Comparison of Three Algorithms for Ensuring Serializable Executions in Parallel Production Systems

James G. Schmolze  
Department of Computer Science  
Tufts University  
Medford, MA 02155 USA  
Email: schmolze@cs.tufts.edu

Daniel F. Neiman  
Department of Computer Science  
University of Massachusetts  
Amherst, MA 01003 USA  
Email: dann@cs.umass.edu

Abstract
To speed up production systems, researchers have developed parallel algorithms that execute multiple instantiations simultaneously. Unfortunately, without special controls, such systems can produce results that could not have been produced by any serial execution. We present and compare three different algorithms that guarantee a serializable result in such systems. Our goal is to analyze the overhead that serialization incurs. All three algorithms perform synchronization at the level of instantiations, not rules, and are targeted for shared-memory machines. One algorithm operates synchronously while the other two operate asynchronously. Of the latter two, one synchronizes instantiations using compiled tests that were determined from an offline analysis while the other uses a novel locking scheme that requires no such analysis. Our examination of performance shows that asynchronous execution is clearly faster than synchronous execution and that the locking method is somewhat faster than the method using compiled tests. Moreover, we predict that the synchronization and/or locking needed to guarantee serializability will limit speedup no matter how many processors are used.

1. Introduction
A production system (PS) is an effective vehicle for implementing a variety of computer systems. Most notable are implementations of successful expert systems such as R1 [14]. Unfortunately, PSs are slow and will require substantial speedup when executing large systems under demanding time constraints [4].

To speed them up, researchers have studied parallel implementations with much of that research focusing on OPS5 [2] or OPS-like languages. Since most CPU time in OPS5 is spent in the MATCH step (over 90% according to [1] and over 50% according to [11]), many efforts have tried to make parallel that one step while leaving the system to continue executing only one rule at a time (e.g., [5, 6, 9, 15, 20, 27, 28]).

To gain even more speedup, several researchers have investigated systems that execute many rule instantiations simultaneously (e.g., [7, 8, 11, 13, 16, 17, 18, 21, 22, 23, 24, 25, 26, 29]). Such parallelism is often called rule or production level parallelism. Each match of a rule is represented by an instantiation, where any given rule can have many instantiations at a given point. When two or more instantiations execute simultaneously, we say they coexecute.

It is possible for these latter systems to produce results that could never have been produced by a serial PS. To deal with this, most of the above systems guarantee serializability (some exceptions are [17, 18, 29]). In other words, they guarantee that the final result could have been attained by some serial execution of the same instantiations that were executed in parallel.1 Numerous algorithms have been offered to guarantee serializability, most of which are based on the seminal work in [7], which enforced serializability by prohibiting the coexecution of pairs of instantiations that interfere with each other. We will soon define interference.

In this paper, we present and analyze three different algorithms that execute multiple instantiations simultaneously while guaranteeing serializable results. Our goal is to measure the overhead that serialization incurs. These algorithms are arguably the three most precise such algorithms in the literature. They use a very precise criteria to determine. We will not address the control problem. Given that most parallel PSs are non-deterministic, there may be many possible serializable results. The control problem is concerned with making the parallel PS produce the preferred result (e.g., see [12, 18]).
mine interference and, as a result, prohibit coexecu-
tions less often, which produces greater concur-
rency. Algorithm A1 operates synchronously, which
means that all the instantiations in the conflict set
are executed in their entirety before new instanti-
ations are considered. Interference checking is per-
formed using tests created during an offline analy-
asis. Algorithm A2 is an asynchronous version of A1.
Algorithm A3 also operates asynchronously but
uses a locking mechanism to prevent interfering
instantiations from coexecuting. A1 appears as al-
gorithm M1–Greedy in [24]. A2 and A3 have not
previously been published.

Section 2 describes the offline analysis used by
algorithms A1 and A2, which are themselves ex-
plained in Sections 3 and 4, respectively. Section 5
discusses algorithm A3. Section 6 analyzes the per-
formance of all three. Section 7 draws conclusions.

