Toward Semantic-Based Parallelism in Production Systems

Shiow-yang Wu, Daniel P. Miranker, and James C. Browne

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188

Abstract

We propose a new approach for the parallel execution of production system programs. This approach embodies methods of decomposition abstraction using declarative mechanisms. Application semantics can then be exploited to achieve a much higher degree of concurrency. In this paper we present the underlying object-based framework of production systems and discuss the ensuing semantic-based dependency analysis technique. In particular, we define a new notion of functional dependency to characterize associative relationships among data objects, which can be used to determine concurrently executable rules.

1 Introduction

A production system is composed of a working memory, a set of rules, and an inference engine. Working memory is a global database composed of data objects called working memory elements (WMEs) representing the system state. A rule is a condition-action pair. The inference engine provides a three-phase cyclic execution model of condition evaluation (matching), conflict-resolution and action firing. A rule with a set of WMEs satisfying the conditions is called an instantiation. The set of all instantiations constitutes the conflict set. In a sequential environment, conflict-resolution selects one instantiation from the conflict set for firing. In a parallel environment, multiple rule instantiations can be selected for firing simultaneously subject to proper correctness constraints. Firing an instantiation executes the selected actions which may add, delete, or modify WMEs in the working memory. The cycle then starts over again.

Initial implementation of production systems suffered from poor performance which prohibited their use in large scale applications [3]. On the other hand, production systems have been assumed to encompass a high degree of parallelism [4], opening the opportunity of performance improvement through parallel processing. However, after over a decade of extensive research effort [14], the speedup achieved by systems with real implementation is quite limited, only about 10-fold, no matter how many processors are used.

In a recent paper [16], we analyzed several commonly used benchmark programs and pointed out that the rather limited success in the past was primarily due to the failure to properly exploit parallelism embedded in the application domains and program structures. Contrary to the conclusion drawn by previous work that the true performance gain from parallelism is quite limited [4], we showed that massive and scalable speedup was indeed achievable with a set of explicit parallel structuring mechanisms.

In this paper, we propose a semantic-based approach for the analysis of rule interference based on associative relationships among data objects. First, a general object-based framework of production systems is proposed both for a formal basis and for the expression of parallelism in a language independent way. Then, a notion of functional dependency is introduced to derive information about whether a rule is self-interfering and about the interference between different rules. We show how the combination of explicit parallel structuring mechanisms and semantic-based analysis technique achieve much higher level of concurrency than traditional techniques.

2 Related Work

Early research on parallel production systems focused almost exclusively on parallel matching [10, 4]. These systems parallelized only the match phase. The speedup is therefore limited by the sequential execution of rules. Multiple rule firing systems parallelize
not only the match phase, but also the act phase by firing multiple rules in parallel [6, 8, 13]. Some systems even fire rules asynchronously [12, 7]. Compile-time syntactic analysis of data dependency graph [6] is used to detect possible interference between rules. Instantiations of compatible rules [8] 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 [5] exploits task-level parallelism and the PARULEL language [15] employs a meta-level rule system to select compatible 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 semantic-based analysis techniques which effectively exploit application parallelism without the high cost of run-time interference detection or instantiation selection.

3 A General Object-Based Framework of Production Systems

The concepts and techniques presented in this paper are language independent. We propose a general object-based framework to abstract away minor details and to capture just the essential features of production systems. Thus the results presented in this paper are generally applicable to any rule language.

3.1 Object Model

We have built our framework on top of a unified object model which can be used to characterize all entities in a rule system.

Definition 1 (Method) A method definition is a triple \((M, P, B)\) where \(M\) is a method name, \(P\) is a set of parameter specifications, and \(B\) is the definition of operations performed by the method. A method invocation is a method name with necessary parameters fully supplied. □

We have deliberately left out the details of how a parameter or body of a method is actually specified. Nor do we restrict the way actual arguments are passed in a method invocation. These issues are not essential to our discussion.

Definition 2 (Class) A class defines a set of objects with similar structure and behavior.

- \(\text{INT, FLOAT, and STRING}\) are primitive classes representing the set of integers, floats, and character strings, respectively.
- An attribute definition is a pair \((a, C)\) where \(a\) is an attribute name and \(C\) is a class name.
- A class is a triple \((C, A, M)\) where \(C\) is a class name, \(A\) is a set of attribute definitions, and \(M\) is a set of method definitions. □

Definition 3 (Object and WME) Objects are defined to model WMEs. They are the basic units of information and behavior encapsulation.

