PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

CELL AND ENSEMBLE ARCHITECTURE FOR THE REWRITE RULE MACHINE*

Sany Leinwand, Joseph A. Goguen† and Timothy Winkler
SRI International, Menlo Park CA 94025

Abstract

The Rewrite Rule Machine (RRM) project at SRI International combines advanced architectural concepts with modern software technology. This combination is based on a novel model of computation called *concurrent term rewriting*, bridging the gap between ultra high level programming and massively concurrent execution.

The RRM architecture may be described at four different levels of granularity. The lowest level is the *cell*, which stores an individual data token. The second level is the *ensemble*, which organizes many cells to represent terms and applies rewrite rules to them under the direction of a single common controller. Each ensemble is planned for implementation as a single custom VLSI chip. The *cluster* level organizes many ensembles to cooperate in solving common problems. Finally, the whole RRM consists of several clusters interconnected in a *network*. Ensembles are ideal for fine-grained concurrency, while clusters also allow coarse-grained concurrency. This *multi-grain concurrency* allows the RRM to exploit the local homogeneity of typical problems.

This paper presents recent results of the Rewrite Rule Machine project. Some topics described in previous papers [3,4,12,8] are only summarized. The focus is on recent results detailing the operation of cells and ensembles. Architectural simulations show the validity of the proposed solutions and prepare for actual VLSI implementations.

1 Introduction

The RRM project at SRI International is a synergistic software and hardware effort to build a novel computer system featuring

- Multi-grain massively concurrent program execution.
- Unusual ease in programming and reprogramming by using ultra high level declarative languages.

1.1 Massively Concurrent Architecture

Concurrent architectures rely on cooperating resources to solve a given problem. Such cooperation can be achieved in several ways:

- Coarse-grain concurrent execution organizes the computation in sizable tasks that seldom exchange information. Components of a coarse-grain architecture are capable of independent operation. The efficiency of data exchanges between tasks is important, but not a critical factor.
- Fine-grain concurrency partitions the problem into very small tasks that constantly exchange data. Fine-grain architectural components are typically incapable of carrying out any meaningful operation alone, but are capable of reaching high performances as an aggregate. The efficiency of data exchanges is of paramount importance.

The RRM architecture exploits concurrency at several levels, achieving a *hierarchical, multi-grain* mode of execution. At one level, the architecture exploits technological advances in VLSI to pack on a single chip many simple processing cells operating in fine-grain mode. At a higher level, several such chips cooperate within a coarse-grain architecture to solve larger problems. As a result, higher performance, typical to fine-grain concurrency, can be attained even when solving problems that are only locally homogeneous.

1.2 Ultra High Level Programmability

One hurdle in the utilization of massively concurrent architectures is the difficulty of programming them. The RRM project demonstrates that ultra high level languages (UHLLs) are the key to combining hardware efficiency with programming ease and flexibility. From the hardware point of view, UHLLs do not prescribe specific orders of execution, and thus provide maximal opportunity for concurrent execution. From the software point of view, UHLLs have features that support all phases of system development, from design to maintenance.

The software component of this project has developed multi-paradigm languages that combine the advantages of functional [1,2], object-oriented [7], and logic programming [5]. A companion paper [6] summarizes these languages and shows how they can be implemented on the RRM. In contrast to traditional programming languages, these languages are *logical*, in the sense that their statements are sentences in a logical system, and their declarative semantics is given by models that satisfy those sentences.

1.3 Term Rewriting Model of Computation

The RRM project is based on a clearly defined model of computation, called *concurrent term rewriting* [3]. Term rewriting operates by applying rewrite rules to data terms consisting of *nodes* organized as a tree structure. A *rewrite rule* is composed of two templates: the rule's *lefthand side* defines a pattern to be matched against subterms occurring in the data term; when the match is successful, the *righthand side* guides the replacement of the subterm instance. *Variables* in a rule's lefthand side denote structures within the

*Supported by Office of Naval Research Contracts N00014-85-C-0417 and N00014-86-C-0450.

†Address from September 1988 onward: Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Road, Oxford OX1 3QD, England (UK).

matched term that can be used to create a new structure according to the righthand side. Although the model uses tree-structured terms, directed acyclic graphs (DAGs) are needed for efficient implementation of shared substructures. Several advantages are accrued:

- Storage can be reduced by *sharing* common data structures.
- Replacement is greatly simplified in the DAG model, since the pure tree data representation requires copying (possibly huge) trees whenever variables occur more than once in the right-hand side of a matched rule.
- DAGs are unavoidable for object oriented programming, since multiple access to objects is essential to that paradigm.

Operationally, a term rewriting computation starts with an initial data term and a set of rewrite rules. These rules are applied to the term (finding matching subterms and replacing them with new structures) until no more matching instances can be found; the initial data term has then been *reduced* yielding the result. Conceptually, this model of computation is inherently concurrent in the sense that several different rewrite rules may be applied at many different data subterms at once. No explicit constructs are required at the language level to achieve or to describe concurrency.

A simple example of term rewriting is illustrated in figure 1. The example shows how to compute the Fibonacci function defined by the equations

```
fibo(0) = 0 .
fibo(s(0)) = s(0) .
fibo(s(s(x)) = fibo(s(x)) + fibo(x) .
```

The example is posed without numerical operations, using natural numbers described with the `s` (successor) and `0` functions (e.g., `3 = s(s(s(0)))`). Figure 1(a) depicts the rewrite rules needed to implement this function. The rules are applied to a data term containing a mixture of `fibo`, `+`, `s`, and `0` symbols. A given data term is reduced to a tree containing no more instances of the `fibo` function. Figure 1(c) shows an initial data tree and, in the middle tree, the final result after applying the set of rules presented in figure 1(a).

