Simulation and Performance Estimation for the Rewrite Rule Machine*

Hitoshi Aida
Dept. of Electrical Engineering
University of Tokyo

Joseph A. Goguen
Programming Research Group
Oxford University

Sany Leinwand,
Patrick Lincoln, José Meseguer, Babak Taheri, and Timothy Winkler
SRI International, Menlo Park CA 94025

Abstract

The Rewrite Rule Machine (RRM) is a massively parallel machine being developed at SRI International that combines the power of SIMD with the generality of MIMD. The RRM exploits both extremely fine-grain and coarse-grain parallelism, and is based on an abstract model of computation that eases creating and porting parallel programs. In particular, the RRM can be programmed very naturally with very high-level declarative languages featuring implicit parallelism. This paper gives an overview of the RRM's architecture and discusses performance estimates based on very detailed register-level simulations at the chip level together with more abstract simulations and modeling for higher levels.

1 Introduction

Following an overview of the Rewrite Rule Machine (RRM) architecture and model of computation, this paper discusses recent performance estimates based on simulation. The architecture is a multi-level hierarchy, which is SIMD at the lower (chip) levels, and MIMD at the higher levels. This enables the RRM to combine the advantages of the SIMD and MIMD approaches. The RRM model of computation is concurrent graph rewriting, which supports extremely fine-grain parallelism, dynamic resource allocation, and simple semantics.

Since performance estimation for a machine like the RRM is difficult, we must carefully justify our approach. We discuss the problems and how we address them in Section 3. Our approach to performance estimation may be summarized as follows: we chose a diversity of problems to stress the design in different ways, including communication, memory, and computation; we chose problems representative of different application areas; and we built and used different simulators to get a variety of performance estimates.

1.1 Multigrain Concurrency and Applications

Many important real-life applications involve a number of diverse, relatively independent processes, many of which are computationally homogeneous. For example, a large simulation problem may involve many independent loosely coupled processes.

Let us call a computation *homogeneous* if at each moment, it consists of many instances of the same instruction being applied to many data items in parallel; sometimes this is called *data parallelism*. While many familiar numerical algorithms have this form, many complex computational tasks are *locally homogeneous* but *globally inhomogeneous*.

Because of its very fine-grain SIMD parallelism at the chip level combined with its flexible coarser-grain MIMD parallelism at the network level that allows different chips to work on very different subtasks of the same problem at once, the RRM can exploit a problem's parallelism at several levels. We call this property *multigrain concurrency*; it makes the RRM very well suited for solving, not only homogeneous problems, but also complex, locally homogeneous but globally inhomogeneous problems in many areas, including discrete event simulation, decision support systems, rapid prototyping, vision, computational geometry, automated deduction, finite element methods, neural nets, and hardware simulation.

1.2 Combining SIMD and MIMD

At present, the two main approaches to massive parallelism are SIMD machines and MIMD multicomputers. Examples of the state of the art in each category are the Connection Machine, CM-2 (Thinking Machines Inc. [14, 4]) and the MP1216 (MasPar Computer Corporation [23]), for SIMD computers; and Mosaic (Chuck Seitz, Caltech [22]), the J-machine (William Dally, MIT [5]), Paragon (Intel Corporation [**?**]), and the CM-5 (Thinking Machines, which simulates SIMD by MIMD broadcast), for MIMD computers. These two approaches are quite different. Each has unique advantages not shared by the other approach. The strength of SIMD machines is their exploitation of fine-grain data parallelism, which makes them a good choice for homogeneous problems; their weakness is their centralized control, executing the same code everywhere, which makes them perform poorly on large nonhomogeneous applications. MIMD machines are much more flexible because they allow different code to be run in different processors simultaneously; however, their communication—typically asynchronous interprocessor message passing over a network—is not well suited to data parallelism.

A key goal of the RRM is to combine the best of

*Supported by Office of Naval Research Contracts N00014-90-C-0210 and N00014-90-C-0086, and NSF Grant CCR-9007010.

Figure 1: Concurrent rewriting of Fibonacci expressions

these two approaches in a single architectural design. It shares with SIMD machines the capability for fine-grain data parallelism, which is carried to an even finer level in the RRM ensemble; however, because of its decentralized MIMD control, the RRM can perform well on both homogeneous and nonhomogeneous problems, whereas SIMD machines can excel only on homogeneous problems. Compared with MIMD machines, the RRM enjoys the same flexibility and generality, based on distributed control and asynchronous message passing, but because the RRM is SIMD at the chip level, it can exploit fine-grain data parallelism locally, even for highly nonhomogeneous applications, whereas, at present, purely MIMD machines can get large degrees of parallelism only at the interprocessor level.

