The Rewrite Rule Machine Node Architecture and its Performance*

Patrick Lincoln, José Meseguer, and Livio Ricciulli

Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA

Abstract The Rewrite Rule Machine (RRM) is a massively parallel MIMD/SIMD computer designed with the explicit purpose of supporting very-high-level parallel programming with rewrite rules. The RRM's *node* architecture consists of a SIMD processor, a SIMD controller, local memory, and network and I/O interfaces. A 64-node *cluster* board is already an attractive RRM system capable of extremely high performance on a variety of applications. A cluster is SIMD at the node level, but it is MIMD at the system level to flexibly exploit the parallelism of complex nonhomogeneous applications. In addition to reporting detailed simulation experiments used to validate the node design, we measure the performance of an RRM cluster on three relevant applications.

1 Introduction

The Rewrite-Rule Machine (RRM) is a Multiple Instruction, Multiple Data/ Single Instruction Multiple Data (MIMD/SIMD) massively parallel computer being designed, simulated, and prototyped at SRI International. The RRM project is unique because it emerged from an initial design search space that was primarily focused on software issues. The outcome of this high-level design effort has been coupled with a bottom-up quantitative approach resulting in an architecture which, while trying to balance complexity, performance and cost in an optimal way, still inherits the important guidelines of the initial theoretical work. Two main characteristics of the overall design are the use of the concurrent rewriting model of computation and the use of active memory.

1.1 RRM Software Model

A rewrite rule $p \longrightarrow p'$ consists of a lefthand side pattern *p* and a righthand side pattern *p'*, and is interpreted as the replacement, called *rewriting*, of *p* by *p'* in some data structure. The RRM's model of computation is *concurrent rewriting*, that is, the process of replacing instances of lefthand side patterns by corresponding instances of rightand side patterns concurrently. Since rule application depends only on the local existence of a pattern, rewrite rules are intrinsically concurrent. A *program* is then a collection of rewrite rules. In its concurrent execution each rule can be applied simultaneously to many instances (SIMD rewriting), and many different rules can each be simultaneously applied to many instances.

Rewrite rules have been used for expressing the implicit parallelism of functional programs in a declarative way, leading to the investigation of so-called *reduction* architectures (see for example [11,21]). However, when generalized adequately [18,16], rewrite rules are not limited to functional computations. They can express with similar ease many other parallel but nonfunctional applications. As explained in [16], concurrent rewriting gives rise to a machine-independent parallel language —Maude [19,16]— in which a very wide range of parallel applications can be easily expressed in a very high level, declarative way. Maude supports three different types of rewriting:

*Supported by Office of Naval Research Contract N00014-92-C-0222.

Term Rewriting. In this case, the data structures being rewritten are *terms*, that is, sytactic expressions that can be represented as labeled trees or acyclic graphs. Functional and symbolic computations are naturally expressible using term rewrite rules.

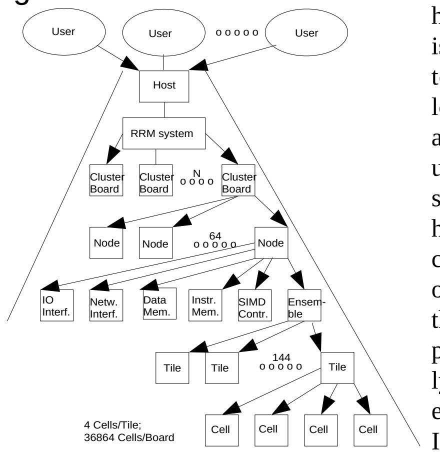
Graph Rewriting. In this case, the data structures being rewritten are *labeled graphs*. A very important subcase is that of graph rewrite rules for which the *topology* of the data graph remains unchanged after rewriting. Many highly regular computations, including many scientific computing applications, cellular automata algorithms, and systolic algorithms fall within this fixed-topology subclass, for which adequate placement of the data graph on a parallel RRM machine can lead to very efficient implementations. The applications used to evaluate the RRM in this paper fall within this category.

Object-Oriented Rewriting. This case corresponds to actor-like objects that interact with each other by asynchronous message-passing. Abstractly, the distributed state of a concurrent object-oriented system of this kind can be naturally regarded as a *multiset* made up of objects and messages; the concurrent execution of messages then corresponds to concurrently rewriting this multiset by means of appropriate rewrite rules. In a parallel machine this is implemented by *communication* on a network, on which messages travel to reach their destination objects. Many applications are naturally expressible as concurrent systems of interacting objects. For example, many discrete event simulations, and many distributed AI and database applications can be naturally expressed and parallelized in this way.

1.2 RRM Hardware Hierarchy

Our parallel programming paradigm diverges from the standard von Neumann model of computation where every execution step requires some interaction between the CPU and data memory. One way of describing the RRM architecture is to imagine a parallel system whose computational units are in its first-level caches. One can think of the SIMD processors as a self-modifiable programmable active store, and of the data memory as conventional passive memory. This organization blurs the distinction between the computational agent and memory, and thus limits the negative effects of random memory access [17].