2. Offline Analysis for A1 and A2

Much of the following is a brief summary of the work
presented in [24], which builds upon the framework
set out in [7].

Let a production rule be written as $P: C_1, ..., C_m$
$\rightarrow A_1, ..., A_n$. $P$ is the name, each $C_i$ is a con-
dition element (CE) and each $A_i$ is an action. Each $C_i$
has a sign of $+$ or $-$, denoted $\text{Sgn}(C_i)$, and a literal,
denoted $\text{Lit}(C_i)$. Similarly for $A_i$. Variables that
appear in positive CEs are bound. All other variables
are free. We only consider the actions of adding to
or deleting from working memory (WM).

An instantiation of $P$, written $i^P$, consists of the
name $P$ plus the working memory elements (WMEs)
that match the positive CEs of $P$. Alternatively, we
can consider $i^P$ to be the name $P$ plus the bindings
of $P$'s bound variables that led to $i^P$ matching. Let
$\text{Match}(x, y, \delta)$ be true iff $\delta$ is a substitution list
that makes $x$ unify with $y$. We will freely use substitu-
tions as substitutions lists. For example, if

\[ \text{PA (OPEN 2E 1232) (WANTS Schmolze 1232)} \]

is an instantiation of rule PA, shown below, it is
equivalent to a substitution list where $<\text{seat}> = 2E,$
$<\text{flight}> = 1232$ and $<\text{passenger}> = \text{Schmolze}.$

PA: (OPEN $<\text{seat}> <\text{flight}>$)
(WANTS $<\text{passenger}> <\text{flight}>$) $\rightarrow$
(REMOVE 1 2)
(MAKE RESERVATION $<\text{passenger}> <\text{flight}>$
$<\text{seat}>$).

Let $\phi(x, \delta)$ denote the result of substituting the
variables in $\delta$ into $x$. Let $IV(x, y, \delta')$ be true iff $\delta'$ is an
independent variable substitution. Here, we re-
name the variables in $y$ so that $x$ and $y$ share no vari-
ables. Finally, let $IVMatch(x, y, \delta, \delta')$ be defined as

\[ IV(x, y, \delta') \land \text{Match}(x, \phi(y, \delta'), \delta'). \]

We call $IVMatch$ an independent variable match.

We say that one instantiations disables another
iff executing one causes the other to match no long-
er. This occurs if the first adds (deletes) a WME that
the other matches negatively (positively). We define
a test for inter–instantiation disabling as follows
(the proofs for all theorems appear in [24]).

Theorem 1: $i^P$ disables $i^Q$ iff $\exists j, k, \delta, \delta'$ such that:

\[ \text{Sgn}(A_j^P) \neq \text{Sgn}(A_k^Q) \land \]
\[ IVMatch(\phi(\text{Lit}(A_j^P), i^P), \phi(\text{Lit}(A_k^Q), i^Q), \delta, \delta')). \]

We say that one instantiations clashes with
another iff executing one would add a WME that
executing the other would delete. We define a test
for inter–instantiation clashing as follows.

Theorem 2: $i^P$ clashes with $i^Q$ iff $\exists j, k, \delta, \delta'$ such that:

\[ \text{Sgn}(A_j^P) \neq \text{Sgn}(A_k^Q) \land \]
\[ IVMatch(\phi(\text{Lit}(A_j^P), i^P), \phi(\text{Lit}(A_k^Q), i^Q), \delta, \delta')). \]

We note that clashing is symmetric, i.e., $i^P$
clashes with $i^Q$ iff $i^P$ clashes with $i^Q$.

We let $I$ be a set of instantiations and define a di-
rected graph called $\text{IDO}(I)$ as the instantiation dis-
abling order. For each $i$ in $I$, there is a node in
$\text{IDO}(I)$. For each distinct $i_j$ and $i_k$ in where $i_k$
disables $i_j$, there is an edge from $i_j$ to $i_k$ in $\text{IDO}(I)$. This
comprises all of $\text{IDO}(I)$.

