Compiling Rewriting onto SIMD and MIMD/SIMD Machines

P. Lincoln, N. Martí-Oliet, J. Meseguer, and L. Ricciulli

Computer Science Laboratory,SRI International, Menlo Park, CA 94025-3493

Abstract. We present compilation techniques for Simple Maude, a declarative programming language based on Rewriting Logic which supports term, graph, and object-oriented rewriting. We show how to compile various constructs of Simple Maude onto SIMD and MIMD/SIMD massively parallel architectures, and in particular onto the Rewrite Rule Machine, a special purpose MIMD/SIMD architecture for rewriting. We show how to compile SIMD graph rewriting onto MIMD/SIMD architectures, and discuss mapping 3-D structures into 2-D SIMD meshes. We show how to compile object-oriented rewriting into efficient MIMD/SIMD code. We thus show that Simple Maude is an efficient, machine-independent parallel programming language.

1 Introduction

Rewriting, that is, the process of replacing instances of a lefthand side pattern by corresponding instances of a righthand side pattern, has been recognized as a basic computational paradigm for implementing functional languages. Rewrite rules are intrinsicly concurrent, since their application depends only on the local existence of a pattern. Thus rewrite rules are well suited for expressing the implicit parallelism of functional programs in a declarative way, leading to the investigation of so-called *reduction* architectures that exploit this type of parallelism (see for example [18, 27]).

There are, however, many applications that are certainly amenable to parallelization but which do not fit well within the functional paradigm. For example, a discrete-event simulation in which many different objects interact with each other usually does not have a natural formulation as a functional program. Such simulations often have a high degree of concurrency, which can be exploited using optimistic parallel simulation methods such as Time Warp [17]. In general, there are many algorithms that are state-oriented or nondeterministic. Therefore forcing all problems and algorithmic solutions into a functional notation is both implausible and ill-advised.

In [1], it is explained how, by adequately generalizing the notion of rewriting, one can arrive at a declarative and machine-independent parallel programming paradigm which can express not only functional computations but also a very wide variety of other highly nonfunctional parallel computations. This paper focuses on techniques for compiling such rewrite rules onto SIMD and MIMD/SIMD architectures. Before discussing these techniques, we briefly summarize our proposed programming paradigm.

1.1 Rewriting Logic

The generalization of rewriting we use is provided by *rewriting logic* [21], a logic of change in which the states of a system are understood as algebraically axiomatized data structures. The basic local (concurrent) changes in a system are axiomatized as rewrite rules corresponding to local patterns that, when present in a state of a system, can change into other patterns.

Rewriting logic is a very general model of concurrency from which many other models can be obtained by specialization. We refer the reader to [21] for a detailed discussion. However, we can briefly mention that labeled transition systems; parallel functional programming, including equational programming and the λ-calculus with explicit substitution; Post systems and related grammar formalisms; concurrent object-oriented programming, including the Actor model [2]; Petri nets; the Gamma language of Banâtre and Le Mètayer [7], and Berry and Boudol's *chemical abstract machine* [8]; CCS [25]; and Unity's model of computation [10] can all be obtained naturally as special cases of rewriting logic without any encoding.

1.2 Maude and Simple Maude

We can use rewriting logic as a machine-independent declarative language in which parallel programs can be specified by means of rewrite rules. However, in general, rewriting can take place *modulo* an arbitrary set of structural axioms E, which could be undecidable.

This suggests considering three subsets of rewriting logic. The most general case of rewriting logic should be considered a specification language. The second subuset where E is assumed to be efficiently implementable gives rise to the Maude language[24, 22] which can be supported by an interpretive implementation adequate for rapid prototyping, debugging, and executable specification. The third, smaller subset discussed in this paper gives rise to Simple Maude, a sublanguage meant to be used as a machine-independent parallel programming language. Program transformation techniques can then support passage from general rewrite theories to Maude modules and from them to modules in Simple Maude.

Simple Maude support three types of rewriting:

Term Rewriting. The data structures being rewritten are *terms*, that is, syntactic expressions that can be represented as labeled trees or acyclic graphs. Many symbolic and artificial intelligence applications can be naturally expressed in this way.

Graph Rewriting. The data structures being rewritten are *labeled graphs*. An important subcase is where the *topology* of the data graph remains unchanged after rewriting. Many highly regular computations, including many scientific computing applications, cellular automata, and systolic algorithms fall within this fixed-topology subclass.

Object-Oriented Rewriting. The data being rewritten are actor-like objects that interact with each other by asynchronous message-passing. The concurrent execution of messages corresponds to concurrently rewriting by means of appropriate rewrite rules. Many applications are naturally expressible as

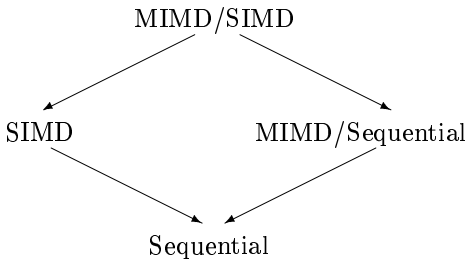


Fig. 1. Specialization relationships among parallel architectures.