Fig. 1



As displayed in Fig. 1, the RRM is a 7-tiered hierarchical architecture. The most basic unit is a 16-bit processing element with 16 registers called a *cell*. Four cells, which share local communication buses, make up a *tile*, and 144 tiles operating in SIMD mode make up an *ensemble*, which is expected to fit on a single die. A *node* consists of a collection of hardware devices that constitute a self-contained computational building block. In our case the node is a tightly coupled design that is tuned to supply the ensemble SIMD processor with enough resources to efficiently sustain computation. A node contains an ensemble, data and instruction memory, and I/O and network interfaces, and is expected to be realized as a multichip module. A *cluster* consists of 64 or more nodes connected on a high-speed network, and fitting on a single board. The Rewrite Rule Machine as a whole is a collection of clusters connected on a network and sharing a common host, which runs a standard operating system and handles user interaction. We view an RRM system with a single cluster as an attractive accelerator for applications such as event-driven simulation, image processing, neural networks, artificial intelligence, and symbolic computation in general. Such single-board system has a raw peak performance of 3.6 teraops and, as explained in this paper, is flexible enough to achieve very good performance on a heterogeneous variety of applications.

1.3 Implementation of Concurrent Rewriting on the RRM

The RRM is designed to exploit the massive parallelism of many types of applications expressed with rewrite rules. Fast SIMD rewriting is supported at the chip level, but the RRM as a whole operates in MIMD/SIMD mode to efficiently and flexibly exploit parallelism at all levels. The RRM can perform globally-SIMD homogeneous computations, but can also effectively exploit heterogeneous MIMD parallelism at the cluster and RRM system levels.

Rewrite rules are surprisingly well-suited to massively parallel computation. The most striking architectural advantage of using rewrite rules for parallel computation is that proper compilation techniques can greatly reduce the need for synchronization [15]. Consistent with our framework, rewrite rules allow our design to favor a solution that exposes the underlying architecture to satisfy synchronization requirements through application-specific software primitives. Our design supports both the shared memory and message-passing communication schemes; shared memory consistency is entirely maintained with barrier synchronization mechanisms and test and set operations, while message passing is supported with a very simple active message scheme [23]. The simplicity of our hardware somewhat increases software complexity, but this allows integration of message-passing and shared-memory communication schemes in a more natural way than in other shared-memory designs [10,13] .

We have developed two compilers mapping rewrite rules to parallel RRM code [2, 15]. The latest compiler exhibits efficiencies within 20% of the corresponding hand-compiled codes. Given the great flexibility of the concurrent rewriting model, we believe that it is possible to compile and parallelize conventional code on the RRM with reasonable ease and efficiency. In this way, support for legacy code written in conventional languages, and integration of such code with new code written in a rewriting language could be achieved.

Terms and graphs are represented by having each RRM cell represent a vertex. Each cell has one register holding a datum labeling a vertex, and a variable small number of registers (two or three) holding the addresses of the child cells. Our indirect addressing scheme allows extreme flexibility in representing a graph; vertices of the same graph could reside in neighboring tiles, in nonneighboring tiles, in different RRM nodes, or in passive memory. A mix of software and hardware mechanisms allows communication to occur between vertices residing in any of the above locations. All cells in an ensemble listen to the same SIMD instructions broadcast by a common controller. The instructions are interpreted depending on the cell's internal state; cells to which the instruction does not apply become inactive. Under SIMD control, cells can communicate with each other to find patterns that are instances of a rewrite rule lefthand side. Many such instances can be found simultaneously within a single ensemble and across multiple RRM nodes; the found instances can then be simultaneously replaced by righthand side patterns. The ensemble's SIMD controller has a feedback mechanism which is used to interrogate cells. In this way, scheduling of code for different rewrite rules can be made conditional to the appropriate data being present in the cells. Different RRM ensembles can then work asynchronously in MIMD/SIMD mode on very different types of data, with each ensemble using only the rules that are relevant for the data it currently has.

1.4 Related Research

Key ways the RRM design differs from massively parallel SIMD machine designs of the past include (1) its MIMD/SIMD character, (2) its use of software-controlled prefetching [12,8], which allows data access to be decoupled from the instruction stream, (3) the extreme simplicity of its SIMD controller and (4) its RISC-like instruction set architecture.

Several other features of the RRM are novel in combination, although most have been seen in earlier machine designs in isolation. As a concrete comparison, Goodyear/NASA MPP [5] has local connections between large numbers of (1-bit) cells; however, cells have minimal computational power and there is no support for indirect addressing. The CM-1 and CM-2 architectures are also composed of SIMD-controlled 1-bit cells and in addition have floating point hardware support. The RRM has no dedicated floating point support, and features much more powerful computational agents (much more active memory and a 16-bit ALU). The CM-5 is a MIMD machine with vector units in each node when fully configured. The vector units could be thought of as a very limited form of SIMD computational agents, but they require significant hand-coded software support and are not designed for symbolic computation. The MasPar line of architectures [6] is another modern SIMD design with some similarity to the RRM. The MasPar architectures utilize 4-bit computational cells which are smaller and can store less than RRM cells. MasPar machines support floating point arithmetic better than the RRM, but lack some of the addressing support, as well as the MIMD/SIMD capabilities found in the RRM.