The example is continued with figure 1(b) showing the rewrite rules for eliminating + operators. By applying them to the term illustrated in the middle of figure 1(c) the term is eventually reduced to a data tree containing only `0` and `s` symbols (i.e., an integer). The tree at the right of figure 1(c) depicts the final result.

Architecturally, term rewriting seems ideal for supporting massively concurrent computation because

- Under a certain simple and quite common assumption (called the Church-Rosser property), the sequencing of rules is immaterial; therefore *rule scheduling* can be dynamically adapted to available resources.
- Actual success or failure of a match at a data subterm where the match should succeed is not critical, since a later attempt will succeed. Therefore *data access* can be adapted locally to architectural resources.

The first feature removes the sequential control bottleneck inherent to the von Neumann model of computation. The second feature strikes a balance between performance (made possible by using local connections) and flexibility (made possible by allowing remote connections that can result in failures) that is absent in most other models of computation capable of handling fine-grain concurrency.

The scheduling of rule execution is important for understanding the RRM architecture. Following are four relevant choices:

1. *Concurrent term rewriting* allows the application of several rules at multiple data sites at once. This could be implemented in a MIMD architecture where multiple controllers direct rewriting at multiple sites. While this is the fastest model (in the sense that it will complete a given task earlier), it requires quite expensive architectural support.
2. *Parallel term rewriting* allows the application of a single rule at many data subterms at once. In traditional architectures this corresponds to SIMD execution: a unique controller broadcasting instructions to many processors. Some problems exhibit very *homogeneous* data structures which are handled very well by this model of computation.
3. *Sequential term rewriting* is the obvious restriction to a single rule rewriting at a single data site.
4. *Concurrent/parallel term rewriting* partitions the data term into *domains*, such that within each domain term rewriting is performed in parallel (typically with different rules for different domains). The choice of data domains is dynamic, reflecting the evolution of term structures. This model adapts the effectiveness of parallel term rewriting to *nonhomogeneous* applications.

2 The Rewrite Rule Machine

The RRM project explores a radically novel organization suitable for implementation in state-of-the-art VLSI technology. The architecture supports massively concurrent execution of locally homogeneous tasks. The concurrent/parallel term rewriting model of computation is the basis of a hierarchical design capable of multi-grain concurrency.

2.1 Architectural Levels

The RRM architecture may be described at the following four levels of granularity:

1. A *cell* stores an individual node component of a data term structure.
2. An *ensemble* coordinates the operation of many cells executing instructions broadcast by a common controller.
3. A *cluster* interconnects many ensembles cooperating on solving a homogeneous or nonhomogeneous task.
4. A *network* is composed of several clusters contributing to the solution of a larger problem.

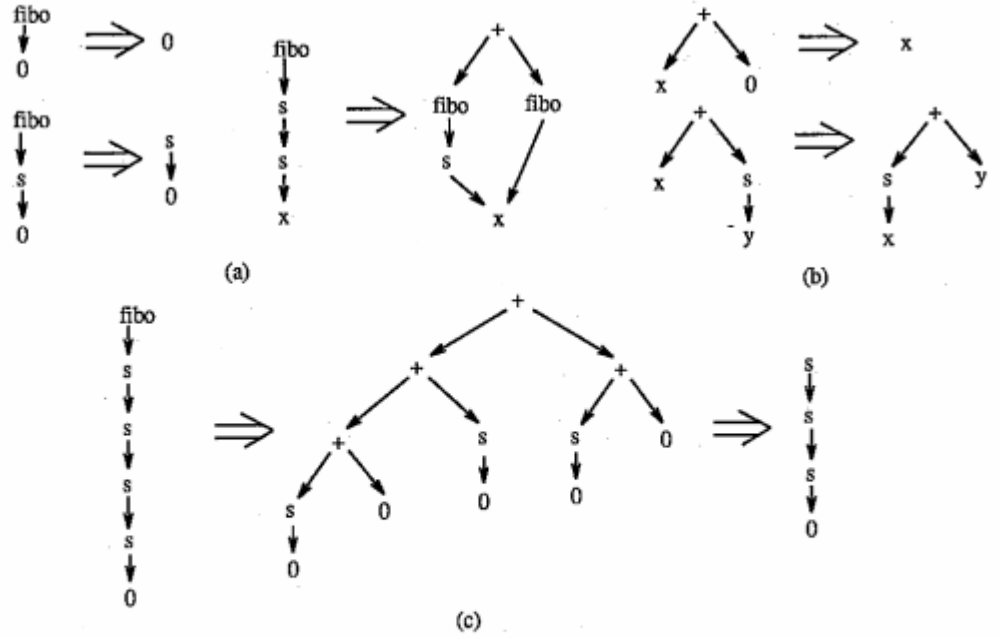

Figure 1: Rewrite Rules for Fibonacci and Addition

The paper focuses mainly on the first two levels, for which new results are presented. However, in order to help the reader understand the complete RRM organization, the other levels are briefly discussed below.

2.1.1 Cell Architecture

A rewrite task is exercised on many thousands, or even hundred of thousands, of cells. Each cell stores one node of the data term, and can also perform simple operations; thus, cells mix storage and computation.