1.3 Programmability

The RRM is programmable in a wide variety of declarative ultra-high-level languages that permit massive exploitation of implicit parallelism and ease creating and porting parallel programs. We believe that declarative languages are good choices for programming such applications as vision, real-time plant control, simulations, and expert systems, because they do not require explicit commitment to specific forms of synchronization or scheduling. These convictions are supported by extensive simulations, and by compilation techniques [12, 1, 20] making functional (e.g., OBJ [8]), object-oriented (e.g., Maude [17], FOOPS [11]) and relational (e.g., Eqlog [10]) programming languages easy to compile into RRM code.

However, it is a fact of life that some parts of large applications programs have already been written, and it may not be practical to rewrite them in a declarative language. Because its flexible model of computation also supports imperative features, a compiler for the RRM from a conventional language, even a sequential one, could be written relatively straightforwardly.

1.4 The Concurrent Rewriting Model of Computation

The RRM's model of computation is **concurrent rewriting**. In this model, data are **terms** constructed from a given set of constant and function symbols, and a program is a set of equations that are interpreted as left to right **rewrite rules**. The lefthand side (abbreviated, LHS) and righthand side (RHS) of a rewrite rule may have variables as well as function symbols. A variable can be instantiated with any term of the appropriate sort, and a set of instantiations for variables is called a **substitution**.

A rewriting computation starts with a given term as its **data** and a given set of rewrite rules as its **program**. Applying a rewrite rule has two phases, called **matching** and **replacement**. The matching phase attempts to find a substitution which yields a subterm of the input term when applied to the rewrite rule's lefthand side. Then, in the replacement phase, the matched subterm, called the **redex**, is replaced by the righthand side of the rule, instantiated with the same substitution. Rules are applied until no more matches can be found; then the resulting term is called **reduced** and considered to be the final result.

In the concurrent rewriting model of computation, more than one rule can be applied at once, and each rule can be applied to many subterms of the given term at once. Let us explain this by example. Here is a simple program to compute the Fibonacci numbers:

```
(1)   fibo(0) = 0
(2)   fibo(1) = 1
(3)   fibo(N) = fibo(N-2) + fibo(N-1)
                  if N > 1
```

If you give `fibo(3)` as data, the top node will match rule (3), thus the whole term will be replaced by

```
fibo(1) + fibo(2)
```

In the next step, the first `fibo` node will match rule (2), and the second `fibo` will match rule (3) again, and the simultaneous application of these rules yields

```
1 + (fibo(0) + fibo(1))
```

in just one step of concurrent rewriting. Figure 1 illustrates these two concurrent rewriting steps, using tree representation for expressions.

We say that a concurrent rewriting computation is **SIMD**, when just one rewrite rule is applied concurrently at each moment; in the RRM, this style of concurrent rewriting is realized by an **ensemble** chip (see Sections 2-2.1). If several rules are concurrently being applied, each to possibly many instances, we have **MIMD** concurrent rewriting; this general case is the correct model for the RRM as a whole. See [9] for general background on the concurrent rewriting model, [6] for definitions of SIMD and MIMD rewriting (called *parallel* and *concurrent* rewriting in that paper), and [18, 19, 17] for a definition of concurrent rewriting as deduction in rewriting logic and a systematic treatment of concurrent object-oriented computation by means of concurrent rewriting.

2 RRM Architecture

The RRM architecture is **hierarchical**, with each unit consisting of a collection of cooperating units at the next lower level. The most basic processing element is the **cell**, with four cells making up a **tile**. An

Figure 2: Hierarchical Structure of the RRM

ensemble chip contains hundreds of cells (576 is our current estimate). A **cluster** is a collection of ensemble chips connected on a board, and the machine as a whole is a **network**. Figure 2 provides a pictorial representation of the RRM hierarchy.

A single ensemble yields very fast extremely fine-grain SIMD rewriting, but RRM execution is coarse-grain MIMD at the cluster and network levels, since each ensemble independently executes its own rewrites on its own data, communicating with other ensembles when necessary.

2.1 Cell, Tile and Ensemble Architecture

The most basic computational element in the RRM is the **cell** [16, 2], which stores one data item with pointers to other cells, and also provides basic computational and communication capabilities; thus cells mix storage, computation and communication. A cell consists of:

- Several **registers** (mostly 16-bit) including:
  - `token`, which encodes the operation or constant symbol of a data node,
  - `left` and `right`, which point to the descendant nodes[1],
  - a 32-bit `marks` register, which holds volatile information (similar to condition codes),
  - `flags`, which holds less volatile information, such as type and reduction status,
  - Twelve general-purpose registers, including `ntoken`, `nleft`, `nright` and `nflags`.
- An **ALU** to operate on and test the contents of registers.
- **Interfaces** to communication channels and the controller.

We divide the silicon area of the ensemble chip into a 12×12 mesh of **tiles**, each with four cells. Adjacent tiles are directly connected by short wires, so that placing logically linked nodes in cells located in adjacent tiles permits very efficient communication. Placing several cells in one tile increases the probability of logically related data being in adjacent cells.

[1] Unary operations only use `left`, and n-ary operations for $n > 2$ are decomposed into binary ones.