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.

1.3 SIMD and MIMD/SIMD Architectures, and the Rewrite Rule Machine

Simple Maude can be implemented on a wide variety of parallel architectures. The diagram in Figure 1 shows the relationship among some general classes that we have considered. Each of these architectures is suited to a different way of performing rewriting computations. Simple Maude has been designed so that concurrent rewriting is relatively easy to implement efficiently in any of these four classes of machines. In the MIMD/Sequential (multiple instruction stream, multiple data) case many different rewrite rules can be applied at many different places at once, but one rule is applied at only one place in each processor. The SIMD (single instruction stream, multiple data) or A-SIMD (for autonomous-SIMD) case corresponds to applying rewrite rules one at a time, possibly to many places in the data. We assume SIMD processing elements are able to execute conditional statements to the extent of being able to deactivate themselves based on internal conditions, and are able to perform address calculations. The RRM ensemble and MassPar-1 are examples of this kind of SIMD processor The MIMD/SIMD case corresponds to applying many rules to many different places in the data, but here a single rule may be applied at many places simultaneously within a single processing node.

Although most of the concepts described later apply equally well to other SIMD and MIMD/SIMD architectures, for concreteness here we describe our target architecture, the Rewrite Rule Machine (RRM) [16, 5, 4, 3], for which many of the techniques described in this paper have been implemented in a prototype compiler. The RRM is a MIMD/SIMD architecture designed with the explicit goal of supporting concurrent rewriting with a 6-tiered hierarchical architecture. The most basic processing element is the *cell*, with four cells making up a *tile*. An *ensemble* is a chip containing about a hundred tiles (144 is our current estimate) and operating in SIMD mode. A *node* is a multichip module containing an

ensemble, local passive memory, I/O hardware and a network interface. A *cluster* is a collection of nodes on a board, and the Rewrite Rule Machine as a whole is a collection of clusters in a cabinet. We are currently focused on the design of a cluster as an economically implementable accelerator for applications such as event-driven simulation, image processing, neural networks, symbolic computing, and artificial intelligence. A single ensemble yields very fast, extremely fine-grain SIMD rewriting, but RRM execution is coarse-grain MIMD at the cluster level, since each ensemble independently executes its own rewrites on its own data, communicating with other ensembles when necessary.

Cells have a 16-bit ALU and 16 registers, each 16-bit wide. Terms and graphs can be naturally represented by each cell holding a node label and pointers to the cells storing the neighboring nodes in the graph. A tile consists of 4 cells which share communication buses. The RRM ensemble is connected as a 2-D mesh of tiles, augmented with special row and column buses to support nonlocal (non-mesh) communcations. 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 contents and internal state; cells to which the instruction does not apply become inactive. Cells can cooperate to find patterns that are instances of a rewrite rule lefthand side. Many such instances can be found in parallel within a single ensemble and across ensembles, and can be replaced by righthand side instances under SIMD control. In the process of rewriting the data in some cells can become garbage; each cell has a reference counter used for garbage collection.

1.4 Compiling Simple Maude onto SIMD and MIMD/SIMD Architectures

We present compilation techniques for term, graph, and object-oriented rewriting. In each case, techniques for both SIMD and MIMD/SIMD architectures are given. The graph rewriting case pays special attention to program transformation techniques taking rules for which globally SIMD lock-step execution is required into more flexible rules for which a less synchronized MIMD/SIMD regime can produce the same answers. The important issue of placement techniques for fixed-topology graph rewriting applications is also studied. In the object-oriented case, message-passing issues, the efficient implementation of multiple inheritance, and object attribute access are discussed. An example of the code generated by our current RRM compiler is given in an appendix.

Here we assume that the rewrite rules are unconditional and left-linear (that is, the lefthand side term of all rewrite rules has no repeated variables). One can easily incorporate simple conditions in the schemes presented below, and use program transformation techniques to remove complex conditions and repeated variables from the lefthand side of rules [4].

2 Compiling Term Rewriting

Given a term rewriting system, the compilation onto MIMD/SIMD architectures is rather involved. First, we describe a general approach to compilation

for SIMD machines. Second, we describe how to lift these compilation ideas to larger MIMD/SIMD architectures composed of many SIMD machines operating in concert, each executing a separate instruction stream.