Section 2 describes in detail the node architecture, gives a brief description of the ensemble, and discusses the (preliminary) cluster architecture used in the simulations. Section 3 discusses our simulation methodology and experiments. We have measured the performance of an RRM cluster on three applications: the DARPA Image Understanding benchmark, a logic level circuit simulation, and a parallel sorting algorithm.

2 RRM Architecture

After a brief description of the system and cluster levels, for which only preliminary designs exist, this section focuses on the detailed architecture of an RRM node by describing and interelating its components.

2.1 RRM System

The RRM system is composed of a number of cluster boards interconnected with a high performance network. A host (a conventional workstation) is responsible for the user interface, compilation, system and high-level synchronization functions. We include a separate I/O network for generality because I/O requirements will depend on the particular application area of the final design. The number of cluster boards employed in the system will depend on both technological issues and performance requirements. For the time being, we focus on a system with one cluster board.

2.2 RRM Cluster Board

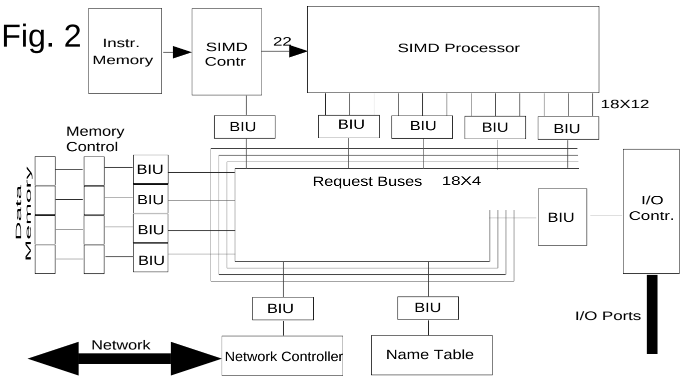
Each RRM cluster board is composed of either 64 or 128 computational nodes; initial estimates indicate that a 64-node cluster implemented in Multi Chip Module technology will fit on a reasonably small board of 40×40 cm. Details of the cluster interconnection topology have not yet been decided; for simulation purposes we model the node-to-node interconnection network as a point-to-point 500-Mbyte/s bidirectional 2-D mesh. We have derived the topology, the link controller architecture and the bandwidth estimates (500 Mbyte/s) from the IEEE SCI standard 1596-1992 [20]. Even though 1-GByte/s communication drivers are already on the market, we prefer to assume 500-Mbyte/s to be conservative on an aspect of the design that we have not yet fully explored.

2.3 Node Architecture

The RRM node architecture augments the ensemble SIMD processor with local memory and with powerful communication capabilities. We have chosen a non-blocking Load/Store scheme so that software-controlled prefetching can allow overlap of computation and communication. We have completely decoupled the data flow from the control flow to parallelize the execution of control and data access operations. One of the interesting results of our design effort is noticing that this paradigm applies to the SIMD world quite well and in some respects allows an overall simplification of the design. As we shall see later, by sharply dividing the execution of control and data access instructions between the SIMD controller and the SIMD processing elements (PEs) one can achieve greater parallelism and at the same time reduce software and hardware complexity.

Fig. 2 is a functional block diagram of the node architecture. The ensemble's cells are continuously fed instructions by the SIMD controller, which steps through the instruction memory. The internal request buses are used for distributing data among the devices of the node. All devices are interfaced to this data path with proprietary bus interface units (BIUs) that, as described below, offer a simple and uniform way of propagating non-blocking split-transaction requests.

Fig. 2



An important characteristic of this architecture is its flexibility; it can be modified by adding and removing BIUs and/or buses to fine tune its performance. Each BIU can be connected to an arbitrary small number of devices and, provided it has enough multiplexers, to an arbitrary number of request buses. The 4-bus configuration depicted above was derived by gathering execution information from a mix of heterogeneous benchmarks (symbolic Fibonacci, sorting, image component labeling, event-driven simulation, image understanding) and by choosing parameters that yield good average performance and at the same time exhibit good hardware utilization. Later, we justify this choice of configuration in more detail.