2.1.2 Ensemble Architecture

An RRM rewrite ensemble consists of many cells, support for their communication needs, a common controller, and local storage for currently applicable rules. Since fast intercell communication is of paramount importance to execution speed, each module is implemented on a single VLSI chip to avoid delays associated with off-chip signals.

A controller broadcasts commands to all the cells in its chip. Following a successful pattern match, the term is rewritten at all places where the pattern was found, according to the rule's righthand side. In general, several cells must cooperate to locally change the structure of the data tree. This involves occupying new cells, and considerable exchange of information. Ensemble execution is fine grain.

2.1.3 Cluster Architecture

A *cluster* contains a large number of rewrite ensembles, some backup memory, and facilities for connecting to a conventional computer storing the complete set of rewrite rules. Of particular interest at this architectural level are communication protocols, rewrite rule distribution and coordination, and so on [3]. Ensembles are relatively independent entities needing relatively little intercommunication. Therefore, cluster execution is coarse-grain.

2.1.4 Network Architecture

A *network* consists of several interconnected clusters. They participate in solving multiple or very large problems. A reasonably small number of clusters is expected for an RRM network, so that a general-purpose interconnection switch would be appropriate. At this level, one has to deal with problems typical of higher grain concurrency, such as executing several reduction problems simultaneously.

2.2 Architectural Features

Efficient execution of the term rewriting model of computation requires extensive cooperation among cells. Such cooperation is needed either to match a rule's lefthand side, or to change the term structure according to a righthand pattern. Therefore fast intercell connections are critical to the RRM design. The following summarizes a number of architectural features of our design and the problems that they address:

1. Cells combine processing and storage capabilities, thus eliminating memory access bottlenecks.

2. The problem of synchronizing the operation of cells within the same ensemble is simplified because they all obey a single shared controller.

3. Cells are intentionally kept as simple as possible, so that an ensemble composed of several hundred cells and a controller can be packed on a single VLSI chip. This avoids delays associated with off-chip signal propagation and enables ensembles to operate at high clock rates.

4. Cells within an ensemble (implemented on a single VLSI chip) are connected by short, dedicated signal wires. VLSI experience and theory [10] show that data can be exchanged on such wires much faster than over nonlocal connections. As a result, only physically adjacent cells within an ensemble are allowed to communicate.

5. It is also clear that cells should be placed on the VLSI chip in a regular tessellation pattern so as not to waste silicon area. The combined constraints of creating a regular pattern and of using local communication imply that cells within an ensemble are interconnected by a regular grid.

3 Cell Architecture

A single term rewrite may be executed by thousands, or even hundreds of thousands, of cells, each storing one data node. Cells can also perform simple computations and can also serve as way stations for data transfer. Thus, cells mix storage, computation, and communication. Although the operations performed by each cell are extremely simple, the RRM architecture achieves great power from the cooperation of many thousands of cells on a common task.

3.1 Cell Organization

Cells are physical realizations of the abstract notion of a tree node. The information associated with a node dictates the minimal requirements for cell design. When a cell is allocated to a node, its resources must be able to accommodate all node information. A cell should therefore be capable of storing

- A *token* that uniquely encodes the node's function symbol or constant.
- Various *flags* summarizing the state of computation at the current node (e.g., match condition checked successfully, reduced subtree, etc.).
- *Pointers* to other structures that serve as arguments to the function symbol.

Partitioning problems into sufficiently local computations keeps the number of distinct function symbols small, so that only a few bits (say 8 to 10) are needed to encode any given token. Pointers are also fairly short (around 10 bits), since the number of cells in a single ensemble is limited by available silicon area. Data terms are represented by binary trees or DAGs where functions of higher arity are represented by several binary ones. Therefore, each cell needs to store at most two pointers.

3.2 Cell Operations

Theoretical studies [9] suggest using the flags stored in nodes for performing pattern matching on data terms. Simple broadcast instructions check the presence of a specific flag or token in a cell. Complex patterns are matched by progressively identifying larger and larger substructures.[1] This requires each cell to have a token (and flag) comparator.

Following a successful match, a new term structure is constructed, including pointers to substructures in the matched pattern. This new structure then replaces the matched subterm. To support this operation, there must be some way to allocate empty cells to new data nodes and to keep pointers to partially constructed structures. The cell design provides three temporary registers for holding pointers to partially built new structures.

[1] This process can emphasize either a bottom-up (from the pattern leaves to its root) or a top-down strategy.

3.3 Numerical Computation

Using the RRM for numerical computation requires additional capabilities at the cell level. Although numerical operations could be implemented from "basic principles" by using only the successor operation on natural numbers (i.e., Peano arithmetic), this would be much too slow. A better solution is to let cells perform simple operations on small (8 to 10 bit) numbers, including signed addition, negation, shift, and bit operations. The incremental cost over the already required equality comparison on tokens is quite small. RRM compilers can build a complete set of arithmetic operations using these built-in cell operations. A novel redundant representation of arbitrary precision numbers as trees of small integers [11] permits highly concurrent arithmetic operations, and thus effectively exploits the RRM capabilities.

