Proceedings of International Conference
on Artificial Intelligence

Parallel Programming of Rule-Based Systems with Decomposition Abstraction

Shiow yang Wu

Institute of Computer Sciences and Information Engineering
National Dong Hwa University
Hualien, Taiwan, R.O.C.

Abstract

Decomposition abstraction is the process of organizing and specifying decomposition strategies for the exploitation of parallelism available in an application. In a recent paper [14], we have developed and evaluated declarative primitives for rule-based programs that expand opportunities for parallel execution. In this paper, we discuss the programming of parallel rule-based systems using decomposition abstraction primitives. We propose methodologies for transforming sequential rule programs into parallel programs and for programming parallel systems from scratch. Preliminary implementation and experimentation results demonstrate scalable and broadly available parallelism.

1 Introduction

Production systems have been shown to be a powerful architecture for intelligent systems [4]. Initial implementations of production systems suffered from poor performance which prohibited their use in large scale applications. Nevertheless, applications of rule-based programming have continued to expand.

Intuition suggests that languages based on the production system model admit a high degree of parallelism [5]. Efforts to exploit parallel processing to increase production system performance have been ongoing for over a decade [1]. However, the maximum speedup achieved by actual implementation rarely exceeds tenfold and has never done so over a general suite of applications no matter how many processors are used.

Most of the existing techniques for parallel production systems are, from a methodology point of view, similar to the techniques used in the parallelization of sequential imperative languages (mostly FORTRAN) [2]. Critical part(s) of the sequential execution is(are) parallelized, or optimizing compilation and transformations are applied to automatically transform a sequential program into a parallel program. This approach has the obvious benefit of its general applicability to existing sequential programs. However, the experiences show that these techniques have met with limited success, both on imperative languages [2] and rule languages [5].

In a recent paper [14], we have developed and evaluated declarative primitives for rule-based programs that expand opportunities for parallel execution. In this paper, we discuss the programming of parallel rule-based systems using *decomposition abstraction primitives*. We propose methodologies for transforming sequential rule programs into parallel programs and for programming parallel systems from scratch. Preliminary implementation and experimentation results demonstrate scalable and broadly available parallelism.

2 Related Work

Early research on parallel production systems focused almost exclusively on *parallel matching* [1, 5]. *Multiple rule firing systems* parallelize not only the match phase, but also the act phase by firing multiple rules in parallel [7, 9, 12]. Some systems even fire rules asynchronously [8]. Compile-time syntactic analysis of *data dependency graph* [7] is used to detect possible interference between rules. Instantiations of *compatible rules* [9] can be fired in parallel. For dependencies that can not be resolved at compile-time, run-time analysis is applied to increase the parallelism.

All techniques above are domain insensitive since parallelism specific to the application domains is not exploited. The benefit of firing multiple rules can easily be overwhelmed by the cost of synchronization and run-time interference analysis [11]. As a result, only limited speedup was achieved.

On the other hand, the SPAM/PSM system [6] exploits *task-level parallelism* and the PARULEL language [13] employs a meta-level rule system to select rule instantiations for parallel execution. These systems achieved better results by exploiting application

specific parallelism. However, the techniques employed tend to be ad hoc or incur excessive overhead.

Our main contributions are to provide abstraction mechanisms and programming methodologies which effectively exploit application parallelism without the high cost of run-time interference detection or instantiation selection.

3 Decomposition Abstraction

Decomposition abstraction(DA) is the process of organizing and specifying decomposition strategies for the exploitation of parallelism available in an application. In this section, we briefly describe a set of DA mechanisms, detailed elsewhere [14], for parallel programming of rule-based systems.

3.1 An Object-Based Framework

We have proposed a general object-based framework and an abstract rule notation for the general applicability of our results and for ease of discussion. The framework is built on top of a unified object model with objects, methods, and classes of the usual meanings. We briefly summarize our framework and the rule notation to set the stage for discussing the programming issues.