2.1 SIMD Compilation of Term Rewriting

Our overall approach is to compile rewrite rules into appropriate SIMD assembly language form, then a subject term is loaded and repeatedly rewritten until no further rewrite rules apply. We assume that the entire term is stored in the active memory of the SIMD processing elements, which we call cells, and for the sake of clarity we will assume in this subsection that every node in the graph representing the program corresponds exactly to one dedicated SIMD cell. This strong assumption may be relaxed in various ways including the storage of multiple nodes in each SIMD cell, and swapping parts of the graph in and out of local (passive) memory, at the cost of slowing the overall rewriting process.

Rewriting as compiled onto SIMD architectures embodies three main phases: matching of the lefthand side, construction of the righthand side, and replacement of one for the other. The matching phase consists of cells comparing their local state to globally broadcast information, as well as fetching information about other cells state through pointers. We use a top-down matching strategy in the RRM. The construction phase consists of two tasks: allocation and data motion. Cells which found successful matches allocate new space for the righthand side of a rule, and then instantiate that new structure with information from the lefthand side match. Finally, if the matching and construction were successful, the old structure is replaced by the new in one step. Reference counts are then updated appropriately.

An example of term rewriting code compiled by our prototype compiler for the RRM is given in an appendix. The code generated by the compiler for term rewriting compares favorably (less than 20% penalty in time and space) to hand coded RRM assembly code.

2.2 MIMD/SIMD Compilation of Term Rewriting

In a MIMD/SIMD environment, we must support the possibility of a rewriting process in one SIMD processor operating concurrently with a separate rewriting process in another, even if there are multiple pointers between the processors. A serious complication in this arena is the possibility that multiple processors operating in MIMD mode share data. If there are two rewrite rules $f(g(h(X))) \Rightarrow i(X)$ and $g(h(X)) \Rightarrow j(0)$, and there is a SIMD cell with label f in one SIMD processor pointing to a cell representing $g(h(1))$ in another SIMD processor, references into deallocated storage can result from pathologic interleaving of the execution of the two rewrite rules. (If rewriting $g(h(1))$ to $j(0)$ occurs after matching $f(g(h(X)))$, but before construction or replacement, then in trying to construct the right hand side $i(X)$, garbage may be encountered. Nonsense can also easily be generated for binary terms.) Of course, one can check for overlapping rewrite rules like those in the example above at compile time. Any rule with a lefthand side pattern which

is not a proper subterm of the lefthand side of any rule will never exhibit this pathologic behavior. Similarly, any subtree node in a rule that matches no root of any lefthand side pattern cannot cause a problem.

In order to prevent erroneous behavior for overlapping rules we must perform some sort of locking procedure for those nodes that are a source of possible conflict. With mild atomicity assumptions at the SIMD cell level, reasonably efficient locking code can be generated by the compiler without relying on expensive hardware mechanisms. We enforce that when a SIMD cell begins the matching phase as the root of a rewrite, it first tests and sets a special locking bit. Further, when a SIMD cell fetches data for a possible match, at some point it must also test and set the same lock bit on that SIMD cell. In this way overlapping rewrites are prevented, and the cost is only paid for rules that are a source of possible conflict. This point should be emphasized again; symbols that are the root of rule-overlaps are locked and unlocked, while all other symbols are treated normally.

2.3 Scheduling Rules

The above discussion has neglected higher-level optimization issues, such as the order in which rewrite rules should be broadcast. A simple scheduling of rules is possible where every rule is executed on each SIMD processor in the order originally input. We attempt to find some principles of locality in SIMD and MIMD/SIMD execution of rewrite rules. For example, if the SIMD cells have some capacity to feed information back to the SIMD processor controller, as they do in the RRM node, one can skip rules involving symbols that simply don't occur, thus speeding the entire rewriting process. Similarly, if a large set of rules share common structure, and that structure is not found in the first attempted match, one can skip the remainder of those similar rules. Also, if a rule is successful, and is known to introduce a new function symbol, rules involving that function symbol may be pre-fetched by the SIMD controller. In MIMD/SIMD computations, if data is allowed to migrate (for example, to relieve localized overcrowding), the SIMD controller may be notified about the introduction of new function symbols, and broadcast only rewrite rules relevant to those symbols. With good initial data layout or a small amount of dynamic reconfiguration, one can arrange data labeled with common symbols in the same SIMD processor. One can then concentrate on only those rewrite rules applicable to the set of labeling symbols present in the local SIMD memory.