3.4 Cell Control

As clarified below, the design requires that each cell

- Obey simple broadcast instructions, typically moving data between registers (temporary, pointer, or token storage) or comparing tokens with broadcast data.
- Attempt to connect to another cell whose address is in a register, and if successful, exchange information with that cell.
- Enter an inactive state whenever a comparison or a connection attempt fails—inactive cells do not "listen" to the broadcast instructions, but can be brought back to attention when a special activation command is broadcast by the controller.
- When in inactive mode or free, use its resources to perform maintenance operations, such as garbage collection and data restructuring.

4 Ensemble Architecture

An RRM ensemble consists of many communicating cells, a shared controller, and local storage for current rules. Each ensemble is implemented on a single VLSI chip to avoid the delay of off-chip signal propagation. A controller broadcasts commands to all cells in its ensemble. Such commands are elementary instructions for pattern matching and pattern replacement.

4.1 Regular Ensemble Mappings

Given the limited flexibility of an efficient physical interconnection structure (for example, a rectangular grid), there appears to be a conflict between accommodating matching and replacement. The best way to achieve fast communication during matching is to ensure that nodes that are logically connected are placed in cells that are physically adjacent.[2] On the other hand, replacement requires creation of new data structures, including pointers to data subterms matched by variables. This task is eased when

[2] As already discussed, it is assumed that only local connections are accommodated.

logically connected cells can be placed in arbitrary cell locations. In [8], these two alternatives are called *physical mapping*, which seeks to place logically linked nodes in physically adjacent cells, and *logical mapping*, which allows placing logically linked nodes in arbitrary cell positions and implements communication requests by some form of message passing.

4.2 Dynamic Multiplexed Mapping

Our solution to the mapping problem, called *dynamic multiplexed mapping*, divides the silicon real estate into in a regular array of *tiles*. Each tile has resources sufficient to implement several cells (8 is the number of cells currently under evaluation). Adjacent tiles communicate directly on short wires, so placing logically linked cells in adjacent tiles permits efficient matching. Changes in term structure are usually handled by finding free cells in the tiles adjacent to the requesting cell. Occasionally, all adjacent tiles are fully booked; then remote cells must be allocated. Pointers to remote cells can also arise when a new structure contains a link to a subterm matching a rewrite rule variable.

Since only local wires are used for connections, nonadjacent cells cannot be accessed directly. Instead, whenever a cell uses a link to a remote cell, the currently broadcast operation is aborted for that particular cell (other cells in the grid may continue their computation) and a special request is issued. As illustrated in Section 6.2, this special request will eventually cause the distant data node to be relocated to a physically adjacent cell.

To summarize, dynamic multiplexed mapping

1. Performs intercell communication at high speed (on local wires) at the expense of occasionally failing to make a connection when remote cell access is needed.
2. Decreases the occurrence of remote connections by increasing the alternatives for placing a new data cell created during replacement.
3. Has several cells per tile, thus making better use of shared resources, such as communication links and special data processing functions.

4.3 Interconnection Capabilities

Since connections are the main resource to be optimized in modern technology, the RRM ensemble architecture provides direct connections only between adjacent tiles. In more detail, each edge in the tessellation mesh is implemented by a *communication port* supporting either duplex or half-duplex data transmission.[3] A port between two tiles is shared by all links between data nodes that happen to be mapped on the same pair of tiles. As a result there is competition for communication bandwidth, with only one request being honored at a time. Rejecting a connection because of lack of a free port is similar to failing to execute a broadcast instruction when a remote connection is involved, and communication failures of either kind are handled by a similar combination of software and hardware.

When the mesh degree is sufficiently high (at least 4), the probability that two cells within the same tile will request communication to the same adjacent tile is lower. The fact that all cells in an ensemble obey similar instructions also helps to minimize the competition for bandwidth.

To the model of computation, failures due to remote connection or insufficient bandwidth appear as occasionally nondeterministic execution, in the sense of unpredictable, unreproducible sequences of events. Such nondeterminism is naturally handled by the concurrent term rewriting model of computation. Indeed, these unpredictable sequences of execution are a bonus for the RRM, because they reduce the danger of deadlock. In some simple situations, two data trees are mapped into the same region of the grid. Match operations can then compete for scarce communication bandwidth, and it is possible that neither may succeed in matching the broadcast pattern. Nondeterministic execution reduces the probability of staying forever in such a deadlocked situation. The rare remaining deadlock situations are eliminated by having the communication ports select at random a winning request whenever connection demands exceed the port's capabilities.

5 Ensemble Operation

Although RRM ensemble execution resembles traditional SIMD execution, in that all the cells execute instructions broadcast by a common controller, there are also some significant differences, as discussed below.

5.1 Execution of Rewrite Commands

A set of rewrite rules is compiled into simple microinstructions and loaded into the controller. Such microinstructions perform matching and replacement operations or implement a sequencing strategy for the rewrite steps.

Each instruction broadcast by the controller is *interpreted in a local context*—this is where the RRM architecture departs from classical SIMD execution. Using the pointers and temporary storage available at the cell level, appropriate cell connections are made according to the logical links in the data tree. By contrast, the central controller in traditional SIMD architectures must be aware of the *physical location* of each data connection. A further departure from classical SIMD execution is that an active cell can momentarily activate another cell in order to request information.

In order to perform locally conditional operations, all cells containing data nodes are activated and then tests are broadcast. Nodes that do not satisfy a test are deactivated and do not execute further instructions until the next global activation. A cell that experiences a communication failure is also deactivated.