We say that two instantiations coexecute if the
time of the execution of their right–hand sides
(RHSs) overlap. We finally arrive at serializability.

Theorem 3: The coexecution of a set of instantia-
tions, $I$, using our parallel model is serializable if
$\text{IDO}(I)$ is acyclic and no two distinct instantia-
tions in $I$ clash.

Interfering instantiations, then, are those that
cause Theorem 3 to be violated. Our algorithms
thus must make sure that, among the set of instan-
tiations that are coexecuting, no two clash and there
is no cycle of disabling relations. Our algorithms all
work at the instantiation level, not the rule level.

Most other published algorithms (e.g., [7, 16]) iden-
tify pairs of rules whose instantiations might dis-
able (or clash), and then prohibit the coexecution of
all their instantiations whether they would actually
disable (or clash) or not. Thus, working at the
instantiation level is much more precise. In fact, in
[24], we show that frequently there is little parallel-
ism to be gained by working at the rule level.

In order to make our algorithms fast, we perform
an offline analysis along the lines set forth in
the above theorems. For each pair of rules, we syn-
thesize a function that takes an instantiation of each
rule as arguments and returns true$\iff$the first will
disable the second. We do the same for clashing.

These functions are produced in Lisp, compiled and

Schmolze and Neiman 493
then stored in tables for fast access and execution.

3. A1: Synchronous using Offline Analysis

All three algorithms begin with a similar architecture which comes from the second author’s dissertation research [19]. We will explain that architecture here, followed by the details of A1.

Given N processors, we assign 1 processor to be the scheduler and the remaining N-1 processors to be demons. The specific jobs of the scheduler and demons differ for the different algorithms, but their basic jobs are the same. The scheduler pulls new instantiations off of the conflict set (CS) and decides whether or not to schedule them. Scheduling an instantiation consists simply of putting it on a shared queue. Each demon pull instantiations off the shared queue — each instantiation goes to exactly one demon — and executes them if the demon decides that execution is appropriate. This basic architecture is shown in Figure 2. The differences between the three algorithms are in the processing that each does to an instantiation, and how each decides whether to schedule and/or to execute the instantiation.

Figure 3 shows the scheduler’s and demons’ algorithms for A1. As can be inferred, the scheduler and demons take turns doing work, and each such pair of turns is called a parallel cycle. The scheduler only begins when the demons are idle, whereupon it takes M instantiations from the conflict set, places them into an array and schedules them. It limits M, the number of instantiations considered per parallel cycle, to be FxN, where F is a constant factor and N is the number of processors. We apply this limit because the time that each demon spends testing for disabling and clashing is proportional to the size of M. We experimented with an F of 2, 4, 8 and 1000 (1000 has the same effect as F=∞) and found that F=2 consistently produced the fastest execution times. All of our results in Section 6 use F=2.

The demons take the instantiations off of the queue one by one, test them for disabling and clashing, and if appropriate, execute them. Multiple instantiations can be tested as such in parallel because each demon will only mark the instantiation it is considering and no other. Moreover, the body of code that executes an instantiation has been written to allow multiple simultaneous executions.

Upon examination of the demon algorithm, one can infer that, after the demons finish, the set of instantiations in A that are still marked in meet the requirements of Theorem 3. As a result of line 3, there is no i<j where both A[i] and A[j] are in and where either A[i] clashes with A[j] or A[i] disables
A2 Scheduler Algorithm
1. If there are instantiations in the CS then go to 2
   else if the demons are quiescent
      (i.e., queue empty and all demons idle)
      then exit else loop to 1.
2. Remove a non-dead instantiation from CS and schedule it.
3. Loop to 1.
A2 Demon Algorithm
1. If demon queue is empty
   then demon is idle & loop to 1.
2. Demon is busy. Remove an instantiation from queue & call it I.
3. Try to acquire locks for I.
4. If successful then schedule I
   else if I is not dead then return I to CS.
5. Loop to 1.