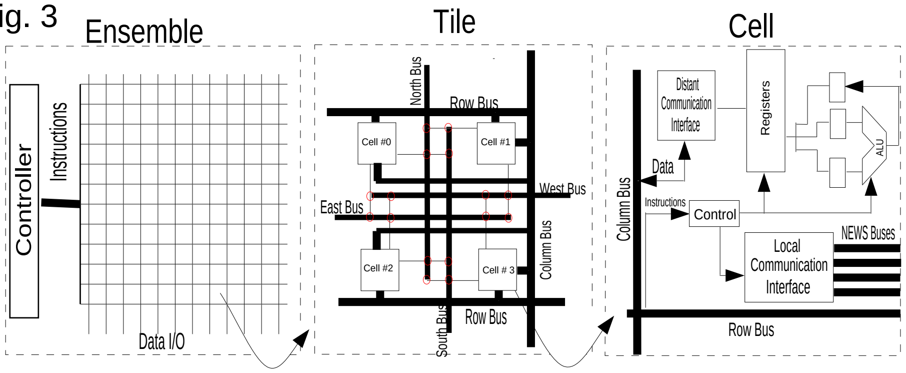
Ensemble

Our SIMD processor, called an ensemble, fits on a single die. The ensemble has been the object of extensive studies in the past [1, 3, 14] and its topology and architecture are based on the results of extensive theoretical and experimental research. For expository purposes we summarize the main characteristics of the ensemble.

The ensemble contains a 12x12 grid of buses and a controller (Fig. 3a). The row buses (really one large unidirectional bus) are used to broadcast SIMD instructions to all cells within the chip, and the column buses are used for data input-output. The controller does not have access to the column buses, which are for the exclusive use of the cells.

Each square formed by the intersection of the buses is called a *tile* (Fig.3b) and contains four 16-bit processing elements called *cells* (Fig. 3c). Each cell is connected to one row bus, to one column bus and to four local 16-bit buses (NEWS). The four local buses allow direct communication between cells of adjacent tiles, and one of the buses (North) allows communication between cells within the same tile. This unique topology offers a large degree of connectivity while trading off hardware simplicity with having to multiplex eight cells on each of the NEWS buses. Non-neighboring cells that cannot communicate through the NEWS buses use the column buses regardless of whether they reside in the same ensemble chip or reside in different nodes. This greatly simplifies both software and hardware at the expense of having to service all non-local communication requests off the ensemble chip even in the case of non-local communication inside the ensemble. A simple fixed-priority scheme synchro-

Fig. 3



nizes the cells' access to the shared buses (local NEWS or column). All arbitrations are explicitly performed by a sequence of SIMD instructions broadcast by the controller. Special hardware support is provided to allow 16 simultaneous 1-bit communication transactions between adjacent local cells without the need for bus arbitration.

Each cell consists of a 16-bit ALU, a dual-ported 16x16-bit register file, communication interfaces, and control logic. Probably the most complex part of the cell is its interface to the column bus. Because of the great throughput needed to allow sustained computation of 576 cells, a lot of effort has been placed on designing an efficient communication scheme. Each cell contains a Finite State Machine designed to receive Load/Store requests and service them autonomously, without interfering with the normal SIMD operations, through the dual-ported register file. No interlock is provided between the Load/Store and the SIMD operations, thus completely relying on software to resolve hazards.

SIMD Controller

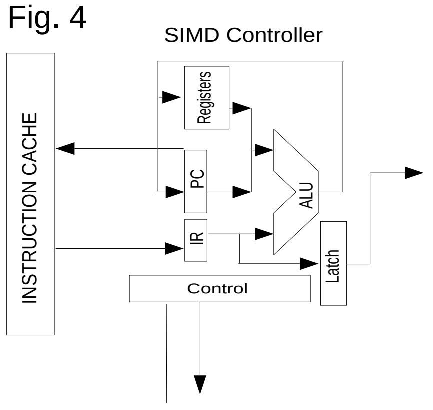
Our SIMD controller is simple enough to fit within the SIMD processor chip. Its simplicity is, in our opinion, of paramount importance because it allows decentralization and simplification of the hardware design and because it permits instructions to be propagated within the chip, therefore allowing faster clock rates.

The controller's hardware (Fig 4) consists of an ALU, a register file, and some control logic. The instruction memory is matched to the controller speed; a secondary program memory can also be included to implement instruction caching. The SIMD controller steps through the program memory and executes or broadcasts instructions.

Our instruction set design closely follows the RISC philosophy to allow only simple elementary instructions and to expose the underlying architecture in order to take advantage of optimizing compilation techniques. Based on our detailed hardware design for the ensemble we are confident that all instructions can execute in two half cycles of 5 ns or at 100 MHz. The RRM uses the A-SIMD mode of execution where, although the controller continuously broadcasts instructions, individual cells may choose to stop executing instructions based on the value of their internal registers. This powerful program control scheme causes control information to be implicit in the ordering of the instruction stream, thus simplifying the hardware design.

As shown in Fig. 4 the instructions in the Instruction Register (IR) can be either placed on a latch to be broadcast to the SIMD cells or can be executed internally by the controller hardware to control the program flow. Synchronization between the controller and the cells is achieved with a simple wired OR mechanism used to determine whether one or more of the 576 cells is in the active state. Besides program flow control mechanisms, the controller also offers some simple hardware support for asynchronous message passing between nodes.