5.1.1 Pattern Matching

Matching consists of finding cells at which the pattern of a lefthand side occurs. The occurrence of a given pattern or subpattern at a particular cell is represented by setting a corresponding flag in that cell. The simplest subpatterns are tokens, for which a flag is set in the cell. A flag representing a larger subpattern is placed in a cell when flags corresponding to its immediate subpatterns occur in its children cells.

[3] A *duplex* wire can simultaneously transmit data in both directions, while data on *half-duplex* wires can be transmitted in only one direction at a time.

5.1.2 Replacement

Following a successful pattern match, the replacement phase creates a new term structure that should replace the matched one. This new structure is gradually grown by allocating new cells. Since communication failures could prevent this new structure from being completed, the replacement algorithm should be capable of being aborted at any stage without corrupting the original data term. The following measures are taken for this purpose:

- The replacement is always performed by constructing the righthand side pattern using newly allocated cells, and possibly pointers to matched subterms.
- The matched data term is using the atomic operation (`commit`) guaranteed to succeed eventually. When the commit operation is finished, the newly constructed righthand side pattern replaces the matched subterm.
- If constructing the righthand side does not succeed (either by not being able to allocate enough new cells, or due to communication failures), the partial structure is deallocated and the matched subterm is not changed.

5.2 Rewrite Cycles and Termination

The model of computation does not prescribe an order or rule application. However, the order in which instructions are broadcast by the controller affects the ensemble performance. For simple examples (most of the test cases analyzed so far), it is sufficient to repeatedly broadcast the same sequence of instructions until the term is reduced. In more complex cases, the controller may check that there are successful match instances before starting a long sequence of instructions directing the corresponding replacement. Checking whether there are any instances of certain marks requires feedback from the cells in an ensemble to the controller. This can be implemented with a simple binary tree network that ORs signals from all cells, but of course, obtaining results from this network will take more time than a single instruction cycle.

Rewrite rules can interact; for example, one rewrite could create the situation where another applies. For this reason rules must be attempted several times, trying to match patterns created by intervening successful replacements. The controller must therefore employ strategies to group rules in related execution cycles that are repeatedly broadcast until all rewrites have been performed. The grouping of rules in sequences and their order of execution is critical to efficient execution.

Closely related to execution strategies is the issue of termination. In principle, according to the term rewriting model of computation, rewrite rules are applied until no more matches succeed. However, since an ensemble cannot handle a complete set of rewrite rules (and moreover, its cells can only store a small part of the data tree), only a subset of the whole rule set is executed at one time. The termination condition becomes more difficult in the presence of such partitioned rewrite rules. Also compounding the problem is the possibility of communication failures, in which case a currently unsuccessful match could succeed at a later time.

5.3 Autonomous Term Relocation

The dynamic multiplexed cell mapping may introduce remote connections that cannot be used for direct data transfer. The novel concept of *autonomous* execution is proposed to handle these situations. A cell that is not currently active (i.e., has not satisfied recent broadcast tests or is free) ignores the instructions broadcast by the controller, and can instead perform simple maintenance operations.

In the case of a remote connection, any request for data transfer through it fails, with the side effect of starting an autonomous process that will eventually relocate the connection target to a cell physically connected to the requester. The strategy used for this is to create an autonomous message cell that "moves" toward the target cell. This is simply achieved by having each autonomous cell allocate another cell closer to the target, copying its state into it, and finally deallocating the current cell. When the target is reached, the reverse move is performed, until the requester is reached. When a cell is relocated, its connections to previously adjacent cells can become remote. In the long run, the effect of the relocation process is to have data structures move around the grid attempting to eliminate all remote connections. The example presented in Section 6.2 illustrates the power of autonomous processes for relocating data in a ensemble.

6 Architectural Simulations

The RRM architecture is currently being validated by simulation, to enable us to explore design choices and to eliminate mistakes and problems before descending to more detailed levels of design. Several simulators, written in Common Lisp, have been developed for testing different abstraction levels:

- *Concurrent term rewriting* simulations in which rewrite rules are the primitive operations; their goal is to establish the amount of concurrency inherent in UHLL programs. Results obtained at this level are presented in [12].
- *Logical ensemble* simulations deal with abstract tree (or DAG) structured data; the rewrite rules are decomposed into simple operations that are broadcast to all the nodes, and arbitrary connections are allowed.
- *Physical ensemble* simulations consider the effect of limited communication resources, stemming from the mapping of cells onto a regular silicon grid.

6.1 Logical Ensemble Simulation

The logical ensemble simulator deals with broadcast instructions, of the kind generated by the RRM compiler. The rewrite example depicted in figure 1 is continued with a detailed presentation of hand-coded instructions suitable for computing Fibonacci. The broadcast program is separated into two rewrite phases, one reducing Fibonacci symbols to a combination of plus and successor symbols—shown in figure 2–and the other eliminating plus symbols (i.e., performing addition in Peano arithmetic)—shown in figure 3.

The registers of a cell are called `token`, `left`, `right`, `vtemp`, `vleft`, and `vright`.