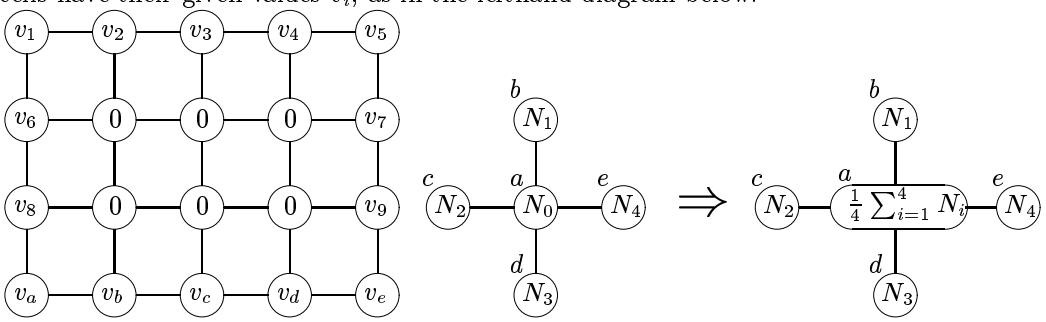
3 Compiling Graph Rewriting

There are two kinds of graph rewriting that we will discuss. One is SIMD graph rewriting, which is similar to SIMD term rewriting. The other, MIMD/SIMD graph rewriting, is similar to MIMD/SIMD term rewriting, where any locally matching graph rewrite rule is applicable. Applications tend to naturally fall into one class of graph rewriting. For example, cellular automata and low-level simulations are much easier to program if one has (the illusion of) complete control over the SIMD lock-step nature of certain graph rewrites. However, for self-timed hardware simulations globally MIMD operation is natural.

3.1 Compiling SIMD Graph Rewriting onto SIMD Machines

Graph rewriting is an area that has received much attention and has been used for a variety of purposes (see for example [13, 28] and references there). In particular, graph rewriting has been extensively used in the compilation of functional languages. However, many highly regular computations can be naturally expressed by graph rewrite rules in which the topology of the graph does not change.

Consider for example the Dirichlet problem from [11]. The problem is to find a solution to the Laplace equation $\nabla^2\Theta = 0$, where the values of Θ are fixed for the boundary cells. Initially, each interior cell has value zero, and the boundary cells have their given values v_i, as in the lefthand diagram below:



As one step of computation, the value of each interior cell is replaced by the average of the values of its four (vertical and horizontal) neighbors. This can be represented by the SIMD graph rewrite rule above on the right, where the labels a, b, c, d, e identify the same nodes before and after the rewrite.

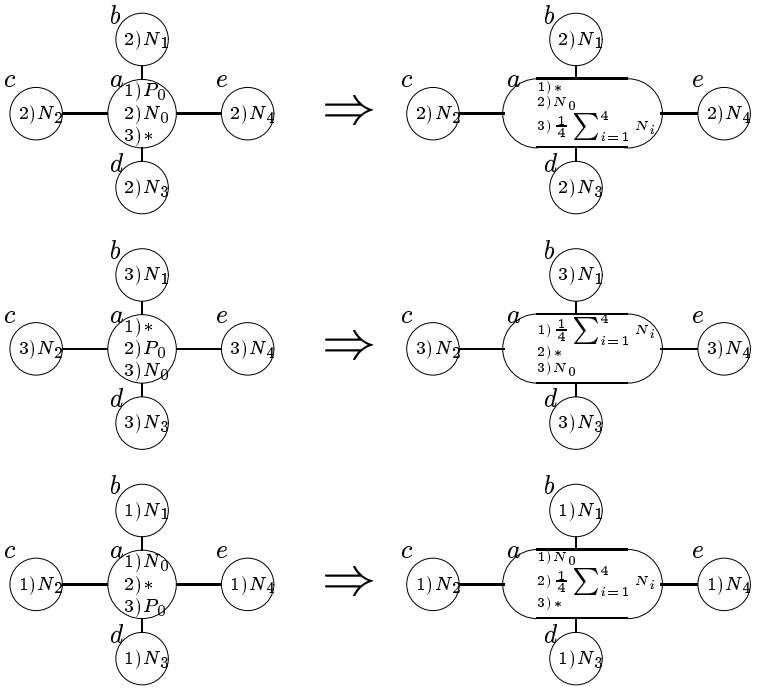
The SIMD compilation of graph rewrite rules is quite similar to that of term rewrite rules. The main difference is the much more careful use of locality in structure creation. For example, in the above rule no new structure has to be created: only the data has to be updated, so the initial placement of the structure becomes permanent. A possible optimization in such cases is to schedule the communication of neighboring graph nodes using knowledge of how the data graph is mapped onto the architecture; this optimization is further discussed in Section 3.4. For some SIMD graph rewriting applications it is crucial that each rewrite rule is applied simultaneously to all its instances, that is, the rewrite must be maximally parallel. However, this restriction can be relaxed by the program transformation into MIMD/SIMD rules described below. Termination of graph-rewriting programs can be detected with simple 1-bit global feedback from all processors as to whether any rewrite was successful in the last main loop. As soon as an entire ruleset has been broadcast without any changes, the computation is complete.

