system of N L-cells in only $2\lceil \lg N \rceil$ link times for any expression. At the end of its partitioning, Magó's machine has identified all RAs, and every L-cell in an RA knows the operator type of its RA. For long expressions this parallel scan easily outperforms serial scanning techniques.

Magó's machine also differs from our two previously discussed systems by always using many processors to store any RA, and many processors to execute any RA. No single processor ever needs to contain an entire RA. In fact, this machine's one-symbol-per-cell storage philosophy means that each L or T processor only needs a small number of registers for its internal memory. Communication between processors brings the necessary data together for computations. This communication and computation takes place among the T-cells, with results travelling back the L-cells. At any moment an active processor element holds only a small piece of an RA's data and performs only a small piece of that RA's computation. Collectively, however, the entire system executes all its RAs simultaneously, and each RA simultaneously involves many processor elements in its execution.

Executing even a single RA in Magó's machine can conceivably invoke many parallel activities. Some or all of this parallelism can go to waste, however, due to an inherent communications bottleneck in Magó's system. This bottleneck comes from the treelike connections of the T-cells. Many functions simply fail to map well to a tree-structured parallel communication scheme, and so some processors must wait while others communicate. Fig. 3-11 shows one simple example of this with worst-case performance. The operator REVERSE reverses the order of the subitems of its input. To execute REVERSE on a tree network requires all the subitems to travel up the tree and pass, one at a time, through its root. Reversing M subitems, then, takes at least O(M) time even though it uses O(M) processors.



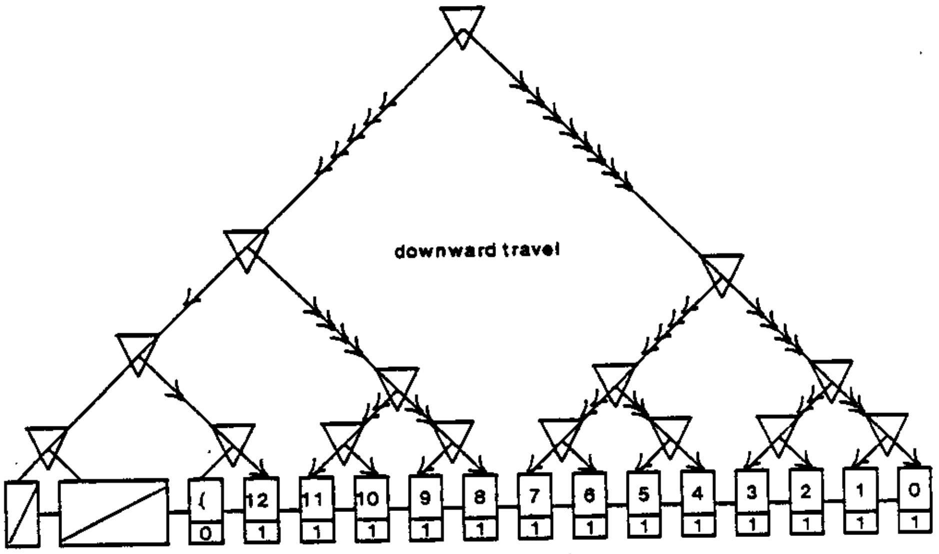


Fig. 3-11: How REVERSE causes a worst-case bottleneck in Magó's machine

Just like our two previously discussed systems, Magó's machine manages its resources serially. Whenever an RA needs more L-cells to hold its result, the symbols in the L-cells must move along their linearly linked row to accommodate the demand. Although Tolle[23] showed that Magó's machine could accomplish this activity without the linear L-cell links, eliminating these links would not make this activity any faster than Magó's. Fig. 3-12 shows an example of how Magó's resource management can constrain program execution. During successive reduction transitions, parts of our example program drastically change in the amount of storage they require. After the first transition, one subexpression has expanded and pushed the others over as it grabbed up more space. Next, it suddenly shrinks, leaving much empty space. After another transition, a different subexpression expands, pushing back the first subexpression and the one in between. Later, it too shrinks. Notice how the subexpression in the middle (the "1") gets pushed back and forth by the other two subexpressions. In Magó's machine, each such push takes up time proportional to the distance of the push. If a program pushes any subexpression a total appreciable distance along the L-cell row, that program could require O(N) time in an system with N L-cells. Thus, the performance of Magó's machine suffers from its serial resource management.

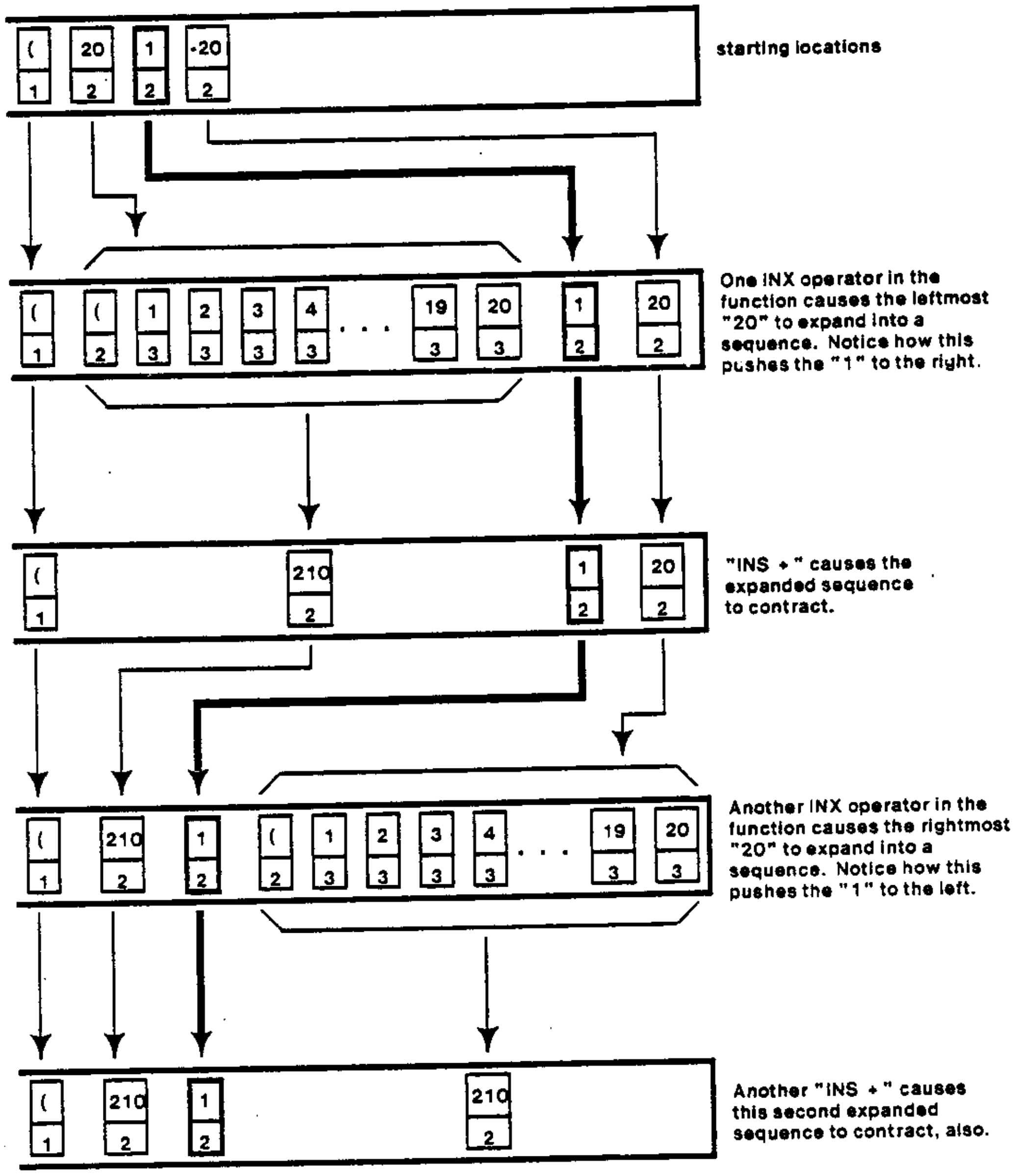
Consider the following function:

([HD,(HD,TL),(INS, + ,INX,HD,TL,TL)] , [(INS, + ,INX,HD),(HD,TL),(NEG,HD,TL,TL)])

Applied to the data object:

(20 , 1 , -20)

Watching the data cells during this computation:



(
1
20
2
1
2
-20
2
starting locations
19
3
20
3
. . .
One INX operator in the function causes the leftmost "20" to expand into a sequence. Notice how this pushes the "1" to the right.
210
"INS + " causes the expanded sequence to contract.
Another INX operator in the function causes the rightmost "20" to expand into a sequence. Notice how this pushes the "1" to the left.
Another "INS + " causes this second expanded sequence to contract, also.

Fig. 3-12: A resource management bottleneck in Magó's machine

3.3 Evaluation of Previous Approaches

Of the three proposals we have just looked at, Magó's machine executes with the most parallelism. Yet, as we saw above, it can easily run into situations which severely hinder its parallel execution, even for programs which it can successfully execute to completion. We still find some of its virtues attractive enough to want in our proposal, however. In particular, we like its following attributes:

ADV1) Parallel scan for RAs

ADV2) Simultaneous execution of all RAs

ADV3) Physical separation of independent activities

In our new proposal we would also like to eliminate the disadvantages of Magó's machine by having the following:

ADV4) Unconstrained parallel communication

ADV5) Parallel demand-allocation of resources

ADV6) Parallel compaction of idle resources

The next chapter proposes a reduction machine multiprocessor which has all six of these advantages and which, surprisingly enough, has an asymptotic hardware cost (in terms of parts count) almost the same as Magó's machine.

4. A New Design Approach

In this chapter we develop and propose a new reduction machine architecture. This new reduction machine has the advantages of Magó's machine, namely, the attributes ADV1, ADV2, and ADV3 we listed in the previous chapter, and also has attributes ADV4, ADV5, and ADV6 which eliminate the disadvantages of Magó's machine. Provided we have a sufficiently parallel sorting network and enough processors to prevent system overflow, this new machine will quickly execute functional reduction language programs as a power-log fast system with unconstrained parallelism.

4.1 Basis for the New Approach

Four observations form the basis for our new design approach. Taking care of all of these together helps us to design a reduction multiprocessor without the troublesome bottlenecks of previously proposed systems.

First of all, we must realize that strictly local multiprocessor activities are sufficient to execute many (but not all) parts of reduction language computations. This comes as a direct result of the tree-structured syntax we saw in 2.6.1. Nearest-neighbor communication among processors can certainly accomplish power-log fast tree reduction (recall Fig. 3-5) and also power-log fast scanning for RAs (as Magó's machine demonstrates). In fact, as long as a multiprocessor reduction machine can quickly form treelike communication structures, we can always put these structures to good use.

On the other hand, some nonglobal reduction activities require more than just nearest-neighbor communication for power-log fast execution. We already saw an example of this with REVERSE in 3.2.2. For another example, consider the meta operator NF (see Appendix A), which requires the system (among other things) to append a copy of the entire data object to each function of a sequence. Since the data object and these functions might initially require more than one processor to hold each of them, the resulting many-to-many communication pattern generally fits no single nearest-neighbor scheme. To avoid bottlenecks in such situations, we need a communications architecture which allows more general communication patterns than nearest-neighbor.

Moreover, two kinds of reduction machine actions require global activities: resource allocation and, if the system design requires it, resource compaction. Having only nearest-neighbor communication will generally slow down both these activities, or (as we saw with the machine of Treleaven and Mole in 3.2.1) even prevent them!

And finally, last but not least, both of the first two above activities (nearest-neighbor and non-nearest neighbor local communication) might have to occur in the system simultaneously. Our new architecture should therefore accomplish both of these in parallel.

4.2 Elements of the New Approach

Fig. 4-1 shows a conceptual diagram of our proposal for a parallel reduction machine.

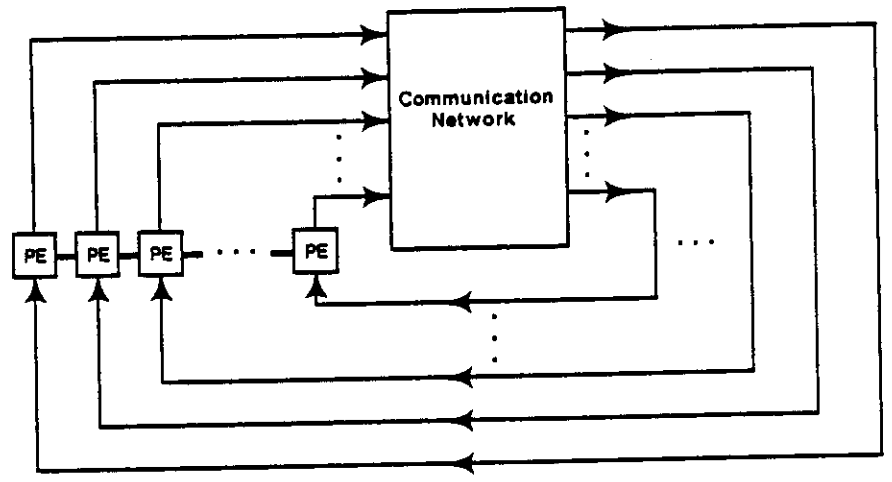


Fig. 4-1: Proposed system architecture

This diagram depicts a row of processing elements (PEs) with communication links linearly connecting them together, and with one input and one output link connecting each PE to a common communication network. At this high level of abstraction, nothing distinguishes our new machine from practically any other multiprocessor (including Magó's machine!). Let's take a closer look at each of the two basic elements of our design to appreciate its unique architecture.

4.2.1 Homogeneous Cellular Resources

A group of identical processing elements perform all the processing and storage for this computer system. These PEs operate independently and asynchronously from one another, co-ordinating their activities only when they communicate to their immediate left or right neighbors, or to the communication network. As we shall see in 4.3.2, the PEs need to communicate to all these places quite regularly. To avoid using any system-wide clock for synchronization, we assume a *self-timed* discipline[24] of co-ordination between all subcomponents of the system. Fig. 4-2 shows the internal components of a PE.

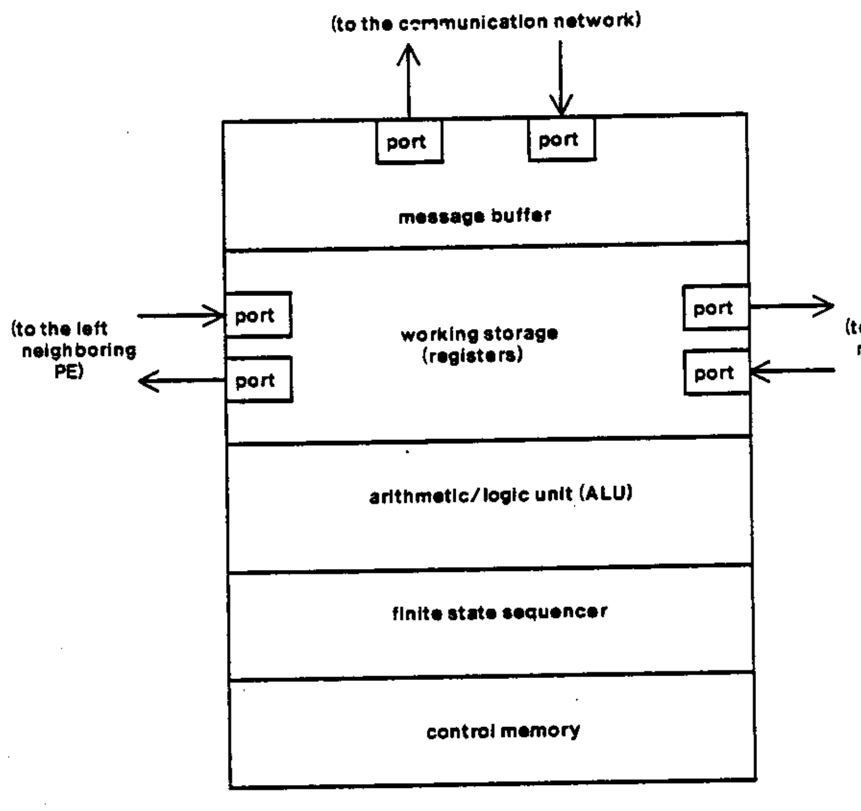

(to the communication network)
port
port
message buffer
(to the left neighboring PE)
port
port
working storage (registers)
port
port
(to the right neighboring PE)
arithmetic/logic unit (ALU)
finite state sequencer
control memory

Fig. 4-2: The Processing Element (PE)

It contains a finite state sequencer which controls an arithmetic/logic unit (ALU) and six communication ports. Three kinds of memory reside in the PE: a control memory for the sequencer (with perhaps a writable portion), an internal storage which contains registers used by the ALU, and a message buffer for communicating to and from the communication network. The pairs of communication ports to its left and right neighbors allow a PE's ALU to access its immediate neighbors' internal storage. The pair of ports to the communication network, on the other hand, must send and receive message packets which travel serially through the communication network. In addition to these six ports the system might also need an extra pair of ports in each PE (not shown in the figure) to communicate to the outside world. Because of the extreme application-dependence of this additional requirement, we discuss it no further in this report.

In our system, each PE has a small fixed number of internal registers (a couple dozen) regardless of the number of PEs in the system. For a system with N PEs, each of these registers need only contain $\lceil \lg N \rceil$ bits of storage (since $\lceil \lg N \rceil$ bits allow each PE in the system to have a distinct address). This implicitly establishes our system's word size. Any internal PE operation must store values which can fit into these registers (or small fixed-size groups of these registers). Doing so not only keeps the internal storage small, but also keeps the rest of the PE simple. This limit on internal storage has important consequences for data storage in the entire system: we might need to decompose into smaller pieces any program value which cannot fit into $\lceil \lg N \rceil$ bits and possibly store it across more than one PE. (Executing programs will then need the ability to deal with such symbolically distributed values, —but such is the case for any machine with fixed word size.) Within each PE, one hardwired register holds the $\lceil \lg N \rceil$-bit *physical address* (PA) which uniquely identifies that PE in the entire system. All the rest of the internal registers in a PE hold the tags, references, and data necessary to execute any of the primitive operations defined for that particular system.

Our multiprocessor architecture with its many interprocessor links makes possible an especially useful kind of data mobility. Because of the small O(lg N) number of bits in each PE, we could, if we had to, transport the entire contents of a PE's internal storage (except for its hardwired physical address) out of that PE, accomplishing this in O(lg N) time or less (depending on whether the data communication occurs bit-serially or in parallel). Travelling as a packet through the communication network, these PE contents could eventually arrive at another PE and then overwrite a new internal storage space. In this sense, the contents of a PE's internal registers (and the writable portion of its control memory) together comprise an abstract entity, —a virtual processor which can perform local computations, relocate from PE to PE, carry information, and independently maintain its own identity. We call this entity a *cell*.

In our machine, cells constitute the only resource we ever need to manage. Although cells can move between PEs, the total number of cells in the system at any moment always matches the total number of PEs. As a result of this, we can regard cells as the true agents of computation in this machine, with the PEs merely acting as part of the framework in which cells accomplish their activities. If we adopt this viewpoint, then every cell minimally possesses all the computational and communicative powers of a PE. In particular, a cell can:

CEL1) compute values from data it contains

CEL2) send and receive messages from its immediate left and right neighbors

CEL3) send and receive messages from other nonadjacent cells in the system

But unlike a PE, a cell can also:

CEL4) move to a new location relative to the other cells

As we shall see in the rest of this chapter, CEL4 makes possible a powerful high-speed communication scheme and a power-log fast resource management method.

4.2.2 Communication Network

The communication network connects the PEs to form the complete system framework in which cells perform their activities. All nonadjacent communication and all cell relocation occurs via this network. By properly designing the communication network we can make both communication and resource management power-log fast, simple, and parallel.

We now describe what kind of network fulfills these ambitious requirements. Fig. 4-3 shows an example of the parallel demand-allocation of resources through a network.

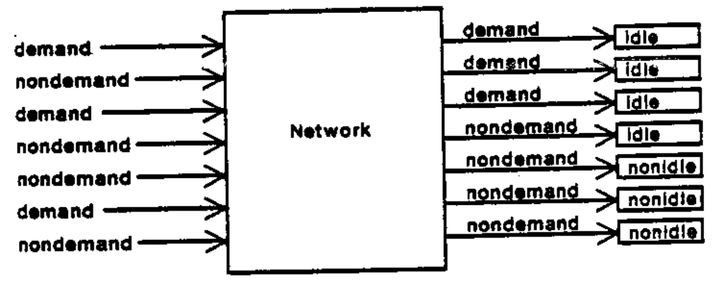


Fig. 4-3: Parallel demand-allocation of resources through a network

On the left side of the network, every input link carries a message containing either a *demand* for more resources, or a *nondemand*. The messages propagate through the network to the outputs on the right side in one-to-one fashion and the network permutes these messages so that the demands go to idle cells. When an idle cell receives a demand it also receives something to do and thereby becomes nonidle. Notice how our example requires all the idle cells to sit at one end of the system. Some process must have previously compacted all the idle cells. Fig. 4-4 shows such a parallel compaction of resources through a network.

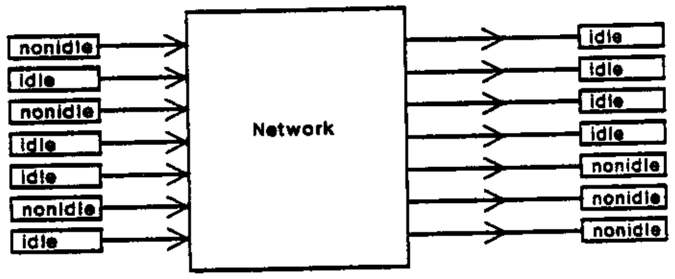


Fig. 4-4: Parallel compaction of resources through a network

In this case, all the cells of the system enter the network with each cell carrying a tag marking it either *nonidle*, or *idle*. During their propagation, the network permutes all the cells so that the idle cells form a single contiguous group at one end of the system when all the cells exit.

We want our network design to at least accomplish both parallel demand-allocation and parallel resource compaction. Fig. 4-5 shows how a relabeling of Fig. 4-3 or Fig. 4-4 demonstrates the equivalence of these two activities. If in Fig. 4-3 we call a demand "0" and a nondemand "1", or if in Fig. 4-4 we tag a nonidle cell "0" and an idle cell "1", we get in either case Fig. 4-5.

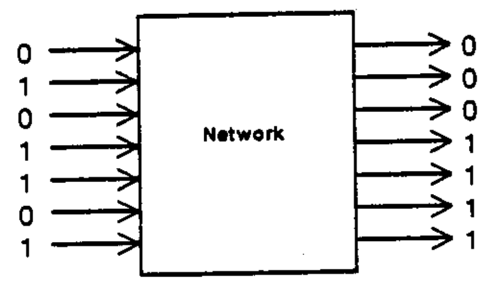


Fig. 4-5: Equivalent parallel activity through a network

In other words, to do parallel resource management our network requires only the parallel ability to segregate the ones and zeroes of any arbitrarily ordered input set of ones and zeroes. The well-known Zero-One Principle[25] tells us that any network which can do this can also do parallel sorting. Hence, our communication network must at least contain a parallel sorter.

In order to show that we need nothing more than a parallel sorter in our network, we must find ways to do all other system communication using this same sorter.

4.3 Basic Activities of the New Approach

4.3.1 Role of Cell Display Addresses

In our new approach, every cell of an N-cell system contains a *cell display address* (CDA) of $4\lceil \lg N \rceil + \lceil \lg \lg N \rceil + 3$ bits (which takes up no more than five internal registers in a PE). Each cell presents its own CDA to the communication network (as a packet header, perhaps) upon entering it and the network sorts all the cells of the system according to these CDAs. Thus, the bits of a cell's CDA determines that cell's position relative to all the other cells when it exits the network.

Fig. 4-6 diagrams the internal organization of CDAs for an N-cell system.

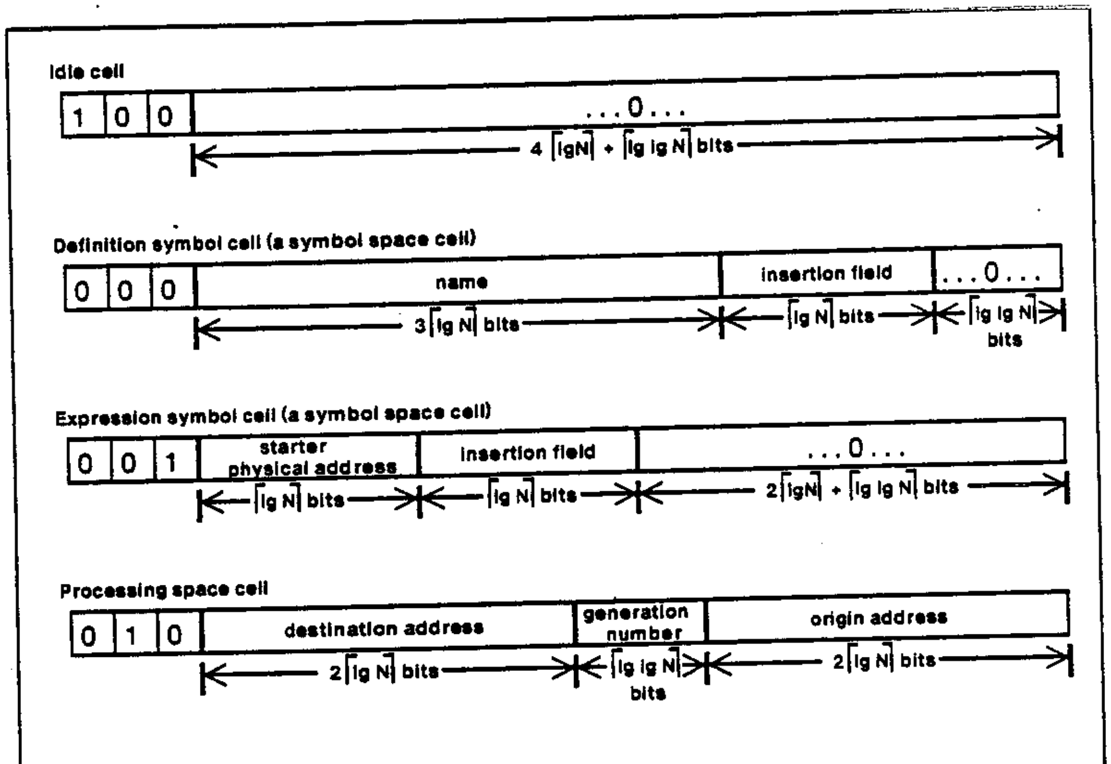

Idle cell
1 0 0 ...0...
4 ⌈lgN⌉ + ⌈lg lg N⌉ bits
Definition symbol cell (a symbol space cell)
0 0 0 name insertion field ...0...
3⌈lg N⌉ bits
⌈lg N⌉ bits
⌈lg lg N⌉ bits
Expression symbol cell (a symbol space cell)
0 0 1 starter physical address insertion field ...0...
⌈lg N⌉ bits
⌈lg N⌉ bits
2⌈lgN⌉ + ⌈lg lg N⌉ bits
Processing space cell
0 1 0 destination address generation number origin address
2⌈lg N⌉ bits
⌈lg lg N⌉ bits
2⌈lg N⌉ bits

Fig. 4-6: Organization of Cell Display Address (CDA)

The leftmost three bits of a CDA labels each cell according to its current activity in the system, and determines the meanings of the other subfields in that CDA. At any moment, our system might contain some *idle cells* and some *nonidle cells.* The system uses the leftmost bit to compact all the idle cells via sorting to one end of the PE row, making them available for reallocation. The group of nonidle cells contains two subgroups distinguishable by their second leftmost CDA bit: *processing space cells* which perform all the nonlocal communication and processing in the system, and *symbol space cells* which hold the symbols of the executing program. The subgroup of symbol space cells, in turn, consists of two further subgroups distinguished by their third leftmost CDA bit: *definition symbol cells* and *expression symbol cells.*

The use of leftmost CDA bits to distinguish all these cell types results in their physical separation into contiguous groups of the same type after each sorting. Fig. 4-7 shows the arrangement of these groups along the PE row after a typical sort.