The microinstructions consist of a sequence of segments of the form

```
(loop (init) (test-token 'fibo)
             (add-mark 'a)
      (init) (test-token 's)
             (add-mark 'b)
      (init) (test-token '0)
             (add-mark 'c)
      (init) (test-tree 'a :lmark 'c)
             (move '0 token)
             (commit 0)
      (init) (test-tree 'a :lmark 'b)
             (add-mark 'd :pointer left)
      (init) (test-tree 'd :lmark 'c)
             (add-mark 'e)
      (init) (test-tree 'a :lmark 'e)
             (fetch left left vleft)
             (move 's token)
             (commit 1)
      (init) (test-tree 'd :lmark 'b)
             (fetch left left vtemp)
             (add-mark 'f)
      (init) (test-tree 'a :lmark 'f)
             (alloc vleft 'fibo)
             (fetch left left vtemp)
             (alloc vright 'fibo)
             (store vtemp vleft left)
             (fetch left vtemp vtemp)
             (store vtemp vright left)
             (move '+ token)
             (commit 2)
```

Figure 2: RRM Code for Fibonacci

```
(loop (init) (test-token '+)
             (add-mark 'a)
      (init) (test-token 's)
             (add-mark 'b)
      (init) (test-token '0)
             (add-mark 'c)
      (init) (test-tree 'a :rmark 'c)
             (fetch left right vright)
             (fetch left left vleft)
             (add-mark 'd)
      (init) (test-mark 'd)
             (commit 0 :pointer left)
      (init) (test-tree 'a :rmark 'b)
             (alloc vleft 's)
             (fetch right left vright)
             (store left vleft left)
             (commit 2)
```

Figure 3: RRM Code for Addition

1. An activation command waking up all inactive cells (`init`).
2. An enabling condition such as
   - `test-token` n enabling only those cells containing specified token "n"
   - `test-mark` x enables cells having flag "x" set
   - `test-tree a :lmark b :rmark c` enables cells having flag "a" set provided their left and right arguments are marked with "b" and "c", respectively (either "b" or "c" may be omitted).
3. A sequence of operations to be performed on all enabled cells:
   - `add-mark` x sets the cell flag "x". A mark can be added to the cell pointed at by a cell's left pointer by (`add-mark x :pointer left`).
   - `remove-mark` x removes mark "x"
   - `move` x y transfers either the token "x" or the contents of register x to y
   - `fetch` x y z moves information from register y in the cell pointed at by x to register z
   - `store` x y z transfers the contents of register x to register z in the cell pointed at by y
   - `alloc` x t allocates a new cell with token "t" and puts a pointer to it in register x
   - `commit` indicates that the structure built under the controller's direction should replace the matched term. As part of this, `vleft` is transferred to `left` and `vright` is transferred to `right`. [4]

This simulator emulates the operations performed locally by each cell, but ignores the effects of distant pointers or limited connection bandwidth. In effect, the constraints introduced by the physical grid are not considered. The simulation can also be used to debug the hand-generated instructions or to find compiler problems.

6.2 Physical Ensemble Simulation

The logical simulator ignores technological constraints limiting the number of cells that can communicate with one another, etc. The next level of simulation models these effects. All cell-to-cell communication requests are checked to see whether they are physically possible (i.e., the requester and target cells are in adjacent grid tiles). If the request is physically possible, then the operation is immediately performed. Otherwise, an autonomous process aimed at moving the target cell to a directly connected tile is started.

The simulation results may be shown graphically as grid "snapshots." The Fibonacci example is continued here by depicting a number of selected pictures in figure 4. The simulation assumes a 4 by 4 rectangular grid connection (each tile can access four neighbors) with half-duplex wires. Each snapshot shows the allocated cells by their token name, and their arguments by arrows corresponding to their "left" and "right" pointers.

[4] The numerical argument is used to coordinate between the inspection of newly created structures and the deallocation of old ones. This is a difficult issue beyond the scope of this paper.