3.2 Converting SIMD to MIMD/SIMD Graph Rewriting

Consider again the Dirichlet example above. Usually, the computation is performed in lock step, enforcing that all cells compute their new values simultaneously. In other words, only maximally-parallel SIMD rewrites are allowed. If one wanted to compute exactly the same sequence of values, but allow asynchronous

computation, one can perform the following transformation. First, where before there was only a single data value stored, we now allow three data values to be stored. Also, we introduce a distinguished data value "$*$" that cannot be matched with a number (say, by type constraints).

We then produce the following three rules: (We take the notational convenience of omitting some attributes of nodes, which are then assumed to be unchanged by the rewrite)



These three rules take the place of the one globally SIMD rule above. The intended operation is that the $*$ stands for the next value to be computed. It is an invariant that when one of these rules applies, the value of P_0 is no longer needed in the computation, and thus can be overwritten.

If one is only interested in the final result of a computation, that is, after things become stable, then, somewhat surprisingly, the original rule suffices. The proof that this one rule, even in MIMD/SIMD mode, leads to the same final result as it does in SIMD mode is not difficult but beyond the capabilities of our compiler to determine, so for MIMD/SIMD machines our compiler would produce the three rule version. Many applications, including most systolic ones, can be naturally expressed as graph rewrite rules. We have described cellular automata such as Conway's game of life, and particle in a cell simulations, lattice gas fluid flow, Dirichlet, sorting networks, and other applications by means of graph rewrite rules.

In general, using the technique just illustrated, one may transform maximally-parallel SIMD rewriting into MIMD/SIMD rewriting by adding clock-time and

past state attributes to all cells. Rewrite rules are then made sensitive to the clock times, enforcing that data from a consistent (artificial) clock time is used in performing all rewrites. After rewriting, a cell must retain its most recent past state to allow its neighbors to compute their next state if they have not already done so.

For example, the original SIMD graph rewriting computation of a 2-D problem can be thought of as a plane (of computation) moving through the third dimension (of simulated clock time) in rigid lock step. The transformed rewrite rules allow this plane to become distorted in the third dimension. The maximum slope of this plane at any time is bounded at any point by the amount of past data retained in each cell. In the above description this amounts to one clock tick per cell. In other words, no cell can compute more than one clock tick ahead of its neighbors. Across a large mesh the global simulated clock skew may be significant. With this transformation, we may thus execute arbitrary graph rewrite rule programs on a loosely coupled MIMD/SIMD machine.

3.3 Compiling MIMD/SIMD Graph Rewriting onto MIMD/SIMD machines

The compilation of MIMD/SIMD graph rewriting is a fairly straightforward extension of the techniques applied to compile term rewriting described earlier, except for the need for a more involved locking procedure and more careful control over (graph) layout for efficiency reasons. In abstract terms, the idea is simply to partition the data into the available multiprocessor memory, and then each SIMD processing unit broadcasts a single rewrite rule at a time. Each MIMD/SIMD graph rewrite rule is executed with the same match, construct, and replace cycle as for term rewriting. The same data locking techniques used in the compilation of term rewriting can be used in the compilation of MIMD/SIMD graph rewriting. However, with graph rewriting it becomes even more critical to obtain most recent information, thus requiring in some cases a complete lockout of all other processes during graph rewriting.

3.4 Mapping

The above sections describing the compilation of graph rewriting assumed that the initial graph was already layed out into the computational units. This section describes some general techniques of data layout in SIMD and MIMD/SIMD machines. Graph mapping is a crucial phase of efficient compilation of parallel programs because it determines how much communication parallelism can be utilized during the course of the computation. We resort to other standard techniques [26] to to handle mapping of more complex non-homogeneous types of graphs.

From an abstract point of view we can define the connectivity of an RRM ensemble (SIMD processor) as the Cartesian product of a completely connected graph on 4 vertices and a large 2 dimensional 4-neighbor mesh. This topology is well suited for symbolic computation because it offers a high degree of connectivity without excessively complicated hardware. When embedding homogeneous graphs

on the RRM it is advantageous to think of the underlying architecture as a two dimensional mesh and utilize the multiplicity of 4 cells per tile to fit a larger 2-D problem by "folding," or to increase the depth that can be handled in a 3-dimensional mapping by a factor of 4.

We have developed a mapping strategy for many kinds of meshes into the 2-D topology of the RRM. For inhomogeneous graphs or graphs for which we haven't developed specialized handling, we resort to randomized heuristic algorithms. The idea is to more-or-less randomly generate an initial placement of the graph (perhaps using specialized mappings for subgraphs), then to repeatedly interchange randomly selected parts of the graph, performing a hill-climbing search to converge to a local minimum of a given communication cost function. This technique was used to efficiently map a hardware simulator by repeatedly moving the location of simulated gates until the communication cost was (locally) minimized. Using the resulting mapping we achieved impressive hardware simulation performance [19].