Fig. 4



Controller messages coming from outside the node contain a predefined vector that points to some part of the program memory; application-specific handlers service messages by executing the appropriate interrupt routines. To keep the controller design as simple as possible we do not anticipate automatic context switch support and nested interruption capabilities. Messages are typically very small and rely on the message handlers for data movement (active message paradigm). This part of the controller can be directly derived from conventional processor design techniques and therefore is not of particular interest at this point.

Bus Interface Units and Bus Architecture

The BIUs (Fig. 5) synchronize information flow between devices within the node. All transactions are non-blocking. (All requests are buffered and, after issuing a request, a device is free to perform other tasks.) All requests and messages consist of either two address words for read requests or one address and one data word for write requests.

It is important to note that a read request, after it has reached its source location and has obtained the necessary data, is transformed into a Write/Reply request that is processed by the hardware as a normal write request, which is then propagated back to the reader. The detection of outstanding read requests is obtained using a mix of software and hardware techniques. The number of request buses determines how many bus transactions can happen in parallel. Depending on the application, internode communication requirements can greatly vary. Here we report the results of some experiments designed to determine a sensible number of request buses to be used in our current node configuration.

Fig. 5

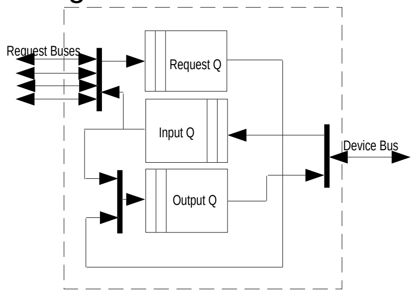
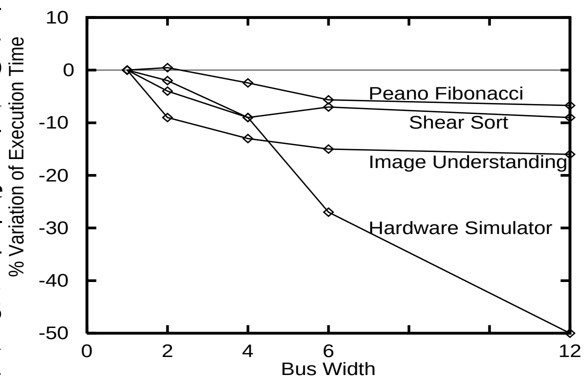


Fig. 6 details the performance variations of a 4-node system (expressed as percentages) when the number of buses, memory units, and SIMD processor BIUs are all varied from 1 to 12. This graph supports the choice of a 4-bus system because, except for the hardware simulator application, the incremental advantage of increasing the bus width beyond 4 is very small. Fig. 6 also helps to convey the novel characteristics of our architect-

Fig. 6

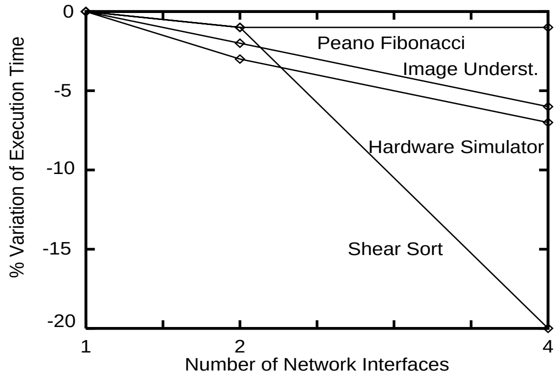


ure. Sorting, Fibonacci, and the hardware simulator applications never use passive memory because the problem size was chosen to fit entirely in active memory. The image understanding benchmark, however, relies on passive memory to store temporary results. This benchmark's small performance variation as the bus bandwidth is reduced reinforces the conviction that our programming model can be quite resilient to memory bandwidth limitations. The hardware simulator is the application that relies most heavily on the internal node communication capabilities because of the extremely high connectivity required by this application.

Network Interface

The network interface supports communication between nodes. It consists of communication drivers and high-speed hardware queues to store incoming and outgoing messages. Although we have not yet committed to a final network topology, we have simulated a 2-D bidirectional mesh with point-to-point links. This part of the node architecture is a good example of how changing specification parameters cause relatively minor changes to the overall system. Our current network interface is assumed to have four bidirectional ports connected to its immediate neighbors; in case, for example, we could only employ an interface with one bidirectional port, four such devices could be placed on a single BIU, thus emulating the original topology with only very localized changes to the design.

Fig. 7



In Fig. 7 we report the result of an experiment aimed at determining a suitable number of network interfaces. This experiment was conducted with a system of 16 nodes. The number of interfaces was varied from 1 to 4, thus measuring the effect of internode communication parallelism. In the 1-interface configuration, packets traveling between nodes can only be sent and received sequentially, while in the 4-interface version packets can be sent and received in parallel from the four NEWS directions. As expected, sorting, which is bound by internode communication bandwidth, shows the highest sensitivity to this parameter and would justify the adoption of multiple interfaces. We have chosen to adopt a more conservative 1-interface base configuration so that our performance results would not depend on an optimistic node-to-node communication mechanism.

