Chapter 4

Asynchronous Computing with RPC Semantics

In this chapter we focus on issues related to concurrency and scheduling within a single user session. We begin by discussing the notion of serial correctness and how this governs the degree of concurrency that can be allowed within a session. Maintenance of serial correctness requires that we capture the serial order of computations. This is done by means of a hierarchical scheme to generate forkstamps. Two extreme scheduling strategies both of which preserve serial correctness, but offer varying degrees of concurrency, are then discussed. Finally we present a framework for the comparative analysis of these and other scheduling schemes.

4.1 Serial Correctness versus Concurrency

In chapter 2 we discussed the synchronization problem caused by concurrent computations and how this can affect serial correctness. To elaborate in more general terms, visualize a set of concurrent computations as a tree such as that shown in figure 4.1. In this figure we see that message manager 1 at the unclassified level has sent messages to one secret object, one top-secret object, and one confidential object in this sequence (we consider message manager 1 to be the ancestor of the three). As these objects are higher in level than unclassified, message filtering has resulted in the creation and concurrent execution of message managers 2, 3, and 4 as children of the root message manager 1.
We can now formally define serial correctness in terms of such a tree.¹

**Definition 4.1** We say a session preserves serial correctness if for any computation $c$ in the session's computation tree, and running at level 1, the following hold:

1. $c$ does not see any updates (by reading-down) of lower level computations that are to its right², in the tree;

2. For any of $c$'s ancestor computations $a$, (i.e., any computation on the path from the root to $c$) $c$ should see only the latest updates made by a just before $a$'s child (or $c$ itself) on this path was forked.

3. For any level $k$ that is not the level of an ancestor of $c$, and $k \leq 1$, $c$ should see the latest updates made by the rightmost terminated computations at level $k$ that are still to the left of $c$.

Given the above definition, let us see the complications concurrency poses to the maintenance of serial correctness. Now if we were to execute the above tree sequentially, the messages sent to higher level objects would be processed in the order given by the labels on the arrows. Note that this order can be derived by a

---

¹It is important to realize that even though the notions of serial correctness and serializability may appear to be analogous, they are not equivalent. Serializability theory in classical transaction management and concurrency control realms reasons about correctness and integrity in terms of the fundamental abstraction of a "transaction". Serial correctness on the other hand, is a more primitive notion as it does not recognize the abstraction or semantics of transactions, and is further more restrictive as it allows only a single serial order (i.e., the order of an RPC-based serial execution of computations (methods)). However, if we were to map individual computations to transactions and derive the transaction serialisation order from the forkstamps, serial correctness amounts to a stricter form of the multiversion concurrency control notion of one-copy serializability [BH87]. We intentionally do not give such a definition as this would give the impression that we are dealing with transactions, and would further introduce unnecessary formal machinery in our exposition.

²A computation $b$ is said to be to the right of a computation $a$, if neither $b$ nor $a$ is an ancestor of the other, and $b$ is encountered later than $a$ in a depth-first traversal of the corresponding session tree, starting at the root. Similarly, a computation to the left will be encountered earlier in a depth-first traversal.
Figure 4.1: A tree of concurrent computations

left-to-right depth-first traversal of the tree. However, with concurrent execution it is possible that message managers 4(C) and 6(S) may terminate well ahead of 3(TS). Therefore our synchronization schemes must ensure that message manager 3 does not see any updates by message managers 4 and 6, since 4 and 6 are to the right of 3.

Solving the above synchronization problem using classical techniques, such as those based on locking and semaphores, is known to be insecure as they open up signaling channels. Also, it is not possible to implement these synchronization mechanisms in a kernelized architecture without introducing trusted subjects since we need the ability to write-down and read-up. Our solution instead relies on a multiversioning scheme. The scheme calls for multiple versions of objects accessed by a session to be kept in memory. Each version is uniquely identified with a timestamp, and can be thought of as a checkpoint in the overall progress of a tree of computations. Thus although 4(C) and 6(S) may terminate well ahead of 3(TS), we are guaranteed that a read-down request from 3(TS) will always read versions that existed before

---

3Note that there is no multiversioning on disk. Even if paging causes these versions to migrate to disk occasionally, they will not be visible to other sessions.
4(C) and 6(S) were started.