Our new ensemble design is simpler and has substantially better overall performance than previous designs [7, 2]. Its simpler instructions allow a faster clock (100 MHz seems a reasonable estimate) and provide much better support for communication between cells.

An ensemble has a single SIMD **controller** that broadcasts its instructions to all cells. The controller can obtain very fast feedback (one clock cycle) about the state of the cells (such as type of data and operation symbols in cells, remote references, success or failure of an instruction, termination, etc.) and can use such feedback to branch to different SIMD code segments. Obeying SIMD instructions, cells can communicate with adjacent cells (each cell has 16 adjacent cells in its 4 adjacent tiles) to find local patterns for rewriting; hundreds of such patterns may be found and transformed simultaneously. Other SIMD instructions allow communication among nonadjacent cells, relocation of data, and input-output.

SIMD concurrent rewriting takes place by broadcasting instructions that implement matching and then replacement of the patterns found. Although for very regular computations it is possible to avoid remote—i.e., not physically adjacent—references within a single ensemble, in general the dynamic nature of the computation will require remote references, and then matching will require *relocation* of some data. This is accomplished with specialized instructions and chip-level hardware support.

We use a reference counting scheme for storage management, both within ensembles and in the RRM as a whole. We have fully simulated the details of this within the ensemble for the examples discussed in Section 3.

2.2 Cluster and Network Architecture

The cluster architectural level corresponds to board-level structure in the actual implementation. At this level, ensemble chips can be arranged in a 2D mesh with fast connections to each of four neighbors, giving 8 connections per ensemble (4 in and 4 out). With current technology, these could be 16-bit-wide connections running at 50 MHz, giving 800 Mbps per connection and 6.4 Gbps total bandwidth per chip. Additional interconnection hardware at the board level beyond the fast, local connections is also

desirable, as in the iWARP [3] and DataWave [21] designs. The performance we assume is not that much beyond that provided by these designs; the iWARP has 8 ports, each 8 bits wide at 40 MHz, giving 320 Mbps per port and 2.56 Gbps total (100 to 150 ns latency), and the DataWave has 8 ports, each 12 bits, at 60 MHz giving 5.76 Gbps total. We are estimating that a cluster will have about 100 ensembles.

The network level interconnection for the RRM has not been fixed. We have been considering the wormhole routing networks of Seitz [22] and Dally [5]. Actual realizations of these designs have achieved high communication rates: 205 Mbps for Ametek 2010, and 200 Mbps for the Intel Paragon [**?**]. For a 2D mesh, average case communication time for 10,000 nodes is estimated at 1885 ns, or 188 clock cycles. For a 3D mesh, the average case communication cost for 10,000 nodes is estimated at 976 ns, or 98 clock cycles.

In general, interchip communication in the RRM is *asynchronous* message passing that imposes no critical timing requirements on the network or switching technology. Thus, the RRM can exploit the best communication technology available, and take advantage of any future improvements. However, the RRM can exploit locality and use fast local interensemble connections at the cluster level to get very high performance for certain problems.

2.3 Interensemble Computation

Sometimes active cells in one ensemble need information from descendents in another ensemble. We call references from one ensemble to another ensemble *distant* references, to distinguish them from the *remote* references which occur from cell to nonneighbor cell within a single ensemble. Although distant references can be reduced by relocating data to ensembles which reference it most often, it is impossible to completely eliminate distant data references, even using static memory allocation, because, in general, structures will not fit in a single ensemble. To efficiently support interensemble communication, we have developed two related mechanisms.

For symbolic computation, where data is laid out dynamically and computation is asynchronous or delay-insensitive, we use an incremental symbolic cache approach. When a distant reference is made, and it is determined that the distant node should not be relocated to the local ensemble, then a *ghost node* is instead allocated in a cell of the local ensemble, and data from the target of the distant reference is copied into the ghost node. However, unlike true relocation, the ghost node is *prevented from being the root of a rewrite*, i.e., is temporarily frozen. Also, a ghost node maintains a copy of the original distant pointer, and thus acts as a passive incremental "symbolic cache" of data which actually resides on another ensemble. After some time, under SIMD control, ghost nodes, flush their data, and use the stored distant pointer to refresh their contents. This flush-refresh of ghost information may be performed at any time. Also, at some times the parent of a ghost may copy the distant pointer from its descendent ghost, and then cause deletion of the ghost.