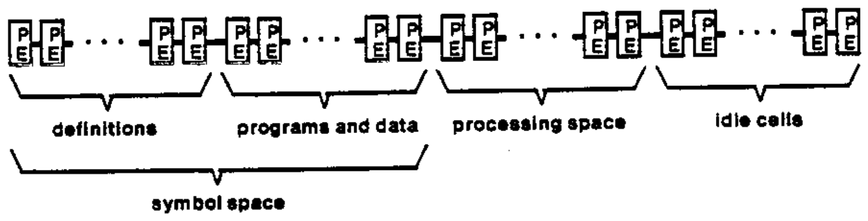


Fig. 4-7: PE row after a typical sort

Notice how the definitions always stay in the leftmost cells of the PE row. We find this convenient because we load the definitions into the system only once with each expression and the definitions remain unchanged throughout the expression reduction. Hence, an expression can safely refer to definition symbol cells without worrying about them changing their locations during a reduction sequence. The idle cells of the entire system always stay in the rightmost PEs, and thus remain available to any RAs needing them. The expression and the processing space sit next to each other and take up all the remaining PE space between the definitions and idle cells. Any series of reduction transitions which terminates always leaves an empty processing space and a result sitting in the expression symbol cells. We shall later see how the rest of each CDA's bits function when we look at the system activities which use them.

4.3.2 System Timing

Even if our system has no system-wide clock, its global sorting activity forces it to have global synchrony. We might easily imagine state changes to ripple through the system like waves, with one set completing before the next starts. Fig. 4-8 shows a timing diagram for our system.

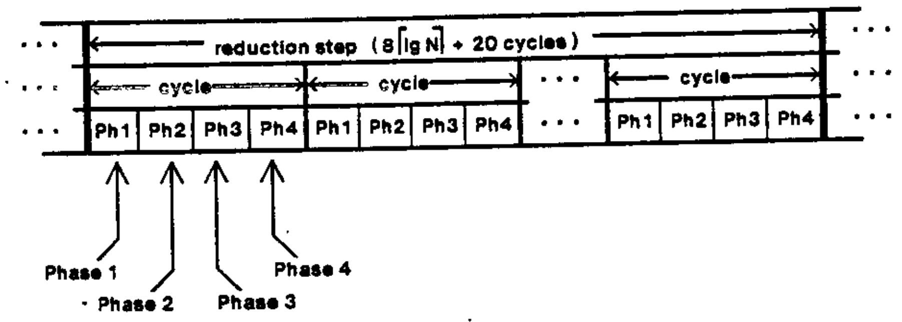


Fig. 4-8: Timing diagram for our system

The smallest time division, the *phase*, corresponds to either one pass of packets through the communication network (i.e., a global sort), or to one round of local internal PE activities which might also involve communication between immediate PE neighbors. We expect both of these alternatives to have similar asymptotic time behaviors: our parallel sort will require no less than O(lg N) time, and the local PE activities will require at most O(lg N) time. (For example, a local arithmetic operation on a couple of ⌈lg N⌉-bit internal PE values could execute in O(lg N) time by using a bit-serial incremental algorithm[26].) The system repeatedly executes the following four phases in the following order:

Phase 1: Internal and adjacent-neighbor cell activities with the cells stationary. (No sorting involved, but asymptotic time behavior no worse than a sort.)

Phase 2: All cells remain stationary but send tagged message packets through the sorter. Each message either requests an idle cell (tag bit = 1) or doesn't (tag bit = 0).

Phase 3: All cells reply to their senders through the sorter (since all received messages in Phase 2). Each reply acknowledges and either grants a request (if the cell was idle) or denies it (if the cell was not idle).

Phase 4: All cells travel through the sorter and redistribute themselves according to their new CDAs. If a previously idle cell becomes nonidle, either the symbol space or processing space *acquires* it. If some nonidle cell becomes idle, it moves to the right-end group of idle cell PEs.

We call one round of these four phases a *cycle*. Resource management, cell permutation, and some local processing happens once per cycle. A sufficient number of cycles can execute a *reduction step.* One reduction step corresponds to finding all the RAs in the system and executing a reduction transition on all of them in parallel. The global synchrony of our system design forces us to use the worst-case number of cycles for every reduction step, since any kind of RA might occur in the program expression at any time. The rest of this chapter presents an execution scheme for this report's example reduction language which requires at most $8\lceil \lg N \rceil + 20$ cycles per reduction step for a system with N cells.

Let us now examine three kinds of multicellular co-operation in this system which give it most of its parallel power.

4.3.3 Data Driven Building of a Tree from its Leaves

Our system repeatedly builds independent binary trees in parallel which start from some or all of its symbol space cells, use processing space cells for connecting nodes, and finally reach a root node which either completes some processing or communicates with a symbol space cell. Fig. 4-9 shows a binary tree diagram and our system's representation (in cells) of the same tree. (Rectangles denote symbol space cells and circles denote processing space cells.)

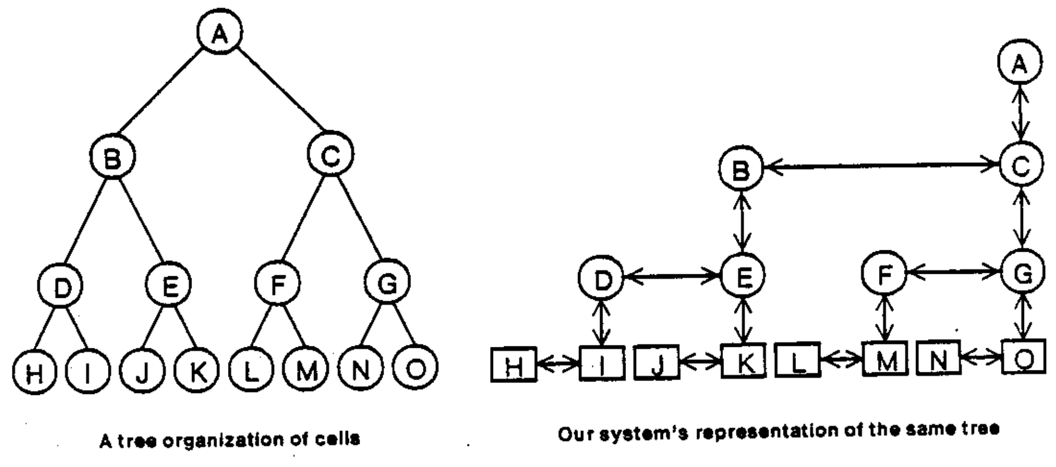

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
A tree organization of cells
Our system's representation of the same tree

Fig. 4-9: Tree structures in our system

Our representation closely resembles Knuth's "natural correspondence transformation" which can represent any tree as a multifurcating binary tree.[27]

Fig. 4-10 shows how our system can build such a binary tree from its leaves. In Phase 1 of the first cycle all the involved cells check both their immediate neighbors for involvement in the same tree. Two cells will become part of the same tree only if they have identical destination address fields in their CDAs (see Fig. 4-6). Assuming that some of them find such neighbors, each member of every *aligned pair* of involved cells remembers the other member, thereby establishing bidirectional pointers between them. Two cells of the same tree constitute an *aligned pair* if both of their $\lceil \lg N \rceil$-bit PAs (physical addresses) differ only in the rightmost bit. In the rest of the first cycle, the right-hand cell of every aligned pair and every leftover unpaired cell simultaneously requests and acquires an idle cell. These new cells become processing space cells and each of them forms its new CDA in the following way:

> The $2\lceil \lg N \rceil$-bit *origin address* field comes from the $\lceil \lg N \rceil$-bit *starter physical address* field and the $\lceil \lg N \rceil$-bit *insertion* field of the expression symbol cell that acquired this cell.
>
> The $\lceil \lg \lg N \rceil$-bit *generation number* contains the value $\lceil \lg N \rceil - 1$.
>
> The $2\lceil \lg N \rceil$-bit *destination address* field depends on the purpose of the tree. It consists of ones during a global parallel scan for RAs. Otherwise, the $2\lceil \lg N \rceil$-bit *destination address* comes from the $\lceil \lg N \rceil$-bit *starter physical address* and the $\lceil \lg N \rceil$-bit *insertion* field of some destination cell's CDA.

As a result of this CDA formation, all these new cells sit adjacently in preserved order in the processing space after phase 4 of the first cycle. The next cycle proceeds just like the first, but now only involves the new cells. After these combine into aligned pairs with cells containing the same generation number and destination address, the right-hand cell of each pair and each leftover unpaired cell requests and acquires an idle cell. These new cells join the processing space, each inheriting the origin and destination addresses of the cell which brought it in, but now their CDAs all have a generation number one less than the new cells of the previous cycle. Successive cycles repeat this process until a level in the tree contains only one cell, —the root of the tree. Even with worst case misalignment in every tree level, this entire tree building process requires no more than $\lceil \lg N \rceil + 2$ cycles. Notice how our CDA formation scheme ensures the segregation of different trees (by using different destination address values) and of different levels in the same tree (by using different generation numbers). The system can therefore perform

Participating symbol space cells combining into aligned pairs:

Acquiring one processing space cell per pair:

Because these processing space cells are all immediately adjacent, they too can combine into aligned pairs with processing space cells of their own generation:

These new aligned pairs, in turn, acquire one next-generation processing space cell per pair:

This process repeats until it reaches a generation which contains only one processing space cell, -- the root of the tree:

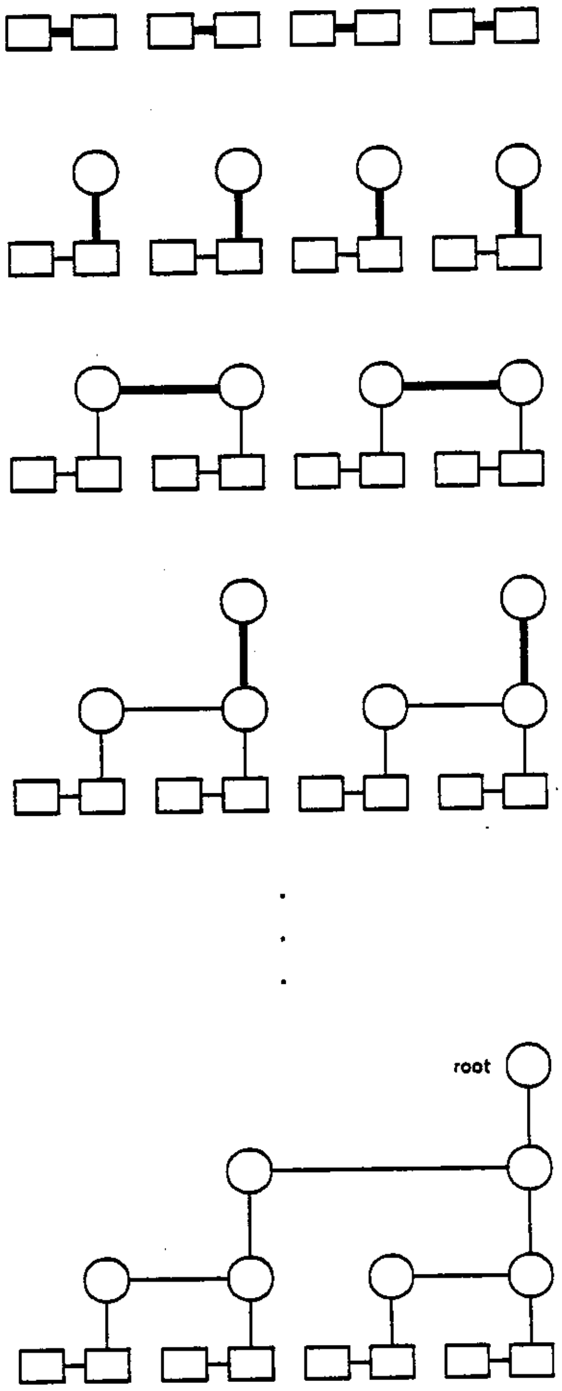


Fig. 4-10: Building a binary tree from its leaves

many independent tree building activities in parallel.

We have two important uses for this kind of tree building. First, it helps us to do a parallel scan for all the RAs in the expression. (We describe this in section 4.4.2.) Second, it makes possible a power-log fast, flexible, and parallel messaging scheme to propagate demands for information in the system. For example, suppose one or more cells throughout the system all want to get the same item of information (a definition or symbol value, say) from a single destination cell whose CDA they all know. By building a tree in the way we discussed above, we can concentrate all their demands until we have a single cell (the root) which acts as an agent on all their behalf. This root then temporarily changes its own CDA to match the CDA of the destination cell. After one additional cycle, the root would then find itself adjacent to its destination cell and able to communicate with it. We now describe the activity whereby the root would propagate the demanded information back to all the requestors.

4.3.4 Data Propagation Through an Already-built Tree

Once our system builds a tree in the way we discussed in 4.3.3, it might want to propagate information upward or downward through it. It can always do this, because the tree inherently contains all of its connectivity information in the addresses of its processing space nodes. The cells of the tree need only a method to make use of this information to pass their messages.

Let's suppose that the processing space root of a tree has received a datum from somewhere and wants to send it to all the leaves. It can calculate the CDA of the tree cell one level below it by adding one to the generation number of its own CDA. As long as this value stays less than or equal to $\lceil \lg N \rceil - 1$, the root knows that it has a tree cell below it to send to. By temporarily changing its own CDA to match the CDA of the cell it wants to send to, the root will find itself next to this cell in the next cycle and able to communicate with it. After this communication the root might restore its old CDA, or it might become an idle cell if we have no further need for it. The cell now containing the datum passes a copy to the other member of its aligned pair (if it exists) and then both of these cells simultaneously repeat the same steps that the root did before them. In this fashion, the root *replicates* the datum and propagates it to all the leaves, never requiring more than $\lceil \lg N \rceil + 2$ cycles total time.

If we need to propagate data in the opposite direction through the tree (i.e., from the leaves to the root), a scheme equivalent to reversing the above one would work. We could, however, also accomplish this same data movement during tree building time if we have all the data available early enough. Note that in either case, leaf-to-root propagation always requires data *concentration* instead of replication, —each tree cell receiving two data items from below should process them in some fashion and only pass upward a single datum. To need to do otherwise would cause a traffic jam in the upper levels of the tree.

4.3.5 Mitotic Insertions of Cells into Expressions

Frequently, our system also needs to insert new cells into specific locations in expressions. For example, an RA might increase in size after a reduction transition, requiring more cells to hold its symbols. To accomplish this in a power-log fast and parallel fashion we use a technique called *mitotic insertion*. Like its name suggests, it closely resembles a cell-replication technique found in nature (mitosis), except that our version uses cell acquisition instead of cell division to double the number of cells.

Fig. 4-11 shows how mitotic insertion works. An expression symbol cell, called the *starter*, decides how many cells to insert and requests an idle cell. The new cell becomes a first generation symbol cell and gets a CDA which places it between the starter and the starter's original neighbor. During the next cycle, the starter and the first generation cell both request idle cells for the second generation. The cell requested by the starter gets a CDA placing it between the starter and the first generation cell. The other new cell gets a CDA which places it between the first generation cell and the starter's original neighbor. Continuing likewise, each successive generation fits in between the cells already inserted in earlier generations. By passing to each new generation the generation number and the total number of cells we wanted to insert, each new cell can decide whether to continue requesting cells or to stop. A sequence of contiguous mitotic insertions completes after no more than $\lceil \lg N \rceil$ cycles (since the entire system only has N cells). Verifying that mitotic insertion did not run out of resources before completion requires

At the beginning of each new reduction transition, the CDA of every expression symbol cell resets to its own physical address (PA) followed by zeroes:

The starter cell acquires an idle cell and puts it into the symbol space to become a first generation cell, with a CDA derived from the starter's own CDA by appending one nonzero bit in its insertion field:

Both the starter and the first generation cell each acquires a new idle cell and puts it into the symbol space to become a second generation cell, each with a CDA derived from the CDA of its corresponding acquirer by appending one nonzero bit in its insertion field:

This process continues through enough generations to insert the proper number of new symbol cells into the expression. Notice how all the new addresses place the new cells between the starter and the starter's original right neighbor!

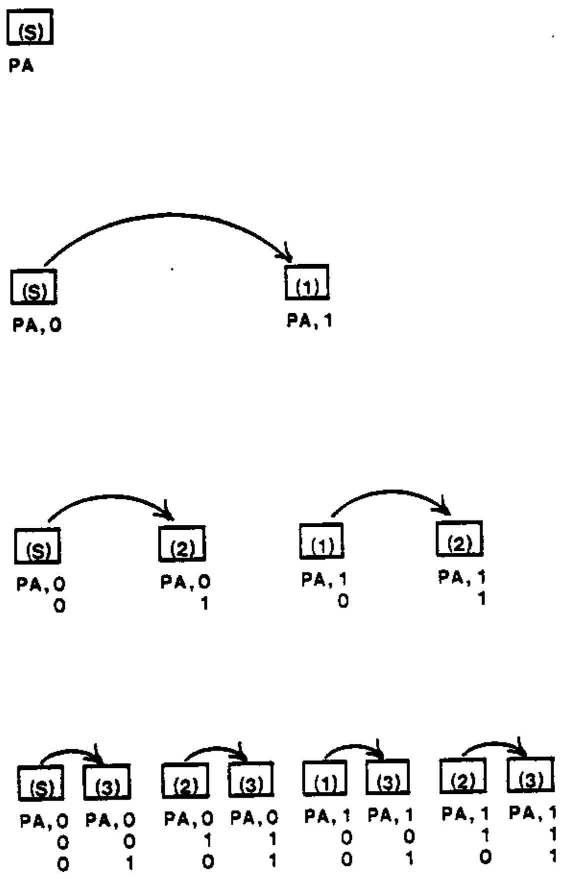


Fig. 4-11: Mitotic insertion of cells into expressions

no more than $\lceil \lg N \rceil$ additional cycles so that all the inserted cells can reply back to their parents in the reverse order of their acquisition. Hence, a contiguous sequence of mitotic insertions never needs more than $2\lceil \lg N \rceil$ cycles total.

Fig. 4-11 also shows the addressing scheme which allows mitotic insertion to give the proper kinds of CDAs to new generations of inserted cells. The starter must always have a CDA made up from a copy of its hardwired physical address in its *starter physical address* field and an *insertion* field of all zeroes. The first generation cell has the same CDA as the starter but with "100...0" in the insertion field. The second generation cell acquired by the same starter has the same CDA but with "010...0" in the insertion field. Likewise, the third generation has "001...0" appended to the starter's physical address, and so on. Similar rules hold for all the cells acquired by later generation cells: take the CDA up to its last nonzero bit and append "100...0" to it, then "010...0", then "001...0", and so on. In the mth generation of this process the m leftmost bits of all these cells' insertion fields will index them in numerical order, as our example in Fig. 4-11 shows.

Note how our address scheme for mitotic insertion insists that the starter always have an insertion field of all zeroes. Since any expression symbol cell might be a future starter, after one sequence of contiguous mitotic insertions ends we want to reset the CDA of every expression symbol cell to its own physical address plus zeroes before starting any others. This resetting enables us to do any number of mitotic insertions while still retaining finite length CDAs. Section 4.4 shows us that any single reduction step requires at most one sequence of contiguous mitotic insertions. We therefore choose to reset all expression symbol cell CDAs right at the start of each new reduction step.

4.4 Executing Reduction Steps

We now examine how our system can execute reduction language programs in a power-log fast and parallel manner. In what follows, we look at how to store expressions, find RAs, execute RAs, and overwrite RAs with their results. To keep implementation options flexible, our description avoids a lot of hardware-specific details and gives us instead a specification level presentation of a set of algorithms which can do the job.

4.4.1 Storage of Expressions

Our system stores the entire functional program in the symbol space cells, one symbol per cell. An expression (or even more than one expression) resides in the expression symbol cells and the definitions reside in the definition symbol cells. Both use the preorder-with-level-number format we described in 2.6.1. This format lets us do without right brackets and commas since every symbol space cell holds one symbol and its corresponding level number.

4.4.2 Location of Reducible Applications

Every reduction step starts with resetting all the expression symbol cell CDAs (see 4.3.5), followed by the parallel location of all RAs. Because the parallel algorithm for this requires the simultaneous co-operation of all the expression symbols cells in the system, our system uses a synchronous global activity to find RAs. Only after finding RAs can we execute them.

The preorder-with-level-number format makes it easy to characterize the RAs of an expression. As Magó[19] points out, an application bracket ("<") with level number i forms the left end of an RA if:

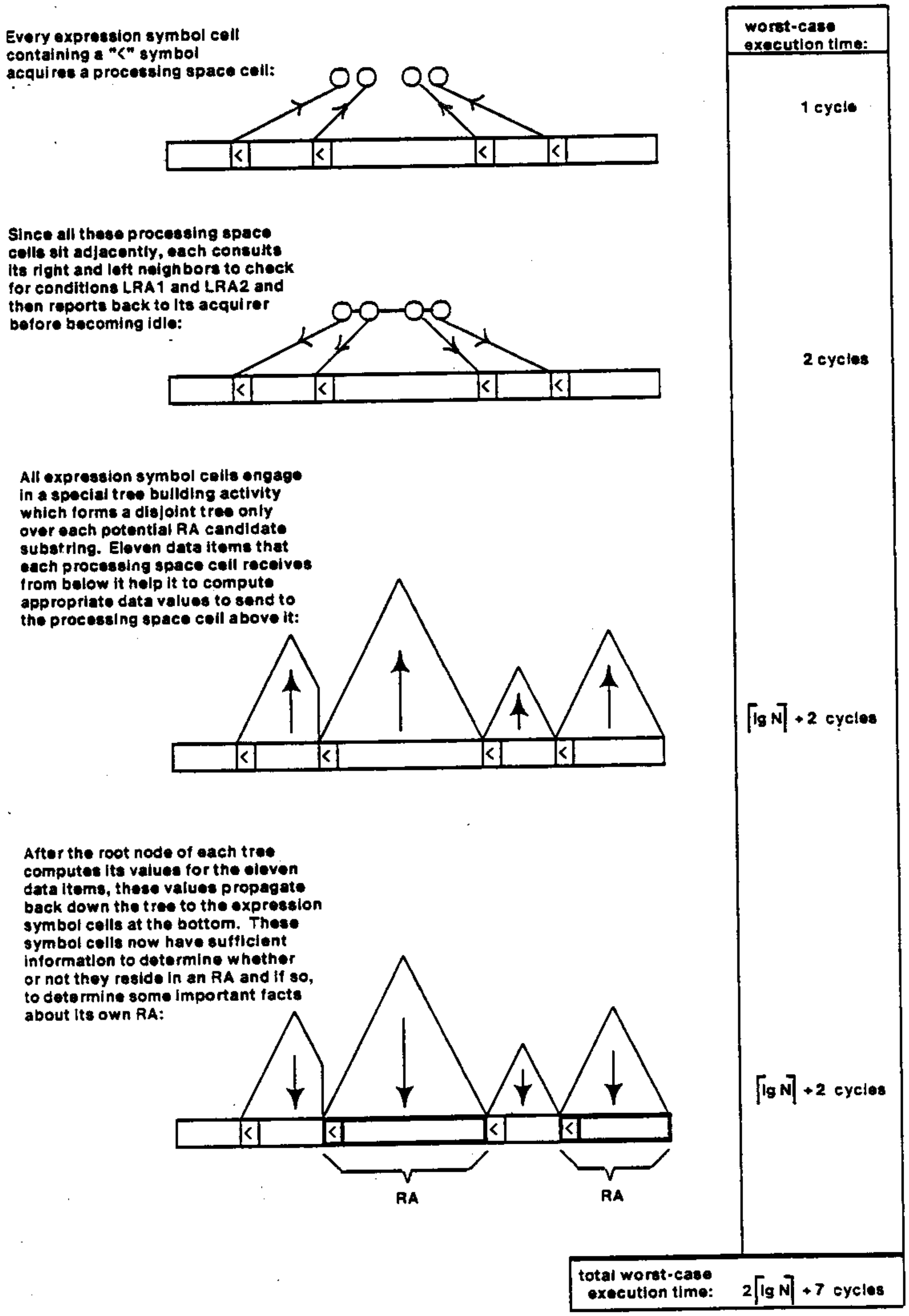
LRA1) the expression has no more application brackets to its right, or

LRA2) the next application bracket to its right has a level number $\leq$ i, or

LRA3) there exists a symbol between the application bracket and the next application symbol to its right with level number $\leq$ i.

Furthermore, if an application symbol with level number i forms the left end of an RA, then the entire RA consists of the application symbol itself and all the contiguous symbols to its right with level numbers > i.

Fig. 4-12 shows our scheme for RA-location (which is much simpler than Magó's scheme). In the first cycle of this process every expression symbol cell containing a "<" symbol requests and acquires a processing space cell. The CDA of each of these new processing space cells contains the PA (physical address) of its requesting <-cell in its origin address field, the value



Every expression symbol cell containing a "<" symbol acquires a processing space cell:

Since all these processing space cells sit adjacently, each consults its right and left neighbors to check for conditions LRA1 and LRA2 and then reports back to its acquirer before becoming idle:

All expression symbol cells engage in a special tree building activity which forms a disjoint tree only over each potential RA candidate substring. Eleven data items that each processing space cell receives from below it help it to compute appropriate data values to send to the processing space cell above it:

After the root node of each tree computes its values for the eleven data items, these values propagate back down the tree to the expression symbol cells at the bottom. These symbol cells now have sufficient information to determine whether or not they reside in an RA and if so, to determine some important facts about its own RA:

worst-case execution time:
1 cycle
2 cycles
$\lceil \lg N \rceil + 2$ cycles
$\lceil \lg N \rceil + 2$ cycles

total worst-case execution time: $2\lceil \lg N \rceil + 7$ cycles

Fig. 4-12: Location of Reducible Applications in our system

$\lceil \lg N \rceil - 1$ in its generation number field, and ones in its destination address field. Hence, by the end of this first cycle all these new processing space cells sit adjacently together in the same order as the symbol space cells which acquired them.

In the second cycle, all these new processing space cells consult their immediate right and left neighbors to check for conditions LRA1 and LRA2. By the end of the second cycle, each of these processing space cells temporarily changes its own CDA to the CDA of the <-cell it must report back to, and thereby moves adjacent to that <-cell.

In the third cycle, each <-cell finds out whether LRA1 or LRA2 holds for it and all the former processing space cells become idle.

Starting in the fourth cycle, all the expression symbol cells start a special tree building activity which continues for at most $\lceil \lg N \rceil + 2$ more cycles. All the processing space cells involved in this activity possess normal tree building CDAs except they have ones in their destination address fields. During this special tree building, every tree cell sends the following eleven data items to the cell immediately above it:

LLN :	the level number of the leftmost symbol sitting below this tree cell.
LCA :	the PA of the leftmost symbol sitting below this tree cell.
LRA :	a tag which signals "yes" only if the leftmost symbol sitting below this tree cell affirmed either LRA1 or LRA2 during the first three cycles of this reduction step.
W :	the leftmost symbol below this tree cell.
X :	the second leftmost symbol below this tree cell, if known. (We tag X "unknown", otherwise.)
Y :	the third leftmost symbol below this tree cell, if known. (We tag Y "unknown", otherwise.)
Z :	the fourth leftmost symbol below this tree cell, if known. (We tag Z "unknown", otherwise.)
TLN :	the level number of the leftmost symbol below this tree cell, exclusive of the one for which we already know LLN, with a level number not greater than LLN. (We tag TLN "unknown" if no such symbol exists.)
TCA :	the PA of the leftmost symbol below this tree cell, exclusive of the one for which we already know LCA, with a level number not greater than LLN. (We tag TCA "unknown" if no such symbol exists.)
RCA :	the PA of the rightmost symbol sitting below this tree cell.