Given a computation, say c, the multiversioning scheme suggested above can provide synchronization when other computations to c's right (in the tree) get ahead of c. But to guarantee serial correctness, we must in addition ensure that c itself does not get ahead of earlier forked computations to its left. For example, under a sequential execution of the tree of computations in figure 4.1, we would expect message manager 2(S) and its descendants (if any) to terminate before message manager 3(TS) to its right, is started. Message manager 3(TS) should thus see all the latest updates by 2(S) and any of its descendants. Allowing arbitrary concurrency may not ensure this. Thus, in addition to multiversion synchronization, we need to enforce some discipline on these concurrent computations by scheduling them in a manner that guarantees serial correctness.

A scheduling strategy which guarantees serial correctness must take into account the following considerations.

- The scheduling strategy itself must be secure in that it should not introduce any signaling channels.

- The amount of unnecessary delay a computation experiences before it is started should be reduced.

The first condition above requires that a low-level computation never be delayed waiting for the termination of another one at a higher or incomparable level. The second consideration admits a family of scheduling strategies offering varying degrees of performance. Some of these are discussed later in the next section.

---

If this were allowed, a potential for a signaling channel is again opened up in a trusted subject architecture.
In summary, the maintenance of serial correctness requires careful consideration on how computations are scheduled as well as on how versions are assigned to process read-down requests. Collectively we have to guarantee the following constraints (as discussed in section 5.2, we assume that every computation is assigned a strictly increasing forkstamp that is consistent with the start order in a sequential execution):

Whenever a computation \( c \) is started at a level \( l \),

- **Correctness-constraint 1**: There cannot exist any earlier forked computation (i.e. with a smaller forkstamp) at level \( l \), that is pending execution;

- **Correctness-constraint 2**: All current non-ancestral as well as future executions of computations that have forkstamps smaller than that of \( c \), would have to be at levels higher or incomparable to \( l \);

- **Correctness-constraint 3**: For each level at or below \( l \), the object versions read by \( c \) would have to be the latest ones created by computations such as \( k \), that have the largest forkstamp that is still less than the forkstamp of \( c \). If \( k \) is an ancestor of \( c \), then the latest version given to \( c \) is the one that was created by \( k \) just before \( c \) was forked.

The above three constraints are sufficient to ensure serial correctness. We now state this formally as a theorem.

**Theorem 4.1** Correctness constraints 1, 2, and 3 are sufficient to guarantee serial correctness of concurrent computations in a user session.

**Proof:**

Constraints 1 and 2 ensure that when computation \( c \) at level \( l \) is started, there will be
no more writes/updates forthcoming from earlier forked non-ancestral computations (the ancestral computations of c are those that are on the path from the root to c, in the computation tree). This guarantees that write operations by non-ancestral computations at levels l or below (and therefore inductively across all levels) will occur in the same relative order as in a sequential execution. Write operations from ancestral computations may, however, be issued in an order different from the sequential execution. Such out of order writes can affect the values obtained by later read operations from higher level methods. However, constraint 3 ensures that read down operations under concurrent execution will obtain the same state as in a sequential execution. To see this, consider any computation such as c at a level l. In a sequential execution all non-ancestral computations at lower levels and with smaller forkstamps than c, would have terminated before c. Thus higher level reads by computations such as c would obtain the last written versions by such non-ancestral computations. The ancestors of c on the other hand would be suspended in a sequential execution, waiting for c and its future children to terminate. Thus read operations issued by c should see the versions written by the ancestors just before they were suspended. Constraint 3 requires this and prevents c from reading out of order writes (versions) of its ancestors.