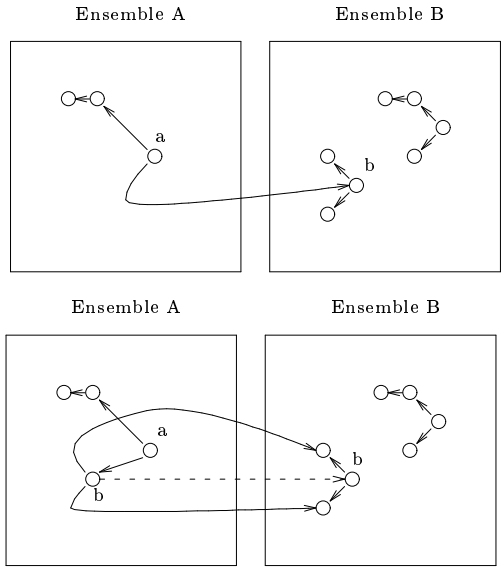

Figure 3: Before and After Creation of Ghost (of **b**)

For example, in Figure 3, in the before (left) picture, ensemble **A** contains a cell labeled **a** which has a distant pointer to a cell labeled **b** in ensemble **B**. In the course of pattern matching, cell **a** requires information from its descendent **b**. In the after (right) picture, a ghost node for **b** has been created in ensemble **A**, and the distant pointer from **a** to **b** has been replaced with a local pointer from **a** to the new ghost of **b**. Thus the ghost of node **b** has distant pointers to the children of **b**, and also has a copy of the original distant pointer (shown as a dashed arrow to node **b** in ensemble **B**). Note that this process cannot continue indefinitely, since ghosts are not allowed to initiate the matching process themselves. Thus even if the structure underneath **b** is large, only that portion of the structure needed to verify a match rooted at **a** is ever copied to ensemble **A**.

The mechanism used in the systolic case is similar in spirit to the symbolic case described above, but can be implemented somewhat more efficiently, due to the locality of reference which (in part) characterizes systolic computations. Because this locality does not change during a computation, we should place elements which communicate frequently on the same ensemble. As in the symbolic case, structures may be too large to fit on a single ensemble, and then we must place portions of the problem on neighboring ensembles, while keeping local copies of the border data current on both ensembles. Since systolic computation is synchronous and delay-sensitive, we must ensure that the border data is updated correctly when it is read by the local ensemble. In general the systolic computation must wait every cycle for the block transfer of data between ensembles.

In Figure 4, ensembles **A** and **B** each contain an area of active cells delineated by the dashed box. Outside this box are border cells which do no necessar-

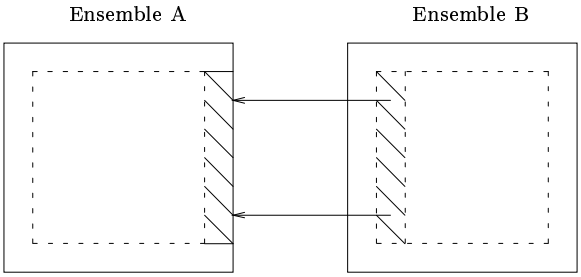

Figure 4: Systolic Interensemble Computation

ily perform computations, but instead store copies of the near-edge cells of neighboring ensembles. Figure 4 shows a block copy of information from active cells in ensemble **B** to (passive) edge cells in ensemble **A**. After information from each neighboring ensemble is copied into ensemble **A**, the next step of computation can proceed.

In many cases we can overlap communication with computation. This potential overlap, or rudimentary pipelining of I/O and computation, is another consequence of our architectural choice of multiple cells per tile. The current design of the ensemble with four cells per tile allows simultaneous systolic computation of four distinct two-dimensional layers at a time. In fact, one or two layers could perform I/O at the same time that the other layers perform their systolic computations. In this way, we may hide some of the potential I/O penalty of inter-ensemble computations.

2.4 Load Balancing

Allocation in an ensemble normally ensures that allocated cells are neighbors of the allocating cell. However, when an ensemble becomes too full, allocations are made on other ensembles. This process can be described as *pushing out* computational subtasks. The SIMD controller can gather (perhaps imprecise) information about the utilization level of an ensemble in order to determine when the ensemble is full. For certain computations, it may be advisable to push out subtasks at the outset. Large symbolic computations usually require building and manipulating very large term structures, which may be distributed over several ensembles when they are initialized, may be distributed explicitly by a specially tuned SIMD broadcast, or may migrate implicitly to neighboring ensembles during computation.

Allocation is important in architectures like the RRM, due to the sensitivity of computation to locality. Thus, initial placement may have a large impact on performance, especially for relatively short computations with large amounts of data. After initial allocation, the compiled SIMD code may explicitly push subcomputations out of an ensemble, perhaps forming a ghost node in its place. Thus the local copy does not perform rewrites itself, although it would still participate passively in other rewrites.

Finally, automatic migration can be performed by pushing subtasks out of an ensemble based on the depth of the subterm from a root node of the ensemble, forcing subcomputations to be pushed out more quickly. However, spreading computation more quickly, and thus more evenly among ensembles, trades off against interensemble communication overhead. The techniques described in Section 2.3 substantially alleviate this overhead, but it still exists.

3 Simulation and Performance Estimation

Estimating the performance of computer systems is a difficult art at best, and is even more difficult for radically new machines that have not yet been built. The performance limitations of simulators mean that large problems are very difficult to run. To test different aspects of a design on the largest possible problems may force using multiple simulators to abstract different details for various choices of performance measure and problem. But then it may be difficult to justify the abstractions, and to ensure that the problems fit the assumptions behind their justifications. For the RRM, these difficulties seem particularly acute, because of the high performance figures that we seek to justify.