- Integers, floats, and character strings are primitive objects.
- If \(a_1, a_2, \ldots, a_n\) are the attribute names of a class \(C\) and \(O_1, O_2, \ldots, O_n\) are objects, then:

\[
O = (a_1: O_1, a_2: O_2, \ldots, a_n: O_n)
\]

is a structural object or simply a tuple. The object is an instance of the class \(C\).
- Tuples are the generalization of WMEs. Each tuple has a unique identifier associated with it. Working memory is a set of tuples. □

Definition 4 (Rule) A rule is a condition-action pair. Conditions can be positive or negative.

- An expression is a quantifier-free first order formula.
- If \(v\) is a variable name, \(C\) is a class name and \(E\) is an 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 5 (Program and System) A program is a pair \((C, R)\) where \(C\) is a set of class definitions and \(R\) is a set of rule definitions. A rule system is a pair \((O, P)\) where \(O\) is a set of tuples and \(P\) is a rule program. \(\square\)

3.2 Execution Model and Semantics

We characterize the semantics by considering rule antecedents as queries to the working memory for selecting a consistent set of objects. The execution of a rule system is defined in terms of state transitions between working memory states.

Definition 6 (State) The state of a rule system is the set of tuples in working memory. \(\square\)

Definition 7 (Instantiation) Pattern matching is modeled by object selection. Given a state \(S\):

- A positive condition \(v : C :: E\) is satisfied in \(S\) if there exists an object of class \(C\) such that \(E\) is evaluated to true. The object (which can be referenced by the variable \(v\)) is said to be selected by the condition element.

- A negative condition \(-v : C :: E\) is satisfied in \(S\) if there does not exist any object of class \(C\) such that \(E\) is evaluated to true.

- A rule is satisfied in \(S\) if there exists at least one set of objects in \(S\) such that all condition elements in the antecedent are satisfied. The set of objects selected by the positive condition elements in the antecedent are satisfied. The set of objects selected by the positive condition elements is called an instantiation of the rule. The set of all instantiations of a rule \(r\) is denoted by \(\text{Inst}(r)\). \(\square\)

Operationally, a rule can be considered as a query to the working memory. The result of the query is a class whose instances are instantiations of the rule.

Definition 8 (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 \(i\) which is in an instantiation of \(r\). We denote such a rule firing by \(S' = S(i)\). \(\square\)

Definition 9 (Execution) An execution of a rule system is a sequence of rule firings that transforms the system from a state to another state. A state is a terminal state if no rule is satisfied under that state. An execution is a terminal execution if the last state in the sequence of rule firings is a terminal state. \(\square\)

It is important to note that in the definitions of rule firing and execution, no restriction is placed on how objects are selected or on which rule instantiation to pick. In other words, no matching technique or conflict resolution strategy is assumed. An execution is not required to be a terminal execution. Thus allowing nonterminating systems.

The framework and execution model above characterize the core concepts and essential features of a sequential production system. We now extend the model to allow simultaneous firing of multiple rule instantiations.

Definition 10 (Interference) If \(i_1\) and \(i_2\) are instantiations of two (possibly the same) rules that are satisfied in a state \(S\), then \(i_1\) interferes with \(i_2\) if any one of the following conditions is true:

1. The execution of \(i_1\) prevents \(i_2\) from being an instantiation in the new state resulting from \(i_1\)'s execution, or vice versa.

2. There exists methods invoked by \(i_1\) and \(i_2\) that modify the same object. \(\square\)

Since a newly created object is always assigned a unique identifier, object creations do not contribute to any interference except when Condition 1 is true. Identical objects with different identifiers are allowed to coexist in our model, which is consistent with most rule languages.

We note that it is possible to weaken Condition 2 above since we need only to avoid conflicting methods to be invoked on the same object. However, such fine-grained parallelism can be easily overwhelmed by the potential complexity. We reserve this issue for future research.