REND : a tag which signals "yes" only if the rightmost symbol sitting below this tree cell has no symbol space cell to its immediate right with LLN > 0, or no symbol space cell at all to its immediate right.

Together, all eleven of these require no more than eleven ⌈lg N⌉-bit registers of internal storage in a PE. Every tree cell receives these corresponding eleven values from each of the one or two tree cells immediately below it. By simple comparisons it chooses its own appropriate values for the same eleven items and sends these, in turn, to the tree cell above it. This special tree building activity also has one additional property: a tree cell never associates with its aligned-pair partner to its right if W="<" or LLN=0 for that partner cell. In this way, a disjoint tree forms over each potential RA candidate substring (where each such substring starts with "<" or a level zero symbol and contains no other "<" symbol within itself).

When any of these disjoint trees finally has a root node, that root node computes its own appropriate values for the eleven data items mentioned above and then immediately sends its version of these back down the tree to all the symbol cells at the bottom. After no more than ⌈lg N⌉+2 additional cycles, every one of these disjoint trees finishes this downward propagation and all their processing space cells become idle. At this point, every expression symbol cell has received enough information from its root to figure out the following:

<u>Whether or not this symbol's expression has completed all possible reduction:</u> If the data items from its tree root have LLN=0, W≠"<", and REND="yes", then the system has completely reduced the expression in which this symbol resides, otherwise it hasn't.

<u>Whether or not this symbol resides within an RA:</u> Either of the following two conditions assures that this symbol resides within an RA:

1) The data items from its tree root have LRA="yes" and a TCA, which if known, is not less than this symbol's PA, or
2) The data items from its tree root have W="<" and a definitely known TCA which is not less than this symbol's PA.

If this symbol fulfills neither of these conditions, then it does not reside within an RA.

<u>The level number and location of the leftmost symbol of this symbol's RA:</u> LLN and LCA from its tree root provide these respective values.

<u>The location of the rightmost symbol properly belonging to this symbol's RA:</u> TCA from its tree root, if known, gives us this value, otherwise RCA from its tree root gives it.

The type of operator in this symbol's RA: Fig. 4-13 shows how data items X, Y, and Z from its tree root tell us whether this RA has a simple or composite operator, whether or not its operator requires definition expansion during this reduction step, and whether its operator involves primitive microcode execution, regular composition, or meta composition.

The process of locating RAs completes no more than 2⌈lg N⌉+7 cycles after the beginning of the current reduction step.

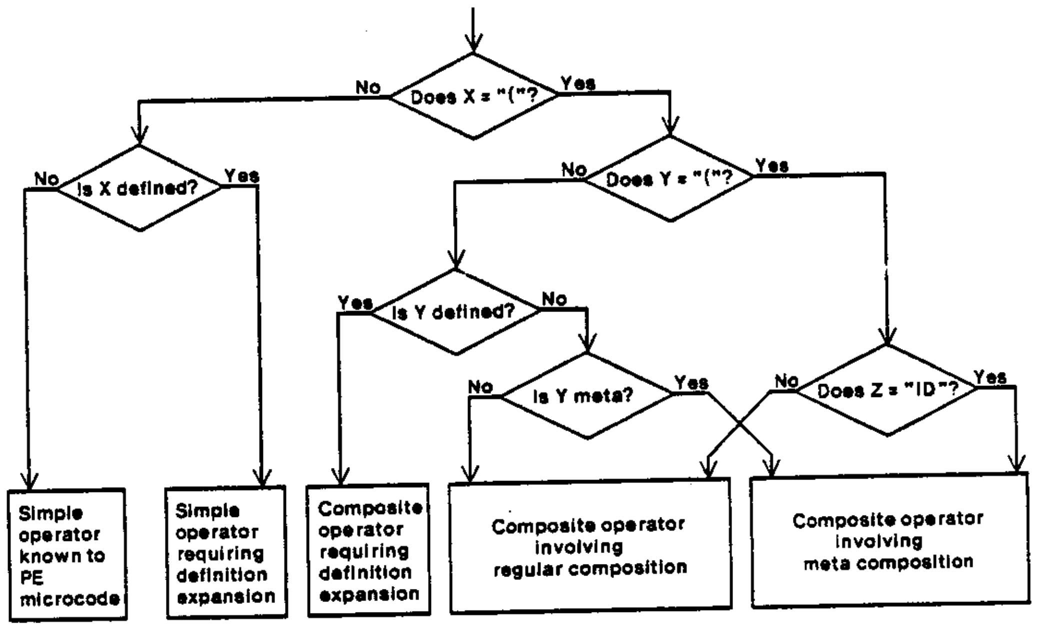

Does X = "("?
No
Yes
Is X defined?
No
Yes
Does Y = "("?
No
Yes
Is Y defined?
Yes
No
Is Y meta?
No
Yes
Does Z = "ID"?
No
Yes
Simple operator known to PE microcode
Simple operator requiring definition expansion
Composite operator requiring definition expansion
Composite operator involving regular composition
Composite operator involving meta composition

Fig. 4-13: Determining operator type for a Reducible Application

4.4.3 Ranking Activities

The information each expression symbol cell gains about its RA partially prepares it to begin participating in the execution of that RA. However, an expression symbol cell still needs additional information about its own relationship to its RA. An element of a sequence very often needs to know the total number of elements in that sequence and its own ordinal position in it. Let's look at a way to accomplish this even if that sequence has nonatomic elements.

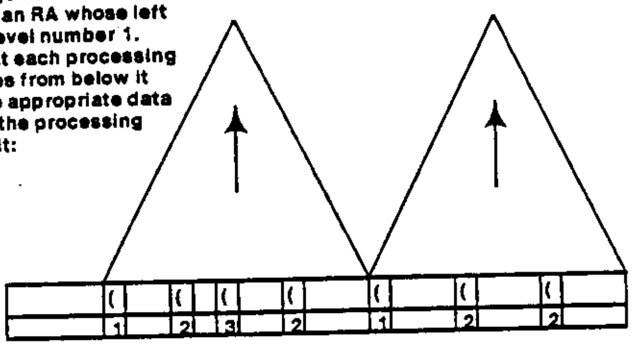
Fig. 4-14 shows how the cells of an RA perform *<+2 ranking.* In general, if the leftmost "<" symbol of an RA has level number p, then *<+q ranking* enables every expression symbol cell of that RA to find out the size of, and its element's ordinal position in the subexpression containing it whose left parenthesis has level number p+q − 1 (if such a subexpression exists). Our example in Fig. 4-14 for <+2 ranking shows how such ranking activities bear a strong resemblance to RA-location.

<+2 ranking starts with a special tree building action by all the expression symbol cells of every RA, taking at most ⌈lg N⌉+2 cycles. All the new processing space cells of these trees have normal tree-building CDAs, with their destination address fields containing the PA of their RA's leftmost expression cell. Thus, the tree cell CDAs look as if all the symbol cells of each RA want to simultaneously send a message to its leftmost cell (although they never actually do so). Actually, we use this kind of CDA-formation to segregate one RA's ranking activity from any other RA's concurrent activities. During this tree-building, every participating tree cell sends the following six data items to the cell immediately above it:

PLN :	the level number of the leftmost symbol sitting below this tree cell.
PCA :	the PA of the leftmost symbol sitting below this tree cell.
BP2 :	the level number of the leftmost symbol of this RA plus 2. (For <+q ranking we would use this RA's leftmost level number plus q and call it BP*q*.)
ECT :	the number of symbols below this tree cell with level number exactly equal to BP2.
ECA :	the PA of the rightmost symbol sitting below this tree cell with level number exactly equal to BP2.
SCA :	the PA of the rightmost symbol sitting below this tree cell.

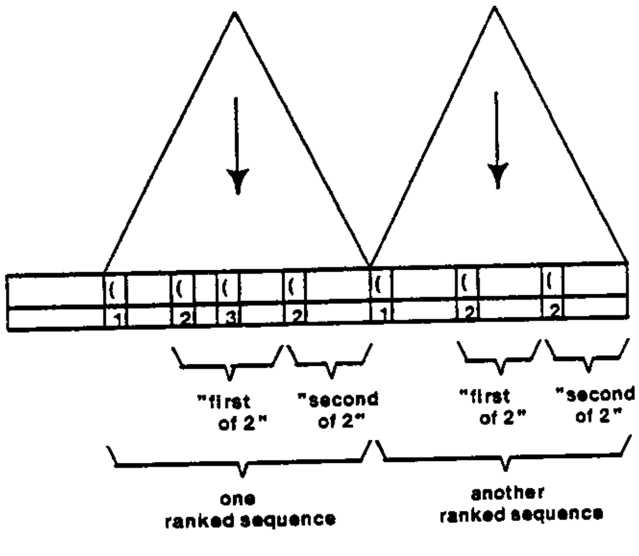
Together, all six of these require no more than six ⌈lg N⌉-bit registers of internal storage in a worst-case execution time:

All expression symbol cells engage in a special tree building activity which forms a disjoint tree over each sequence in an RA whose left parenthesis has level number 1. Six data items that each processing space cell receives from below it help it to compute appropriate data values to send to the processing space cell above it:



$\lceil \lg N \rceil + 2$ cycles

After the root node of each tree computes its values for the six data items, these values propagate back down the tree to the expression symbol cells at the bottom. In addition to this, two more computed values, LENO and HENO, also arise in each processing space cell for each cell below it. When each symbol cell at the bottom receives all eight data items from the processing space cell immediately above it, it then has all the information about its subexpression's rank in the level 1 sequence containing it:



$\lceil \lg N \rceil + 2$ cycles

total worst-case execution time: $2\lceil \lg N \rceil + 4$ cycles

Fig. 4-14: $< + 2$ Ranking

PE. Every tree cell receives these corresponding six values from each of the one or two tree cells immediately below it. By simple comparisons and an arithmetic addition it computes its own appropriate values for the same six values, sends these new values to tree cell above it, and stores in itself a copy of the ECT values it received from both tree cells immediately below it. (If one of these tree cells doesn't exist below it, it stores zero for that corresponding ECT value.) This special tree building activity also has an additional property: a tree cell never associates with its aligned-pair partner to its right if PLN < BP2 for that partner cell. In this way, a nondegenerate disjoint tree forms only over each sequence in the RA whose left parenthesis has level number BP2 - 1. These correspond to all the sequences whose elements we want to rank.

When any of these disjoint trees finally has a root node, that root node computes it own appropriate values for the six data items mentioned above and then immediately sends its version of these back down the tree along with two additional values, LENO and HENO, where LENO = 1 and HENO = ECT for the root. During downward propagation, every tree cell propagates a copy of the root's six data items to the tree cells immediately below it, and computes new values of LENO and HENO for each of them based on the old ECT values they previously sent upward. Fig. 4-15 shows how to do this calculation.

After no more than $\lceil \lg N \rceil + 2$ additional cycles, every one of the disjoint trees finishes this downward tree activity and all their processing space cells become idle. At this point, every expression symbol cell of the RA has enough information to figure out the following:

<u>Whether or not this symbol participates in this <+2 ranking:</u> If this symbol has a level number < BP2 then it doesn't, otherwise it does.

<u>The total number of elements (note: not necessarily symbols) in the ranked sequence containing this symbol:</u> ECT from its tree root gives us this value.

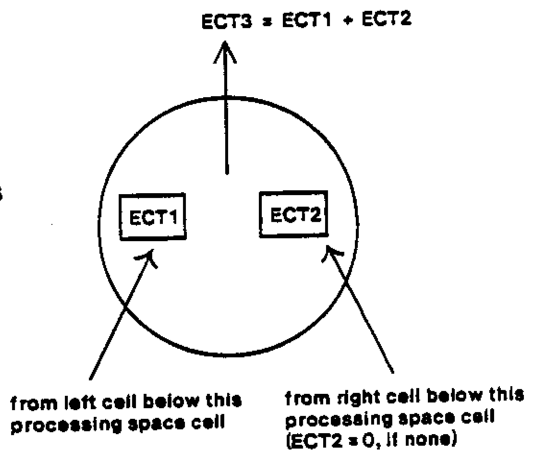
<u>The ordinal position of the element containing this symbol in this sequence:</u> This symbol's LENO (which should equal its HENO) gives us this value.

<u>The level number and location of the leftmost symbol of this sequence:</u> PLN and PCA from its tree root provide these respective values.

<u>The location of the leftmost symbol of the rightmost element in this sequence:</u> ECA from its tree root gives us this value.

<u>The location of the rightmost symbol properly belonging to this sequence:</u> SCA from its tree root gives us this value.

Storage of ECTs during upward data movement:


ECT3 = ECT1 + ECT2
ECT1
ECT2
from left cell below this processing space cell
from right cell below this processing space cell (ECT2 = 0, if none)

Calculations of LENOs and HENOs during downward data movement:

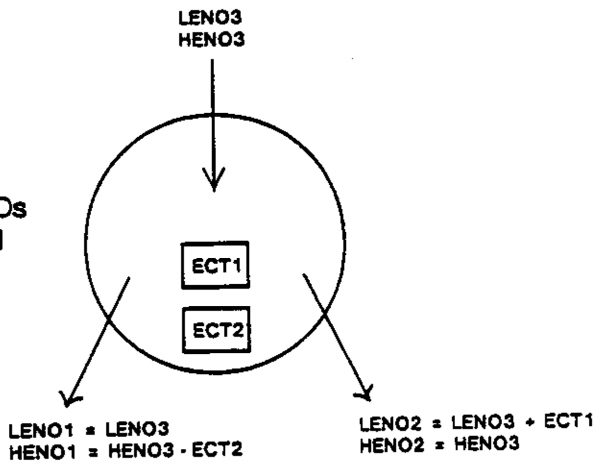


Fig. 4-15: Rank calculations during <+2 ranking

(For <+2 ranking only) The location of the operator/operand boundary in this symbol's RA: If the PCA from its tree root has the value LCA+1 (recall that LCA came from the RA-location activity), then the tree root's SCA gives us the location of the rightmost operator symbol, otherwise if the SCA from its tree root points to the rightmost symbol of the RA, then the tree root's PCA gives us the location of the leftmost operand symbol.

Hence, a ranking process (<+2, or otherwise) completes no more than 2⌈lg N⌉+4 cycles after it starts.

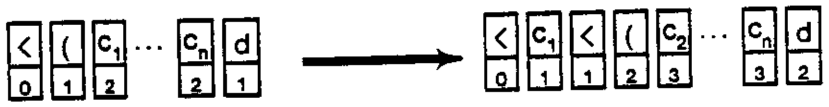
4.4.4 Executing Reducible Applications

After a reduction step performs RA-location the next action by each RA depends on the operator part of that RA. Depending on whether the operator part contains regular composition or meta composition, or whether it invokes the direct application of a user-defined function or a primitive function, one of several activities take place.

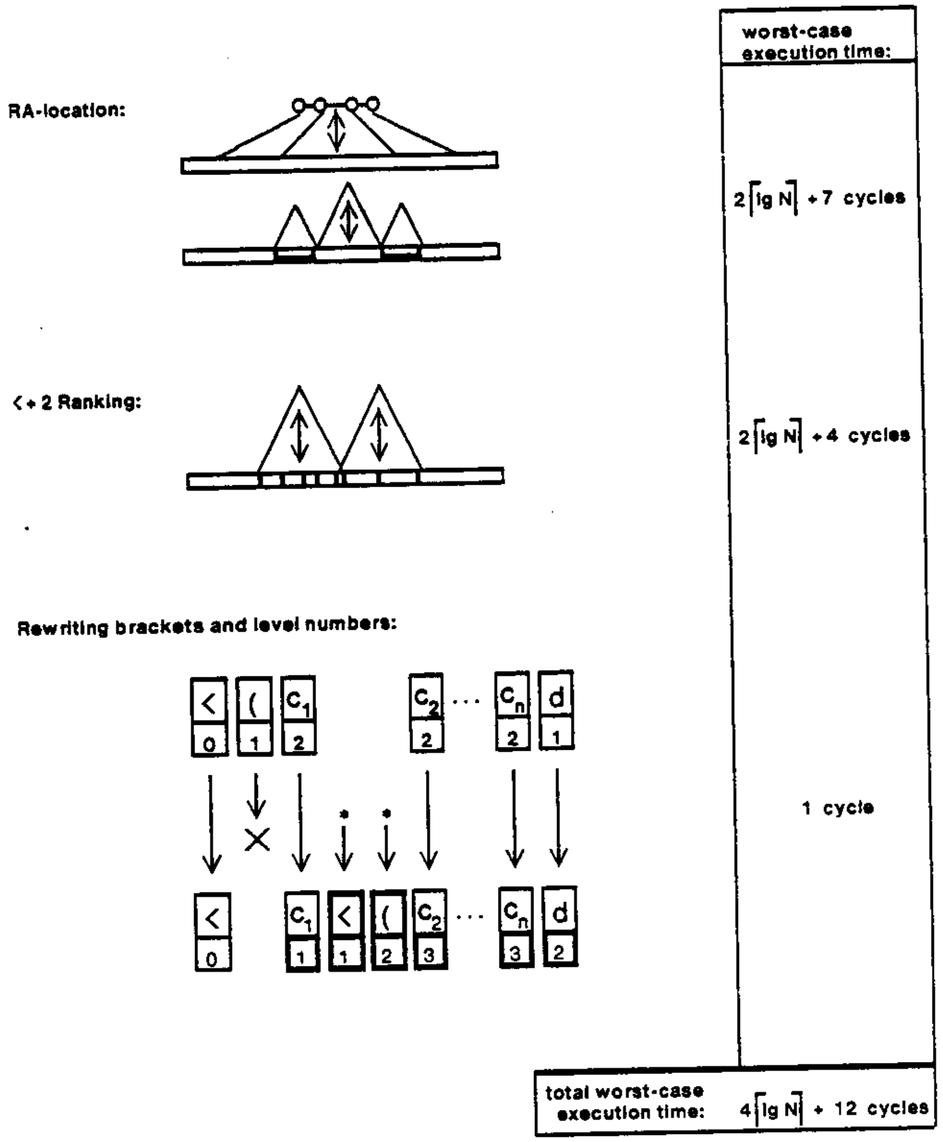
Regular Composition. Fig. 4-16 shows what regular composition looks like in our system's internal representation and how our system reduces an RA which has regular composition in its operator part.

Notice how this reduction requires <+2 ranking followed by rewriting brackets and changing level numbers. After the <+2 ranking, every cell of this RA knows exactly what to do because RA-location and <+2 ranking have together given each of them enough information. The symbols in the first element of the operator sequence must all decrement their level numbers by one, while the symbols in all elements of the rest of the operator and all of the operand must increment their level numbers by one. In the meantime, the leftmost parenthesis in the RA must disappear while a new application symbol ("<") and parenthesis must appear between the first and second operator elements and have the proper level numbers. All of this can happen within one system cycle after the <+2 ranking.

Our system's internal representation of a reduction involving regular composition:



How our system reduces such an instance of regular composition:



worst-case execution time:
RA-location:
$2\lceil \lg N \rceil + 7$ cycles
<+2 Ranking:
$2\lceil \lg N \rceil + 4$ cycles
Rewriting brackets and level numbers:
1 cycle
total worst-case execution time:
$4\lceil \lg N \rceil + 12$ cycles

Fig. 4-16: Reducing expressions involving regular composition

Meta Composition. Fig. 4-17 shows what meta composition looks like in our system's internal representation and how our system reduces an RA which has meta composition in its operator part.

Notice how this reduction requires <+2 ranking followed by copying the first operator element to the immediate left of this RA's leftmost parenthesis, decrementing by one the level numbers of these new cells, and incrementing by one all the level numbers of the operand's symbols. All of this can happen within one system cycle after RA-location and <+2 ranking, because copying the entire first element of the operator sequence can proceed in parallel as follows: All the symbols of the first element acquire idle cells. Every one of these new cells gets a symbol space CDA which has the PA of the RA's application symbol in its starter address field and the PA of its acquiring cell in its insertion field. Thus, the new symbol cells all sit in preserved order between the RA's application symbol and the former leftmost parenthesis of the RA. Since these new symbol cells also receive symbols and level numbers from their acquiring symbol cells, they comprise a new copy of the first operator element.

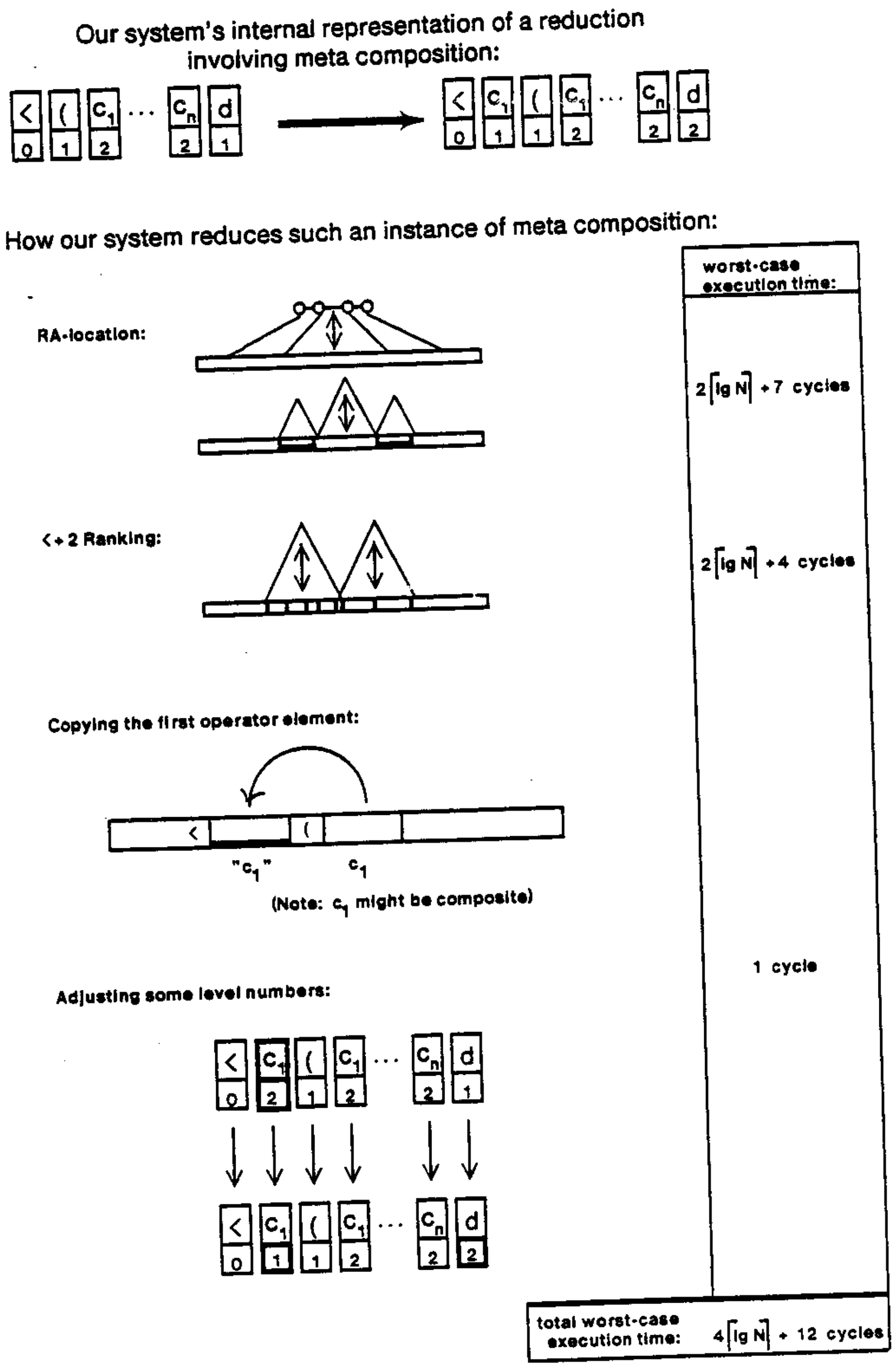

Our system's internal representation of a reduction involving meta composition:
How our system reduces such an instance of meta composition:
worst-case execution time:
RA-location:
$2\lceil \lg N \rceil + 7$ cycles
<+2 Ranking:
$2\lceil \lg N \rceil + 4$ cycles
Copying the first operator element:
"c_1"
c_1
(Note: c_1 might be composite)
1 cycle
Adjusting some level numbers:
total worst-case execution time:
$4\lceil \lg N \rceil + 12$ cycles

Fig . 4-17: Reducing expressions involving meta composition

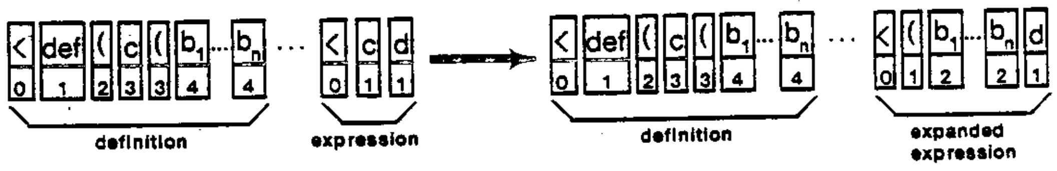
Direct Application of a User-Defined Operator. Fig. 4-18 shows how an RA might have an operator part consisting of a single name which the system does not recognize as a primitive function. This name must appear in a definition in the definition symbol space of the system, otherwise this RA reduces to $\perp$.