Definition 1 (Rule) *A rule is a condition-action pair. Conditions can be positive or negative.*

- *If v is a variable name, C is a class name and E is a quantifier-free first order expression, then $(v : C :: E)$ is a* **positive condition** *and $-(v : C :: E)$ is a* **negative condition**.
- *A* **rule** *is a triple (P, N, M) where P is a non-empty set of positive conditions, N is a set (possibly empty) of negative conditions, and M is a set of method invocations.*
- *A positive or negative condition is termed a* **condition element**. *The set of all condition elements is called the* **antecedent**. *The set of method invocations is called the* **consequent**. □

Definition 2 (Instantiation) *The* **state** *of a rule system is the set of objects in working memory. Given a state S, a rule is satisfied in S if there exists at least one set of objects such that all positive conditions are satisfied and none of the negative conditions are satisfied. The set of objects satisfies the positive condition elements is called an* **instantiation** *of the rule.* □

Definition 3 (Rule Firing) *If S is a state, r is a rule which is satisfied in the state, the result of* **firing** *the rule is a new state S' obtained from S by invoking the methods in the consequent of r on the set of objects which is an instantiation of r.* □

Definition 4 (Parallel Rule Firing) *Two instantiations are* **compatible** *if their execution do not interfere with each other. A set of instantiations is compatible if the instantiations are pair-wise compatible. The result of* **parallel firing** *of compatible instantiations is a new state obtained by invoking all methods on corresponding objects of the instantiations.* □

3.2 Set Selection Conditions

A positive condition element enclosed in square brackets is a *set selection condition* denoting that all qualified objects should be processed by the consequent and that they can be processed independently. The rule below specifies that for a department d, select all poor employees and raise the salary of each one of them by 10%.

rule Raise_All_Poor_Employees {
(d : *Dept*),
[e : *Emp* :: $e.dept == d.name \wedge e.salary < 10000$]
→
$e.salary = e.salary + e.salary/10$ }

3.3 Aggregate Operators

The set of objects selected by a set selection condition can also be processed as a whole by *aggregate operators* such as **count**, **sum**, **max**, **min**, and **avg**. In a set selection condition [$v : C :: E$], v denotes an individual and $v*$ the whole set of selected objects, respectively. For example, the following rule computes the number of poor employees in a department.

rule *Count_Poor_Employees* {
(d : *Dept*),
[e : *Emp* :: $e.dept == d.name \wedge e.salary < 10000$]
→
$d.poor\_emps$ = **Count**($e*$) }

3.4 ALL Combinators

For complex decomposition involving multiple classes of objects, the *ALL combinator* is used to group together several condition elements into an *ALL condition* to denote that any consistent collection of objects and set objects (for set selection conditions) can be considered independent, and therefore all of them can be processed in parallel. The following rule specifies that the number of poor employees of each departments can be computed in parallel.

rule *Count_All_Poor_Employees* {
All ((d : *Dept*),
[e : *Emp* :: $e.dept == d.name \wedge e.salary < 10000$])

$d.poor\_emps = \textbf{Count}(e*)$ }

3.5 DISJOINT Combinators

The *DISJOINT combinator* is used to combine several condition elements into a *DISJOINT condition* for denoting that objects matching the enclosed conditions are to be decomposed in a disjoint pattern. In other words, for any two instantiations of a rule with DISJOINT combinator, as long as the selected set of objects for the enclosed conditions are disjoint, they are parallel executable. The rule below specifies that all teams can be formed at the same time as long as the selected employees are mutually disjoint.

rule *Team_All_Employees* {
Disjoint (
($e1 : Emp :: e1.team ==$ **unknown**),
($e2 : Emp :: e2.team ==$ **unknown**),
($e3 : Emp :: e3.team ==$ **unknown**))

$e1.team = \textbf{new}\ Team(e1, e2, e3)$,
$e2.team = e3.team = e1.team$ }

3.6 Contexts

A rule of the form

rule r **in context** T { ... }

denotes that the rule r is designed for context T. A *context rule* of the form

$$T \vdash T_1, T_2, \ldots, T_n$$

specifies that context T is causally dependent on contexts $T_1, T_2, \ldots, T_n$ which means, to solve T, all $T_1, T_2, \ldots, T_n$ must be solved first. For example, the context rule below specifies that before working on salary adjustment, we must perform profit evaluation and salary survey.