A3 Scheduler Algorithm
1. If there are instantiations in the CS then go to 2
   else if the demons are quiescent
      (i.e., queue empty and all demons idle)
      then exit else loop to 1.
2. Remove a non-dead instantiation from CS. Call it I.
3. Try to acquire locks for I.
4. If successful then schedule I
   else if I is not dead then return I to CS.
5. Loop to 1.
A3 Demon Algorithm
1. If demon queue is empty
   then demon is idle & loop to 1.
2. Demon is busy. Remove an instantiation from queue & call it I.
3. Execute I.
4. Loop to 1.

Figure 4: A2: Asynchronous algorithm using disabling/clashing tests.

Figure 5: A3: Asynchronous algorithm using locks.

A[j]. Thus, no two in instantiations clash (remember, clashing is symmetric) and there is no cycle of in disabling relations (since we have prevented any forward links of A[i] disabling A[j]).

4. A2: Asynchronous using Offline Analysis

A1 is synchronous, and as such, wastes a considerable amount of time. Given that different instantiations take different amounts of time to execute, some demons sit idle while waiting for other demons to finish. An asynchronous algorithm eliminates this wasted waiting time (as was argued in [17, 18]), and so we designed A2.

Figure 4 shows A2, an asynchronous version of A1. Here, the scheduler simply takes instantiations from the CS and schedules them, thereby performing very little work. The scheduler also checks for an empty CS and quiescence, which signals no further executions. This is similar to A1, however, the scheduler does no waiting.

The demons also behave similarly to the demons in A1. However, we must carefully define the execution time of an instantiation. An instantiation is said to be executing from the time that a demon removes it from the queue to the time that the demon either discards it or finishes executing its RHS. A list E of the instantiations currently executing is maintained where E is a critical resource. Writers to E have unique access but multiple readers can have simultaneous access. In addition, and due to the asynchrony of the system, it is possible for an instantiation to become disabled while a demon is testing it in line 5. In that case, the system marks the instantiation as dead and no further processing is performed on it.

For reasons similar to those for A1, A2 obeys Theorem 3 and yields a serializable result.

5. A3: Asynchronous using Locks

Figure 5 shows A3, which is considerably different from A1 and A2. Here, we try to acquire locks instead of checking for disabling/clashing; we will soon explain how this works. Moreover, since this occurs in the scheduler, lock acquisition is a serial process, which eliminates the potential for deadlock. By the time an instantiation is scheduled, the system has already determined that it should execute, so the demons perform no decision-making; they simply take instantiations off the queue and execute them.

Schmolze and Neiman 495
The locking scheme would be simple but for the use of negative CEs. While it is easy to lock a WME that is already in the WM, it is not so easy to lock the “non-existence” of a WME, as implied by a negative CE. However, we have designed an efficient way to use the Rete net [3] to identify precisely the set of WMEs that would match the negative CE. Sellis et al [23] also use locks for serializing but for negative CEs, they lock an entire class of WMEs.

In the Rete net, when a WME is positively matched, a token representing that element is concatenated to a set of tokens being propagated through the network. We can similarly create a pseudo-token corresponding to a successful match of a negated element. This token represents a pattern of the working memory elements that would disable this instantiation. This pattern is simply the set of tests encountered by the working memory element as it proceeds through the matching process; specifically, the inter-element alpha tests preceding the NOT node, concatenated to the tests performed by the NOT node and unified with the positively matched tokens in the instantiation.

For example, if we had a rule such as the one shown below, the pseudo-token would have the form (class = B) (element(1)=wombat) (element(2)=koala). Thus, any currently executing instantiation that creates an element matching this pattern would disable an instantiation of PK stimulated by the working memory element (A wombat).

(P PK (A <x>) WM = { (A wombat) }
  - (B <x> koala) →
  (B <x> koala))

When a rule instantiation is created, we thus have two sets of tokens: the WMEs matching the left-hand-side (LHS) and negative pattern tokens. To use the latter, we must also do the following. Before each instantiation is scheduled, we develop a list of all the WMEs that it will add when it is executed. This is reasonable as the formation of elements is usually inexpensive. Immediately before the instantiation is executed, we post all the WMEs it is about to add onto a global ADD list. We now explain the operation of lock acquisition in detail.