Fig. 4-18 also shows how our system reduces an RA with a defined operator by a definition expansion which substitutes the correct subexpression for the defined name. After RA-location the expression cell containing the nonprimitive name acquires a processing space cell and uses it to try and propagate a *lookup demand* to the appropriate definition space cell. Just as we saw in 4.3.3, when other RAs simultaneously try to propagate similar lookup demands to the same definition cell, a tree forms in the processing space to concentrate their demands and distribute the resulting reply from the definition cell to all of them. The source of any lookup demand knows the CDA of its lookup destination in the definition space, because the first cell of any definition has a CDA uniquely derived from the symbolic name it defines (see Fig. 4-6). Thus, if a given definition exists, lookup demands trying to find it always reach it after no more than $\lceil \lg N \rceil + 2$ cycles. If a given name actually has no corresponding definition in the system, the lookup demands searching for it will always fail to find it within the same amount of time. The result of a lookup demand propagates back down the tree in at most $\lceil \lg N \rceil + 2$ additional cycles. Successful lookups provide the following two items of information to all the demanding cells:

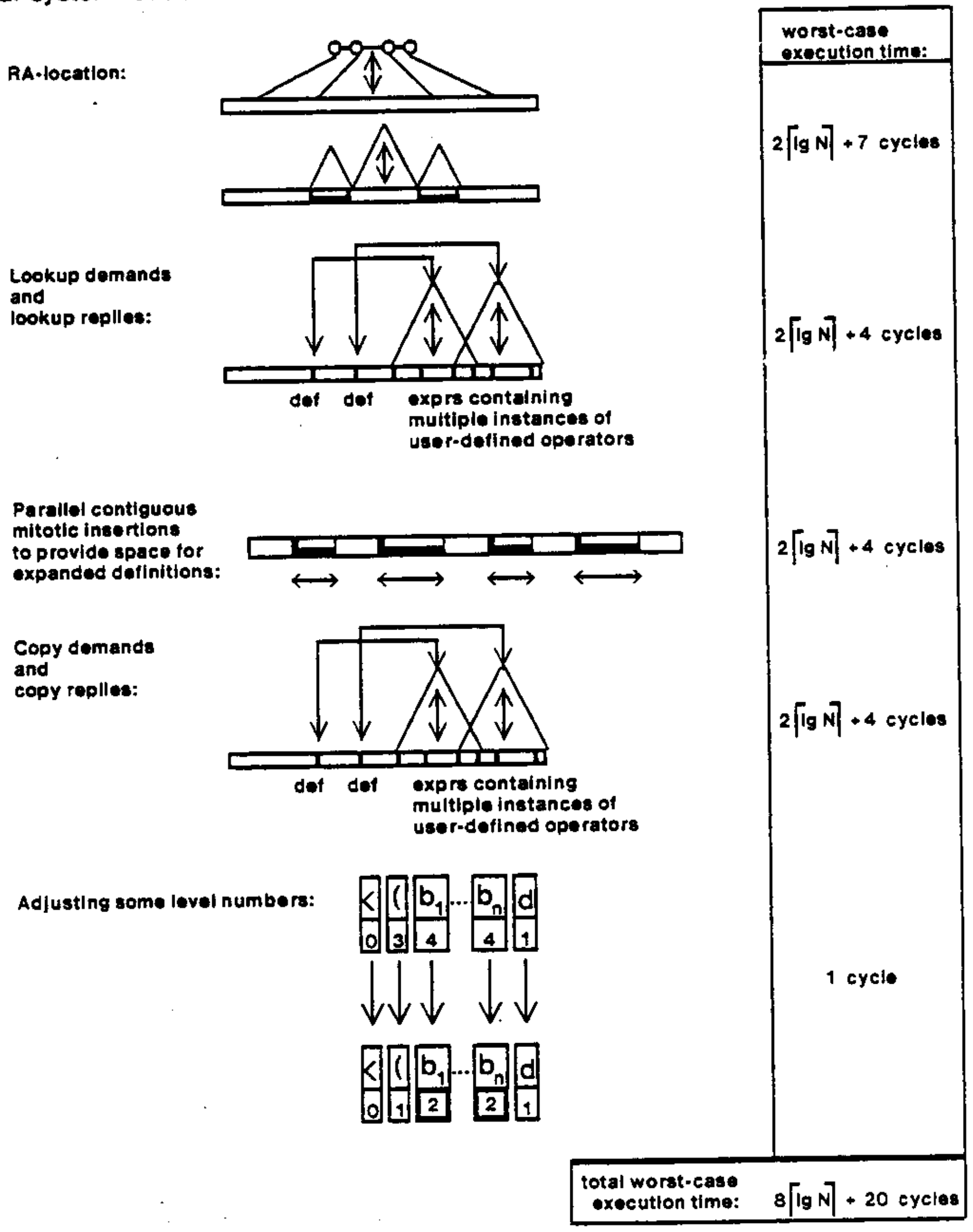
DCA :	the CDA of the leftmost cell of the subexpression to substitute for the defined name (confirming that the lookup succeeded).
DCN :	the number of cells in this subexpression.

When an expression cell containing a nonprimitive name finally receives these two data items in reply to its lookup demand, it knows how much expression symbol space to reserve for its definition expansion (from DCN) and where to copy the definition from (DCA gives the leftmost location). These expression cells then become starters for one round of contiguous mitotic insertions which reserves the necessary expression symbol space and requires at most $2\lceil \lg N \rceil + 4$ cycles to finish. This reserved space shows up as empty symbol cells immediately adjacent to each nonprimitive name symbol involved. In the insertion field of their CDAs, mitotic insertion automatically informs each newly inserted cell of its ordinal position with respect to its starter cell. Hence, every one of these new cells has enough information (from DCA and its own rank) to calculate (by simple concatenation) the CDA of the definition symbol it needs to copy (see Fig. 4-6). Upon finishing this, every one of these new cells acquires a processing space cell and uses it to propagate a *copy demand* to the definition symbol it needs to copy. Like before, when other RAs simultaneously try to copy the same cells, a tree forms in the processing space for each destination to concentrate the demands and distribute the replies. Thus, after no more than $2\lceil \lg N \rceil + 4$ more cycles, every new cell has a symbol and a relative level number. By adding the starter's level number to the relative level number, each new cell obtains its own correct level number and definition expansion has completed. The entire definition expansion process takes no more than $6\lceil \lg N \rceil + 12$ cycles after RA-location.

Our system's internal representation of a reduction involving definition expansion:



How our system reduces such an instance with definition expansion:



worst-case execution time:
RA-location:
2⌈lg N⌉ +7 cycles
Lookup demands and lookup replies:
def def exprs containing multiple instances of user-defined operators
2⌈lg N⌉ +4 cycles
Parallel contiguous mitotic insertions to provide space for expanded definitions:
2⌈lg N⌉ +4 cycles
Copy demands and copy replies:
def def exprs containing multiple instances of user-defined operators
2⌈lg N⌉ +4 cycles
Adjusting some level numbers:
1 cycle
total worst-case execution time: 8⌈lg N⌉ + 20 cycles

Fig. 4-18: Direct application of a user-defined operator

Direct Application of a Primitive Operator. An RA might have an operator part consisting of a single name which the system recognizes as a primitive function. In such cases, the system microcode resident in a PE (or in a cell) controls the RA execution.

Below we look at the primitive meta operator NF and the primitive regular operator HD. These two examples give us some of the flavor of the power-log fast parallel primitives possible in our proposed system. Appendix B show how our system can execute all of our example reduction language's primitive functions in a power-log fast fashion.

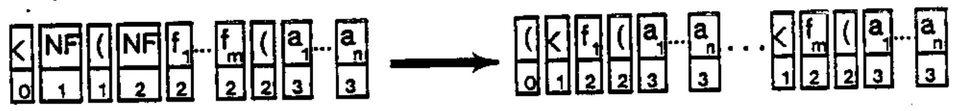
Fig. 4-19 shows what executing the primitive meta operator NF looks like in our system's internal representation (note how meta composition reduction has already occurred), and how our system can accomplish this for an RA.

Notice how this reduction requires <+2 ranking, operand copying, bracket rewriting, and the deletion of two atoms (both "NF"). All of this can happen as follows: After RA-location and <+2 ranking, the two "NF" symbols and the parenthesis between them must disappear. In the meantime, every <+2 -ranked element except the rightmost one (the operand) must insert a "<" symbol to its immediate left with the proper level number, and mitotically insert SCA - ECA+1 cells to its right (enough to hold a complete copy of the operand). Each of these newly inserted cells then copies the appropriate operand cell so that a complete copy of the operand sits with each new application thus formed. Once again, since this copying happens in parallel, trees form in the processing space to concentrate copy demands and distribute duplicate replies. Note how no level numbers need to change during this entire reduction step. This whole process requires no more than $4\lceil \lg N \rceil + 8$ additional cycles after RA-location and <+2 ranking.

Fig. 4-20 shows what executing the primitive regular operator HD looks like in our system's internal representation, and how our system can accomplish this for an RA. Notice how this reduction requires <+2 ranking followed by deletion of all the RA's cells except for the leftmost element of the operand sequence, and then decrementing by two the level numbers of these remaining cells. All of this can happen within one cycle after RA-location and <+2 ranking.

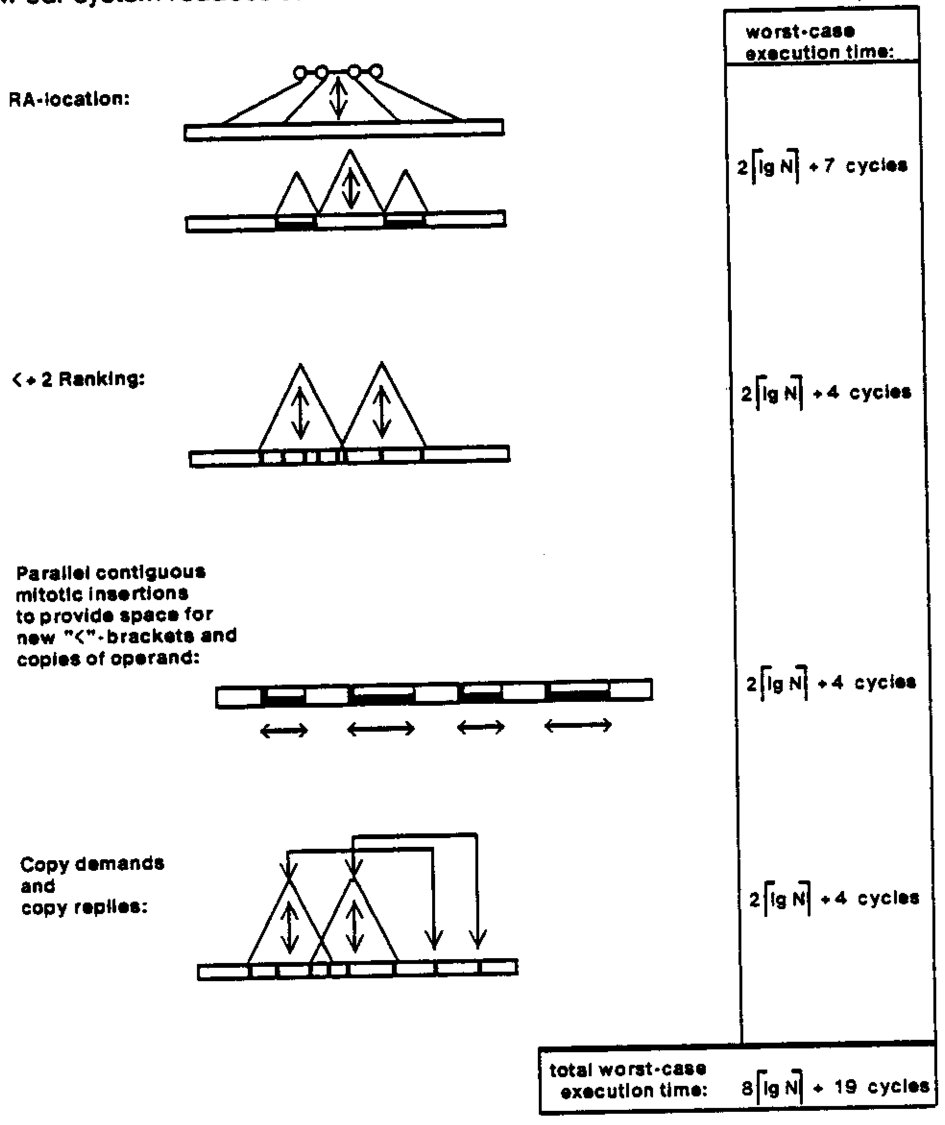
As Appendix B clearly shows, no reduction step for our example reduction language ever requires more than $8\lceil \lg N \rceil + 20$ cycles in our N-cell system. Even TRANS (matrix transpose) executes power-log fast. (Most other proposed systems can't even include TRANS as a primitive!) Appendix C suggests how certain other powerful primitive operators not found in our example reduction language could also execute power-log fast in our system, including even a sort! But this shouldn't surprise us. Our system's communication network, remember, includes a sorter.

Our system's internal representation of a meta reduction involving NF:



< NF (NF f1 ... fm (a1 ... an
0 1 1 2 2 2 2 3 3
(< f1 (a1 ... an ... < fm (a1 ... an
0 1 2 2 3 3 1 2 2 3 3

How our system reduces such an instance:



worst-case execution time:
RA-location:
2⌈lg N⌉ + 7 cycles
< + 2 Ranking:
2⌈lg N⌉ + 4 cycles
Parallel contiguous mitotic insertions to provide space for new "<"-brackets and copies of operand:
2⌈lg N⌉ + 4 cycles
Copy demands and copy replies:
2⌈lg N⌉ + 4 cycles
total worst-case execution time:
8⌈lg N⌉ + 19 cycles

Fig. 4-19: Execution for primitive meta operator NF

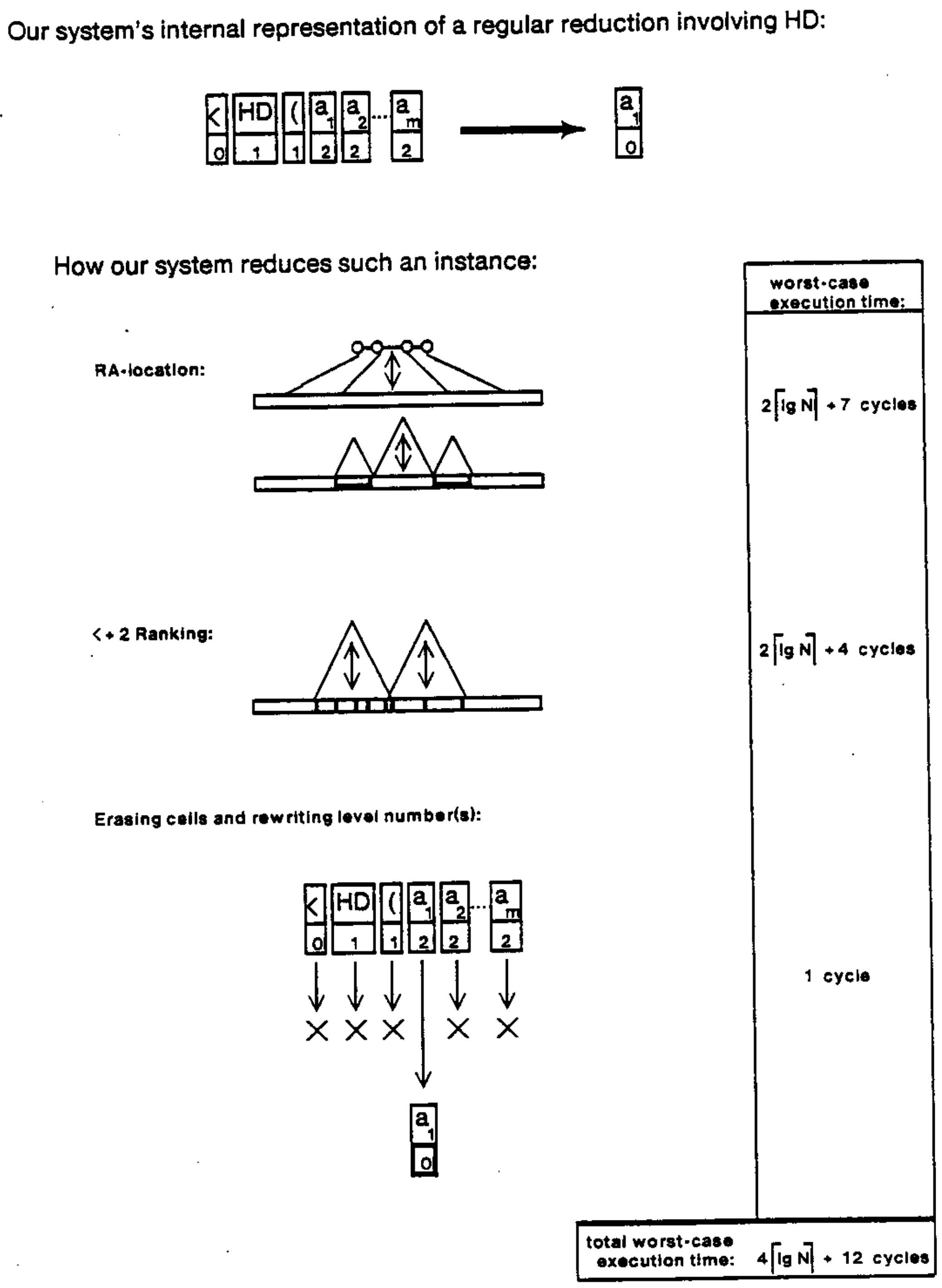

Our system's internal representation of a regular reduction involving HD:
< HD (a_1 a_2 ... a_m
0 1 1 2 2 2
a_1
0
How our system reduces such an instance:
worst-case execution time:
RA-location:
$2\lceil \lg N \rceil + 7$ cycles
< + 2 Ranking:
$2\lceil \lg N \rceil + 4$ cycles
Erasing cells and rewriting level number(s):
1 cycle
total worst-case execution time: $4\lceil \lg N \rceil + 12$ cycles

Fig. 4-20: Execution for primitive regular operator HD

5. Design Evaluation

5.1 Assumptions

Our system's evaluation depends upon a specific set of assumptions about our system. We assume we have our computations expressed as functional programs in our reduction language. Furthermore, we assume that these programs have a great deal of inherent parallelism, i.e., they contain a large number of RAs during most of their execution. In such cases, the execution semantics for reduction language programs of sections 2.4.3 and 2.4.4 gives us a conceptually fast way to carry out these computations: parallel RA-reduction. To ensure that every reduction step can always invoke fully parallel RA-reduction, we also assume that the computer system executing these computations has enough resources (processing elements) to avoid system overflow or resource exhaustion. Defending our conclusions requires us to justify the legitimacy of these assumptions. Let us examine them one at a time.

5.1.1 Computation Expressed in Our Reduction Language

We assume that our computations are expressed as functional programs written in the reduction language described in this report. At the same time, this report claims no special-purpose properties for this particular functional language. On the contrary, our simple reduction language serves merely as an illustrative example to help demonstrate a powerful execution methodology. The reduction step descriptions of Appendix B should suggest similarly powerful schemes to execute the primitive operators of any other general or special-purpose reduction language. (Appendix C contains examples of some additional possible primitive operators which demonstrate this power.)

However, even the simple reduction language of this report contains the foundations for vast expressive power. In one sense, it already has universal computational capability since it contains the equivalents of the primitive operators found in pure Lisp (CAR, CDR, CONS, ATOM, EQ, COND)[12]. But more important with respect to user programming, its definition facility provides a means of language extension. A programmer can define new nonprimitive operators in terms of existing operators (both primitive and nonprimitive) already present in the system. In fact, given the execution scheme for definition expansion we saw in section 4.4.4, a programmer might easily define a new primitive operator as well in the following way: write the microcode to execute the new operator and place this microcode (instead of a symbolic expression) into a DEF expression cell along with the name of the new operator. During execution, then copy replies would bring back microcode instead of expression cells to the requestors (see Fig. 4-18).

Hence, even though this report only presents a single reduction language with which to evaluate our proposed system, we should easily expect our conclusions to hold for a wide variety of more complex general and special-purpose reduction languages.

5.1.2 Large Amount of Inherent Parallelism

We assume that our functional programs possess a great deal of inherent parallelism, i.e., that throughout most of their execution their expression bodies contain many reducible applications to reduce in parallel. This assumption appears at the beginning of this report and motivates practically everything else in it. In chapter 1, we took notice only of example applications requiring massively parallel computations. Our hope to achieve upward-scalable, power-log fast performance seems to depend on the program expression of this inherent parallelism.

Nevertheless, we should note in passing that even if a functional program fails to have any great amount of parallelism, our system organization can still take advantage of what little expressed parallelism it might happen to have. In our system, even a strictly sequential program (or a strictly sequential portion of a parallel program) will properly execute. If any additional parallelism happens to lurk in the program's reduction language expression, the system automatically takes advantage of it without any programmer intervention. (This fulfills one important goal, first mentioned in section 1.2.1, of automatic system adaptation to varying amounts of program parallelism.) Hence, our system does not need program parallelism for correct execution, it just needs it to achieve fast execution.

5.1.3 Sufficient Resources

To ensure that every reduction step always invokes fully parallel RA-reduction, we assume that our computer system always has enough resources (processing elements) to avoid system overflow or resource exhaustion. With cheap enough hardware (both processors and network components) we could perhaps provide for this assumption by shear quantity of PEs. To ensure that the symbol space cells could always perform the worst-case tree building activity, the total system would only need 2 plus twice as many cells as the number of cells in the largest expected symbol space. This kind of requirement should not surprise us, however. Even conventional computer systems need enough main and auxiliary memory to execute a given program, otherwise they will fail to properly complete it. Hopefully, the simplicity and homogeneity of our proposed system's processors and network will help keep down the costs of providing sufficient resources in our system as well.

Should our proposed system ever need more cells during execution than it has available, our present execution scheme results in disaster: The system logjams with active cells trying to acquire nonexistent idle cells before the end of the current reduction step. Because they cannot succeed in this endeavor, this logjam possibly repeats itself again and again in later reduction steps without any further progress in the computation. If we refrained from offloading any symbol cells, but temporarily switched to some nonparallel execution scheme which used fewer processing space cells, we might conceivably unblock some of these logjam situations at the cost of more execution time, but this could not always work in general: If the symbol space consumed all the cells in the system and continued to demand more before it could reduce any RAs, the computation would surely come to a halt. Situations like this force us to look for other methods to solve this problem, such as the virtual memory approach which Frank[28] suggests for Magó's machine (swapping suspended RAs to auxiliary memory). We would need to take great care in designing such a virtual execution mode for overflow cases, since it would be very easy to reintroduce system bottlenecks that severely hurt system parallelism, even if the system overflows were only slight. Perhaps we could find some way to swap RAs in parallel, or some way to temporarily "hide" the symbols of suspended RAs in distributed fashion so that we could suddenly appear to have many new idle cells. Issues like these constitute important topics for further investigation, because until we have some appropriate way to deal with insufficient resource situations, we just cannot overflow our system gracefully!

5.2 Performance: Space and Time

Given the assumptions we have just examined, we now undertake to compare the performance of our proposed system architecture with the other approaches mentioned in this report.

5.2.1 Space

For the purposes of our comparisons we shall adopt the "primitive parts count" metric to express hardware costs. Measuring computer systems in this way, the total cost of a system linearly depends on the number of bits in its registers and I/O ports, the number of wires communicating individual bits of data in its system network, and the number of logic gates in its ALU. This may or may not realistically reflect its actual hardware costs, which also depend on the space-efficiency of these parts' arrangement in an actual system. We shall leave the investigation of efficient layouts in 2- and 3-dimensional space to future reports, since much work remains unfinished in this field.

A conventional uniprocessor system with N memory cells, then, has at least an O(N) parts count. If each of its N memory cells contained O(lg N) bits, for example, then the whole system would have O(N lg N) hardware cost.

The serially-scanning multiprocessors of Patel and of Treleaven and Mole, on the other hand, have a very different hardware cost behavior. Although either might contain, say, N processors, each processor needs its own auxiliary memory to store all the symbols of the largest size RA which a processor may sometime need to reduce. If we place no restrictions on RA size, the largest possible RA might contain O(N) symbols. Furthermore, each symbol, to make it distinguishable in the system, might need O(lg N) bits in its representation. Thus, the entire serially-scanning multiprocessor might have $O(N^2 \lg N)$ hardware cost!

In designing parallel-scanning multiprocessors, such as Magó's machine and the system proposed in this report, we have more hope of achieving O(N lg N) hardware cost because each of the N processing elements in either case contains only a fixed number of O(lg N)-bit registers. We still need to consider whether the interconnection network of the system in question adds any additional asymptotic hardware cost. The tree network of Magó's machine does not increase its asymptotic hardware cost, since the number of wires in any tree network is directly proportional to the number of the PEs in that tree and the data path width between PEs. Can we say the same thing about this report's proposed system? We answer this question in section 5.2.3 after discussing our network in more detail.

5.2.2 Time

Since communication dominates the time costs of computation in all the systems we consider in this report, we shall adopt a time metric based on the time it takes for a single wire to transmit one bit between processors (ignoring wire length). This unit of time is quite appropriate because hardware scalability eventually encourages us to use bit-serial communications between processors to help keep down the number of wires in really huge multiprocessors. In fact, if we use bit-serial primitive operations within processors (as suggested in section 4.3.2) we not only reduce the ALU hardware, we also make it possible to overlap primitive computations with the serial communications, thereby masking any asymptotic effects these primitive computations might have on system execution time. Thus, this report's way measuring execution time fits quite well to the parallel systems under our investigation.

A conventional uniprocessor, then, takes at least O(N) time to execute a functional program with O(N) problem size. It might take worse than O(N) because many functional primitive operators, like AA or TRANS, are not exactly "primitive" on a uniprocessor, —each may itself take O(N) time or worse just to complete. The serially-scanning multiprocessors of Patel and of Treleaven and Mole, even with O(N) processors, can still take at least O(N) time to execute a functional program: If some functional program contained an RA of O(N) problem size, this RA would still have to eventually wind up in the local memory of a single processor before it could execute. From that point, it would execute as slowly as in a conventional uniprocessor.

We now examine the execution times of the parallel-scanning multiprocessors. Magó[19] gives the following expression for the execution time of his reduction multiprocessor:

$$\text{time/RA} = S + \lceil m/M \rceil * S$$

where:

m	is the number of messages between cells necessary to reduce the RA.
M	is the worst-case number of uncombineable messages among the cells of the RA that the system can deliver during a 14-state system cycle.

and where S, the time to execute a 14-state system cycle, is given by:

$$S = 14 * t * \lceil \lg N \rceil + K$$

where:

t	is the time it takes for a state change to propagate one level in the tree. (This is typically a small constant. Assume $t=1$ for our purposes.)
N	is the number of leaf cells (L cells) in the tree ($\lceil \lg N \rceil$ is the number of levels in the tree.)
K	is the extra time required during a 14-state system cycle to bring in microprograms, send messages between cells, and to complete storage management.

We can already see from these formulas that the asymptotic time behavior of Magó's machine critically depends upon the asymptotic behaviors of m, M, and particularly K. For example, if m and M had the same asymptotic orders of growth, and if K were really a constant independent of N (as Magó claims), then Magó's machine would indeed execute power-log fast (in fact, log-fast). Such is not the general case, however, and Magó's machine can actually have O(N) execution time per RA, or even worse.

Let us now examine an example of how Magó's machine can have an execution which is not power-log fast. Recalling our discussion at the beginning of this section, we assume bit-serial communication within Magó's machine for scalability (at worst, this only multiplies the execution time by O(lg N)). For the time being, we also ignore the loading of microcode (assume resident firmware in each cell). From the definition of K above, then, we get:

$$K = M\lceil \lg N \rceil + c20\lceil \lg N \rceil$$

where:

c	is the number of symbols moved during storage management during a 14-state cycle. (The factor $20\lceil \lg N \rceil$ comes from the approximately twenty registers of $\lceil \lg N \rceil$ bits each in every cell of Magó's machine.)

Now in some realistic situations RAs executing in Magó's machine might possibly have:

$$c = O(N)$$

$$m = O(M)$$

(One simple example of this might be a large RA getting pushed around by other RAs while it, say, tries to REVERSE its elements. Other difficult permutations for Magó's machine include such useful ones as the butterfly, bit-reversal, and shuffle permutations.[29]) Substituting these into the above formulas gives for Magó's machine:

$$\text{time/RA} = [(O(N)/M)+1]^{*}(O(N)+14+M)^{*}(\lceil \lg N \rceil)$$

Whereupon, we see that no matter what asymptotic behavior we assume for M, Magó's machine still needs O(N) time or longer to complete such reduction steps.

The new computer system design which this report introduces has a very different and much more promising asymptotic behavior. Let us assume (like we did with Magó's machine) bit-serial communication and no need to load microcode. Taking into account our system's four phases per cycle, its $8\lceil \lg N \rceil + 20$ cycles per reduction step, and its ~20 registers per cell, we obtain:

$$\text{time/reduction step} = ((1/2)8\lceil \lg N \rceil + 20)^{*}((2)3P+1)^{*}(20\lceil \lg N \rceil)$$

where:

P	is the time required to perform a parallel sort

and where the factors of "1/2" and "2" come from the fact that to allow worst-case tree building, the maximum number of symbols should actually be about half the total number of cells in our system. As we can see from the above formula, if we can perform a power-log fast parallel sort, then our system will also have power-log fast execution for its reduction steps. The asymptotic behavior of P, then, determines the asymptotic performance behavior of our system.

Let's investigate P. Fig. 5-1 shows the time performance (and hardware costs) of several high performance parallel sorting networks in terms of the number of cells, N, each could support for our system (and ignoring the bit-serial communication overhead, an O(lg N) factor which our formula above already takes into account).