4.2 Maintaining Global Serial Fork Order

We now discuss an implementation consideration for our scheduling schemes which has to do with maintaining knowledge of the equivalent global serial order in which computations are forked within a user session. In scheduling various computations, such knowledge is used to determine when a computation will be started. In an architectural framework without multilevel trusted subjects, no single system component has a global view (such as the tree in figure 4.1) of the entire set of com-
Figure 4.2: Generation of forkstamps for a session's computation tree

computations as they progress. In coordinating various computations, an individual level manager has to determine where in the global serial fork order, the computations at its level belong. One could be tempted to pursue a solution requiring the value of a global real-time clock to be appended to every message manager (computation) as it is forked. However, computations are not always forked in the equivalent serial order and thus a solution based on a real-time clock will not always work.

Consider now a hierarchical scheme to generate forkstamps that is independent of the scheduling strategy used. The forkstamps so generated, reflect the equivalent serial order of execution of the computations. Figure 4.2 shows a tree of computations and the forkstamps generated for it. Every message manager except the root, is assigned a unique forkstamp by the parent issuing the fork. The scheme starts by assigning an initial forkstamp of 0000 to the root message manager 1(U). Every subsequent and immediate child of the root is then given a forkstamp derived from this initial one by progressively incrementing the most significant (leftmost) digit by
one. To generalize this scheme for the entire tree, we require that with increasing depth along any path in the tree, a less significant digit be incremented. In general for a security lattice with a longest maximal chain of \( n \) elements, we need to reserve \( p \cdot (n - 1) \) digits for the forstamp. In a lattice with \( l \) levels, and \( c \) compartments, \( n = l + c \). The value of \( p \) would depend on the maximum degree of a node in a computation tree. For example if we assume that any computation sends a maximum of 99 messages to higher levels, then setting \( p = 2 \) would be sufficient. Even with large lattices and a high number of messages sent to higher levels, these numbers are reasonable.

We now show that the above forstamping scheme captures the intended global serial order.

**Theorem 4.2** The hierarchical forstamping scheme preserves global serial order.

**Proof:**

We prove the above by contradiction. Consider any pair of computations \( c_1 \) and \( c_2 \) with forstamps \( f_1 \) and \( f_2 \) respectively, such that \( c_1 \) was forked before \( c_2 \) (i.e., \( c_1 \) is to the left of \( c_2 \) in the computation tree). Assume \( f_2 < f_1 \). In other words, \( c_1 \) is forked earlier than \( c_2 \) but we assume that the former is given a later forstamp. Let us see if this assumption can be contradicted.

By virtue of the fact that the forked computations form a tree, there will always be a common ancestor node \( a \) from which the paths to \( c_1 \) and \( c_2 \) fan out. Consider the edges from \( a \) to \( c_1 \) and \( c_2 \) as belonging to the paths \( p_1 \) and \( p_2 \), respectively. Now consider the immediate children of \( a \), say \( a_1 \) and \( a_2 \) that lie on paths \( p_1 \) and \( p_2 \), respectively (for paths of length one this will be \( c_1 \) and \( c_2 \)). Assume that we are using forstamps with \( k \) digits denoted \( d_1, d_2, \ldots, d_k \), with \( d_1 \) being the most significant
digit. Of these \( k \) digits, let \( d_a \) be the digit assigned to (and incremented by) computation \( a \), and \( d_{a/a_1} \) and \( d_{a/a_2} \) the values of the digit \( d_a \) for the forkstamps assigned to \( a_1 \) and \( a_2 \), respectively. Now since the ancestor computation \( a \) forked \( c_1 \) before \( c_2 \), it follows that the path \( p_1 \) will be to the left of path \( p_2 \). Hence, our forkstamping algorithm will assign forkstamps to \( a_1 \) and \( a_2 \) derived from \( a \)'s forkstamp such that \( d_{a/a_1} < d_{a/a_2} \). Now every descendant of \( a_1 \) (or \( a_2 \)) on the path \( p_1 \) (or \( p_2 \)) including \( c_1 \) (or \( c_2 \)) will retain \( a_1 \)'s (or \( a_2 \)'s) value for the digits \( d_1, d_2, \ldots, d_k \). Hence it follows that the value of the digit \( d_{a/c_1} \) will always be less than \( d_{a/c_2} \). Further, successive descendants of \( a_1 \) and \( a_2 \) in these paths will increment only digits \( d_{a+1}, d_{a+2}, \ldots, d_k \). These digits are less significant in place value than the digit \( d_a \). Thus the value of the forkstamp \( f_1 \) of \( c_1 \) will be less than the forkstamp \( f_2 \) of \( c_2 \). This contradicts the assumption that we stated out with, and hence the proof. \( \square \).