Memory Controller, I/O Controller, and Addressing

The flexibility of our node architecture allows tuning the memory subsystem to a required throughput. Our base configuration uses four memory BIUs, one memory controller per BIU, and assumes that memory is matched to the memory controller speed. Because of the adoption of the active memory paradigm, the applications we have developed so far make very little use of passive memory and therefore are marginally influenced by the memory subsystem characteristics. Addressing is a part of our design that has been left underspecified because it is usually not a critical aspect of computer designs. For the moment we do not simulate any indirect system-level addressing mechanisms and assume instead hard-wired addresses. In the future we plan to include a standard virtual memory mechanism to handle a larger memory address space. I/O ports are memory mapped and are accessed just like any other memory location.

3 Simulation and Performance Results

We describe here our simulation methodology and performance measurements for an RRM cluster of 64 or 128 nodes. Although communication-to-computation ratio, bus contention, network throughput and other performance metrics are all important measurements, we chose to report only the wall clock time. We have chosen to do so because this is the only performance evaluation measure that allows easy comparison of the RRM cluster with other designs to give an accurate relative account of the RRM estimated performance.

A register transfer-level simulator of an RRM cluster has been implemented. The simulator holds a very detailed description of all the hardware down to the register level; it uses the libraries provided by the general-purpose simulation package Csim [9]. This package is an extension of the C language that allows very efficient process-oriented event-driven simulations. Each device of each node is a separate process that interfaces with other processes through synchronization lines (events) and hardware queues (mailboxes). This simulation scheme is very similar to a Verilog Hardware Description Language (VHDL) type of behavioral simulation. Contention is carefully taken into account at all levels, and timing (the amount of time each process takes to perform a given operation) is derived from a careful analysis of the hardware as it would be implemented with realistic high-end microelectronics technology. All chips are clocked at 100 MHz. Request bus transactions execute at 50 MHz, while node-to-node packets travel at the rate of 500 Mbyte/s. Since the RRM compiler is only partly complete, we hand-compiled the benchmarks in RRM assembly language. Based on our experience, we expect the compiled code to perform within ±20% of this handwritten code.

Since the network architecture for the cluster has not yet been determined, further simulation work will be required. However, since our communication assumptions are based on existing off-the-shelf technologies, the performance estimates derived from the present simulation experiments are well-grounded.

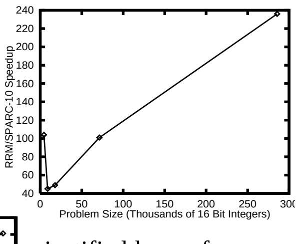
3.1 Performance Estimates

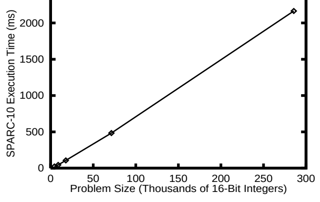
Sorting was implemented with a new version of the Shear Sort algorithm [22]. Even though our particular implementation is architecture-dependent, the ideas we used can be easily extended to other architectures offering good connectivity of their computational agents. The trick is to lay out the problem in a manner allowing efficient communication for both the normal 2-D pattern necessary for the Shear algorithm and for longer-range links among the elements of the list. We have found that the register usage to hold long range pointers is fully justified by performance improvements. Another important improvement to the algorithm discussed in [22] is the fact of keeping a sublist of el-

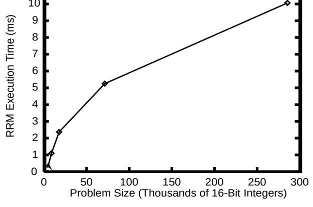
Fig. 8

16-Bit Integer Sorting

Nodes	Problem Size	RRM Time (ms)	SPARC-10 Time (ms)	Speedup
1	4464	0.22	23.20	104.6
2	8928	1.00	45.00	45.0
4	17856	2.16	106.20	49.1
16	71424	4.78	483.20	101.1
64	285696	9.15	2164.60	236.4







ements in each processor, thus avoiding the need of alternating shuffle exchanges between odd and even locations.

In Fig. 8 we report the speedup obtained by a 64-node RRM cluster over an optimized quicksort implementation on a SPARC-10/41 with 48 Mbytes of memory. The anomalous speedup behavior between 4464 and 8929 is due to the internode I/O overhead, which becomes predominant when the data size grows beyond the active memory available in a single RRM node. Notice that the RRM's parallel performance is vastly better than the sequential version, with execution time growing much slower as the problem size approaches the active memory size of the RRM cluster. We anticipate some performance degradation when the data set size grows beyond the active memory available; this will be the object of future studies.