Parallel Sorting Network	Number of Comparators Required	Worst-case Sort Time (P)
j-dimensional mesh sorter of Thompson and Kung	$O(jN)$	$O(N^{1/j})$ (not power-log fast)
odd-even or bitonic sorter of Batcher	$O(N \lg^2 N)$	$O(\lg^2 N)$ (power-log fast)
perfect-shuffle sorter of Stone	$O(N)$	$O(\lg^2 N)$ (power-log fast)
perfect-shuffle or cube-connected sorter of Nassimi and Sahni (where $Q^{1+1/k} \leq N$)	$O(Q^{1+1/k})$	$O(k \lg Q)$ (power-log fast)

Fig. 5-1: The asymptotic impact of several choices of parallel sorting network for our system

From this chart we see that the j-dimensional mesh sorter of Thompson and Kung[30] can sort N cells in $O(N^{1/j})$ time using O(jN) comparators. Although it has the simplest topology of all the networks under our consideration, the system it would produce would not execute power-log fast. The odd-even and bitonic sorters of Batcher[31] on the other hand, can sort N cells in $O((\lg N)^2)$ time using $O(N(\lg N)^2)$ comparators. Either of them would produce a power-log fast system, but at the expense of considerably more hardware than a mesh sorter. The perfect shuffle sorter of Stone[32] however, can sort N cells in $O((\lg N)^2)$ time using O(N) comparators. This sorter realizes power-log speed with far less hardware than the other two above-mentioned sorters. Moreover, if we have some flexibility in the number of comparators available in some perfect shuffle interconnection, Nassimi and Sahni[33] have described how this same perfect-shuffle sorter can sort Q cells in O(k lg Q) time using only $O(Q^{1+1/k})$ comparators (where k is a constant and $Q^{1+1/k}$ must be less than the total number of cells in the system). Fig. 5-2 shows how setting k to different values can greatly vary this sorter's asymptotic time and space behaviors. Thus, a perfect shuffle sorter provides some interesting time and space tradeoffs to make in implementing our system. For the time being, therefore, we choose a perfect shuffle for our system network.

Value of k for a perfect-shuffle or cube-connected sorter of the type Nassimi and Sahni describe	Number of Comparators Required	Worst-case Sort Time (P)
$k = 1$ (so that $Q^2 \leq N$)	$O(Q^2)$	$O(\lg Q)$
$k = \lg Q$ (so that $2Q \leq N$)	$O(Q)$	$O(\lg^2 Q)$
$k = 1/\epsilon$ (where ϵ is small)	$O(Q^{1+\epsilon})$	$O(\frac{1}{\epsilon} \lg Q)$

Fig. 5-2: How different values of k impact the time and space performance of the parallel sorters of Nassimi and Sahni

5.2.3 Hardware Cost

The low O(N) hardware cost of a perfect shuffle sorter enables us to combine both PEs and communication network into an integrated whole: connect the PEs themselves into a perfect shuffle network and let them carry out the comparator functions of a perfect shuffle sorter. Fig. 5-3 diagrams an example of such a system for the small-scale case of eight PEs. The total hardware cost of such a system would just be the total O(N lg N) PE cost plus the total O(N) interconnection hardware cost, or just a total of O(N lg N).

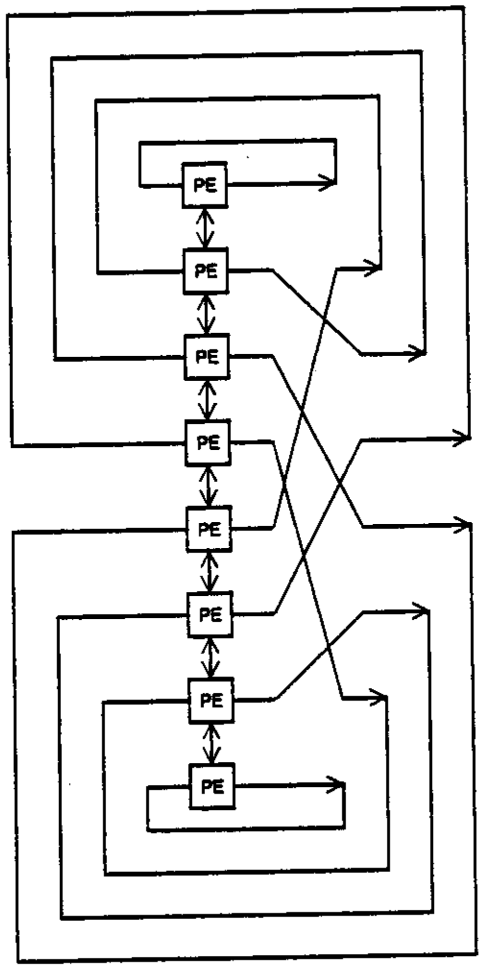


Fig. 5-3: An eight-PE system incorporating a perfect shuffle

5.2.4 Execution Speed

Consider Fig. 5-4 which graphs both O(N) and power-log fast execution times versus N. Notice how the power-log fast curve, after a fast initial rise, eventually flattens out into an approximate plateau which becomes more nearly horizontal as N gets larger. This plateau implies that after a point, increasing N has only a small effect on the execution time, and hence, doubling N after this point almost doubles the parallel execution capacity (throughput) of a power-log fast system. The O(N) curve, on the other hand, eventually increases linearly with N, which implies that its throughput eventually becomes nearly constant, even when N increases any amount. These respective behaviors constitute the original motivations to design a power-log fast parallel processor, such as the one this report proposes.

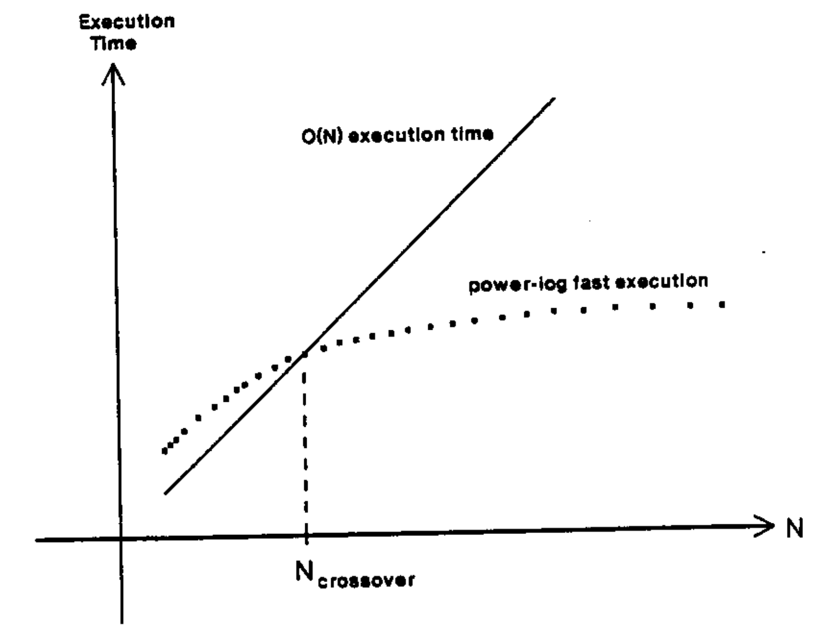


Fig. 5-4: O(N) and power-log fast execution times versus N

We have already seen how our proposed multiprocessor reduction machine has a faster worst-case asymptotic execution than even Magó's parallel multiprocessor, and in fact, it even achieves power-log fast execution speed for any reduction step it can execute. But asymptotic descriptions can hide constant multipliers and lower-order terms which might significantly affect the relative execution speeds of systems we might actually want to construct. Looking back at Fig. 5-4, we see that for $N < N_{crossover}$, the power-log fast execution time is actually *worse* than the O(N) execution time. For this reason, we want to try some numbers in our formulas to more meaningfully compare our system's performance with Magó's, and in particular, find $N_{crossover}$.

Keeping the same assumptions as before (bit-serial communication between cells and resident firmware in each cell), we choose for Magó's machine:

$$c = N/2$$

$$m = N/2$$

(This might correspond, say, to the limiting case of an RA trying to REVERSE N/2 symbols while it gets pushed all the way across the system by storage management.) Substituting into the appropriate formulas of section 5.2.2 gives for Magó's machine:

$$\text{worst time/RA} = [(N/2M)+1]*(10N+14+M)*(\lceil \lg N \rceil)$$

For the machine proposal of this report, we make the same assumptions (bit-serial communications between PEs and resident firmware in each PE) as before, and use a perfect shuffle organization with Stone's sorting algorithm[32] so that:

$$P = (\lceil \lg N \rceil)^2 - \lceil \lg N \rceil$$

Substituting into the appropriate formula of section 5.2.2 gives for our machine:

$$\text{worst time/reduction step} = (4\lceil \lg N \rceil+20)*(1+6[(\lceil \lg N \rceil)^2 - \lceil \lg N \rceil])*(\lceil \lg N \rceil)$$

This formula is independent of both storage management behavior and parallel communication complexity so long as no system overflow occurs. With both of these exact formulas for execution time, we need only specify M for Magó's machine and try out values for N in both formulas in order to compare the respective performances of both machines.

Setting $M=1$ would give Magó's machine the notable property of having equal-sized state times throughout its 14-state cycle (except for the state transitions during its storage management). It would also force Magó's machine to perform operations like REVERSE only one element at a time per cycle, because of the implied limit of 1 on uncombineable messages per cycle. For $M=1$, $N_{crossover}$ is in the approximate vicinity of 256. This means that for same-sized systems executing identical collections of massively parallel, message intensive, and storage-varying programs, this report's proposed machine could execute faster than Magó's machine for all

$N>256$.

Setting M to asymptotically greater values than 1 would disrupt locality of effect in Magó's machine, because a single message-intensive RA (say, one involving REVERSE) could then slow down the entire system when a 14-state cycle paused to deliver M messages before continuing. Nonetheless, we could claim that storage management also disrupts locality of effect in the same way, so let's give increasing M a try. Setting $M=\lceil \lg N \rceil$ would raise $N_{crossover}$ to just a little over 1024, still a rather small value for a "massively parallel system". Even $M=\lceil \lg N \rceil^2$ would only raise $N_{crossover}$ to just over 4096. If we were to go all the way to the maximum and set $M=N/2$, Magó's machine would then take care of all the messages from every RA in the system in a single (and very irregularly timed) 14-state cycle. $N_{crossover}$ would then reach its maximum value of approximately 100,000.

$N=100{,}000$ corresponds to a fairly large system at the present time. This large crossover value might tend to suggest sufficient parallel computation powers for even tree-structured multiprocessors like Magó's machine. We must temper this judgment, however, with the following observations: This report's proposed machine can have much more powerful primitive operations (like TRANS and even SORT) than Magó's machine. Our proposed machine, unlike Magó's machine, can also perform fully parallel definition lookups of any number of functions by any number of RAs and still need only to keep one copy of each definition in its lookup area. Moreover, once N increases past $N_{crossover}$, further increases in N for our proposed machine increases its parallel capacity in a nearly proportional manner without its patterns of storage management or parallel communication affecting this rate of increase.

6. Conclusions

6.1 What Have We Accomplished Here?

This report has investigated the fully parallel execution of general-purpose functional programs. By proposing a new system design and execution methodology in somewhat concrete detail, we have found at least one way to meet the goals we set in chapter 1. In particular, our new system design achieves all of the following:

It automatically detects and takes advantage of the maximum amount of program-expressed parallelism in the programs it executes.

It directly executes a functional reduction language and has ample facilities to implement many powerful primitive operators for this language.

It contains a large number of simple and identical processing elements, interconnected in a manner which allows all general interprocessor communication patterns.

It behaves as a power-log fast system for any computation it can execute to completion, and, using our parts count metric, it costs only O(N lg N) total hardware to implement a system containing O(N) PEs.

As a result of our worst-case evaluative approach, these achievements provide some special insights into the parallel execution of functional programs in general. For example, we can say that with a sufficient number processing elements we eventually reach a point where doubling the number of these PEs can nearly double the total system throughput for a very wide class of massively parallel computations.

In addition to meeting these goals, our proposed system also solves, in power-log fast time, the two rather more general problems of resource management: allocation and compaction. Inefficient resource allocation and compaction (in the guise of "free list handling" and "garbage collection") have long plagued the conventional execution of computations written in functional-style programming languages. Moreover, systems based on object-oriented message-passing semantics have similar resource management problems. Hence, our new approach might someday find application not only in functional programming systems, but also for implementing parallel execution in object-oriented systems like PLASMA[34] and SMALLTALK[35].

6.2 What Might We Do With All of This?

The practicality of using our proposed system design for any real system depends on what we really want in terms of system "generality". If a planned system will only ever execute a single computation (e.g., an FFT network), then custom-designed, optimally configured hardware will always work best for such a system. On the other hand, some hardware configurations may work well over whole specialized classes of problems (e.g., tree networks for hierarchical divide-and-conquer computations). More generally, however, the universe of computational problem-solving has many nasty surprises where certain useful computations fail to map at all well to a given specialized parallel communication architecture (e.g., REVERSE in a tree network). When the planned system needs to retain predictable worst-case execution performance in spite of such computational surprises, then we have the need for a system design like this report's proposal.

Many challenges still await the adventurous builder of any instance of our proposed system. We have already noted the system overflow problem (chapter 5) that might occur should the system run out of execution space, and also the necessity for decomposing program data into $\lceil \lg N \rceil$-bit chunks (chapter 2) so that each processing element in the system needs only a limited amount of internal storage. Other important issues include testing and debugging such a large multiprocessor, reliability considerations (e.g., building a fault-tolerant parallel sorting network), and many application-dependent system integration issues. All of these issues deserve additional investigation.

For applications with $N<100{,}000$ using our new approach is appropriate when functional expressions might need to perform complex symbolic rearrangements during execution (e.g, sorting, permuting, matrix transposition), when expressions might need to perform many definition lookups (e.g., when the programmer uses a hierarchical approach to deal with the complexity of a computation), and when the symbolic storage behavior of the computation might vary between extremes (e.g., as it will often do when a single system executes several different programs simultaneously). Any application with real-time parallel processing will easily produce such situations, including all of the applications mentioned in chapter 1 (image processing, control systems, simulation systems, and complex signal synthesis).

For applications with N>100,000 the power-log speeds of our new approach's reduction steps become quite significant in its overall execution time. Regardless of problem size, the system performs any executable parallel reduction step in what almost looks like constant time (recall the logarithmic plateau in the power-log curve of Fig. 5-4), and thus closely approximates proportional scalability. Note that we have this behavior even though we assume PEs communicate bit-serially!

At the present time systems of 10^5 PEs constitute the limiting horizon of our technological capabilities. The future will probably bring us systems with far more than 10^5 PEs, perhaps using different technologies than today's (maybe molecular logic?[36]). At that time we will find it quite a challenge to put together such huge numbers of processing elements and still obtain general-purpose systems where the processors help each other more than hinder each other. Hopefully, system designs like the proposal of this report will inspire some of the possible solutions to this problem.

References

1. Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", Communications of the ACM, 21(8):613-641, August 1978.

2. Burton, F.W., and M.R. Sleep. "Executing Functional Programs on a Virtual Tree of Processors", Proceedings of the MIT Conference on Functional Programming Languages and Computer Architectures, Portsmouth, New Hampshire, October 1981, pp.187-194.

3. Backus, J. "Programming Language Semantics and Closed Applicative Languages", Proceedings of the ACM Symposium on the Principles of Programming Languages, Boston, Mass., October 1-3, 1973, ACM, NY, NY, 1973, pp.71-86.

4. Berkling, K.J. "Reduction Languages for Reduction Machines", Proceedings of the Second Annual Meeting of Computer Architecture, University of Houston, January 20-22, 1975, IEEE, NY, NY, 1975, pp.133-138.

5. Henderson, P. Functional Programming: Applications and Implementation, John Wiley and Sons, New York, 1980.

6. Hehner, E. "Positive Concurrent Programming", oral presentation September 9, 1980 at the Xerox Palo Alto Research Center.

7. Wu, C.L. "Interconnection Networks", guest editor's introduction, Computer, 14(12):8-9, December 1981.

8. Pease, M.C. "The Indirect Binary n-Cube Microprocessor Array", IEEE Transactions on Computers, 26(5):458-473, May 1977.

9. Johnston, J.B. "The Contour Model of Block Structured Processes", Proceedings of the ACM Symposium on Data Structures in Programming Languages, SIGPLAN Notices, 6(2), February 1971.

10. Budnik, P., and D.J. Kuck. "The Organization and Use of Parallel Memories", IEEE Transactions on Computers, 20(12):1566-1569, December 1971.

11. Ackerman, W.B. "Data Flow Languages", Proceedings of the AFIPS NCC 1979, NY, NY, June 4-7, 1979, AFIPS Press, Montvale, NJ, 1979, pp.1087-1095.

12. McCarthy, J., et al. LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Mass., 1965.

13. Henderson, P., and J.H. Morris, Jr. "A Lazy Evaluator", Third ACM Symposium on Principles of Programming Languages, Atlanta, Ga., January 19-21, 1976, ACM, NY, NY, 1976, pp.95-103.

14. Keller, R.M., G. Lindstrom, and S.S. Patil. "A Loosely-Coupled Applicative Multi-Processing System", Proceedings of AFIPS NCC 1979, NY, NY, June 4-7, 1979, AFIPS Press, Montvale, NJ, 1979, pp.613-622.

15. Ashcroft, E.A., and W.W. Wadge. "Lucid, a Nonprocedural Language with Iteration", Communications of the ACM, 20(7):519-526, July 1977.

16. Ackerman, W.B., and J.B. Dennis. "VAL – A Value-Oriented Algorithmic Language: Preliminary Reference Manual", MIT Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., TR-218, June 1979.

References

17. Arvind, K.P. Gostelow, and W. Plouffe. "An Asynchronous Programming Language and Computing Machine", Department of Information and Computer Science, University of California, Irvine, Technical Report 114a, December 1978.

18. McJones, P. "A Church-Rosser Property of Closed Applicative Languages", IBM Research Laboratory, San Jose, Calif., IBM Research Report RJ1589, May 1975.

19. Magó, G.A. "A Network of Microprocessors to Execute Reduction Languages", International Journal of Computer and Information Sciences, 8(5):349-385 (Part 1), 8(6):435-471 (Part 2), 1979.

20. Patel, D.R. "A System Organization for Applicative Programming", Computer Science Department, University of California, Los Angeles, UCLA-ENG-8204, March 1981.

21. Treleaven, P.C., and G.F. Mole. "A Multi-processor Reduction Machine for User-defined Reduction Languages", Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, England, Technical Report 150, January 1980.

22. Magó, G.A. "A Cellular Computer Architecture for Functional Programming", Spring COMPCON 80 Digest of Papers, San Francisco, Calif., February 25-28, 1980, IEEE, NY, NY, 1980, pp.179-187.

23. Tolle, D.M. "Coordination of Computation in a Binary Tree of Processors: A Machine Design", Ph.D. dissertation in preparation (1980), University of North Carolina at Chapel Hill.

24. Seitz, C.L. "System Timing", Introduction to VLSI Systems, edited by C. Mead, and L. Conway, Addison-Wesley, Reading, Mass., 1980, pp.218-262.

25. Knuth, D.E. The Art of Computer Programming, Vol.3: Sorting and Searching, second edition, Addison-Wesley, Reading, Mass., 1973, pp.224-225.

26. Trivedi, K.S., and M.D. Ercegovac. "On-line Algorithms for Division and Multiplication", IEEE Transactions on Computers, 26(7):681-687, July 1977.

27. Knuth, D.E. The Art of Computer Programming, Vol.1: Fundamental Algorithms, second edition, Addison-Wesley, Reading, Mass., 1973, pp.333-334.

28. Frank, G.A. Virtual Memory Systems for Closed Applicative Language Interpreters, Ph.D. dissertation, University of North Carolina at Chapel Hill, 1979.

29. Parker, D.S. "Notes on Shuffle/Exchange-Type Switching Networks", IEEE Transactions on Computers, 29(3):213-222, March 1980.

30. Thompson, C.D., and H.T. Kung. "Sorting on a Mesh-Connected Parallel Computer", Communications of the ACM, 20(4):264-271, April 1977.

31. Batcher, K.E. "Sorting Networks and their Applications", Proceedings of the AFIPS SJCC, Atlantic City, NJ, April 30 - May 2, 1968, Thompson Book Company, Washington, DC, 1968, pp.307-314.

32. Stone, H. "Parallel Processing with the Perfect Shuffle", IEEE Transactions on Computers, 20(2):153-161, February 1971.

References

33. Nassimi, D., and S. Sahni. "Parallel Permutation and Sorting Algorithms and a New Generalized-Connection-Network", Computer Science Department, University of Minnesota, Technical Report 79-8, April 1979.

34. Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages", Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., AI Working Paper 92, April 1976.

35. Kay, A. and A. Goldberg. "Personal Dynamic Media", Computer, 10(3):31-41, March 1977.

36. "Whatever Happened to Molecular Electronics?", IEEE Spectrum, 18(12):17, December 1981.

Appendix A: Definitions of Our Example Primitive Functions

A.1 Meta Operators

NF (N Functions)

$\langle(NF,f_1,f_2, \ldots ,f_m),x\rangle$ reduces to $(\langle f_1,x\rangle,\langle f_2,x\rangle, \ldots ,\langle f_m,x\rangle)$

Comment: NF enables many functions to simultaneously operate on the same argument(s). We also use the square brackets, [...], to abbreviate (NF ...), so that equivalently:

$\langle[f_1,f_2, \ldots ,f_m],x\rangle$ reduces to $(\langle f_1,x\rangle,\langle f_2,x\rangle, \ldots ,\langle f_m,x\rangle)$

CN (ConditioN)

$\langle(CN,f_1,f_2,f_3),x\rangle$ reduces to

$\langle(CN,\langle f_1,x\rangle,f_2,f_3),x\rangle$ if f_1 is not a constant function,

$\langle f_2,x\rangle$ if $f_1=T$,

$\langle f_3,x\rangle$ if $f_1=F$,

otherwise it reduces to $\perp$.

C (Constant)

$\langle(C,v),x\rangle$ reduces to v

Comment: We also use C(v) to represent (C,v), so that equivalently:

$\langle C(v),x\rangle$ reduces to v

AA (Apply to All)

$\langle(AA,f),(a_1,a_2, \ldots ,a_m)\rangle$ reduces to $(\langle f,a_1\rangle,\langle f,a_2\rangle, \ldots ,\langle f,a_m\rangle)$

Comment: AA enables the same function(s) to operate on many arguments.

INS (INSert)

$\langle(INS,f),x\rangle$ reduces to

$\langle f,(a1,\langle(INS,f),(a_2, \ldots ,a_m)\rangle)\rangle$ if $x=(a_1,a_2, \ldots ,a_m)$ and $m>1$,

a_1 if $x=(a_1)$,

$(\ )$ if $x=(\ )$.

Comment: Notice how INS produces recursion via meta composition.

AI (Associative Insert)

$\langle(AI,f),x\rangle$ reduces to

$\langle f,(\langle(AI,f),(a_1, \ldots ,a_{\lceil m/2\rceil})\rangle,\langle(AI,f),(a_{\lceil m/2\rceil+1}, \ldots ,a_m)\rangle)\rangle$

if $x=(a_1,a_2, \ldots ,a_m)$ and $m>1$,

a_1 if $x=(a_1)$,

$(\ )$ if $x=(\ )$.

Comment: AI directly produces tree reduction involving a single implicitly repeated operator part ("f" above).

A.2 Regular Operators

ID (IDentity)

$\langle ID,x \rangle$ reduces to x

Comment: When the empty sequence, ( ), occurs in the operator part of an application, we can say: ( ) = ID and ID = ( ), and thus, either one can serve as the concrete representation of $\emptyset$.

HD (HeaD)

$\langle HD,x \rangle$ reduces to

a_1 if $x=(a_1,a_2, \ldots ,a_m)$ and $m \geq 1$,

otherwise it reduces to $\perp$.

TL (TaiL)

$\langle TL,x \rangle$ reduces to

$(a_2, \ldots ,a_m)$ if $x=(a_1,a_2, \ldots ,a_m)$ and $m \geq 2$,

( ) if $x=(a_1)$,

otherwise it reduces to $\perp$.

AP (APply)

$\langle AP,x \rangle$ reduces to

$\langle x_1,x_2 \rangle$ if $x=(x_1,x_2)$,

otherwise it reduces to $\perp$.

EQ (generalized EQuals predicate)

$\langle EQ,x\rangle$ reduces to

T if $x=(x_1,x_1)$,

F if $x=(x_1,y_1)$ and $x_1 \neq y_1$,

otherwise it reduces to $\perp$.

ATOM (ATOMic predicate)

$\langle ATOM,x\rangle$ reduces to

T if $x=(\ )$, or $x \neq \perp$ and x is atomic,

F if $x \neq \perp$ and x is not atomic,

otherwise it reduces to $\perp$.

NULL (NULL sequence predicate)

$\langle NULL,x\rangle$ reduces to

T if $x=(\ )$, or $x=ID$,

F if $x \neq \perp$ and $x \neq (\ )$ and $x \neq ID$,

otherwise it reduces to $\perp$.

UN (UNion)

$\langle UN,x\rangle$ reduces to

$(a_1,a_2, \ldots ,a_m)$ if $x=(a_1,(a_2, \ldots ,a_m))$ and $m>1$,

(a_1) if $x=(a_1,(\ ))$,

otherwise it reduces to $\perp$.

+,-,*,/ (arithmetic operations)

<+,x> reduces to the value a_1+a_2 if $x=(a_1,a_2)$ and a_1,a_2 are numerical values

<-,x> reduces to the value $a_1 - a_2$ if $x=(a_1,a_2)$ and a_1,a_2 are numerical values

<*,x> reduces to the value a_1*a_2 if $x=(a_1,a_2)$ and a_1,a_2 are numerical values

</,x> reduces to the value a_1/a_2 if $x=(a_1,a_2)$ and a_1,a_2 are numerical values

INX (INdeX)

<INX,x> reduces to

(1,2, ... ,m) if x has the numerical value m,

otherwise it reduces to $\perp$.

REVERSE (REVERSE a sequence)

<REVERSE,x> reduces to

$(a_m,a_{m-1}, \dots ,a_2,a_1)$ if $x=(a_1,a_2, \dots ,a_m)$ and m>1,

a_1 if $x=(a_1)$,

( ) if x=( ),

otherwise it reduces to $\perp$.

TRANS (TRANSpose 2-dimensional array)

<TRANS,x> reduces to

$((a_{11},a_{21}, \dots ,a_{j1}),(a_{12},a_{22}, \dots ,a_{j2}), \dots ,(a_{1k},a_{2k}, \dots ,a_{jk}))$

if $x=((a_{11},a_{12}, \dots ,a_{1k}),(a_{21},a_{22}, \dots ,a_{2k}), \dots ,(a_{j1},a_{j2}, \dots ,a_{jk}))$ and j,k>1,

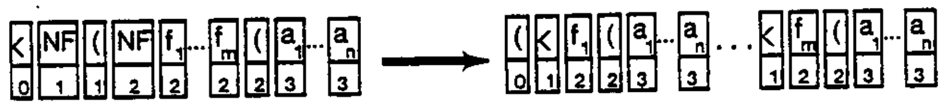
$((a_{11}))$ if $x=((a_{11}))$,

otherwise it reduces to $\perp$.