An obvious implication of our forkstamping scheme is that only unique and acyclic forkstamps are generated. We conclude this section by stating this as a corollary to the above theorem.

**Corollary 4.1** The forkstamping scheme generates forkstamps that are unique and acyclic.

**Proof:**

The proof follows from the following propositions:

1. Two computation nodes that lie on different paths originating from a common ancestor will have their forkstamps varying on the significant digit (position) that is incremented by the ancestor.

2. Nodes that lie on the same path will have forkstamps that increase with the length of the path.
3. The nodes form a tree and as such there are no loops or parallel paths.

4. The forkstamps form a totally ordered set. □

4.3 A Family of Scheduling Strategies

We now present a family of scheduling schemes. These schemes offer varying tradeoffs between performance and complexity when implemented under different architectures. We present two schemes, namely conservative and aggressive, that lie towards the ends of a spectrum of scheduling schemes. We also briefly mention a hybrid scheme for which the performance lies somewhere in between the above two. We also present a framework and metric for the comparative analysis of these schemes.

4.3.1 A Conservative Level-by-level Scheduling Scheme

Consider first a level-by-level scheduling scheme. We characterize this approach as being conservative, as opposed to being aggressive, since the objective here is not to maximize concurrency. In other words, a computation may be unnecessarily delayed before being started even if its earlier execution would not violate serial correctness. Although this scheme is not optimal in terms of performance, it does give insights into how concurrent computations can be scheduled and completed in a simple, yet secure, correct and distributed fashion. The conservative scheme maintains the following invariant:

Inv-conservative: A computation is executing at a level $l$ only if all computations at lower levels, and all computations with smaller fork stamps at level $l$, have terminated.
(a) Hasse diagram for a lattice

(b) Hasse diagram for a partial order

Figure 4.3: Conservative level-by-level scheduling in lattice and partial order
Thus the basic idea is to execute forked computations in a bottom up fashion in the lattice, starting with the lowest level. At any point, only computations at incomparable levels can be concurrently executing. We thus begin with the root computation and allow it to run to completion. Meanwhile, all higher level computations that are forked by the root are unconditionally queued in forkstamp order at these higher levels, by the respective level managers. Upon termination of the root, its level manager signals that it is okay to release computations at all immediate higher levels by sending a WAKE-UP message to these levels. Thus when a level manager receives a WAKE-UP message from all immediate lower levels, it proceeds to dequeue and execute computations at its level one at a time in forkstamp order. Note that, at this point, this level manager is guaranteed that no more fork requests will be forthcoming from lower levels. Eventually, the level manager will find its queue to be empty. The next higher levels are then released through WAKE-UP messages.

For a more visual explanation of this level-by-level scheduling strategy, consider the lattice and partial order in figures 4.3(a) and 4.3(b). Consider the lattice first. On termination of the root computation at level \([U,\{\}\)] , WAKE-UP messages are sent to all the immediate higher levels \([C,\{A\}], [C,\{B\}], [C,\{D\}]\), and queued computations at these levels are then released. Next, computations at \([S,\{B,D\}]\) are started when all those at the immediate lower levels \([C,\{B\}]\) and \([C,\{D\}]\) have terminated. Eventually, computations at the highest level \([TS,\{A,B,D\}]\) are started on the termination of computations at levels \([S,\{A\}]\) and \([S,\{B,D\}]\) followed by the receipt of a WAKE-UP message from each of these levels. Now consider the partial order in figure 4.3(b). When the root computation at level \([U,\{A\}]\) terminates, a WAKE-UP message is sent to all the immediate higher levels, namely \([C,\{A,B,D\}]\) and \([C,\{A,B,E\}]\) and computations at these levels may thus be running concurrently. Computations at level \([S,\{A,B,D\}]\) are released when a WAKE-UP message is received
from the only dominated class which is \([C,\{A,B,D\}]\). Similarly when computations at \([C,\{A,B,E\}]\) terminate, a WAKE-UP message is sent to level \([S,\{A,B,E\}]\) to release queued computations.