Definition 11 (Compatibility) Two instantiations are said to be compatible if they do not interfere with each other. A set of instantiations is compatible if the instantiations are pair-wise compatible. \(\square\)

Since compatible instantiations do not interfere with each other, they can be executed in parallel. Our definitions of interference and compatibility are similar to the corresponding definitions in [6, 7, 8, 13] which are all essentially originated from database concurrency control theory [1]. However, we extend it to a general object-based context which allows any type of method instead of just the add, delete, and modify operations as in most previous work on parallel production systems.

754
Definition 12 (Parallel Rule Firing) The result of parallel firing of two compatible instantiations in a state is a new state obtained by invoking all methods on corresponding objects of the two instantiations. Likewise, the parallel firing of a set of compatible instantiations \( I \) in a state \( S \) is to invoke all methods on corresponding objects of all instantiations. The parallel firing is denoted by \( S' = S(I) \). □

Because of the non-interference requirement between parallel executable instantiations, the resulting state of the parallel firing is the same as the result of execution of the set of instantiations in sequence following any order. To state it more precisely, if \( I = \{i_1, \ldots, i_n\} \) is a set of parallel executable instantiations in a state \( S \), then

\[
S(I) = S(i_{j_1})(i_{j_2}) \ldots (i_{j_n})
\]

where \( j_1, j_2, \ldots, j_n \) is any permutation of \( n \).

4 Class Relationships and Rule Compatibility

Under the general framework presented in last section, we now describe our semantic-based interference analysis technique. Our approach is motivated by the observation that, more often than not, class relationships provide valuable hints on data decomposition patterns that actually happen at run time but are not necessarily clear at design or compile time. We show that this information can often be used in determining the semantic compatibility (i.e. parallel executability) of instantiations of the same rule or between different rules.

As an intuitive example, consider the following rule from the corporation application domain.

\[
\text{rule Team Fairness} \{ \\
\quad ( t : \text{Team} ), \\
\quad [ e : \text{Employee} :: e.\text{name} = t.\text{name} \land e.\text{salary} < t.\text{min. wage} ] \\
\quad \\
\quad e.\text{salary} = t.\text{min. wage} \\
\}
\]

As we presented in [16], a positive condition enclosed in square brackets is a set selection condition which is to select all objects satisfying the condition. The rule is to raise the salary of all under-paid employees in a team. In general, different instantiations of this rule can not be executed in parallel because the same employee may be a member of different teams. On the other hand, if each team is associated with a unique and disjoint set of employees, then different instantiations will select different teams with disjoint set of employees. Apparently, all such instantiations can be fired in parallel. The key point is on the relationship between instances of the Team and the Employee class. We call such relationship functional dependency which is formally characterized in the following definitions.

Definition 13 (Class Relation and Scheme) A class relation scheme (or simply scheme) is an ordered set of class names. A class relation on a class relation scheme with \( n \) class names is an \( n \)-ary relation among instances of the corresponding classes. □

For a class relation \( A \), we denote the scheme on which \( A \) is defined by \( \text{Sch}(A) \). A class relation can be considered as a collection of classes with certain relationship. Note that an element of an \( n \)-ary class relation is an ordered set of \( n \) objects, one from each corresponding class in the scheme. For an element \( a \in A \) and a scheme \( X \subseteq \text{Sch}(A) \), the notation \( a(X) \) denotes the ordered collection of objects in \( a \) which are from classes in \( X \). We note that from the definition above, \( a(X) \subseteq a \) and \( a(\text{Sch}(A)) = a \).

Definition 14 (Functional Dependency) Let \( X \) and \( Y \) be the schemes of two class relations \( R_x \) and \( R_y \). The functional dependency

\[
X \rightarrow Y
\]

holds on \( R_x \) and \( R_y \) if

1. Each element in \( R_x \) is associated with a unique element in \( R_y \).
2. For all \( a_1, a_2 \in R_x \) and the associated \( b_1, b_2 \in R_y \),
   \[
   a_1 \neq a_2 \Rightarrow b_1 \cap b_2 = \emptyset. \quad □
   \]

As an example from the corporation application domain discussed earlier, the functional dependency \( \{\text{Team}\} \rightarrow \{\text{Employee}\} \) holds when each team is associated with a unique and disjoint set of employees.