Appendix B: Implementing Our Example Primitive Functions

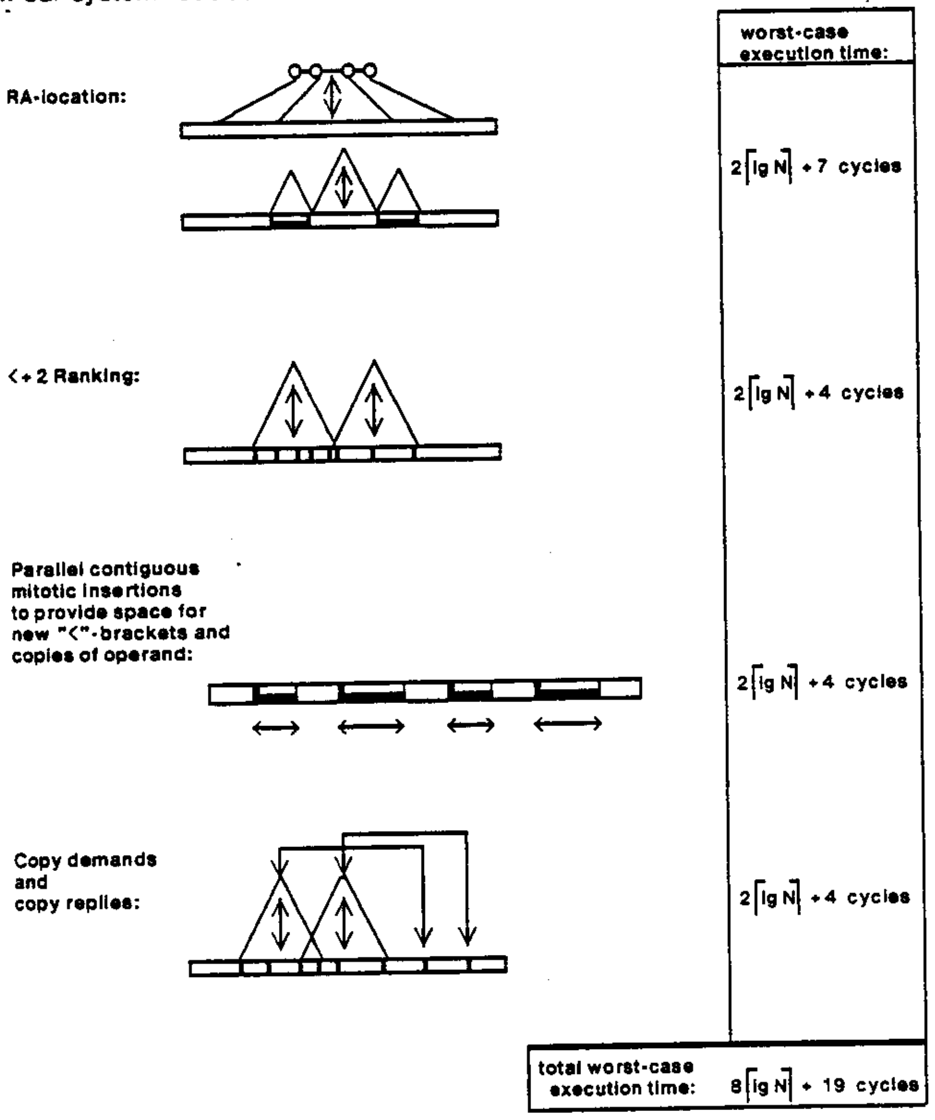
The following figures show how our system implements the primitive functional meta operators and operators of our example reduction language.

Our system's internal representation of a meta reduction involving NF:



< NF (NF f_1 ... f_m (a_1 ... a_n
0 1 1 2 2 2 2 3 3
(< f_1 (a_1 ... a_n ... < f_m (a_1 ... a_n
0 1 2 2 3 3 1 2 2 3 3

How our system reduces such an instance:



worst-case execution time:
RA-location:
2⌈lg N⌉ + 7 cycles
<+2 Ranking:
2⌈lg N⌉ + 4 cycles
Parallel contiguous mitotic insertions to provide space for new "<"-brackets and copies of operand:
2⌈lg N⌉ + 4 cycles
Copy demands and copy replies:
2⌈lg N⌉ + 4 cycles
total worst-case execution time:
8⌈lg N⌉ + 19 cycles

Fig. B-1: Execution for primitive meta operator NF

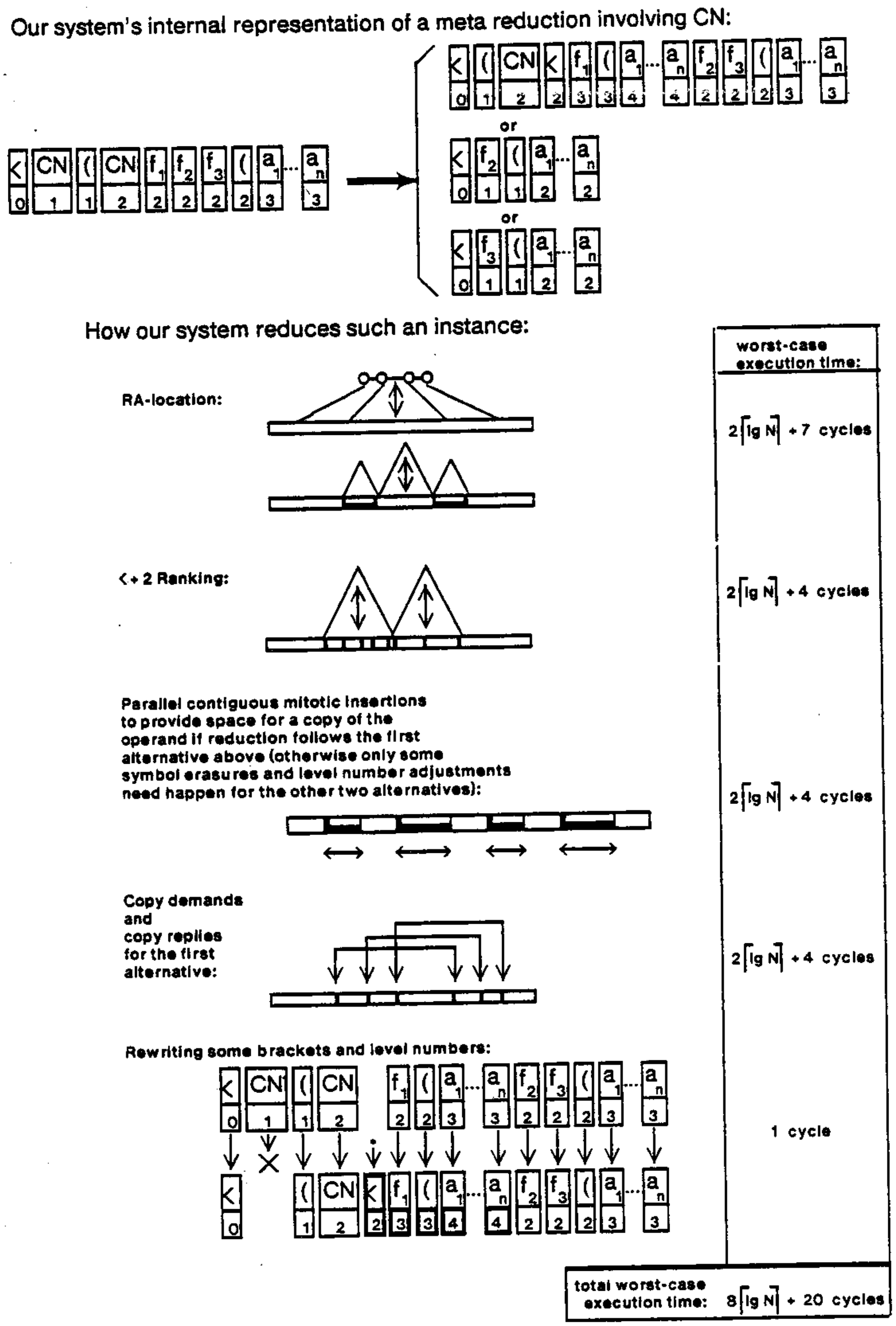

Our system's internal representation of a meta reduction involving CN:
or
or
How our system reduces such an instance:
worst-case execution time:
RA-location:
2⌈lg N⌉ + 7 cycles
< + 2 Ranking:
2⌈lg N⌉ + 4 cycles
Parallel contiguous mitotic insertions to provide space for a copy of the operand if reduction follows the first alternative above (otherwise only some symbol erasures and level number adjustments need happen for the other two alternatives):
2⌈lg N⌉ + 4 cycles
Copy demands and copy replies for the first alternative:
2⌈lg N⌉ + 4 cycles
Rewriting some brackets and level numbers:
1 cycle
total worst-case execution time: 8⌈lg N⌉ + 20 cycles

Fig. B-2: Execution for primitive meta operator CN

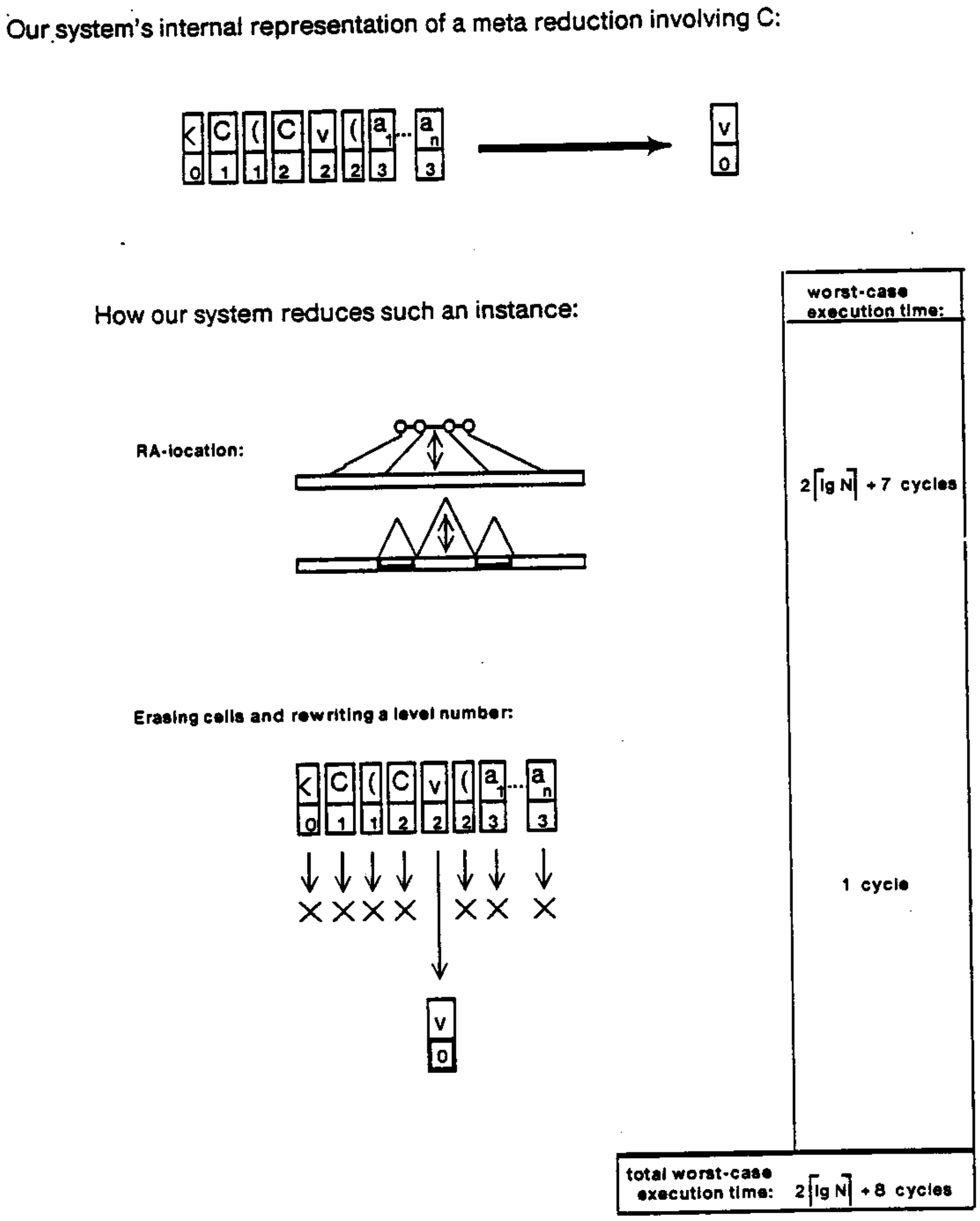
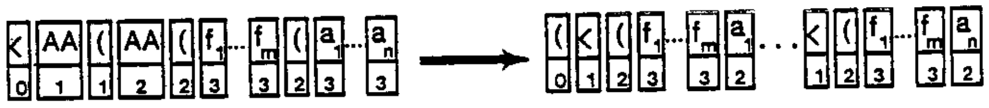

Our system's internal representation of a meta reduction involving C:
How our system reduces such an instance:
worst-case execution time:
RA-location:
2⌈lg N⌉ + 7 cycles
Erasing cells and rewriting a level number:
1 cycle
total worst-case execution time:
2⌈lg N⌉ + 8 cycles

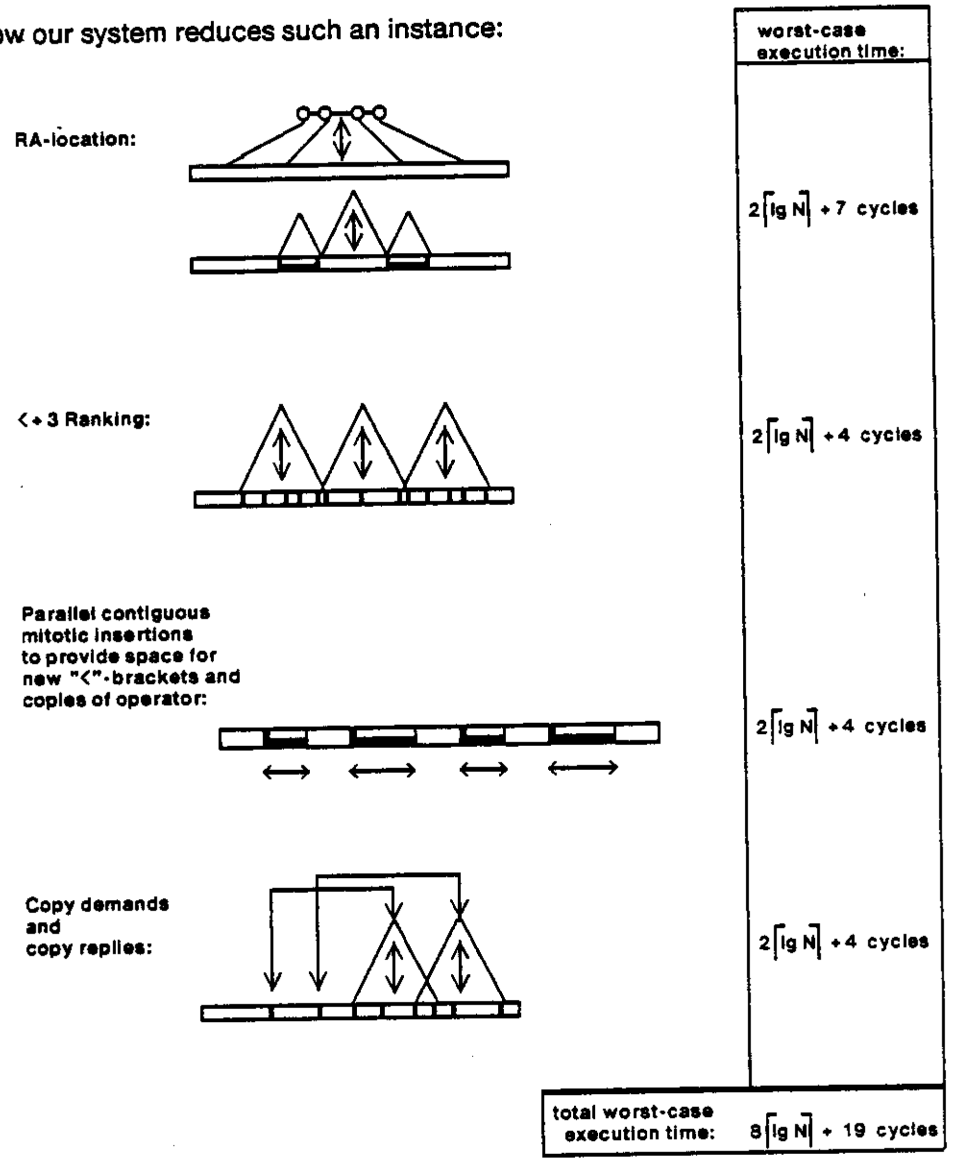
Fig. B-3: Execution for primitive meta operator C

Our system's internal representation of a meta reduction involving AA:



< AA (AA (f1 ... fm (a1 ... an
0 1 1 2 2 3 3 2 3 3
(< (f1 ... fm a1 . . . < (f1 ... fm an
0 1 2 3 3 2 1 2 3 3 2

How our system reduces such an instance:



worst-case execution time:
RA-location:
$2\lceil \lg N \rceil + 7$ cycles
<+3 Ranking:
$2\lceil \lg N \rceil + 4$ cycles
Parallel contiguous mitotic insertions to provide space for new "<"-brackets and copies of operator:
$2\lceil \lg N \rceil + 4$ cycles
Copy demands and copy replies:
$2\lceil \lg N \rceil + 4$ cycles
total worst-case execution time:
$8\lceil \lg N \rceil + 19$ cycles

Fig. B-4: Execution for primitive meta operator AA

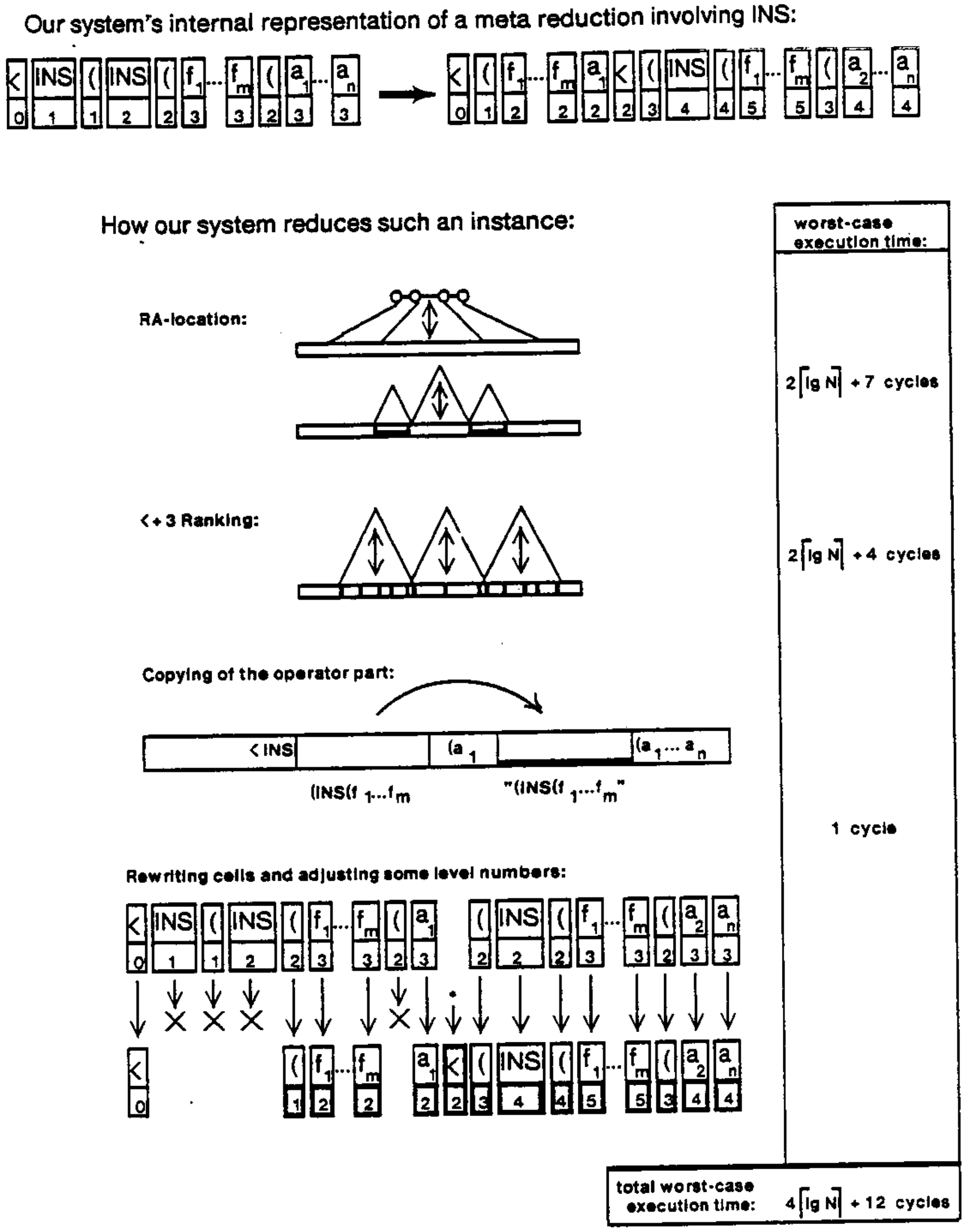
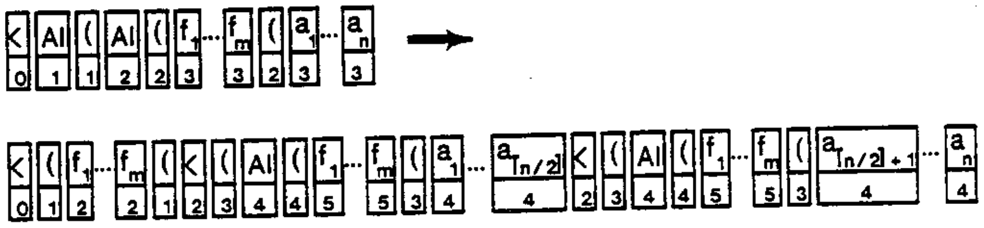
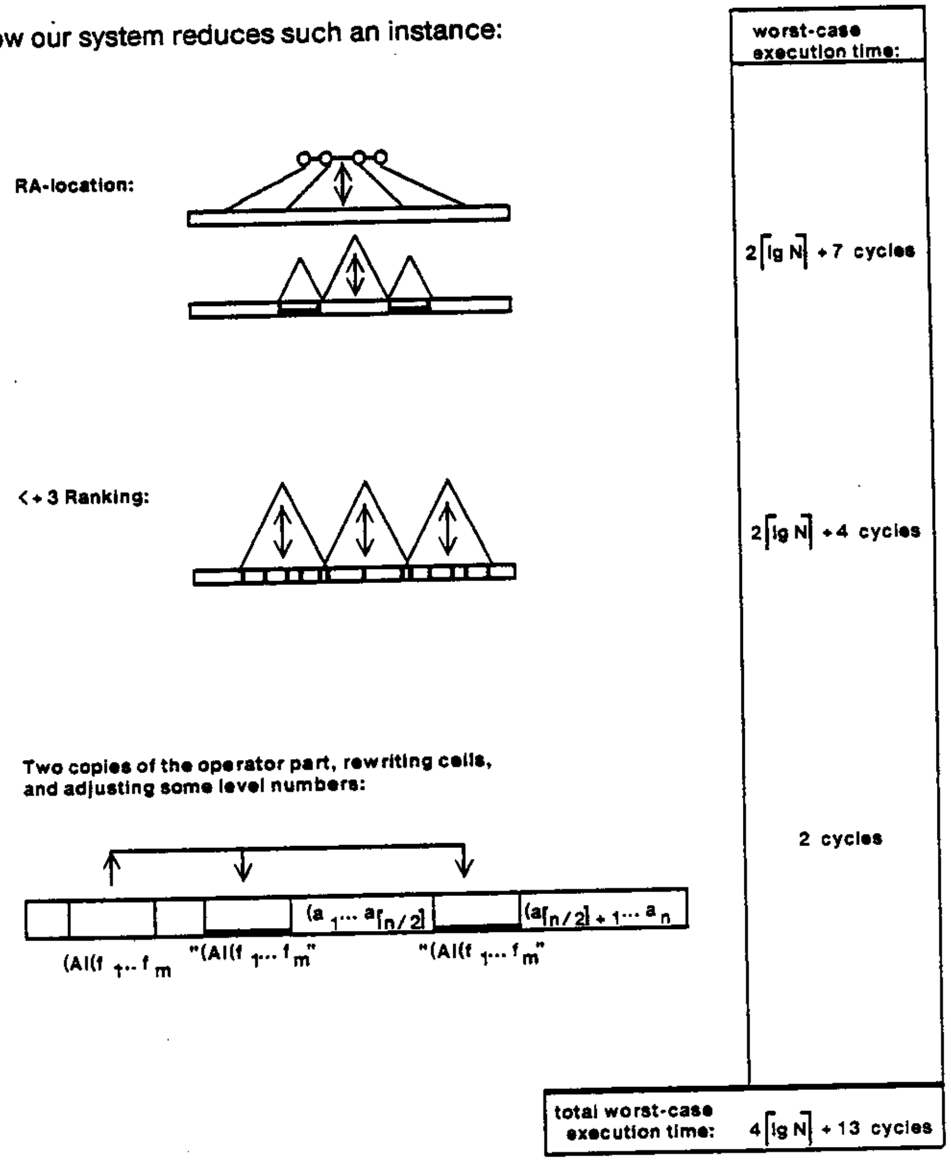

Our system's internal representation of a meta reduction involving INS:
How our system reduces such an instance:
RA-location:
<+3 Ranking:
Copying of the operator part:
<INS
(a 1
(a 1... a n
(INS(f 1...f m
"(INS(f 1...f m"
Rewriting cells and adjusting some level numbers:
worst-case execution time:
2⌈lg N⌉ + 7 cycles
2⌈lg N⌉ + 4 cycles
1 cycle
total worst-case execution time:
4⌈lg N⌉ + 12 cycles

Fig. B-5: Execution for primitive meta operator INS

Our system's internal representation of a meta reduction involving AI:



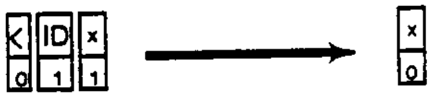
How our system reduces such an instance:



worst-case execution time:
RA-location:
$2\lceil \lg N \rceil + 7$ cycles
<+3 Ranking:
$2\lceil \lg N \rceil + 4$ cycles
Two copies of the operator part, rewriting cells, and adjusting some level numbers:
2 cycles
(AI(f 1.. f m
"(AI(f 1... f m"
(a 1... a⌈n/2⌉
"(AI(f 1... f m"
(a⌈n/2⌉+1... a n
total worst-case execution time:
$4\lceil \lg N \rceil + 13$ cycles

Fig. B-6: Execution for primitive meta operator AI

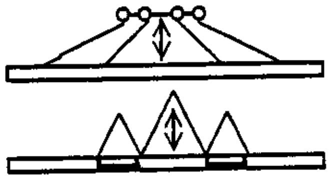
Our system's internal representation of a regular reduction involving ID:



How our system reduces such an instance:

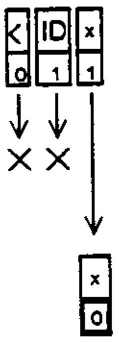
worst-case execution time:

RA-location:



$2\lceil \lg N \rceil + 7$ cycles

Erasing cells and rewriting a level number:

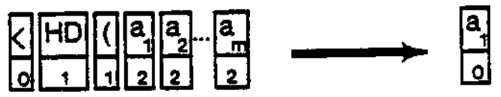


1 cycle

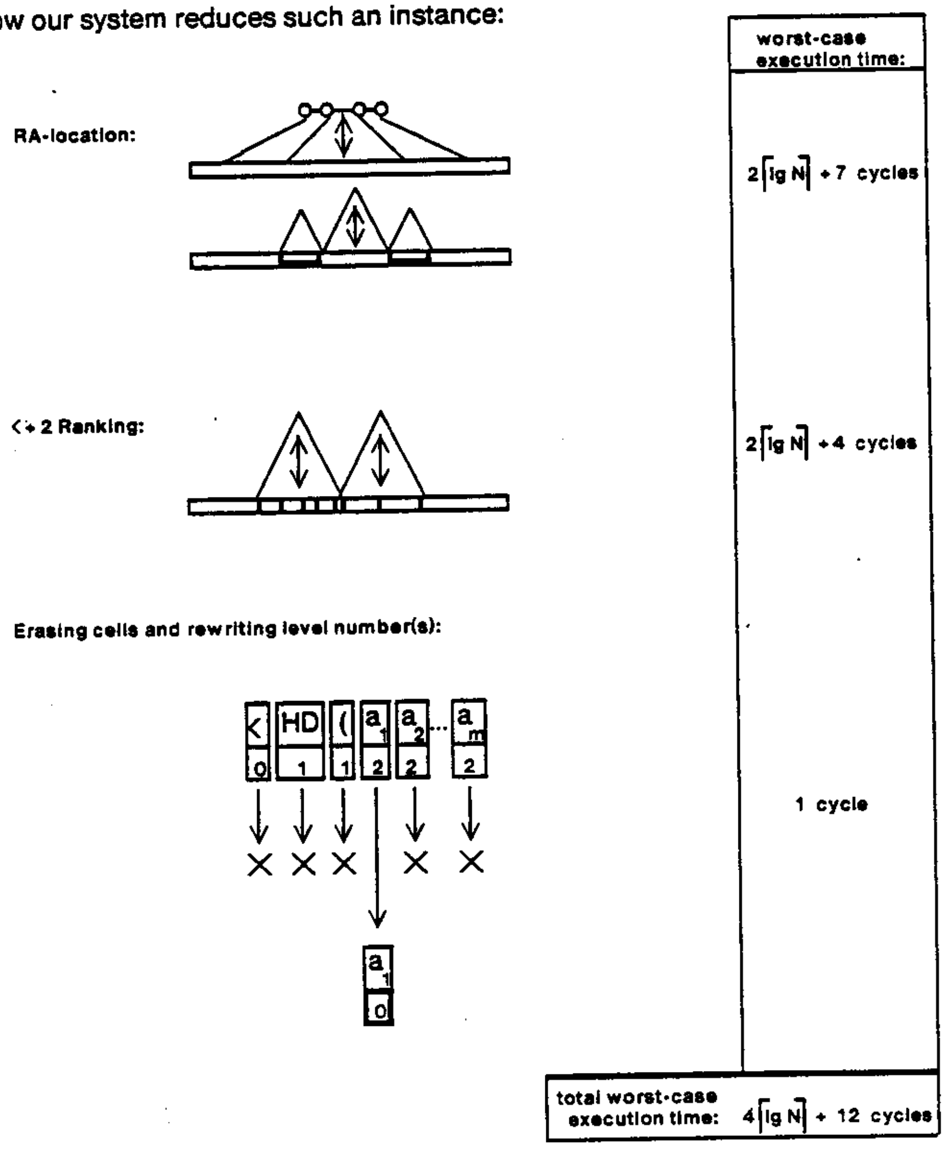
total worst-case execution time: $2\lceil \lg N \rceil + 8$ cycles

Fig. B-7: Execution for primitive regular operator ID .