Our simulations at the ensemble level have a great level of detail and give quite accurate performance estimates, but our overall performance estimates for the RRM are still preliminary, and more studies and experiments are required to increase their accuracy. The present estimates are based on detailed ensemble simulations, high-level interensemble simulations, estimates of communication requirements, and analysis using simple approximate models. More definitive performance estimates will require more detailed simulations and analytic studies for a wider collection of examples and applications.

The performance models are based on simple predictions of the computation times for specific strategies for performing the computations. Beginning in Subsection 3.3, we discuss RRM performance predictions for a variety of examples which were chosen because their patterns of computation are representative of different kinds of computations; they represent basic examples of general symbolic computations (numeric Fibonacci and the TAK function), highly regular symbolic computations (sorting), and systolic computations (fluid flow and a simple hardware simulator).

When describing RRM performance at the cluster or network levels, we specify efficiency as a percentage of the ideal performance. The ideal performance corresponds to a linear extrapolation of a single ensemble's performance, i.e., a linear speedup. We will also give "idealized Sun-relative speedup," which simply is the product of the number of ensembles, the Sun-relative speedup, and the efficiency.

3.1 Ensemble Simulations and Performance

The new ensemble design and estimates of its performance have been validated by running a variety of benchmarks on a new ensemble simulator written in C which models the ensemble computation in great detail at the register transfer level.

We assume a 100 MHz clock and a 12×12 array of tiles requiring approximately six million transistors. These figures seem achievable since speeds and sizes of this kind have already been demonstrated. For example, the 1991 Hot Chips conference [15] presented two chips with 100 MHz clocks (one of them with 4.1 million transistors), and another chip with 14 million transistors.

There are many different performance measures for machines, including machine instruction execution rates, and actual elapsed time. The most *intrinsic* ensemble performance estimate is the number of clock cycles needed for a given computation. By assuming a specific clock rate, this measure can be translated into seconds. However, some *relative comparison* of performance between the ensemble and existing sequential processors is also desirable. We use the *Sun-relative speedup* for this purpose. To obtain this comparative measure we write one program in ensemble SIMD code or with rewrite rules, and another in efficient C. By comparing the actual performance of the C program on a Sun workstation with the performance of the SIMD code on the ensemble simulator, we obtain for each problem a speedup measure "Sun-relative speedup." In our case, we take a Sun SparcStation IPC as the basis for comparison. This could also be used to assign a "MIPS" rating to the ensemble by multiplying this speedup by the published MIPS ratings of the specific Sun workstation, which is roughly 15 MIPS for the SparcStation IPC. In most cases, the aim is to compare a good algorithm for a problem on the RRM with a good sequential algorithm on a Sun. In some cases, the optimized sequential Sun version involves significant variations from the algorithm used on the RRM. When we discuss each benchmark below, at the ensemble level and levels above, we will mention the specific assumptions made.

3.2 Interensemble Simulations and Communication Requirements

We developed a high-level interensemble simulator for the RRM to study interensemble communication. This simulator models the RRM as a 2D array of clusters, each a 2D array of ensembles. An overall 2D topology was chosen because it is a relatively modest and unproblematic interconnection structure. The simulator was instrumented to track communication at different levels, so that we could estimate communication requirements for the RRM as a whole and between clusters. The high level of abstraction of this simulator means that the results are not precise predictions of the behavior of the RRM, but we do expect large scale behavior to be roughly similar.

We have used the high-level interensemble simulator on certain examples to estimate upper bounds on the communication demands of an ensemble. The following summarize the estimated communication requirements of an ensemble: the estimated ensemble I/O rate was 160–520 Mbps (estimate based on specially instrumented interensemble simulations), given this estimate we can see that pins are not a bottleneck (4 Gbps for 100 pins at 40 MHz), and the communication capacity seems to be in the range of newer network and interconnection designs (which were discussed in Section 2.2).

3.3 Performance Estimates for TAK

The TAK benchmark is a subtle modification of the function Ikuo Takeuchi originated specifically to test Lisp systems. The modification accidentally introduced by Richard Gabriel and John McCarthy makes the function more difficult to optimize, but preserves its simple, recursion-intensive structure. We have implemented TAK for the RRM and in C for purposes of comparison. The Lisp and C code are shown below:

```
(defun tak (x y z)
  (if (not (< y x))
      z
      (tak (tak (1- x) y z)
           (tak (1- y) z x)
           (tak (1- z) x y))))
```

```
tak(x,y,z) register int x,y,z;
{ int r1, r2, r3;
  while (1) {
    if (x<=y) return z;
    r1 = tak(x-1,y,z);
    r2 = tak(y-1,z,x);
    r3 = tak(z-1,x,y);
    x = r1; y = r2; z = r3; }}
```

Because our most detailed simulations are limited to a single ensemble, we have used the arguments $12, 8, 4$, instead of the more traditional $18, 12, 6$. The RRM code completes this benchmark in 22,428 cycles, while the C version finishes in .0015 seconds on a SparcStation IPC. This leads to a Sun-relative speedup of 6.7 (= .0015/.00022428). We currently don't have cluster or RRM estimates for this example.