Salary_Adjustment $\vdash$
Profit_Evaluation, *Salary_Survey*

3.7 Semantic Interference Analysis

In a recent paper [14], we proposed a semantic-based approach for the analysis of rule interference based on associative relationships among data objects. A new notion of *functional dependency* was introduced. As an example, if a team uniquely determines the set of players associated with the team, then the functional dependency

$$\{Team\} \rightarrow \{Players\}$$

holds for the application. We have shown that functional dependencies can be used to effectively derive information about whether a rule is self-interfering and about the interference between different rules.

4 From Sequential to Parallel

In this section, we present a set of heuristics to assist the programmer in converting an existing sequential program into a parallel program using the DA mechanisms.

4.1 Repeatedly Firing Rules

Commonly, rules that fire repeatedly in a sequential program can actually be fired in parallel. For example, the following rule calculates the GPA for all students.

```
rule Calculate_GPA {
    (s:Student :: s.GPAdone == NO)
-->
    s.GPA = calculate_GPA(s),
    s.GPAdone = YES }
```

Another case is a set of rules that explicitly simulate a loop. One rule initializes the loop. One or more rules constitute the body and the last one detects the end. For example, the following three rules calculate the GPAs under a task control.

```
rule Calculate_GPA_Loop_Init {
    (t:Task :: t.task == PREVIOUS)
    (s:Student :: s.GPAdone == NO)
-->
    t.task = CALCULATE }

rule Calculate_GPA_Loop_Body {
    (t:Task :: t.task == CALCULATE),
    (s:Student :: s.GPAdone == NO)
-->
    s.GPA = calculate_GPA(s),
    s.GPAdone = YES }

rule Calculate_GPA_Loop_End {
    (t:Task :: t.task == CALCULATE),
   -(s:Student :: s.GPAdone == NO)
-->
    t.task = NEXT }
```

Both types of repeated firing rules can be easily transformed into DA rules using set selection condition as follows.

```
rule DA_Calculate_GPA {
    [s:Student :: s.GPAdone == NO]
-->
    s.GPA = calculate_GPA(s),
    s.GPAdone = YES }
```

Heuristic 1 *Transform a rule that fires repeatedly on a class of objects by changing the condition that matches the class into a set selection condition.*

4.2 Accumulation Rules

There is no construct in sequential rule languages to do aggregate numerical computations such as counting, computing the sum, maximal, minimal, average, etc. Instead, they are "simulated" by a set of rules that implement counters and loops as the following example.

```
rule Count_Straight_A_Students_Init {
     (t:Task :: t.task == CALCULATE),
    -(s:Student :: s.GPAdone == NO)
-->
     t.task = COUNT,
     count = 0 }

rule Count_Straight_A_Students_Body {
     (t:Task :: t.task == COUNT),
     (s:Student :: s.GPA == 4.0
        && s.counted == NO)
-->
     count = count + 1,
     s.counted = YES }

rule Count_Straight_A_Students_End {
     (t:Task :: t.task == COUNT),
    -(s:Student :: s.counted == NO)
-->
     print_count(count),
     t.task = NEXT }
```

This type of rules can be transformed into a single DA rule using the set selection conditions and aggregate operators.

```
rule DA_Count_Straight_A {
     [s:Student :: s.GPA == 4.0]
-->
     count = Count(s*),
     print_count(count) }
```

Heuristic 2 *Transform a set of rules for accumulation into a single DA rule by changing the condition that matches the class of objects to be accumulated into a set selection condition, and by using appropriate aggregate operators on the selected set in the consequent.*