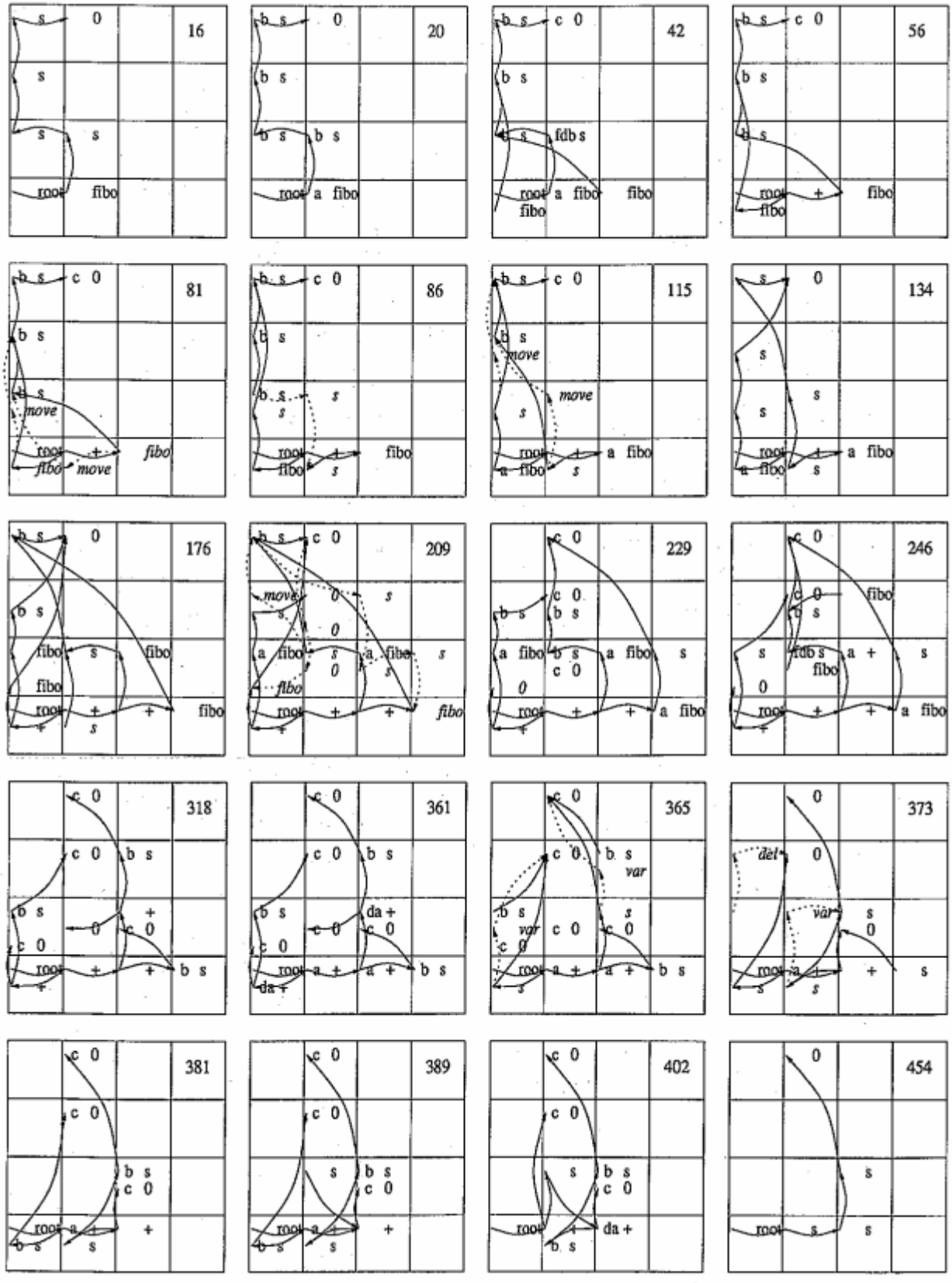

Figure 4: Grid Rewrite Simulation

The snapshot in figure 4 at clock 16 depicts the initial data term as loaded on the grid. The "root" token indicates the start of the term; the starting term for reduction consists of the function symbol `fibo` with an argument consisting of a sequence of `s` operators applied to `0`. Although each of the 16 tiles is able to accommodate several data nodes, the nodes normally are evenly distributed about the grid.

The instructions listed in figure 2 are applied to this data term. At clock 20 the interesting function symbols have been identified by means of flag settings. Since an instance of `fibo(s(s(x)))` has been found, the ensemble proceeds to build a new righthand pattern. The snapshot for clock cycle 42 shows the original `fibo` cell having allocated two adjacent cells that were loaded with `fibo` tokens and have a pointer set to the original argument with one `s` removed, in one case, and two removed, in the other. The old matched term is replaced by the new construction in a single `commit` operation. The resulting term is depicted in figure 4 at clock 56.

The term as rewritten contains remote pointers; when attempting to exchange information on such links the matching process fails, and requests for relocating the term are issued. Figure 4 at clock 81 indicates these requests (actually messengers) by dotted arrows. Both `fibo` operators request their arguments. The details of the algorithm for moving cells on the grid are too complex for this presentation, but the result is shown at clock 86, where one of the moved arguments has been located "above" the `fibo` operator, while the second one is still moving closer. At the completion of these moves there are new remote pointers that in their turn trigger request messages. These can be seen at clock 115, attempting to bring closer the next level of `fibo` arguments.

Eventually the structure is contiguous enough for another set of rewrites, as shown at clock 134. Similar to the execution of the first rewrite, the two `fibo` operators are identified and new structures are allocated for each. Notice the concurrent rewriting at two sites, corresponding to the parallel term rewriting model. The term after the replacement is depicted at clock 176. The initial `fibo` operator has now spawned four new instances of itself.

Just as in the previous steps, the "active" operators are identified and new righthand structures built. But since there are four rewrite sites, there is a high probability of requests conflicting over scarce resources. For example, the snapshot for clock 209 depicts three separate move requests independently finding their targets in the grid. After a few more clock cycles, the structure is contiguous enough to match more rewrite rules. As depicted at time 229, a `fibo(0)` has been replaced by `0`, and all other `fibo` arguments are in adjacent positions. One more rewrite step is shown at clock 246, when another two instances of `fibo` are rewritten. Finally, as illustrated by the snapshot at time 318, all instances of the `fibo` operator have been eliminated, and the term is "reduced" with respect to this rule set. The data term represented corresponds to the tree shown in the middle of figure 1(c).

Next, the rewrite rules for addition (see figure 3) are applied. As depicted at clock 361, the `+` and `s` operators have been marked. The snapshot for clock 365 shows two `+` instances being rewritten.

The next set of pictures is similar to the Fibonacci reduction sequence. The snapshot at clock 373 unveils some of the messages used during `commit`. At time 381, two of the `+` operators have been eliminated, and the remaining two have their arguments in adjacent locations. Their rewrite is shown at clocks 389 and 402. Eventually the data term is fully reduced. The result, as shown at clock 454, is (as expected) 3.

6.3 Architectural Explorations

The grid level simulator also allows experimentation with different choices for several important architectural parameters, such as

- The grid size and the tile mesh structure.
- The number of cells accommodated by each tile.
- The communication bandwidth between tiles and the port allocation strategy used in case of conflicts.
- Allocation strategies for new cells.