3-dimensional rectilinear structures are very useful for a wide variety of problems. We give a brief overview of our current methods for mapping 3-dimensional meshes onto 2-dimensional grids. We assume that the neighborhood of a point in the 3-D problem space is composed of the 6 nearest neighbors in the x, y, and z directions. A useful notion to measure the success of a map from some problem space onto a parallel hardware platform is *dilation*. A problem space can be mapped with (maximum) dilation n into some architecture when all neighboring points in the problem space are no more than n communication hops away when mapped on the architecture. This is a useful metric as it tends to predict a scaling factor from maximum performance (for problems that can be mapped perfectly, with dilation 1). For 3-D problems we also consider the dilations restricted to each dimension separately, called the x-, y-, and z-dilations.

One can lay out a 2-D problem perfectly into a 2-D mesh (with dilation 1). However, one cannot map a large 3-D structure onto a 2-D (computing) surface with dilation independent of the dimensions of the problem. Given a 3-D problem space of dimensions x, y, z, where we assume that $x \geq y \geq z$, a naive mapping approach is to lay out x copies of a perfect mapping of $y \times z$ problem spaces. The x-dilation is then z, while both y and z-dilations are 1. Thus, the maximum dilation is z, the sum of dilations is $z+2$, and the average dilation is $\frac{z+2}{3}$. However, a 3-D problem space with minimum dimension z (that is, $z \leq x$ and $z \leq y$) can be mapped onto a 2-D mesh with x-dilation $\lceil \sqrt{z} \rceil$, y-dilation $\lceil \sqrt{z} \rceil$, and z-dilation 1. Thus the average dilation is $\lceil \frac{2\sqrt{z}+1}{3} \rceil$. This mapping may be viewed as squashing the third (z) dimension down into small squares of size $\sqrt{z} \times \sqrt{z}$. By analysis of total reachable spaces, it can be shown that one cannot do much better. That is, the average dilation of any mapping of this general sort is at least $\frac{2}{3}\sqrt{z}$.

It is possible to use this technique repeatedly to map meshes of larger dimension into meshes of lower-dimension. In particular, a 4-dimensional space with dimensions x, y, z, t, with $x \geq y \geq z \geq t$ can be mapped onto a 2-D mesh with x-dilation $\lceil \sqrt{zt} \rceil$, y-dilation $\lceil \sqrt{zt} \rceil$, z-dilation $\lceil \sqrt{t} \rceil$, and t-dilation 1. Note that if $x = y = z = t$, the maximum dilation of this mapping is x, and the average dilation is more than $x/2$, so these mappings are of limitted interest for large

cube-like structures of large dimension.

These optimized mappings allow reasonably efficient layout of 3-D problem structures of relatively large size on 2-D meshes. In the case of the RRM, we have 4 cells per tile, and thus can handle, for example, 3-D structures with minimum dimension $z = 1024$ with average dilation 11, by using the mapping for $z = 256$ and packing four z-neighbors in each tile.

Problems larger than the available active memory can be mapped into passive memory, and then broken into chunks of appropriate size and swapped in and out of active memory for computation.

Another example, hexagonal meshes can be mapped nicely into a 2-D 4-neighbor mesh. The maximum dilation under our mapping for a 2-D hexagonal grid is 2, with average dilation 4/3. An 8-neigbor graph used in an image processing application was mapped with maximum dilation 2. Graphs of larger connectivity can be handled with larger dilations, where in a 2-D mesh the connectivity grows as $2d * (d + 1)$.

4 Compiling Object-Oriented Rewriting

In a concurrent object-oriented system the concurrent state (or *configuration*) typically has the structure of a multiset made up of objects and messages. Therefore, we can view configurations as built up by a binary multiset union operator which we can represent with empty syntax as

```
subsorts Object Msg < Configuration .
op __ : Configuration Configuration -> Configuration [assoc comm id: null] .
```

where the multiset union operator `__` is declared to satisfy the structural laws of associativity and commutativity and to have identity `null`. The subtype declaration

```
subsorts Object Msg < Configuration .
```

states that objects and messages are singleton multiset configurations, so that more complex configurations are generated out of them by multiset union.

As a consequence, we can abstractly represent the configuration of a typical concurrent object-oriented system as an equivalence class $[t]$ modulo the structural laws of associativity, commutativity and identity, i.e., as a multiset of objects and messages.

An *object* in a given state is represented as a term

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where O is the object's name or identifier, C is its class, the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*. The set of all the attribute-value pairs of an object state is formed by repeated application of the binary union operator `_,_` which also obeys structural laws of associativity, commutativity, and identity; i.e., the order of the attribute-value pairs of an object is immaterial.