3.4 Performance Estimates for Numeric Fibonacci

A strategy for computing numeric Fibonacci—which yields a simple approximate model for estimating performance—is to do the computation directly if it fits in one ensemble, and otherwise apply the last of the rewrite rules for `fibo` below

```
fibo(0) = 0
fibo(1) = 1
fibo(N) = fibo(N-1) + fibo(N-2) if N>1
```

once, and then push out the subcomputation of `fibo(N-2)`, to proceed in parallel with that of `fibo(N-1)`, which may either be done locally or may push out further subcomputations. This strategy always keeps a significant subcomputation for the current ensemble. Detailed ensemble simulations allow quite accurate estimates of time required for n up to 10 (it is linear in n). By comparing with the time required to run the same algorithm in C on a Sun workstation, we obtain a Sun-relative speedup of 6.7. The cost for larger n is the time to set up the subcomputations, plus the maximum of the cost to finish the local subcomputation and the cost to finish the pushed out subcomputation, plus the cost to finish the computation. Assuming that network I/O can be overlapped with SIMD broadcast, but that transferring a simple

expression like `fibo(10)` out of an ensemble or transferring a result such as `2584` takes just a small number of SIMD instructions, the complete time to compute the numeric Fibonacci can be modeled by a recursive function allowing different assumptions about the network communication delays. For very fast networks, the network communication times and the computation times (for setup and finishing) are roughly comparable, so that network I/O cannot dominate the overall computation time (usually it will be overlapped with computation).

The cost of numeric Fibonacci within an ensemble is approximated by

$$\text{fibens}(n) = 250 \times n - 50$$

for $n \geq 3$. The approximate cost to compute the n-th Fibonacci, for $n \geq 10$, is then

$$\begin{aligned}\text{fibgen}(n) = {} & \text{simdcost} + \\ & \max(\text{fibgen}(n-1), \\ & \quad \text{fibgen}(n-2) + \text{pushcost})\end{aligned}$$

where simdcost is the SIMD execution cost to setup the subcomputations, push out, pull in, and finish the Fibonacci computation (approximately 300 clock cycles), pushcost is the cost to do two I/O operations (estimated to be less than 200 clock cycles for 10,000 ensembles, see Section 2.2), and $\text{fibgen}(n) = \text{fibens}(n)$ for $n < 10$. With these estimates the simdcost dominates, I/O is overlapped, and efficiency is very good. For larger n, $\text{fibgen}(n) = 300 \times n - 455$. For a 10,000 ensemble RRM, the predicted worst case efficiency for this example is 88%, which seems quite encouraging. The idealized Sun-relative speedup is 59,000.

3.5 Performance Estimates for Sorting

A simple way to sort a sequence of numbers on an RRM ensemble is to use a two-dimensional (2D) exchange sort that uses both "bubblesort" exchanges of consecutive elements of the sequence and "shortcut" exchanges between nonconsecutive elements. By appropriate placement of the sequence within an ensemble, both types of exchanges can be accomplished by simple, local transformations. For a 23×23 array of values we can form a linear sequence of numbers in the array by going down the first tile column, up the second column, and so forth. We can also establish horizontal shortcut links between list elements that are adjacent elements of the same row. By folding the 2D array twice, it is possible to embed the array in an ensemble and fit a list with 23×23 ($= 529$) elements inside an ensemble in this way in such a way that all links are direct neighbor-to-neighbor connections. The 2D exchange sort algorithm alternates bubble sort exchanges between consecutive elements in the sequence with shortcut exchanges between nonconsecutive, but horizontally adjacent elements. For a list of length n placed in this manner, the time to do a 2D sort within single ensemble is proportional to $\sqrt{n}$, and requires approximately $221 \times \sqrt{n} - 468$ clock cycles. The average number of instructions for either the bubblesort or the shortcut exchanges phases is 42, giving a main loop size of 84. Comparing with the time taken by a simple quicksort algorithm written in C and running on a Sun workstation yields a Sun-relative speedup of 127. Uniformly distributed random data was used for the tests.

At the interensemble level, one can use the same pattern, i.e., ensembles in a mesh and interchanges in the long chain or rows, but interchanges will always exchange the maximal value from one ensemble with the minimal value from the next. For this problem, the computation within an ensemble has a different structure than the structure at the cluster level and higher. The simple, fixed connectivity is one advantage of this approach; it should be possible to allocate ensembles so that all I/O connections are local, best-case links. The data would be broken into chunks, that start interchanging data internally and across ensemble boundaries, in one of two directions, at their endpoints. When data items are exchanged across a boundary, an item is pushed out and another is pulled in, preserving the size of the chunk in the ensemble. It seems better not to have a lock step process, in which data items are always exchanged, but instead to only exchange data when there is a need, e.g., when a new value has been interchanged into an end position.