A number of other examples have been simulated: additional versions of Fibonacci, matrix transposition, bubble sort, and a highly parallel tree sort. Through these simulations the algorithms for match and replacement have been validated. The simulations also showed that the techniques for resource management do not pose unacceptable overhead.

7 Novel Architectural Features

This section summarizes some of the novel features of the RRM architecture. They are derived from the new model of computation and stress the importance of communication resources.

7.1 Noncritical Scheduling

The concurrent term rewriting model of computation has been useful in freeing the architecture from concerns with sequential execution, separate data access protocols, etc. Since this model requires that rules are executed until the data tree is fully reduced, rewrite rules must be executed cyclically, but in arbitrary order. This permits any rewrite attempt to be abandoned at any moment, provided that the data term is left in a consistent state. This approach allows the RRM architecture to use an extremely fast connection scheme that is *optimal on the average*, without the need for complex logic to handle overloading. Moreover, cells that are disabled because of resource allocation problems (e.g., communication bottlenecks) will get another chance in the next cycle of execution for the same rewrite rule.

7.2 Self-Organizing Processing

In contrast to traditional fine-grain concurrent architectures, the operations broadcast by the ensemble controller are interpreted in the context of each cell, thus achieving unusual flexibility. For example, a traditional SIMD operation requests each active cell to access the cell above it. But an ensemble operation can request that each cell access the cell indicated by one of its link registers. Local instruction interpretation greatly simplifies compilation, and also improves RRM efficiency, because requests for scarce resources are resolved in a context specific to each cell.

The independent execution capabilities of cells are further exploited by autonomous execution modes. Any cell that is either inactive or free during a particular broadcast operation is able to use its resources for other worthwhile tasks. The result is a self-organizing array of cells that either interpret a broadcast stream of instructions or reorganize themselves to make best use of available global resources.

7.3 Sharing by Dynamic Data Passing

Data sharing is difficult for massively concurrent architectures because it requires expensive coordination, which conflicts with the goal of high performance. Concurrent architectures solve this problem by using either message-passing or data-passing. Message-passing organizations rely on explicit data access requests, and typically incur large overhead. They are most suitable for coarse-grain concurrent execution, where this kind of data access occurs sufficiently rarely. Static data-passing organizations (e.g., pipelined processors) move data in a prearranged pattern, such that whenever a shared piece of information is needed in a processing cell at a particular moment, it arrives there on time. Static data-passing ensures high-performance execution, at the expense of great inflexibility and painful programming.

The RRM does not incur the high overhead associated with message passing or the control inflexibility typical for data passing. Data passing need not be static for an architecture that is self-organizing in the sense described above, since cells not enabled for a particular broadcast operation can use their resources to relocate data. For high performance, this dynamic data passing makes use of local communication only. Any cell requesting access to a data item must relocate the data item to an adjacent cell. Data sharing is thus achieved by dynamically allocating resources to cells, such that cells requesting access take turns at becoming adjacent to the data-holding cell.

Acknowledgments

We thank our fellow members of the Rewrite Rule Machine project, Dr. José Meseguer, Prof. Hitoshi Aida, and Prof. Ugo Montanari, with whom we have had extensive discussions of ideas presented in this paper.

References

[1] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 52–66, Association for Computing Machinery, 1985.

[2] Kokichi Futatsugi, Joseph Goguen, José Meseguer, and Koji Okada. Parameterized programming in OBJ2. In Robert Balzer, editor, *Proceedings, Ninth International Conference on Software Engineering*, pages 51–60, IEEE Computer Society Press, March 1987.

[3] Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In Robert Keller and Joseph Fasel, editors, *Proceedings, Graph Reduction Workshop*, pages 53–93, Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[4] Joseph Goguen, Claude Kirchner, José Meseguer, and Timothy Winkler. OBJ as a language for concurrent programming. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, Second International Supercomputing Conference, Volume I*, pages 195–198, International Supercomputing Institute, Inc. (St. Petersburg FL), 1987.

[5] Joseph Goguen and José Meseguer. Eqlog: equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363, Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179-210, September 1984.

[6] Joseph Goguen and José Meseguer. Software for the rewrite rule machine. In *Proceedings, International Conference on Fifth Generation Computer Systems*, ICOT, 1988. To appear.

[7] Joseph Goguen and José Meseguer. Unifying object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477, MIT Press, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986; also, Technical Report CSLI-87-93, Center for the Study of Language and Information, Stanford University, March 1987.

[8] Sany Leinwand and Joseph Goguen. Architectural options for the rewrite rule machine. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, Second International Supercomputing Conference, Volume I*, pages 63–70, International Supercomputing Institute, Inc. (St. Petersburg FL), 1987.

[9] Ugo Montanari and Joseph Goguen. *An Abstract Machine for Fast Parallel Matching of Linear Patterns.* Technical Report SRI-CSL-87-3, Computer Science Lab, SRI International, May 1987.

[10] Jeffrey Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1983.

[11] Timothy Winkler. *Numerical Computation on the RRM.* Technical Report, SRI International, Computer Science Lab, 1988. To appear, Technical Memorandum Series.

[12] Timothy Winkler, Sany Leinwand, and Joseph Goguen. Simulation of concurrent term rewriting. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, Second International Supercomputing Conference, Volume I*, pages 199–208, International Supercomputing Institute, Inc. (St. Petersburg FL), 1987.