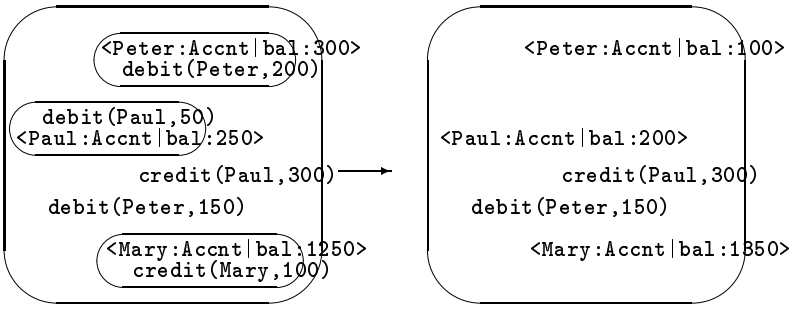


Fig. 2. Concurrent rewriting of bank accounts.

For example, bank account objects in a class `Accnt` may be defined with just one attribute `bal` that is a natural number corresponding to the current balance. Concurrent interaction with the account can be achieved by means of `credit` and `debit` messages that add or subtract a given amount of money. The corresponding updates of the account are specified by the two rewrite rules

```
credit(A,M) < A : Accnt | bal: N > => < A : Accnt | bal: N+M > .
debit(A,M) < A : Accnt | bal: N > => < A : Accnt | bal: N-M > if N >= M .
```

where the second rule can only be applied if the condition `N >= M` is satisfied, that is, if the account has enough funds.

Figure 2 provides a snapshot in the evolution by concurrent rewriting of a simple configuration of bank accounts. The system evolves by concurrent multiset rewriting of the configuration using the rewrite rules of the system. Intuitively, we can think of messages as "traveling" to come into contact with the objects to which they are sent and then causing "communication events" by application of rewrite rules.

The two rules for bank accounts above illustrate the asynchronous message passing communication between objects supported by the Simple Maude language. In general, for concurrent object-oriented modules we only allow conditional rules of the form

$$
\begin{aligned}
(\ddagger)\quad & (M(O))\ \langle O : F \mid atts \rangle \\
& \longrightarrow (\langle O : F' \mid atts' \rangle) \\
& \quad \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\
& \quad M'_1 \ldots M'_q \\
& \quad \mathit{if}\ C
\end{aligned}
$$

involving at most one object and one message in their lefthand side, where the notation $(M(O))$ means that the message $M(O)$ is only an optional part of the lefthand side. In addition, the target object itself is optional, and new objects and messages may be created.

An efficient way of realizing rewriting modulo associativity and commutativity by communication is to associate object identifiers with specific addresses in the

virtual address space of a parallel machine and then messages are sent directly to the address of their target object.

There are several options regarding the practical implementation of general message receipt, but the following protocol appears to be a good tradeoff between time, space, and complexity. The protocol begins with a standard message-send, where the destination of the message is the (unique) object location (object id) of the recipient[1]. Upon arrival, messages are queued in a data structure that tests for immediate applicability of rewrite rules with respect to that message. In the case that the message cannot be accepted immediately, the message is incorporated in a compact data structure that supports periodic access. The protocol requires that messages periodically be retried to ensure the fairness required at the language level. Since several objects of related classes may reside in the same SIMD processor, several objects may be updated concurrently by the SIMD broadcast of one rewrite rule. The actual rewriting of a message and an object, as well as the additional rewriting required for the evaluation of the new attribute values can be implemented using the SIMD and MIMD/SIMD term rewriting techniques described in Section 2. However, both inheritance and attribute access can be implemented more efficiently by special techniques described below. More experimentation is needed to measure the performance of object-oriented rewriting on large examples from realistic applications. However, we believe that using the hardware-supported message passing of the RRM we can effectively implement this style of object-oriented rewriting very efficiently.

4.1 Inheritance Hierarchy and Attribute Access

Support for multiple inheritance for classes is provided by the order-sorted type structure of rewriting logic [15] so that if C is a subclass of C', then C is a *subsort* of C'. For more details on the type theory of class inheritance and the desugaring of rules we refer the reader to [22, 23].

In order to support run-time class information, the compiler must incorporate an algorithm for determining when an object is of a class that is subsumed by some other class. That is, at run-time we must determine if a given object is, by class-membership and inheritance, a member of some class. This operation is performed at every application of a rewrite-rule. We support this operation very efficiently with an encoding of the inheritance hierarchy with bitvectors based on [6]. The idea is essentially to embed the partial order of inheritance into a Boolean lattice. The Boolean lattice is then encoded using one bit-position for each element of the original partial order. The bitvector encoding each element of the lattice is then the bitwise-*or* of the bitvector encoding of its descendents and a 1 in the bit-position of the partial-order element that lattice element represents, if any. Thus any bitvector code can be read as a set, where each 1 indicates the presence of some element from the partial order (which has a unique bit-position). This encoding technique reduces the cost of class-matching to $n/16$ instructions

[1] In a SIMD implementation this takes place inside the SIMD processor, whereas in a MIMD/SIMD implementation it must in general be supported by the network connecting the SIMD processors.

on a machine with 16 bit word-size where n is the size of the (largest connected component in the) input hierarchy. Methods exist to reduce the cost to either $log(n)$ (expected, still $n/16$ worst-case), or linear in the depth of the classes being compared, although these have larger constant factors [6, 14, 9, 12]. Note that this Boolean lattice representation applies more generally to any set of rewrite rules in which variables are typed in an order-sorted type structure.