Our system's internal representation of a regular reduction involving HD:



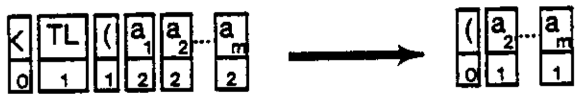
How our system reduces such an instance:



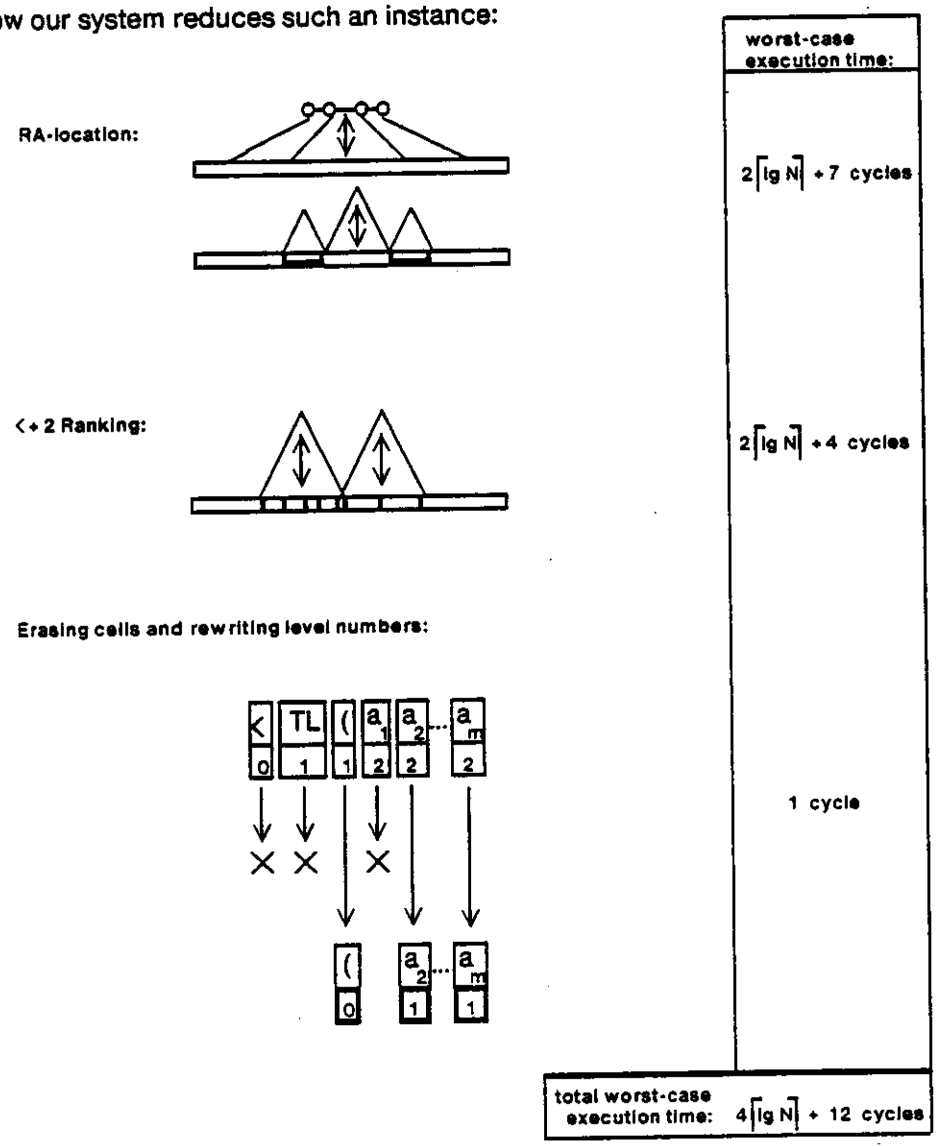
worst-case execution time:
RA-location:
$2\lceil \lg N \rceil + 7$ cycles
<+2 Ranking:
$2\lceil \lg N \rceil + 4$ cycles
Erasing cells and rewriting level number(s):
<
HD
(
a_1
a_2
...
a_m
0
1
1
2
2
2
a_1
0
1 cycle
total worst-case execution time:
$4\lceil \lg N \rceil + 12$ cycles

Fig. B-8: Execution for primitive regular operator HD

Our system's internal representation of a regular reduction involving TL:



How our system reduces such an instance:



worst-case execution time:
RA-location:
$2\lceil \lg N \rceil + 7$ cycles
< + 2 Ranking:
$2\lceil \lg N \rceil + 4$ cycles
Erasing cells and rewriting level numbers:
1 cycle
total worst-case execution time: $4\lceil \lg N \rceil + 12$ cycles

Fig. B-9: Execution for primitive regular operator TL

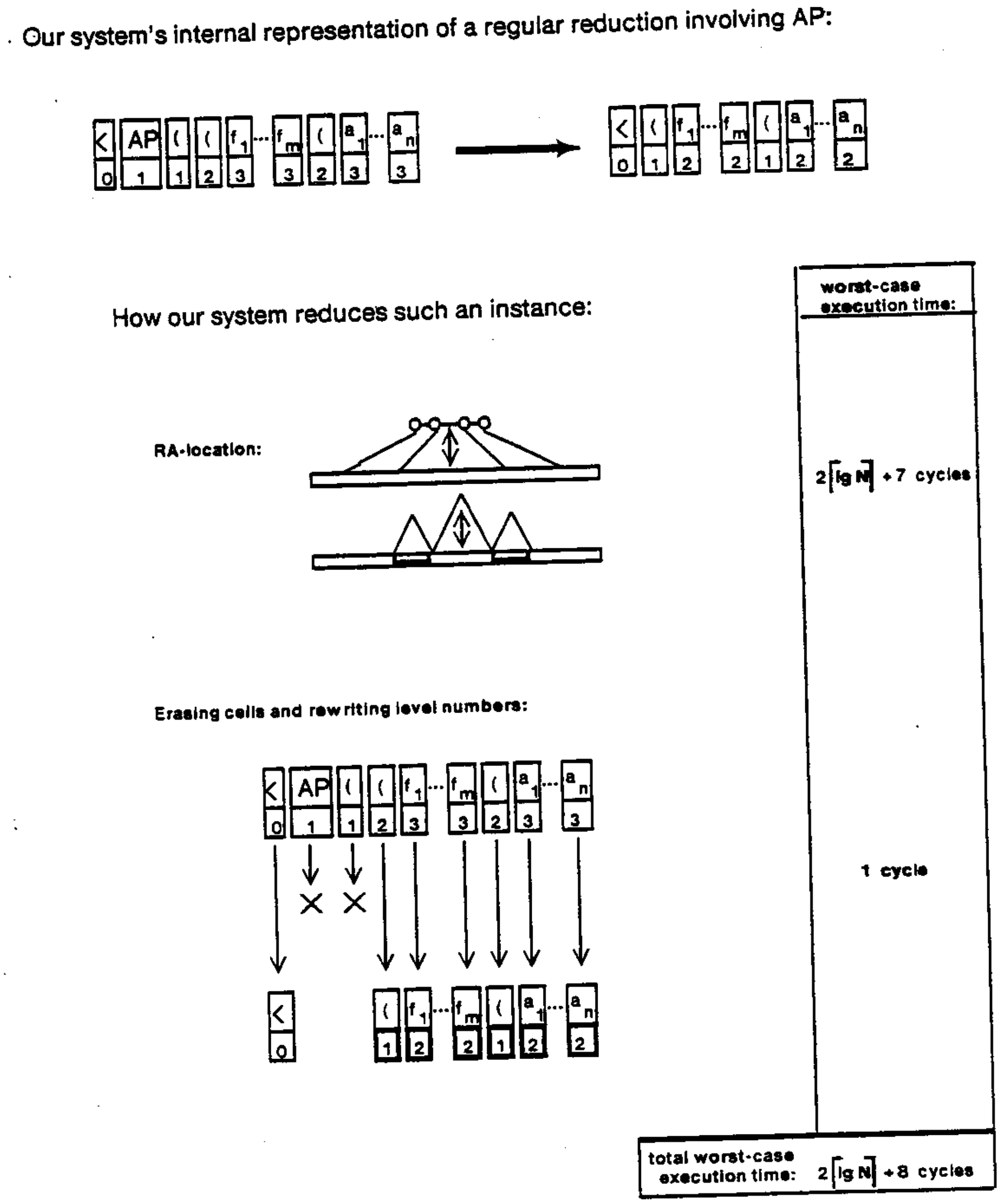
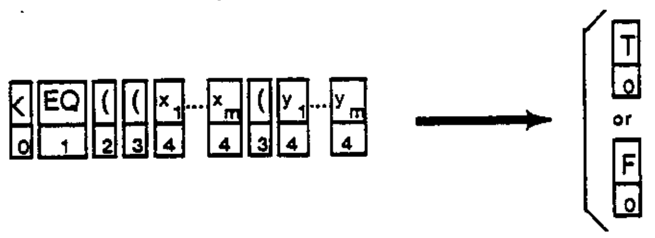
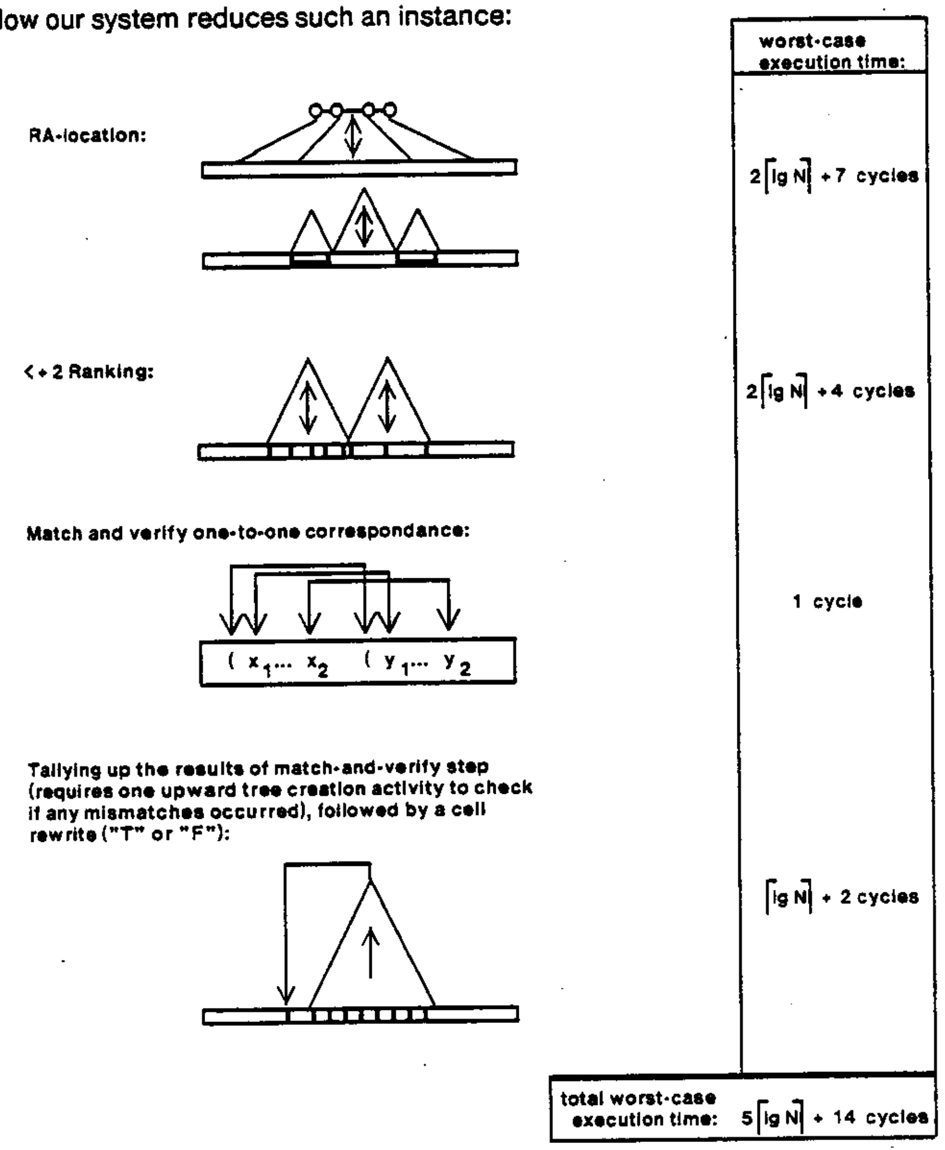

Our system's internal representation of a regular reduction involving AP:
< AP ((f1 ... fm (a1 ... an
0 1 1 2 3 3 2 3 3
< (f1 ... fm (a1 ... an
0 1 2 2 1 2 2
How our system reduces such an instance:
worst-case execution time:
RA-location:
2⌈lg N⌉ +7 cycles
Erasing cells and rewriting level numbers:
1 cycle
total worst-case execution time: 2⌈lg N⌉ +8 cycles

Fig. B-10: Execution for primitive regular operator AP

Our system's internal representation of a regular reduction involving EQ:



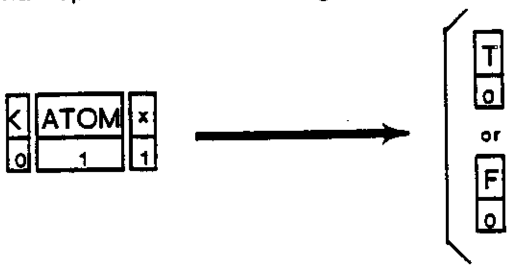
How our system reduces such an instance:



worst-case execution time:
RA-location:
$2\lceil \lg N \rceil + 7$ cycles
<+2 Ranking:
$2\lceil \lg N \rceil + 4$ cycles
Match and verify one-to-one correspondance:
(x_1... x_2 (y_1... y_2
1 cycle
Tallying up the results of match-and-verify step (requires one upward tree creation activity to check if any mismatches occurred), followed by a cell rewrite ("T" or "F"):
$\lceil \lg N \rceil + 2$ cycles
total worst-case execution time: $5\lceil \lg N \rceil + 14$ cycles

Fig. B-11: Execution for primitive regular operator EQ

Our system's internal representation of a regular reduction involving ATOM:



How our system reduces such an instance:

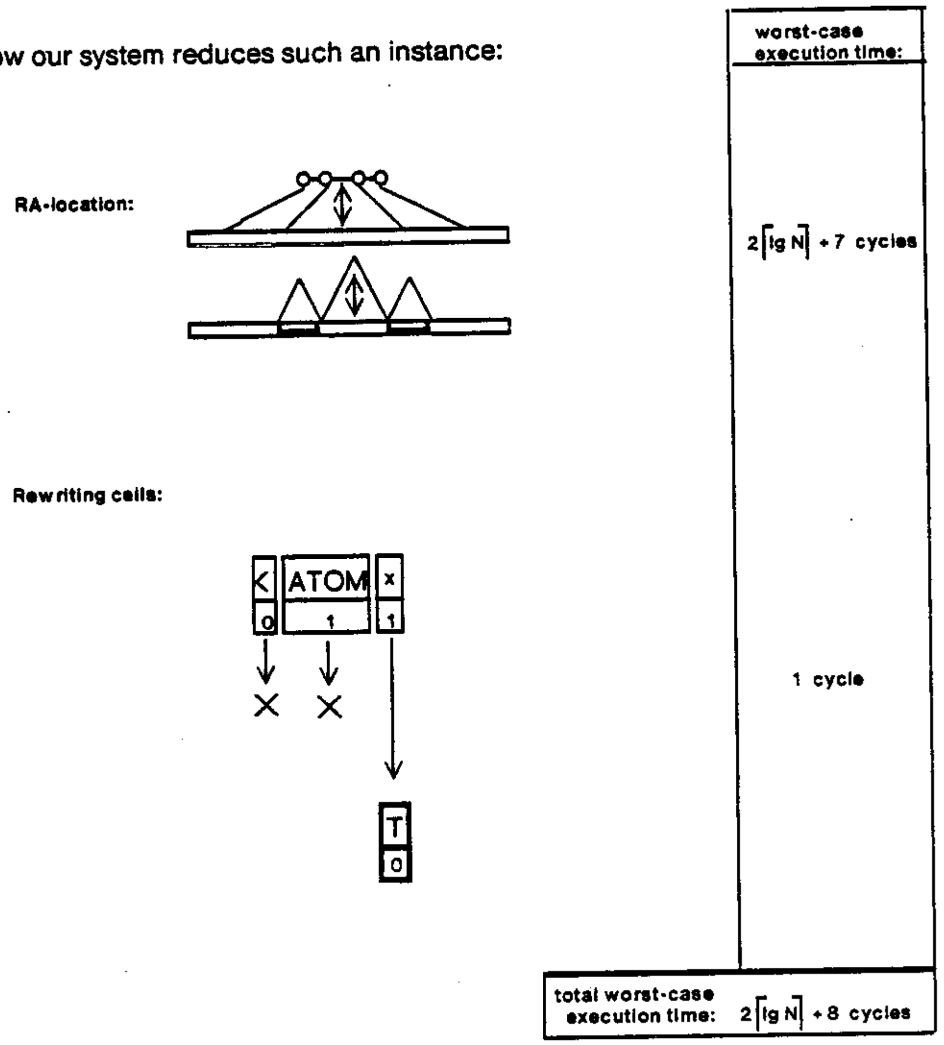
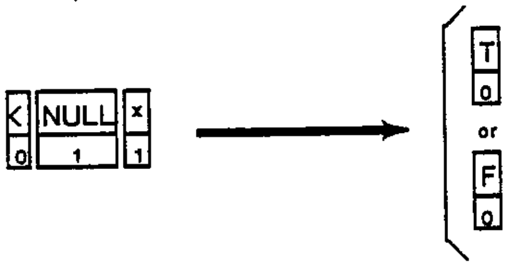

worst-case execution time:
RA-location:
2⌈lg N⌉ + 7 cycles
Rewriting cells:
K
ATOM
x
0
1
1
T
0
1 cycle
total worst-case execution time: 2⌈lg N⌉ + 8 cycles

Fig. B-12: Execution for primitive regular operator ATOM

Our system's internal representation of a regular reduction involving NULL:



How our system reduces such an instance:

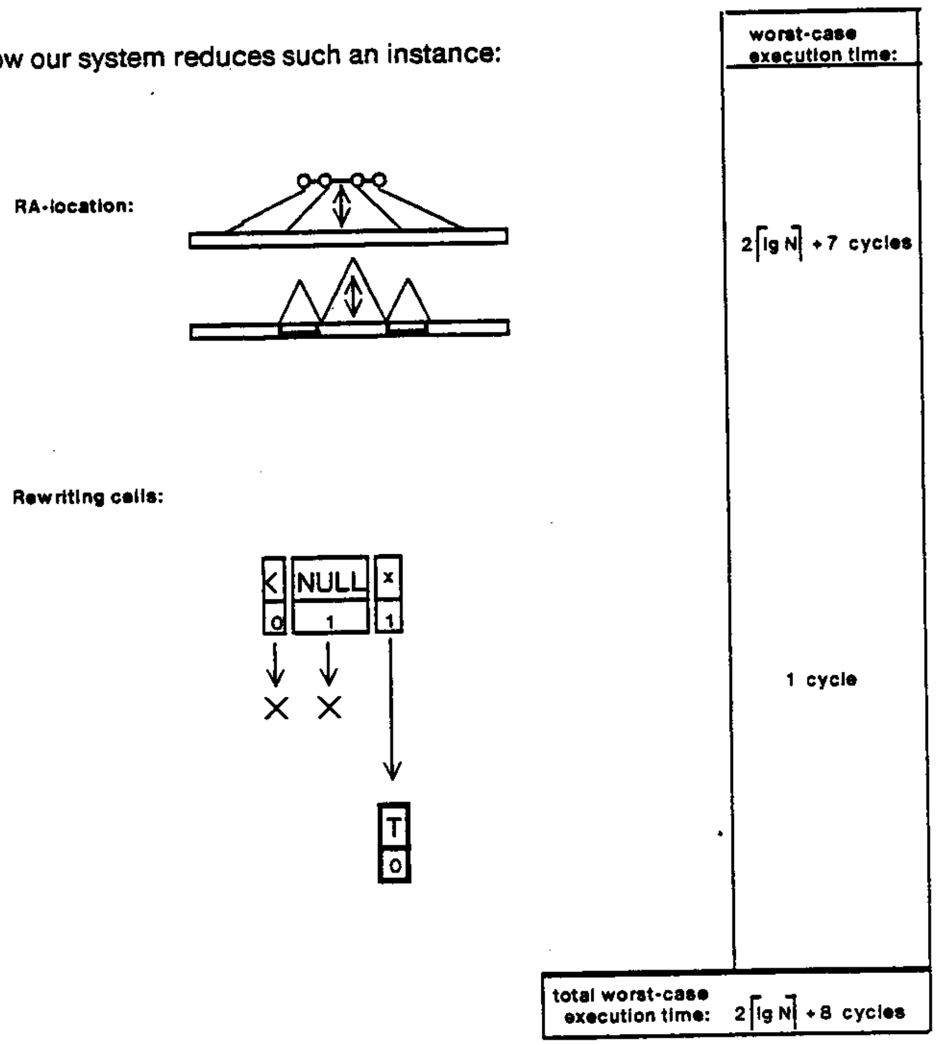
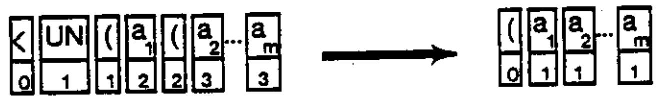
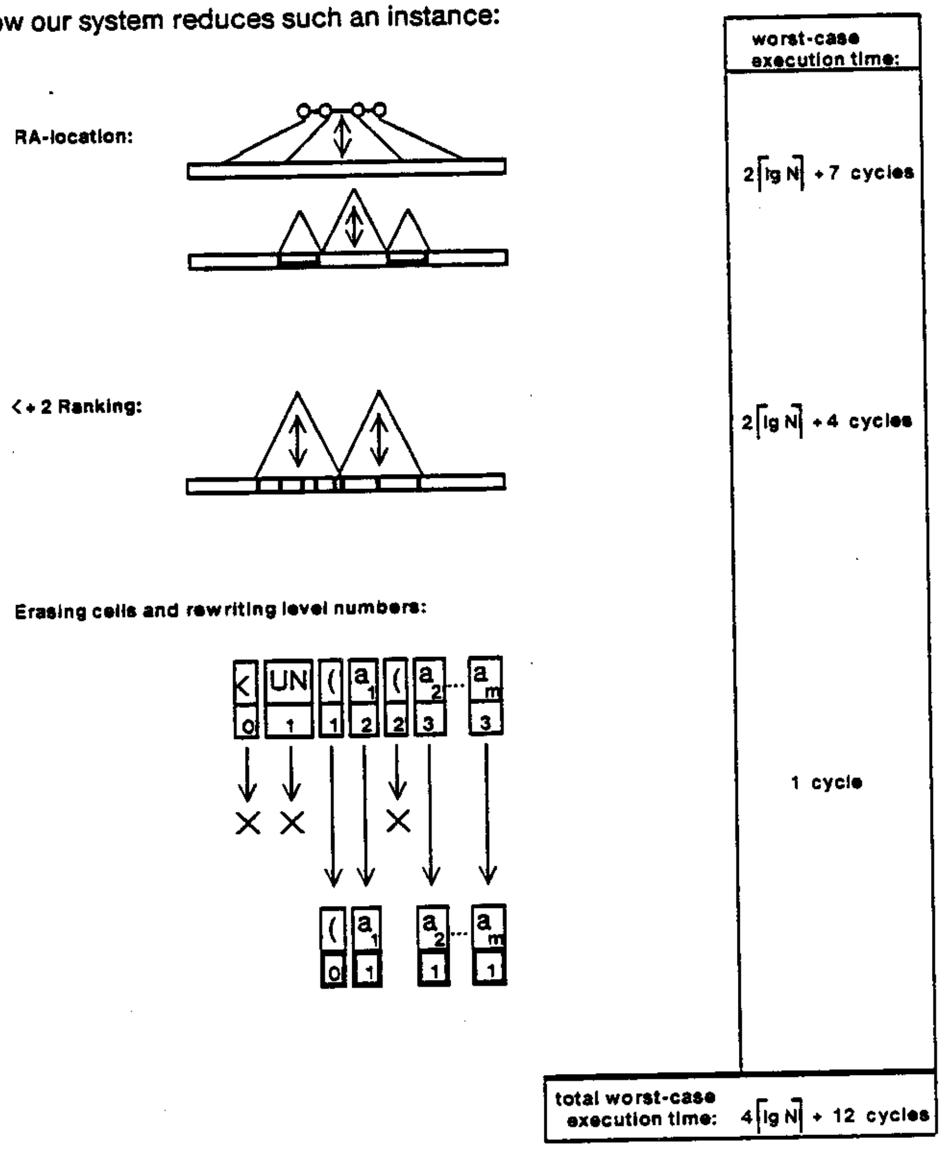

worst-case execution time:
RA-location:
2⌈lg N⌉ +7 cycles
Rewriting cells:
K
0
NULL
1
x
1
T
0
1 cycle
total worst-case execution time: 2⌈lg N⌉ +8 cycles

Fig. B-13: Execution for primitive regular operator NULL.