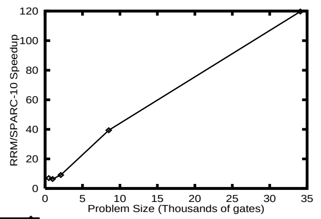
Hardware simulation is representative of a wide class of applications that fall under the category of Discrete Event Simulation. We have simulated a 540-gate LSI design consisting of several cascaded binary counters used for digital image processing. Each one of the logic gates in the LSI design is mapped to an RRM cell. Each gate can have a maximum of 5 inputs and can be programmed to have a maximum delay of 15 time steps. Mapping of the network was performed off-line to minimize distant connections. We replicated the same circuit enough times to obtain a suitable number of gates for the different experiments.

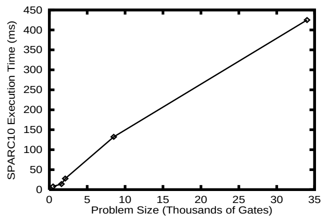
Fig. 9 reports the performance of a 64-node RRM and the Mentor Graphics Quick-Sim simulation tool run on a SPARC-10/41 for 100,000 iterations. The Quick-Sim execution time was estimated by subtracting the time taken to simulate one time step form the time taken to simulate the 100,000 steps to mask out the effects of system-level overhead. These results point out the great versatility of the RRM interconnection network by indicating good performance figures even for an application where the connectivity required is extraordinarily high. The largest example required a total of 64,592 connections between gates of which 68% (44195) required, at each time step, the use of the distant communication mechanisms.

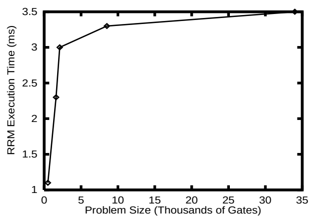
Fig. 9

Gate-Level Hardware Simulator

Nodes	Problem Size	RRM Time (ms)	SPARC-10 Time (ms)	Speedup
1	548	1152	8120	7.05
2	1080	2204	14100	6.4
4	2144	3037	27760	9.1
16	8528	3359	132250	39.37
64	34064	3547	424730	119.74







The DARPA Image Understanding Benchmark for Parallel Computers [24] is a good benchmark because it allows direct performance comparisons with other parallel machines and because it is composed of different phases which test different performance aspects of a design.   The benchmark consists of detecting and abstracting a pattern of rectangles embedded in a cluttered color digital image, and then matching the resulting model with a set of given hypotheses. We have not yet completed this benchmark; therefore, we report only the execution times of the low- and intermediate-level processing parts which detect and abstract the pattern of rectangles from an input test image of size 512x512x8 bits. For this benchmark we relaxed the assumption of a 64 node board and increased the number of nodes to 128 to allow a more fair comparison with the ASP and the IUA architectures.

Fig. 10 contains the reported execution times of several parallel machines [7] with the addition of the RRM performance. Notice that the RRM favorably compares with even the fastest reported simulated execution times that are based on massive special-purpose signal processing designs. We expect the symbolic processing phase of the rest of the benchmark to perform very well in comparison with other machines, given the fact that the RRM was originally designed to support symbolic computation. A fair performance comparison should point out that the ASP, IUA and RRM execution times were obtained through simulation and with

Fig. 10

(D)ARPA Image Understanding Benchmark for Parallel Computers
Low- and Intermediate-Phase Cumulative Execution Times(Sec.)

	Sun-4	FX80/8	Seq81/8	Warp	CM2/64k	ASP	IUA	RRM/128
Connected Comp.	23.07	35.99	78.25	21.72	0.5	0.1509	0.0003	0.0049
Rectangle Detection	4.36	13.92	7.6	20.97	1.26	0.0157	0.0683	0.0898
Total	27.43	49.91	85.85	42.69	1.76	0.1666	0.0686	0.0947

substantial development efforts, while the other execution times were obtained with ‘‘real’’ machines and in some cases required minimal software development time. The clock rates of the ASP and IUA machines were at the time of the simulations (1989) 20 MHz and 10 MHz, respectively; although this might suggest a technological imbalance (the RRM is clocked at 100 MHz) a more careful analysis of the architectures points out that the RRM’s high clock rate is justified by its RISC-like design and on-chip controller; in addition, our understanding is that the ASP and IUA clock rate estimates would still be reasonably adequate today and have not been much influenced by recent advances in microelectronic technology.

4 Conclusion

We think that our design is well-suited for massively parallel computation because it unifies state-of-the-art computer architecture and hardware solutions with a well-understood and mature high-level programming paradigm. Our declarative model of computation allows parallelism to be exploited at many levels simultaneously while reducing synchronization overhead. We have shown very good performance of an RRM cluster on a set of representative applications. We have also laid down the basis for further tuning of our base architecture to application requirements and technological constraints, thus providing design flexibility that will be very useful for future implementations. In the near future we will develop and simulate more applications and experiment with a range of network architectures for the cluster. The current RRM compiler will be extended to handle a wider class of rewrite rules and will be enriched with optimization techniques. In addition, a hardware prototype of the SIMD processor will be built using the SPLASH-2 FPGA system [4].