In addition to this class-matching operation, the compiler must also support attribute access efficiently. The interpretive approach of maintaining an association list from attribute (field) names to values can be greatly improved upon. For hierarchies without multiple inheritance one can easily allocate fixed-length arrays to store values, and translate field-lookup operations into direct array-access. With multiple inheritance one must be more careful, in some cases leaving empty "padding" in the array in order to leave room for fields declared for incomparable elements that are shared by common descendents. This process complicates compilation but in fact maintains constant-time field access at run time. More flexible methods combining partial array allocation (say, for frequently-accessed attributes declared high in the hierarchy) and partial use of association lists (for other attributes) allow run-time additions of new fields to existing objects and circumvent the need for padded (space inefficient) arrays.

5 Concluding Remarks

We have argued that, by generalizing term rewriting so as to encompass also graph and object-oriented rewriting, a very wide spectrum of parallel programming applications can naturally be expressed with rewrite rules in a declarative and machine-independent way. We have then presented methods to efficiently compile these very general types of rewrite rules onto SIMD and MIMD/SIMD parallel machines. Since rewriting is a very simple way of implementing and parallelizing not only functional languages but also many other declarative and constraint-based languages (see [27, 20]), our compilation techniques are directly applicable not only to Maude, but also to the parallel implementation of many other declarative languages on SIMD and MIMD/SIMD machines.

Our techniques are applicable under relatively general assumptions such as the A-SIMD capabilities enjoyed by a good number of SIMD and MIMD/SIMD machines. However, for the sake of concreteness we have also reported on our compilation experience for the RRM, a particular MIMD/SIMD machine in this class specifically designed to support parallel rewriting.

Important areas requiring further research include: a tighter integration of mapping and compilation techniques for graph rewriting; further experimentation with compilation of object-oriented rewriting; efficient code scheduling for a set of rewrite rules; and techniques for efficiently handling very large problems that do not fit in active memory. In addition, the machine-independence of our proposed paradigm and compilation techniques should be demonstrated in practice by developing implementations on several SIMD, MIMD/SIMD, and MIMD/Sequential machines.

References

1. . Making parallel programming machine-independent. DRAFT. Submitted to PARLE'94, November 1993.
2. G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, 1988.
3. 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.
4. 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.
5. 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.
6. Hassan Ait-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11:115–146, 1989.
7. J.-P. Banâtre and D. Le Mètayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.
8. Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. In *Proc. POPL'90*, pages 81–94. ACM, 1990.
9. Yves Caseau. Efficient handling of multiple inheritance hierarchies. In *OOPSLA'93*, Washington, September 1993.
10. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
11. K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, 1992.
12. Veronica Dahl and Andrew Fall. Logical encoding of conceptual graph lattices. In *Proc. 1st International Conference on Conceptual Structures*, 1993.
13. H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Graph Grammars and their Application to Computer Science.* Springer LNCS 532, 1991.
14. Gerard Ellis. Efficient retrieval from hierarchies of objects using lattics operations. In *Proc. 1st International Conference on Conceptual Structures*, Quebec City, August 1993.
15. Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
16. Joseph Goguen, José Meseguer, Sany Leinwand, Timothy Winkler, and Hitoshi Aida. The rewrite rule machine. Technical Report SRI-CSL-89-6, SRI International, Computer Science Laboratory, March 1989.
17. D.R. Jefferson. Virtual time. *Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.
18. R. Keller and J. Fasel, editors. *Proc. Workshop on graph reduction, Santa Fe, New Mexico.* Springer LNCS 279, 1987.
19. Patrick Lincoln, José Meseguer, and Livio Ricciulli. The MIMD/SIMD Rewrite Rule Machine architecture and its performance. To Appear, November 1993.

20. Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.
21. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
22. José 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.
23. José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In Oscar M. Nierstrasz, editor, *Proc. ECOOP'93*, pages 220–246. Springer LNCS 707, 1993.
24. 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.
25. Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.
26. M.J.Berger and S.H.Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-35(5):570–580, 1987.
27. Simon Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.
28. M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. vanEekelen, editors. *Term Graph Rewriting.* Wiley, 1993.

This article was processed using the LaTeX macro package with LLNCS style