4.3 Nested Rules

Nested rules are often used when objects of several related classes need to be processed repeatedly. Suppose we want to count the number of students in all departments.

```
rule Count_Students_Init {
     (d:Dept :: d.counted == NO)
-->
     d.count = 0 }

rule Count_Students_Body {
     (d:Dept :: d.counted == NO),
     (s:Student :: s.dept == d.name &&
                   s.counted == NO)
-->
     d.count = d.count + 1,
     s.counted = YES }

rule Count_Students_End {
     (d:Dept :: d.counted == NO),
    -(s:Student :: s.dept == d.name &&
                   s.counted == NO)
-->
     d.counted = YES }
```

This is a standard nested loop over two classes of objects. They can be transformed into a single DA rule using the ALL combinator.

```
rule DA_Count_Students {
    All((d:Dept :: d.counted == NO),
        [s:Student :: s.dept == d.name])
-->
    d.count = Count(s*),
    d.counted == YES }
```

Heuristic 3 *Transform nested rules into a single DA rule using the ALL combinator. Enclose all conditions that match the target objects into the combinator.*

4.4 Disjointness Rules

In sequential rule languages, rules that fire repeatedly on disjoint partitions of data objects must have the disjointness property explicitly specified. As an example, the following rule assigns projects to groups of three students.

```
rule Assign_Projects {
    (p :Project :: p.assigned == NO),
    (s1:Student :: s1.assigned == NO),
    (s2:Student :: s2.assigned == NO
        && s2 != s1),
    (s3:Student :: s3.assigned == NO
        && s3 != s1 && s3 != s2)
-->
    p.assigned = YES,
    s1.proj = s2.proj = s3.proj = p.name,
    s1.assigned = s2.assigned = YES,
    s3.assigned = YES }
```

This type of rule is inefficient, hard to read and counter intuitive. With DISJOINT combinator, the rule above can be transformed into a much better DA rule.

```
rule DA_Assign_Projects {
   Disjoint(
     (p :Project :: p.assigned == NO),
     (s1:Student :: s1.assigned == NO),
     (s2:Student :: s2.assigned == NO),
     (s3:Student :: s3.assigned == NO))
-->
   p.assigned = YES,
   s1.proj = s2.proj = s3.proj = p.name,
   s1.assigned = s2.assigned = YES,
   s3.assigned = YES }
```

Heuristic 4 *Transform rules that fires repeatedly on disjoint partitions of data objects using DISJOINT combinator. Enclose all conditions involving the disjointness test and remove the test.*

4.5 Secrete Messages to Explicit Contexts

The use of so-called "secret-messages" is a common technique to emulate procedural control. This technique employs a designated WME (usually called the *goal element*) to control the phases of execution. For example, the following two rules perform the computation and phase change, respectively.

```
rule Calculate_GPA {
    (g:Goal :: g.task == CALCULATE),
    (s:Student :: s.GPAdone == NO)
-->
    s.GPA = calculate_GPA(s),
    s.GPAdone = YES }

rule Calculate_GPA_to_Print_Results {
    (g:Goal :: g.task == CALCULATE),
   -(s:Student :: s.GPAdone == NO)
-->
    g.task = PRINT_RESULTS }
```

This programming style can be easily mapped into DA context mechanism as follows.

```
rule DA_Calculate_GPA
in_context CALCULATE_GPA {
    [s:Student :: s.GPAdone == NO]
-->
    s.GPA = calculate_GPA(s),
    s.GPAdone = YES }

PRINT_RESULTS |- CALCULATE_GPA
```

Heuristic 5 *Transform rules that match goal element into DA rules that use explicit contexts. Replace phase changing rules with context rules.*

5 Programming in Parallel

In this section, we discuss issues about programming parallel rule systems directly using decomposition abstraction.

5.1 A Simple Course Scheduling System

We use a course scheduling application as an example. The problem is to schedule courses by assigning instructors, time, and classrooms. Each instructor can teach a number of courses. An instructor should be assigned no more than three courses. A student is registered in a unique department and can take a number of courses. Each course is two hours long and to be assigned to the time slot of 8:00am, 10:00am, 1:00pm, or 3:00pm during weekdays. Each department building has a number of classrooms. A classroom has a fix capacity. Some classrooms have special equipments.