Acknowledgments We are saddened by the untimely loss of our colleague and friend Dr. Sany Leinwand. Sany contributed much to the RRM project; particularly to its architecture, simulation, and VLSI aspects. We gratefully thank Prof. Joseph Goguen for his fundamental contributions to the RRM and for his most valuable assistance and advice, and Prof. Hitoshi Aida for his crucial help in earlier stages of the RRM design. Besides their other contributions to the RRM project, we especially thank Mr. Babak Taheri for his help during the early stages of the RRM node design, and Mr. Timothy Winkler for his important contributions to the RRM ensemble design and simulation.

References

1. H. Aida, J. Goguen, S. Leinwand, P. Lincoln, J. Meseguer, B. Taheri, and T. Winkler. "Simulation and Performance Estimation for the Rewrite Rule Machine". *In Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation,* pages 336—344. IEEE, 1992.

2. H. Aida, J. Goguen, and J. 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.

3. H. Aida, S. Leinwand, and J. 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.

4. Davis E., Arnold J., Buell D. "SPLASH-2". *In Proceedings of the ACM Symposium on Parallel Algorithms and Architectures, 1992*.

5. K. Batcher. "The Architecture of Tomorrow's Massively Parallel Computer". In *Frontiers of Massively Parallel Scientific Computing, September 1986.* NASA CP 2478.

6. T. Blank. "The MasPar MP-1 architecture". *In CompCon 1990*, 1990.

7. C.Weems, E. Riseman, and A. Hanson. "The DARPA Image Understanding Benchmark for Parallel Computers". *Journal of Parallel and Distributed Computing, 11(1), 1991*.

8. Veidenbaum A. Gornish E., Granston E. "Compiler-directed data prefetching in multiprocessors with memory hierarchies". *In Proceedings of the 1990 International Conference on Supercomputing, 1990*.

9. H. Schwetman "Csim: A c-based, process-oriented simulation language". *MCC Technical Report.*

10. Gupta A., Heinlein J., Gharachorloo K. " Integrating Multiple Communication Paradigms in High Performance Multiprocessors". *Technical Report CSL-TR-94-604*, Computer Systems Laboratory, Stanford University, 1994.

11. R. Keller and J. Fasel, editors. *Proc. Workshop on graph reduction*, Santa Fe, New Mexico. Springer LNCS 279, 1987.

12. Levy H., Klaiber A. "An architecture for Software-controlled Data Prefetching". *In International Symposium on Computer Architecture 1991,* volume 19-3, May 1991.

13. Agarwal A., Kubiatowicz J. "Anatomy of a Message in the Alewife Multiprocessor". *In Proceedings of the 7th ACM International Conference on Supercomputing, 1994*.

14. 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 1988,* Tokyo, Japan, pages 869—878. ICOT, 1988.

15. P. Lincoln, N. Martí-Oliet, J. Meseguer, and L. Ricciulli. "Compiling Rewriting onto SIMD and MIMD/SIMD Machines". *To Appear in PARLE'94, 1994.*

16. P. Lincoln, N. Martí-Oliet, and J. Meseguer. "Specification, Transformation and Programming of Concurrent Systems in Rewriting Logic". G. Belloch, K. M. Chandy, and S. Jagannathan (editors), *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms*, American Mathematical Society, Providence, RI, 1994.

17. Active Memory Technology LTD. "Introducing the DAP/cp8 Range". *DAP Series Technical Overview*, April 1990. Sales Support Note 7.

18. J. Meseguer. "Conditional Rewriting Logic as a Unified Model of Concurrency". T*heoretical Computer Science,* 96(1):73—155, 1992.

19. J. Meseguer. "A logical Theory of Concurrent Objects and Its Realization in the Maude Language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314—390. MIT Press, 1993.

20. Microprocessor and Microcomputer Standards Subcommittee. "IEEE standard for scalable coherent interface". IEEE Standard, 1992.

21. S. Peyton-Jones. "The Implementation of Functional Programming Languages". *Prentice Hall*, 1987.

22. Sen S. Scherson I. "Parallel Sorting in Two-Dimensional VLSI Models of Computation. *IEEE Transactions on Computers*, Feb 1989.

23. T. von Eicken, D. Culler, S.C. Goldsten, and H.E. Schauser. " Active Messages: a Mechanism for Integrated Communication and Computation". In *Proceedings of the 19th International Symposium of Computer Architecture,* May 1992.

24. C. Weems, E. Riseman, A. Hanson, and A. Rosenfeld. "IU Parallel Processing Benchmark." In *Proceedings of the Computer Society Conf. Computer Vision and Pattern Recognition, 1988.*