Since an instantiation can also be considered as an ordered set of objects (one for each positive or set selection condition), a rule \( r \) actually defines a class relation whose elements are exactly the set of instantiations of the rule, i.e. \( \text{Inst}(r) \). The scheme of \( \text{Inst}(r) \), denoted by \( \text{Scheme}(r) \), is the ordered set of class name components of positive and set selection conditions of \( r \).
Except for borrowing the terminology, functional dependency as defined here is quite different than in databases [9]. In database systems, the notion of functional dependency is defined at the attribute level and is used primarily in the normalization process. We generalize the concept to the class level and use it to identify the parallelism in rule systems. Functional dependencies are considered as specifications of data decomposition across class boundaries, which are shown below to play a crucial role in determining the compatibility between instantiations of the same or different rules.

**Definition 15** Let \( r \) be a rule. The access set of \( r \), denoted by Access(\( r \)), is the set of all class names referenced in the antecedent of \( r \). The write set of \( r \), denoted by Write(\( r \)), is the set of class names with objects that are modified (including creation and deletion) in the rule.

**Definition 16** (Dominant Set) Let \( r \) be a rule and \( C \) be a class relation scheme. \( C \) is a dominant set of \( r \) if:

1. \( C \subseteq \text{Scheme}(r) \),
2. for all \( i, j \in \text{Inst}(r) \) \((i \neq j \Rightarrow i(C) \neq j(C)) \).

A dominant set of a rule is simply a set of class names sufficient to discriminate between different instantiations of the rule.

**Theorem 1** (Self Compatability) Let \( r \) be a rule and \( A, B, C \) be three class relation schemes that are subsets of Scheme(\( r \)) satisfying the following conditions:

1. \( C \) is a dominant set of \( r \) and \( C \subseteq A \)
2. \( A \rightarrow B \)
3. \( \forall c \in \text{Write}(r) (c \in B \lor c \notin \text{Access}(r)) \)

then all instantiations of \( r \) are compatible (i.e. parallel executable).

**Proof sketch:** In any given state, let \( i \) and \( j \) be instantiations of \( r \) such that \( i \neq j \).

\[
\begin{align*}
    i & \neq j \\
    \Rightarrow & \ i(C) \neq j(C) \\
    \Rightarrow & \ i(A) \neq j(A) \\
    \Rightarrow & \ i(B) \cap j(B) = \emptyset \\
    \Rightarrow & \ i(\text{Write}(r)) \cap j(\text{Write}(r)) = \emptyset \\
    \Rightarrow & \ i \text{ and } j \text{ do not interfere with each other} \\
    \Rightarrow & \ i \text{ and } j \text{ are compatible}
\end{align*}
\]

The central idea of this theorem is that functional dependency implies disjoint decomposition of objects selected by the instantiations of a rule. As long as the objects modified in the consequent belong either to the decomposition or to classes which do not affect the satisfiability of the rule, no instantiations will interfere with each other. We will have examples later in this section. We first generalize this idea to the analysis of interference between multiple rules.

**Definition 17** (Partially Mutual Exclusion) Let \( p, q \) be rules and \( C \) be a class relation scheme. We say that \( p \) and \( q \) are partially mutual exclusive on \( C \), denoted by \( p \prec_C q \), if

1. \( C \subseteq \text{Scheme}(p) \) and \( C \subseteq \text{Scheme}(q) \)
2. For any two instantiations \( i, j \) of \( p \) and \( q \) respectively, \( i(C) \neq j(C) \).

Partially mutual exclusion simply means that \( p \) and \( q \) can not have instantiations containing the same set of objects of classes in \( C \). The simplest and most common case is when \( C \) contains a single class referenced in both \( p \) and \( q \) but tested on disjoint values of the same set of attributes. Since the values are disjoint, \( p \) and \( q \) can not select the same object in \( C \).

Note that no requirement is placed on selected objects that are not of the classes in \( C \). Therefore, partially mutual exclusive rules may still interfere with each other. However, in many cases, partially mutual exclusive rules can be determined to be parallel executable with the help of functional dependencies as indicated by the following theorem.