5.2 Identify Application Objects

The very first step is to analyze the problem and identify application objects. Any object-oriented analysis and design techniques can be applied. The key is to think in terms of application instead of implementation. Any noun that is mentioned more than once in the problem statement is probably a good candidate. We identify the following application objects: **departments**, **courses**, **instructors**, **students**, **time slots**, and **classrooms**. Each type of objects should be defined by a proper class definition.

5.3 Identify Functional Dependencies

The next thing to do is to identify the relationships between application objects. We are particularly interested in the functional dependencies between objects of different classes. For the sample application, we can identify the following functional dependencies.

```
{ Department } --> { Classroom }
{ Department } --> { Course }
{ Department } --> { Student }
```

5.4 Identify Tasks

This is to figure out a solution plan. A problem can be solved by identifying the transformations that need to be applied on the data objects to produce the desired results. Each transformation corresponds to a task. A task can be further decomposed recursively into subtasks until a subtask is manageable. Each task can be represented by a context which is designed separately. For the purpose of illustration, we adopt a simple solution plan by identifying the following possible tasks:

- Gather information.
- Schedule courses that require special equipments.
- Schedule courses for senior faculty.

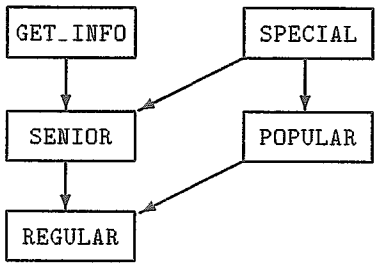

Figure 1: Partial Order Derived from the Context Rules of the Course Scheduling Problem

- Schedule popular courses.
- Schedule regular courses.

5.5 Identify Parallelism

The arguably most important step, as performance is concerned, is to identify potential parallelism in the applications. An effective technique is to analyze the problem alone two dimensions: *function decomposition* and *data decomposition*.

Function Decomposition Function decomposition involves the partitioning of the solution plan into subtasks as in last section, and the identification of task structure. Task structure is represented by the causal dependencies between tasks which can be specified by context rules. For the example problem, we have:

```
SENIOR  |- SPECIAL
POPULAR |- GET_INFO, SPECIAL
REGULAR |- GET_INFO, SPECIAL, SENIOR,
           POPULAR
```

The set of context rules specifies a partial order which can be used to identify independent contexts that can be executed in parallel. Figure 1 shows the partial order derived from the context rules above. Clearly, the GET_INFO and SPECIAL contexts can be executed in parallel. Similarly for the SENIOR and POPULAR contexts.

Data Decomposition Data decomposition involves the partitioning of data objects for data-parallel or SPMD style computation. In rule languages, this is achieved through pattern matching in the antecedents and concurrent execution of multiple instantiations. In particular, we represent the transformations that need to be applied on data objects as rules. The pattern matching will dynamically decompose the data into desired partitions for processing in parallel.

5.6 Writing DA Rules

After all the steps in previous sections have been performed, writing DA rules is more a specification process than a design process. There are some guidelines to follow though:

1. Write a set of rules for each context.
2. Write a rule for each transformation identified in the data decomposition process.
3. Within a rule, add a positive condition for each type of objects to be transformed. Add negative conditions to specify additional constraints. Spell out the transformations as actions in the consequent. Use DA mechanisms if the rule corresponds to a SPMD style transformation.

We present some rules for the sample application. In the GET_INFO context, we need the number of registrants for each course. This is done by using a set selection condition to select all students of a course and the aggregate operation **Count** to compute the number, and finally the ALL combinator to do all courses in parallel. Note that we use <| instead of $\in$ as the set membership operator for ease of typing.

```
rule Count_Regs in_context GET_INFO {
   All((c: Course :: c.counted == NO),
       [s: Student :: c.name <| s.take*])
-->
   c.registrants = Count(s*),
   c.counted = Yes }
```