1. Each WME has a read counter and write flag. Each instantiation has a read and write list. As each instantiation, I, enters the CS, we add to its write list each WME matched on its LHS that would be modified or removed by I. The remaining WMEs matched on I's LHS are placed on its read list. Next, we see if any of the WMEs on the read or write list have their write flag set. If so, we discard I because it will soon be disabled by another instantiation that is already executing.

If a WME on the write list has its read counter > 0, we do not execute I and instead, place it back on the CS. In this way, we do not disable another instantiation that is already executing while giving I another chance later. If I has not been discarded or put back on the CS, we proceed.

2. We compare I's negated pattern tokens against the list of WMEs on the ADD list. If any match, then I is discarded as it will soon be disabled by an instantiation already executing. Otherwise, we proceed.

3. We now acquire the locks, which amounts to incrementing the read counters for the WMEs on the read list and setting the write flags for the WMEs on the write list. We also post the WMEs to be added to the ADD list.

The demon also has some extra tasks. After it finishes executing an instantiation I, it removes the elements that I added to the ADD list and decrements the read counters for those WMEs on I's read list. We note that accessing and modifying the ADD list, read counters and write flags must be performed in critical code.

A3 does not check for clashing. While space does not permit our discussing it here, it turns out that the implementation of WM as a multiset in OPS5 makes testing for clashing unnecessary (e.g., we could go back to A1 and A2 and safely remove the clashing tests).

6. Performance Analysis

In order to determine the cost of serialization, we have implemented the three algorithms and run them against a benchmark PS that we call Toru-Waltz-N. Toru-Waltz-N began with Toru Ishida's implementation of Dave Waltz's constraint propagation algorithm to identify objects from line drawings [30]. It was modified by the second author to increase the available rule concurrency by combining the initialization and processing stages and to allow rules to be asynchronously triggered. The time for serial execution is 12.79 seconds. In our parallel system, if we turn off the checks for serializability, the best time we obtain is 1.2 seconds with 15 processors. Thus, 10.7 is the maximum possible speedup for this benchmark and for our software without serialization. Any further reductions in speedup are due to the serialization component of our algorithms.

Figure 6 shows the speedups attained by the three algorithms. Clearly A3 performed the fastest.

The text of the Toru-Waltz-N benchmark plus a discussion of its implementation and performance can be found in [19].
attaining a maximum speedup of 5.70 (runtime of 2.24 seconds), followed by A2 with a maximum speedup of 4.64 (runtime of 2.75 seconds), and followed, finally, by A1, with a maximum speedup of 2.80 (runtime of 4.57 seconds). However, when A3 was run, additional mechanisms were used: match and action parallelism. Space does not allow us to describe those algorithms here, but we estimate that they reduce the run time by about 0.5 seconds. Subtracting 0.5 from A2’s best time of 2.75 seconds yields 2.25 seconds, or a speedup of 5.68. The run times of A2 and A3 are thus very similar.

It is clear that the speedup for each algorithm tapers off at around 10 to 12 processors, indicating that these maximum speedups are close to the absolute maximums for these algorithms and benchmark. The speedups realized fall quite short of 10.7, which was achieved without serialization. Serialization thus appears to cut the potential speedup roughly in half for this benchmark.

The reason for A1 being slowest is simple. A1 is performing nearly the same work as A2, but has the disadvantage of being synchronous. Given that different instantiations take different amounts of time to execute, this amounts to a waste of processor time as demons sit idle while waiting for other demons to finish. An asynchronous algorithm eliminates this wasted waiting time (as was argued in [17]).