In order to get estimates at the cluster level, a special simulator was written in C that simulates the two-level 2D sorting algorithm and calculates clock-count estimates. Note that, because of reduction of the bandwidth through a cross section of the machine, one expects that sorting at the cluster level should be at least 23 times slower than within an ensemble. Since there are 100 ensembles at the cluster level, one might still see some further speedup; however, the algorithms are more complex and less efficient. If very fast neighbor-to-neighbor connections can be used at the cluster level (and this should also be possible at the level of the RRM as a whole), then exchanging data with a neighbor should take only 5–10 clock cycles. The phases consist of local plus global linear exchanges, with an additional smallest to largest shortcut, and local plus global row exchanges. The additional cost, due primarily to communication, of global operations is estimated at 20 to 30 instructions. The estimated time to sort in a cluster was compared against the time to quicksort on a Sun giving an estimated Sun-relative speedup, for a 100 ensemble cluster, of 114.

For a wormhole routing network, when data items are exchanged between two ensembles, a round trip message is required with estimated time, assuming a single hop is required, of perhaps 20 clock cycles. The estimated idealized Sun-relative speedup for the RRM would be very close to the cluster case. It is very possible that the network latency could be overlapped with other computation, and the increase in the total computation time compared with the cluster case should not be more than 20%.

3.6 Performance Estimates for Fluid Flow

Fluid dynamics can be studied using a two dimensional cellular automaton model [13]. This computational model is nearly ideal for the RRM, due to its very regular structure heavily using instructions that efficiently interchange bits among neighboring cells.

The same communication pattern could be used for many other 2D processing and cellular automata problems. In fact, we have implemented Conway's game of Life using these same techniques, and have achieved similar performance. Many other problems, such as certain vision algorithms, stress analysis and particle diffusion in solids, fit this pattern of computation.

We have implemented a version of the cellular automata approach based on a regular two dimensional hexagonal lattice. Each cell is connected to its six neighbors by links which may hold at most one particle traveling in each direction in each time step. We use unit time steps, unit particle masses, and unit velocity. Each particle is completely described by the link it currently resides on, and all particles have constant kinetic energy, and zero potential energy. At each time step, particles move along their links, possibly interact with other particles at the center of a hexagonal cell, and move to some other link.

We have implemented this model using one RRM cell to simulate each hexagonal cell of the model. Each RRM cell contains six bits which encode the presense or absence of outgoing particles on the links to its six neighbors. Communication is handled by transferring the six bits from each cell to the appropriate neighbor. Computation is handled by performing certain bitwise operations (such as `and`, `or`, `equal`) and a form of table lookup.

We used 1000 iterations of 529 hexagonal cells as the benchmark. Assuming that the ensemble chips will have a clock speed of 100 MHz, the whole benchmark should run in 2.2–2.6 ms. There are multiple ways to implement this problem in C for comparison. The fastest implementation we developed (using register declarations for variables, changing the way table lookup was handled, moving conditional expressions out of the main loop) ran in 1.4 seconds. This results in a Sun-relative speedup of between 400 and 670 for a single ensemble.

The instruction count for the main loop for this problem is about 220 instructions. We estimate that the communication overhead within a cluster, using neighbor-to-neighbor connections, could be as low as 48 clock cycles (6 bits $\times$ 4 cells per tile $\times$ 2). The transfers of marks between ensembles can take place in 12 bit parallel transfers (one cell for each tile on the edge of the ensemble). This gives 268 clock cycles per main loop or 2680 ns at 100 MHz. This gives a cluster-level performance that is 82% of ideal (= 220/268).

3.7 Performance Estimates for a Hardware Simulator

It is possible to do a simple kind of hardware simulation on the RRM extremely fast. The code to simulate two-input NAND and OR gates, where the output state of a gate is represented by status of a specific mark, has only 24 instructions. This simple simulator cannot simulate arbitrary circuits, since there can be lay-out problems; gates must be close to the gates that produce their input signals. For a specific very simple circuit, comparison of the simulations with a highly optimized C program running on a Sun workstation gives a Sun-relative speedup estimate for an ensemble of 533, and for an optimized C program that is a more general circuit simulator an estimated speedup of 1,500.

The cluster-level performance estimate is close to a linear scale up of this. Only a single mark per edge cell needs to be transferred across the ensemble boundaries. It seems reasonable to assume that this could be done in 8 clock cycles. The cluster performance would then be 75% of ideal (= 24/32). The idealized Sun-relative speedup for a cluster would be from 40,000 to 110,000.

4 Summary and Conclusions

First we summarize the performance estimates for the RRM, and then we briefly discuss the status of the project.