Our system's internal representation of a regular reduction involving UN:



How our system reduces such an instance:



RA-location:
<+2 Ranking:
Erasing cells and rewriting level numbers:
worst-case execution time:
2⌈lg N⌉ + 7 cycles
2⌈lg N⌉ + 4 cycles
1 cycle
total worst-case execution time: 4⌈lg N⌉ + 12 cycles

Fig. B-14: Execution for primitive regular operator UN

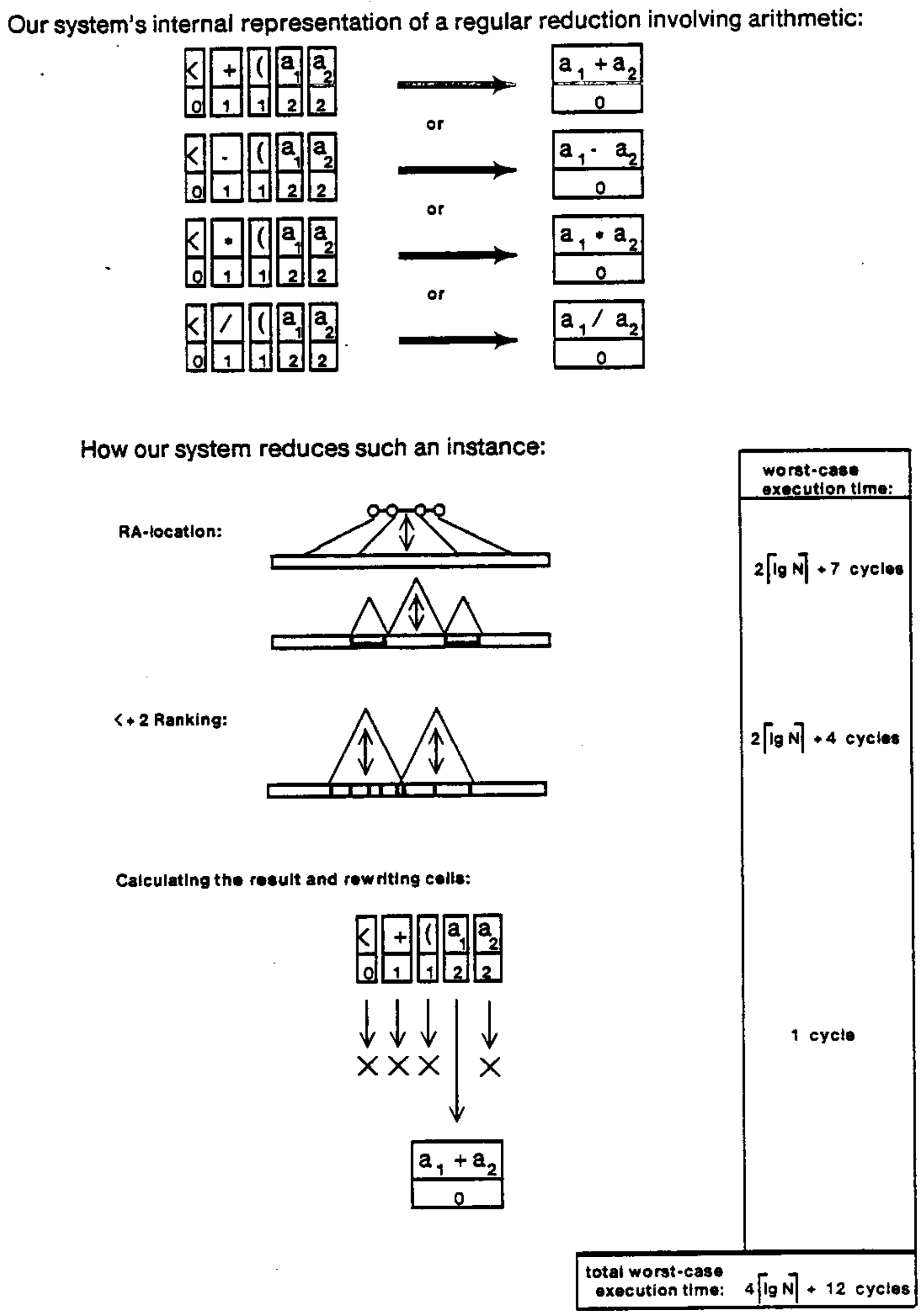
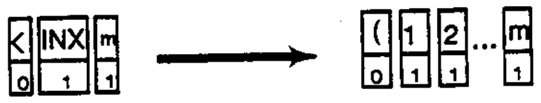

Our system's internal representation of a regular reduction involving arithmetic:
or
or
or
How our system reduces such an instance:
RA-location:
<+2 Ranking:
Calculating the result and rewriting cells:
worst-case execution time:
2⌈lg N⌉ + 7 cycles
2⌈lg N⌉ + 4 cycles
1 cycle
total worst-case execution time: 4⌈lg N⌉ + 12 cycles

Fig. B-15: Execution for primitive regular arithmetic operator

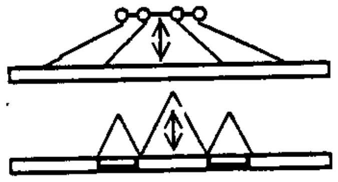
Our system's internal representation of a regular reduction involving INX:



How our system reduces such an instance:

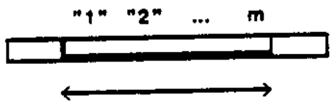
worst-case execution time:

RA-location:



$2\lceil \lg N \rceil + 7$ cycles

Mitotic insertions for the correct number of numbered cells:

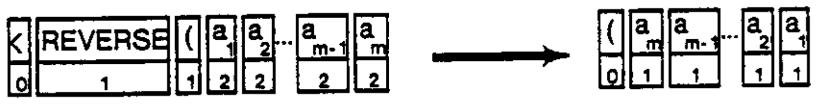


$2\lceil \lg N \rceil + 4$ cycles

total worst-case execution time: $4\lceil \lg N \rceil + 11$ cycles

Fig. B-16: Execution for primitive regular operator INX

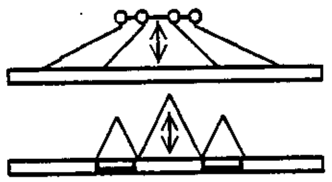
Our system's internal representation of a regular reduction involving REVERSE:



How our system reduces such an instance:

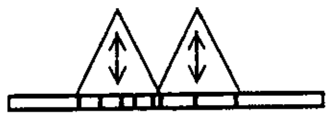
worst-case execution time:

RA-location:



$2\lceil \lg N \rceil + 7$ cycles

< + 2 Ranking:



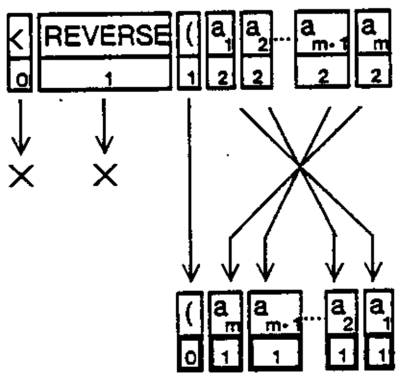
$2\lceil \lg N \rceil + 4$ cycles

Erasing cells, rewriting level numbers, and permutation of cells:

(After < + 2 Ranking each cell has enough information to determine the new positional location of its element in a reversed sequence. If each cell now forms its new CDA by concatenating:

PCA , new element position , old cell PA

each cell can move to its proper new location.)

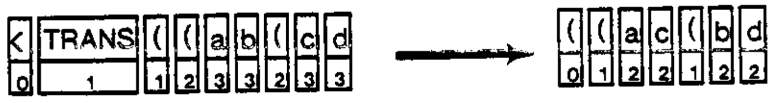


1 cycle

total worst-case execution time: $4\lceil \lg N \rceil + 12$ cycles

Fig. B-17: Execution for primitive regular operator REVERSE

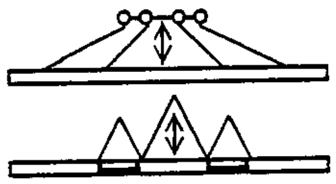
Our system's internal representation of a regular reduction involving TRANS:



How our system reduces such an instance:

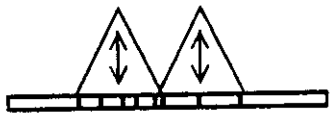
worst-case execution time:

RA-location:



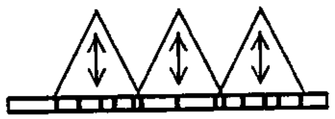
$2\lceil \lg N \rceil + 7$ cycles

<+2 Ranking:



$2\lceil \lg N \rceil + 4$ cycles

<+3 Ranking:



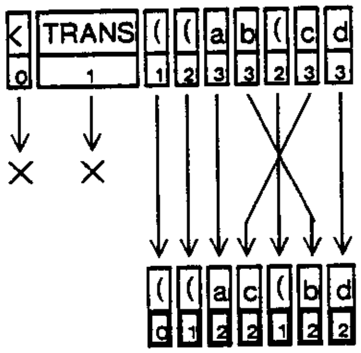
$2\lceil \lg N \rceil + 4$ cycles

Erasing cells, rewriting level numbers, and permutation of cells:

(After <+2 and <+3 Ranking each cell has enough information to determine the new positional location of its element in a transposed matrix, even when this matrix is represented as a linear array. If each cell now forms its new CDE by concatenating:

PCA , new row , new column , old cell PA

each cell can move to its proper new location.)



1 cycle

total worst-case execution time: $6\lceil \lg N \rceil + 16$ cycles

Fig. B-18: Execution for primitive regular operator TRANS

Appendix C: Some Additional Primitive Functions

(Note: these all happen to be regular operators.)

SUN (Set UNion)

$\langle SUN,x \rangle$ reduces to

$(z_1,z_2, \ldots ,z_p)$ if $x=((x_1,x_2, \ldots ,x_m),(y_1,y_2, \ldots ,y_n))$,

and where $\{z_i | i \leq p \leq m+n\} = \{x_i | i \leq m\} \cup \{y_i | i \leq n\}$

(with all duplicates removed),

otherwise it reduces to $\perp$.

SINT (Set INTersection)

$\langle SINT,x \rangle$ reduces to

$(z_1,z_2, \ldots ,z_p)$ if $x=((x_1,x_2, \ldots ,x_m),(y_1,y_2, \ldots ,y_n))$,

and where $\{z_i | i \leq p \leq m+n\} = \{x_i | i \leq m\} \cap \{y_i | i \leq n\}$

(with all duplicates removed),

otherwise it reduces to $\perp$.

SHUF (SHUFfle permutation)

<SHUF,x> reduces to

$(x_1,x_{(m/2)+1},x_2,x_{(m/2)+2}, \ldots ,x_{m/2},x_m)$ if $x=(x_1,x_2, \ldots ,x_m)$, $m>1$, and m is even,

$(\ )$ if $x=(\ )$,

otherwise it reduces to $\perp$.

Comment: This example permutation demonstrates how easily we might use this execution methodology to implement any rank-calculable permutation.

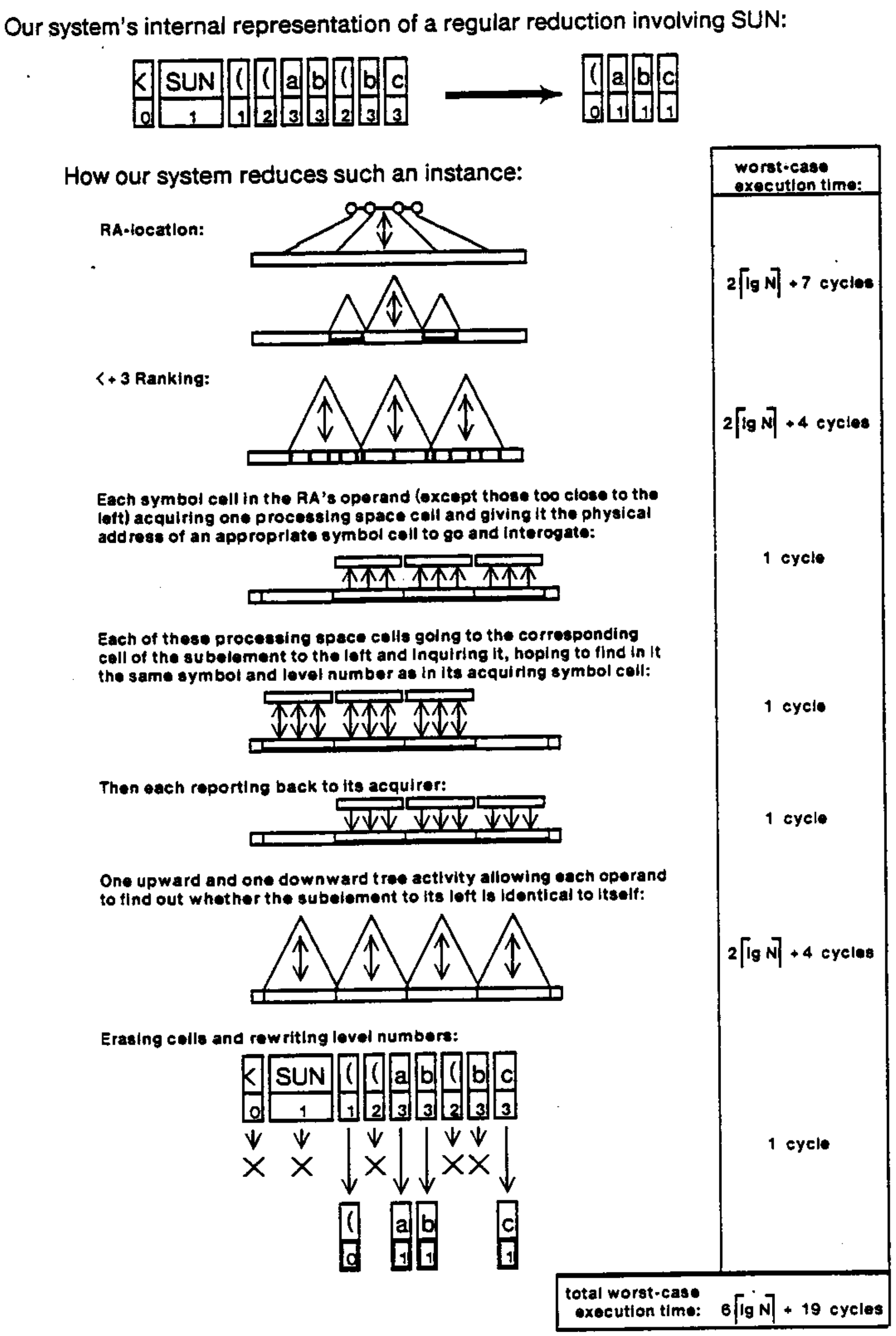
SORTF (SORT based on First subelement indices)

<SORTF,x> reduces to

$((b_1,x_{b_1}),(b_2,x_{b_2}), \ldots ,(b_m,x_{bm}))$ if $x=((a_1,x_{a_1}),(a_2,x_{a_2}), \ldots ,(a_m,x_{a_m}))$,

where $(b_1,b_2, \ldots ,b_m)$ is $(a_1,a_2, \ldots ,a_m)$ sorted into ascending order,

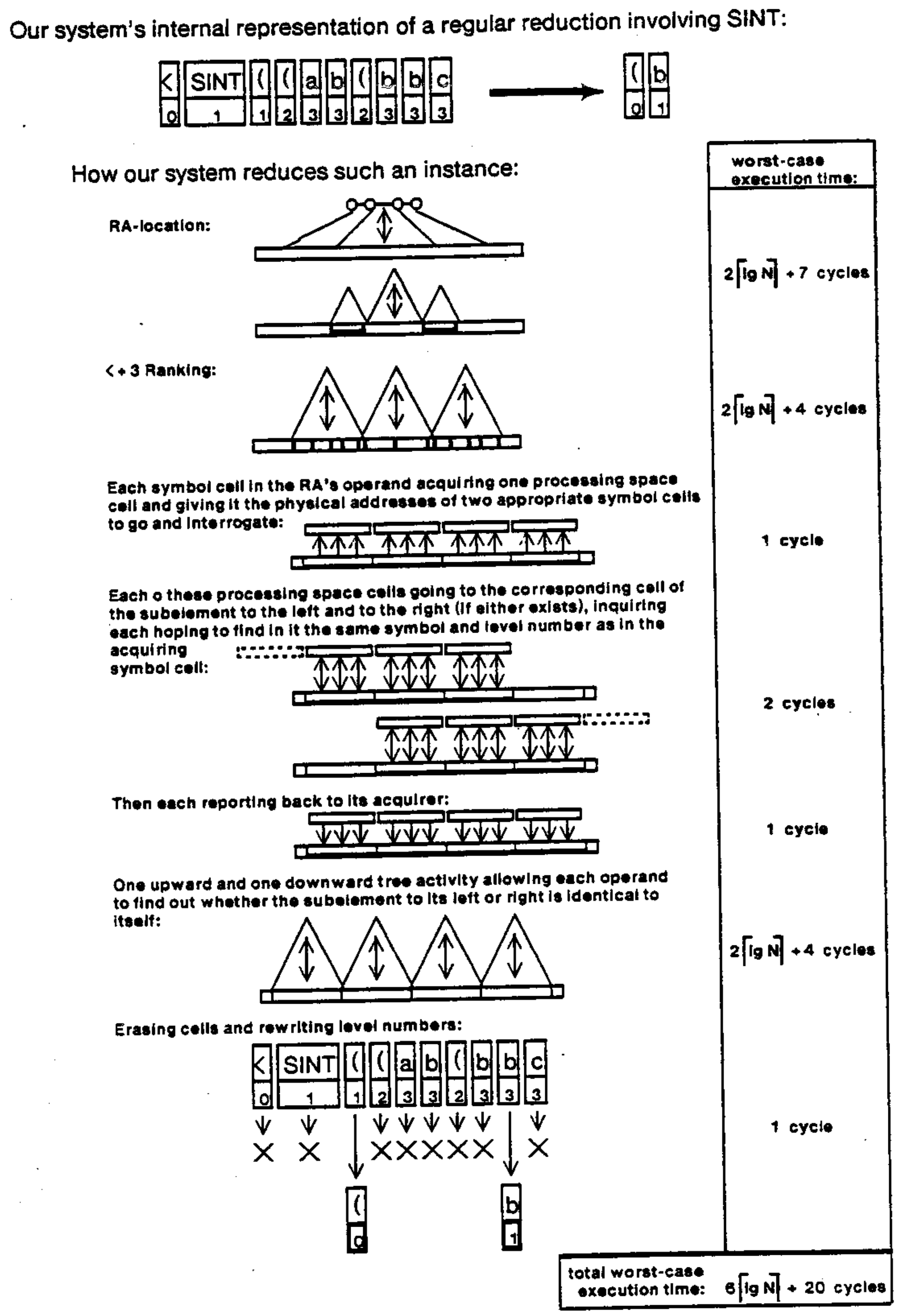
otherwise it reduces to $\perp$.


Our system's internal representation of a regular reduction involving SUN:

How our system reduces such an instance:

	worst-case execution time:
RA-location:	2⌈lg N⌉ + 7 cycles
< + 3 Ranking:	2⌈lg N⌉ + 4 cycles
Each symbol cell in the RA's operand (except those too close to the left) acquiring one processing space cell and giving it the physical address of an appropriate symbol cell to go and interogate:	1 cycle
Each of these processing space cells going to the corresponding cell of the subelement to the left and inquiring it, hoping to find in it the same symbol and level number as in its acquiring symbol cell:	1 cycle
Then each reporting back to its acquirer:	1 cycle
One upward and one downward tree activity allowing each operand to find out whether the subelement to its left is identical to itself:	2⌈lg N⌉ + 4 cycles
Erasing cells and rewriting level numbers:	1 cycle
total worst-case execution time:	6⌈lg N⌉ + 19 cycles

Fig. C-1: Execution for primitive regular operator SUN


Our system's internal representation of a regular reduction involving SINT:

How our system reduces such an instance:

RA-location:

<+3 Ranking:

Each symbol cell in the RA's operand acquiring one processing space cell and giving it the physical addresses of two appropriate symbol cells to go and interrogate:

Each o these processing space cells going to the corresponding cell of the subelement to the left and to the right (if either exists), inquiring each hoping to find in it the same symbol and level number as in the acquiring symbol cell:

Then each reporting back to its acquirer:

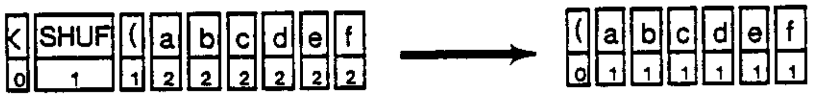
One upward and one downward tree activity allowing each operand to find out whether the subelement to its left or right is identical to itself:

Erasing cells and rewriting level numbers:

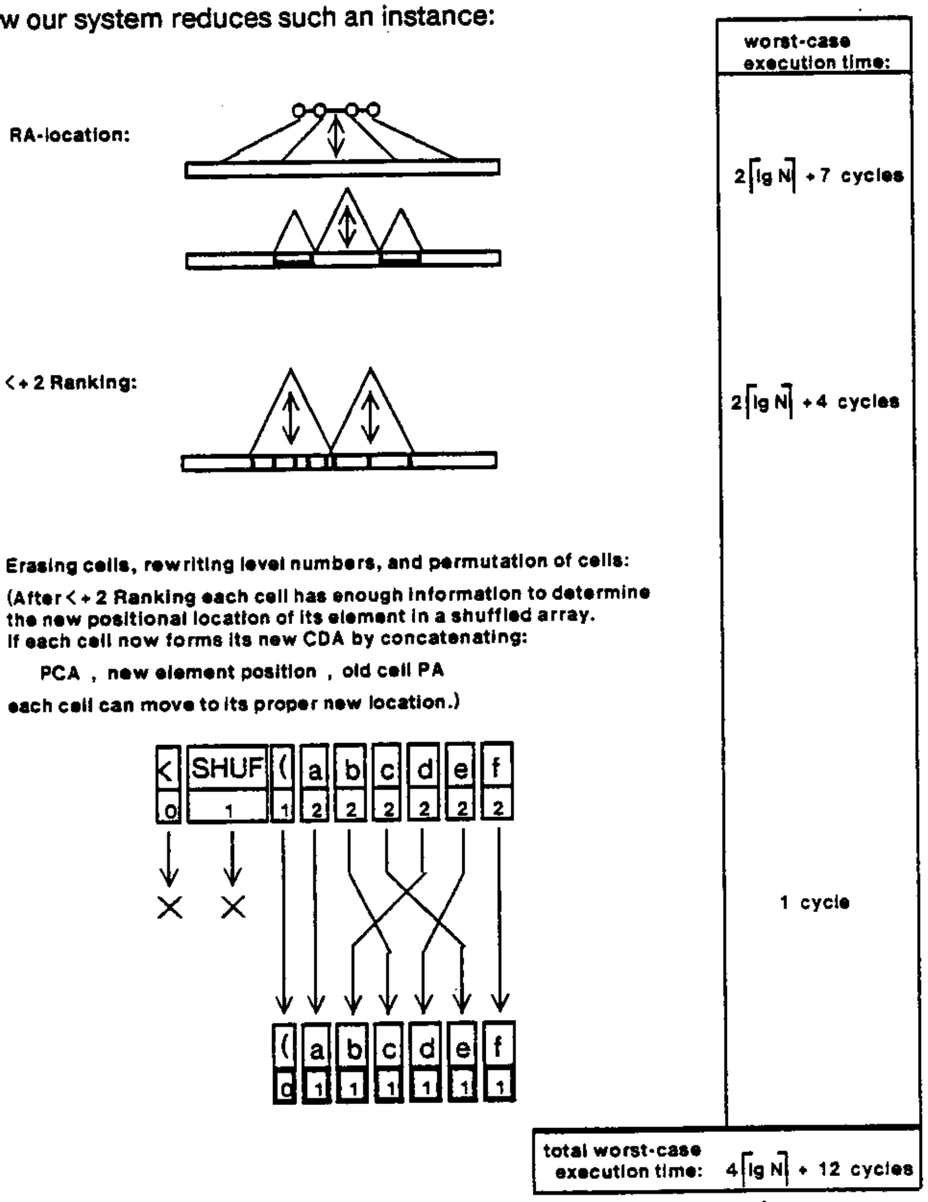
worst-case execution time:
$2\lceil \lg N \rceil + 7$ cycles
$2\lceil \lg N \rceil + 4$ cycles
1 cycle
2 cycles
1 cycle
$2\lceil \lg N \rceil + 4$ cycles
1 cycle
total worst-case execution time: $6\lceil \lg N \rceil + 20$ cycles

Fig. C-2: Execution for primitive regular operator SINT

Our system's internal representation of a regular reduction involving SHUF:



How our system reduces such an instance:



worst-case execution time:
RA-location:
2⌈lg N⌉ + 7 cycles
< + 2 Ranking:
2⌈lg N⌉ + 4 cycles
1 cycle
total worst-case execution time: 4⌈lg N⌉ + 12 cycles

Erasing cells, rewriting level numbers, and permutation of cells:

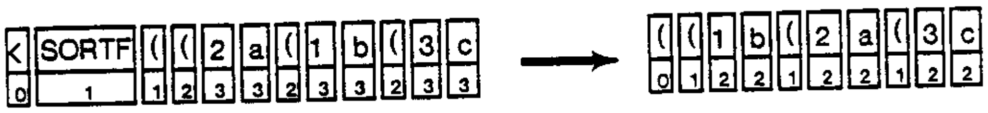
(After < + 2 Ranking each cell has enough information to determine the new positional location of its element in a shuffled array.
If each cell now forms its new CDA by concatenating:

PCA , new element position , old cell PA

each cell can move to its proper new location.)

Fig. C-3: Execution for primitive regular operator SHUF

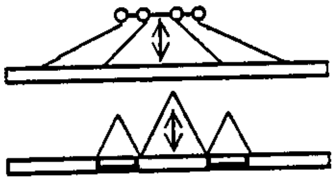
Our system's internal representation of a regular reduction involving SORTF:



How our system reduces such an instance:

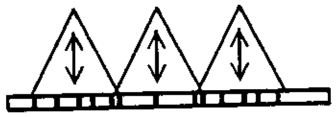
worst-case execution time:

RA-location:



2⌈lg N⌉ + 7 cycles

Each parenthesis at exactly the < + 2 level remembering the value of the cell to its immediate right in this RA (the element's elemental sort index), followed by a < + 3 Ranking which also propagates this remembered value to all the other leaves in the same tree (i.e., in the same element):

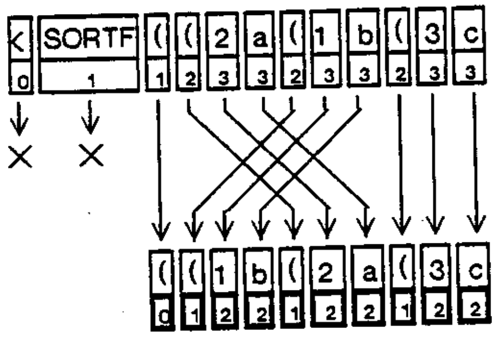


2⌈lg N⌉ + 4 cycles

Erasing cells, rewriting level numbers, and permutation of cells:
(After < + 2 Ranking each cell has enough information to determine the new positional location of its element in a sorted array.
If each cell now forms its new CDA by concatenating:

LCA + 3 , elemental sort index , old cell PA

each cell can move to its proper new location.)



1 cycle

total worst-case execution time: 4⌈lg N⌉ + 12 cycles

Fig. C-4: Execution for primitive regular operator SORTF