**Theorem 2** (Pair-Wise Compatibility) If \( p, q \) are two distinct rules, and \( A, B, C \) are class relation schemes that are subsets of both Scheme(\( p \)) and Scheme(\( q \)) such that the following conditions are satisfied:

1. \( p \prec_C q \) and \( C \subseteq A \)
2. \( A \rightarrow B \)
3. \( \forall c \in \text{Write}(p) (c \in B \lor c \notin \text{Access}(q)) \)
4. \( \forall c \in \text{Write}(q) (c \in B \lor c \notin \text{Access}(p)) \)

then \( p \) and \( q \) are compatible and therefore parallel executable.
Proof sketch: In any given state, let \( i \) be an instantiation of \( p \) and \( j \) be an instantiation of \( q \).

\[
\begin{align*}
p & >\succ c \quad q \\
\Rightarrow & \quad i(C) \neq j(C) \\
\Rightarrow & \quad i(A) \neq j(A) \\
\Rightarrow & \quad i(B) \cap j(B) = \emptyset \\
\Rightarrow & \quad i(\text{Write}(p)) \cap j(\text{Write}(q)) = \emptyset \\
\Rightarrow & \quad p \text{ and } q \text{ are parallel executable} \quad \blacksquare
\end{align*}
\]

Again, the central idea of this theorem is that as long as objects modified in \( p \) and \( q \) can be determined as non-overlapping with the help of functional dependency, instantiations of \( p \) and \( q \) do not interfere with each others.

Even with their general applicability to many cases, the two theorems above are less complicated than they appear. Continuing with our examples in the corporation application domain, if a team is associated with a set of disjoint employees as team members, then the functional dependency \( \{\text{Team}\} \rightarrow \{\text{Employee}\} \) holds. We note that this semantic information can be easily supplied by the programmer (similar to the identification of key attributes in database systems). With functional dependency and the fact that a team can be uniquely identified by its name, we can immediately determine that all instantiations of the \textit{Team\_Fairness} rule can be fired in parallel using Theorem 1.

As another example, the following two rules can be determined to be parallel executable by Theorem 2.

\begin{verbatim}
rule Facilities\_Research {
  ( t : Team :: t.dept == "research" )
  [ e : Employee :: e.team == t.name ]
  \rightarrow
e.equipment = "AXP500X(Alpha)"
}

rule Facilities\_Sales {
  ( t : Team :: t.dept == "sales" )
  [ e : Employee :: e.team == t.name ]
  \rightarrow
e.equipment = "PowerBook"
}
\end{verbatim}

In this case, the two rules are partially mutual exclusive on \textit{Team}. With the help of functional dependency, they can be statically determined to be parallel executable.

In general, any type of class relationship which implies certain pattern of association or partitioning in

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{waltz_speedup.png}
\caption{Waltz Speedup on Different Problem Sizes.}
\end{figure}

the application domain is of great help in the determination of proper decomposition for parallel processing. Mechanisms for expressing these relationships are therefore of great value for capturing application parallelism in program development.

5 Performance Assessment

We have conducted extensive experiments on a parallel rule execution engine we developed to evaluate the performance of our decomposition abstraction mechanisms and semantic-based interference analysis technique. With proper granularity management and scheduling strategy, we were able to obtain close to linear speedup on all benchmark programs we tested. Figure 1 is one of the simulation results on the Waltz benchmark program. It demonstrates both the effectiveness and scalability of our approach. Because of the space limit, we cannot present all results in this paper. Please refer to [17] for a through presentation of our approach to parallel production systems and the complete set of performance results.

6 Conclusions and Future Work

We have shown in this paper that, contrary to the conclusion drawn by previous research, production systems and rule-based programs do encompass high degree of concurrency and it is possible to effectively exploit such level of parallelism. These results are obtained when a programmer is provided with decomposition abstraction mechanisms to specify the natural
parallelism inherited in the application domain. The novel approach of using data relationships (functional dependency in particular) in the derivation of information for parallelism is a promising direction that has never been recognized before.

We are embodying our decomposition abstraction mechanisms on the Venus/C++-based modular rule language [2] targeting Sequent Symmetry shared-memory multiprocessors. We choose Venus because it is probably the first sequential rule-based programming language to provide both a declarative syntactic and semantic mechanism to support top-down modular design of rule-based programs. The embodiment of our mechanisms instantly converts the language into a parallel rule language, which is called Venus/DA. The language system is not only intended to serve as a test-bed for semantic-based exploration of parallelism in production systems, but also to be used for the planned extension of rule-based technology into database applications.

References