Conceptually, A2 and A3 perform the same type of interference checking, although they use very different mechanisms to do so. It is interesting to see that they offer similar potential for speedup. If we take a brief look at these mechanisms, we see that A2 requires $O(M)$ time to check interference for each instantiation, where $M$ is the number of instantiations currently coexecuting; each instantiation must be checked against all those currently coexecuting. It turns out that A3, when checking negative CEs, also requires $O(M)$ time per instantiation since the size of ADD depends on $M$. However, when A3 checks positive CEs, the time required is not dependent on the size of $M$ and, instead, is constant per rule, which makes this type of check very fast.\footnote{In [17], the second author argues that this is a good reason for preventing interference only for positive CEs and \emph{not} for negative CEs, even though this falls short of guaranteeing serializability.}

To discover some of the limitations of these algorithms, we broke down the execution times of A2 and A3 further. Figure 7 shows a bar chart of the total run times for A2 along with the average time spent by the demons doing their two main tasks: testing for interference and executing instantiations. It is clear that the time spent testing increases with the number of demons. This makes sense because the average number of coexecuting...
instantiations increases with the number of demons, so there is more testing to be done. The time spent executing rules decreases since there are more demons, and so each one executes fewer instantiations. Therefore, A2 will always be slower than a system that does not guarantee serializability because of this testing time.

In A3, the interference checking is done via a locking mechanism. However, this mechanism must run serially in order to avoid deadlocks between instantiations simultaneously attempting to acquire locks. All locking is thus performed in the scheduler. The most time consuming portion of the locking mechanism is the portion that deals with negated tokens. If the cost of checking negated tokens against the ADD list is expensive as compared to the time needed to execute an instantiation, then the benefits of asynchronous execution are lost. In order to form an estimate of the overhead associated with negated tokens, we note that the processing performed when matching each working memory element being asserted against each negated pseudo-token pattern is essentially equivalent to the time of a beta node activation within the Rete net (for the check against the ADD list) and two memory node activations (one for each addition or deletion to the ADD list). This approximation is reasonable as the tests contained within the negated pseudo-tokens are derived from the NOT nodes which generated them. The beta nodes are the most time-consuming component of the pattern matching process and the number of beta nodes executed can be used to create an estimate of relative costs. Using the statistics gathered by Gupta [6], we note that the average instantiation activates approximately 40 beta node and memory operations (of course, the actual figures depend on the size and complexity of the CEs). Thus the runtime detection of interactions due to negated tokens may incur costs of as much as 10% of the cost of actually executing the rule for each negated condition in the rule. Because the detection of interference must be carried out within a critical region of the scheduler, an overhead of this magnitude would limit the potential parallelism within the system to a factor of 10 (assuming one negative CE per rule on average, which agrees roughly with the measurements in [6]), exclusive of other scheduling costs.4

7. Conclusions

We conclude that one pays a fairly high price for ensuring serializability. While the design and implementation of these algorithms could certainly be optimized further, each incurs an unavoidable overhead. For A3, this overhead appears to be at least 10%. For A2, we do not have a firm estimate for the minimum overhead, but it is clear that the overhead increases with the number of processors, suggesting a firm limitation to speedup. For A1, we have shown that its performance will always lag behind that of A2. Overall, a speedup of 10 appears to be an absolute limit for A3, and probably applies to A2 and A1 as well. In addition, we find that our serializable algorithms obtain roughly half the speedup obtained by a similar parallel system that does not guarantee serializability.

One could potentially improve throughput by modifying our algorithms. For example, in A3 we could perform a compilation-time analysis on the rule set and divide the rules such that several lock acquisition processes could be used. Because our results depend on analyzing the ratio between lock acquisition times and rule execution times, mechanisms such as those described here may be more suitable for environments such as blackboard systems in which the units of execution are of a higher granularity. Another possibility for decreasing the time required to acquire locks is to apply micro-level parallelism to the checking process such that new candidate instantiations are compared against executing instantiations in parallel, along the lines suggested in [9].

While we have concentrated on the detection of rule interactions in this paper, the overhead analysis is appropriate for any overhead such as control scheduling or heuristic pruning that has to occur within a critical region. We thus feel that this type of research is essential to our understanding of the applicability of parallelism to artificial intelligence research in general.

References


4. We can model a system of this type as an \( M/M/\infty \) queue [10].