For a 10,000 ensemble RRM, our present estimates are as follows:

- Raw peak performance: 576 trillion operations per second.
- For general symbolic applications (the numeric Fibonacci problem is taken as a typical example and the TAK function is a secondary example):
  - Ensemble Sun-relative speedup is roughly 6.7.
  - RRM performance with wormhole network at 88% efficiency gives an idealized Sun-relative speedup of 59,000.
- For highly regular symbolic applications (the sorting problem is taken as a typical example):
  - Ensemble performance is a Sun-relative speedup of 127.
  - Cluster-level performance is a Sun-relative speedup of 114.
  - RRM performance is estimated at over 80% efficiency (relative to the cluster performance) yielding a Sun-relative speedup of over 91.
- For systolic applications (a 2D fluid flow problem is taken as a typical example; a secondary example is a hardware simulator):
  - Ensemble performance is a Sun-relative speedup of 400–670.
  - Cluster-level performance, which should be attainable in practice, is 82% efficiency. This yields idealized Sun-relative speedups of 33,000–55,000.

The RRM project is now in an advanced stage of its proof-of-concept phase that will culminate in building and demonstrating an ensemble prototype, and in further performance and application studies to evaluate the expected behavior of the RRM as a whole.

Acknowledgements

We thank Dr. Tom Blank, Dr. Keith Bromley, Dr. Bill Carlson, and Prof. Al Despain for comments and suggestions that have helped improve the exposition of this paper.

References

[1] Hitoshi Aida, Joseph Goguen, and José Meseguer. Compiling concurrent rewriting onto the rewrite rule machine. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems, Montreal, Canada, June 1990*, pages 320–332. Springer LNCS 516, 1991.

[2] Hitoshi Aida, Sany Leinwand, and José Meseguer. Architectural design of the rewrite rule machine ensemble. In J. Delgado-Frias and W.R. Moore, editors, *VLSI for Artificial Intelligence and Neural Networks*, pages 11–22. Plenum Publ. Co., 1991. Proceedings of an International Workshop held in Oxford, England, September 1990.

[3] S. Borkar, R. Cohn, G. Cox, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWARP: an integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339. IEEE Press, 1988.

[4] Thinking Machines Corporation. Connection machine technical summary. 1990.

[5] William Dally. Network and processor architecture for message-driven computers. In R. Suaya and G. Birtwistle, editors, *VLSI and Parallel Computation*, pages 140–222. Morgan Kaufmann, 1990.

[6] J.A. Goguen. Semantic specifications for the rewrite rule machine. In A. Yonezawa, W. McColl, and T. Ito, editors, *Concurrency: Theory, Language and Architecture*, pages 216–234. Springer LNCS, Vol. 491, 1990.

[7] J.A. Goguen, S. Leinwand, J. Meseguer, and T. Winkler. The rewrite rule machine, 1988. Technical Report PRG-76, Oxford University, Programming Research Group, 1989.

[8] Joseph Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy Winkler. An introduction to OBJ3. In Jean-Pierre Jouannaud and Stephane Kaplan, editors, *Proceedings, Conference on Conditional Term Rewriting, Orsay, France, July 8-10, 1987*, pages 258–263. Springer LNCS 308, 1988.

[9] Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In R. Keller and J. Fasel, editors, *Proc. Workshop on Graph Reduction, Santa Fe, New Mexico*, pages 53–93. Springer LNCS 279, 1987.

[10] 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.

[11] Joseph Goguen and José Meseguer. Unifying functional, 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.

[12] Joseph Goguen and José Meseguer. Software for the rewrite rule machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 628–637. ICOT, 1988.

[13] J. Hardy, B. Hasslacher, and Y. Pomeau. Lattice gas automata for the Navier-Stokes equation. *Physical Review Letters*, 56:1505, 1986.

[14] W. Hillis. *The Connection Machine*. MIT Press, 1985.

[15] *HOT Chips III*. IEEE, 1991. Record of Symposium held at Stanford University August 26–27, 1991.

[16] S. Leinwand, J.A. Goguen, and T. Winkler. Cell and ensemble architecture for the rewrite rule machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 869–878. ICOT, 1988.

[17] José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.

[18] José Meseguer. Rewriting as a unified model of concurrency. In *Proceedings of the Concur'90 Conference, Amsterdam, August 1990*, pages 384–400. Springer LNCS 458, 1990.

[19] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[20] José Meseguer and Timothy Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, pages 253–293. Springer LNCS 574, 1992. Also Technical Report SRI-CSL-91-08, SRI International, Computer Science Laboratory, November 1991.

[21] Ulrich Schmidt and Knut Caesar. Datawave: A single-chip multiprocessor for video applications. *IEEE Micro*, 11(3):22–25, June 1991.

[22] Charles L. Seitz. Concurrent architectures. In R. Suaya and G. Birtwistle, editors, *VLSI and Parallel Computation*, pages 1–84. Morgan Kaufmann, 1990.

[23] Arthur Trew and Greg Wilson, editors. *Past, Present, Parallel: A Survey of Parallel Computing at the Beginning of the 1990s*. Springer-Verlag, 1991.