The SPECIAL context is special in that some courses need special equipments that are available only in certain classrooms.

```
rule Schedule_Special in_context SPECIAL {
   (t:Time),
   Disjoint(
     (c:Course :: c.scheduled == NO
        && c.special_equip != NULL),
     (i:Instructor :: c.name <| i.teach*
        && i.assigned < 3),
     (r:Classroom :: t.time <| r.slots*
        && c.special_equip <| r.equip*))
-->
   c.instructor = i.name,
   c.classroom = r.number,
   c.time = t.time,  c.scheduled = YES,
   i.assigned = i.assigned + 1,
   r.slots* = r.slots* - t.time }
```

For the SENIOR context, we schedule just one course for whatever a senior instructor wants to teach.

```
rule Schedule_Senior in_context SENIOR {
   (t:Time),
```

```
    Disjoint(
      (c:Course :: c.scheduled == NO)
      (i:Instructor :: i.is_senior == YES
         && c.name <| i.teach*
         && i.assigned < 1),
      (r:Classroom :: t.time <| r.slots*))
-->
    c.instructor = i.name,
    c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.slots* = r.slots* - t.time }
```

The POPULAR context schedules courses with number of registrants exceeding a threshold.

```
rule Schedule_Popular in_context POPULAR {
    (t:Time),
    Disjoint(
      (c:Course :: c.scheduled == NO
         && c.registrants > THRESHOLD),
      (i:Instructor :: i.is_senior == NO
         && c.name <| i.teach*
         && i.assigned < 3),
      (r:Classroom :: t.time <| r.slots*))
-->
    c.instructor = i.name,
    c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.slots* = r.slots* - t.time }
```

Now rest of the courses can be assigned simply by pattern matching.

```
rule Schedule_Regular in_context REGULAR {
    (t:Time),
    Disjoint(
      (c:Course :: c.scheduled == NO),
      (i:Instructor :: i.is_senior == NO
         && c.name <| i.teach*
         && i.assigned < 3),
      (r:Classroom :: t.time <| r.slots*))
-->
    c.instructor = i.name,
    c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.slots* = r.slots* - t.time }
```

6 Preliminary Implementation

We have conducted a good variety of experiments on a preliminary implementation of our decomposition abstraction mechanisms. The run-time system was built on top of an object-oriented thread package called Presto[3] on the Sequent Symmetry shared-memory multiprocessors. Because of the space limit, we only presents the set of results on a combinatorial search program for seat assignment called MANNERS.

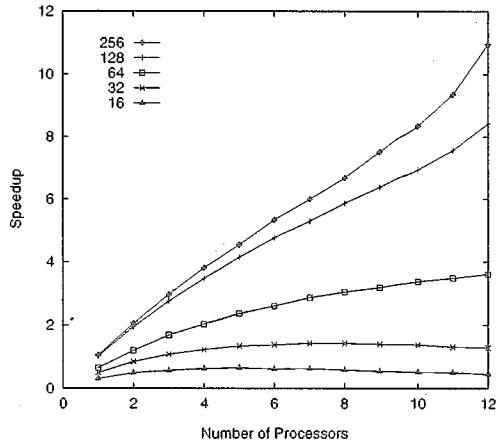

Figure 2: MANNERS overall speedup on different problem size.

6.1 Overall Speedup and Scaling Results

Figure 2 shows the overall speedup results of the MANNERS program on problem size of 16, 32, 64, 128, and 256 guests. Each point is the mean of at least 10 runs. This set of experiments is a good indication of the effectiveness and scalability of the DA mechanism. First of all, the system exhibits good speedup behavior. We also have the desired behavior of scalable speedup, both in terms of number of processors and problem size.

6.2 Parallel vs. Sequential

Our implementation is based on the LEAPS algorithm [10] for searching instantiations. Multiple LEAPS engines (LEs) work together to search for compatible instantiations and fire them as soon as they were found. Figure 3 demonstrates a comparison of the number of key operations done by the LEs and the sequential system. We can see that the extra work due to parallelism is within a constant factor of the sequential execution work. The constant does not increase with the problem size either. Also note that the total number of rule firing of the parallel version is smaller than the sequential version because of the context mechanism employed which eliminates the context switching rules.

7 Conclusions and Future Work

Decomposition abstraction and the mechanisms we proposed raise the level of abstraction from implementation level to application level. Rules are much easier

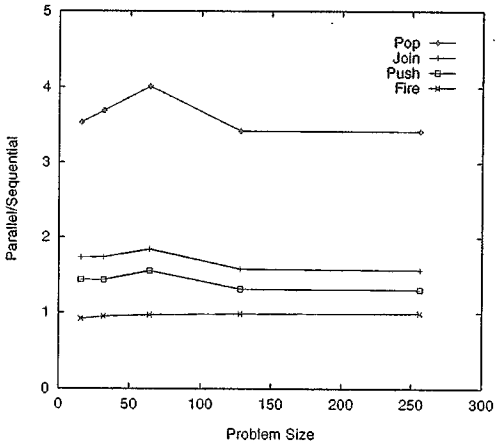

Figure 3: Parallel vs. sequential execution.

to write since they are closer to application semantics. The number of rules tend to be significantly less than corresponding sequential program. In writing the rules, however, it is still the programmer's responsibility to ensure the correctness of the specified semantics.

On the other hand, it is possible to detect violation of specified semantics before the program execution. One of our primary future research directions is to develop theories and techniques for consistency checking and correctness validation of DA programs.

Another direction of our future research is to formally compare the programmability, complexity, and effectiveness of decomposition abstraction with other approaches.

References

[1] José Nelson Amaral and Joydeep Ghosh. Speeding up production systems: From concurrent matching to parallel rule firing. In L. Kanal, V. Kumar, H. Kitano, and C. Suttner, editors, *Parallel Processing for Artificial Intelligence*, chapter 1. Elsevier Science Publishers B.V., 1993.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[3] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A system for object-oriented parallel programming. *Software - Practice and Experience*, 18(8):713–732, August 1988.

[4] B. G. Buchanan and E. H. Shortliffe. *Rule-Based Expert Systems.* Addison-Wesley, Reading, MA, 1984.

[5] Anoop Gupta, Charles Forgy, and Allen Newell. High-speed implementation of rule-based systems. *ACM Trans on Computer Systems*, 7(2):119–146, May 1989.

[6] Wilson Harvey, Dirk Kalp, Milind Tambe, David McKeown, and Aleen Newell. The effectiveness of task-level parallelism for production systems. *Journal of Parallel and Distributed Computing*, 13(4):395–411, December 1991.

[7] T. Ishida and Salvatore J. Stolfo. Toward the parallel execution of rules in production system programs. In *IEEE Intl. Conf. on Parallel Processing*, pages 568–574, 1985.

[8] Chin-Ming Kuo, Daniel P. Miranker, and James C. Browne. On the performance of the CREL system. *Journal of Parallel and Distributed Computing*, 13(4):424–441, December 1991.

[9] Steve Kuo and Dan Moldovan. Implementation of multiple rule firing production systems on hypercube. *Journal of Parallel and Distributed Computing*, 13(4):383–394, December 1991.

[10] Daniel P. Miranker and David A. Brant. An algorithmic basis for integrating production systems and large databases. In *Proc. 6th Intl. Conf. on Data Engineering*, pages 353–360, February 1990. The LEAPS algorithm.

[11] Daniel E. Neiman. *Design and Control of Parallel Rule-Firing Production Systems.* PhD thesis, University of Massachusetts at Amherst, September 1992.

[12] James G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13(4):348–365, December 1991.

[13] Salvatore J. Stolfo, Ouri Wolfson, Philip K. Chan, Hasanat M. Dewan, Leland Woodbury, Jason S. Glazier, and David A. Ohsie. PARULEL: Parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing*, 13(4):366–382, December 1991.

[14] Shiow-yang Wu, Daniel P. Miranker, and James C. Browne. Decomposition abstraction in parallel rule languages. *IEEE Trans. on Parallel and Distributed Systems*, 1996. to appear.