Figures 4.4(a) through 4.4(g) illustrate the progressive execution of the computation tree in figure 4.1, as governed by the level-by-level scheduling scheme. At each stage the termination of a computation results in the start-up of another. In this example, there can only be one computation executing at any given moment as the lattice is totally ordered. More generally, we could have multiple computations running, provided they are at incomparable levels. As shown in figure 4.4(a), the startup of the root computation has resulted in its forked children to be queued (the unborn computations have not yet been created, and are shown in the figures for visual completeness only). The subsequent termination of the root (see figure 4.4(b)) has resulted in the forked child, at the lowest level 4(C), to be executed.

4.3.2 An Aggressive Scheduling Scheme

We now describe an aggressive scheduling algorithm. It is governed by the following invariant:

**Inv-aggressive:** A computation is executing at a level \(l\) only if all non-ancestor computations, in the corresponding computation tree, with smaller fork stamps at levels \(l\) or lower, have terminated.

We characterize this as an "aggressive" scheme as every attempt is made to execute a forked computation immediately. The above invariant implies that if a computation is denied immediate execution, then there must be at least one non-ancestral lower level computation with an earlier forkstamp, that has not terminated. The invariant ensures that the correctness constraints 1 and 2 are never violated. The correctness
of read-down operations is again dependent on multi-versioning.

The major differences between the aggressive and conservative schemes can be summarized as follows:

- On being forked, a computation may be immediately started, if doing so would not violate the invariant inv-aggressive.

- The termination of a computation may result in the start-up of the next queued computation at the same level as well as multiple computations at other higher levels.

- A wake-up is sent to a higher level only if there exists at least one queued computation pending execution at the higher level.

- A level may receive multiple wake-up messages before all its queued computations are released.

Figure 4.5 illustrates how a tree of computations can advance to termination under the aggressive scheme. In particular, we note that the termination of a computation may result in multiple start-ups of others at higher levels, even with a totally ordered security lattice, so long as the invariant is not violated (see figure 4.5(c) where computations 3(TS) and 6(S) are started on termination of 2(S)). We also observe that with aggressive scheduling, by the time the first four terminations have occurred, namely, 1(U), 2(S), 3(TS), and 5(TS), the entire tree of computations has been released for execution (see figure 4.5(e)). Now compare the progress of this tree under conservative scheduling where the first four terminations as shown in figure 4.4 (e), still leaves four others queued and awaiting execution. In summary, the tree progresses to termination at a much faster rate, under the aggressive scheduling scheme.
4.3.3 Hybrid Schemes

We now consider a variant of the level-by-level scheduling scheme. It is a hybrid scheme as it combines both conservative and aggressive approaches. The basic idea is simple. As in the conservative level-by-level scheme, we execute computations on a level-by-level basis. However, when a computation is allowed to be active (by virtue of its level), we allow its immediate children to execute as well, if doing so would not violate serial correctness. Figure 4.6 illustrates how a tree of concurrent computations advances to completion under this hybrid scheme.

To get a quick comparison of the conservative, hybrid, and aggressive schemes, consider the trees in figures 4.4(e), 4.6(e), and 4.5(e), each with four terminated computations. In the conservative scheme, this leaves two computations still pending, while in the hybrid scheme we have only one computation pending execution. On the other hand, with the aggressive scheme, the termination of four computations leaves no computations pending start-up.

4.3.4 A Framework and Metric for Comparative Analysis

The conservative and aggressive schemes discussed above can be seen as two that approach the ends of a spectrum of secure and correct scheduling strategies. This is because it is meaningless to come up with any algorithm that does worse than the conservative one, in terms of the degree of concurrency allowed. At any given time, if there is a computation active at a maximal level in the lattice, then no other computations may be concurrently active. The conservative scheme thus exhibits the least meaningful degree of concurrency within a session. The only way to do worse would be to allow computations at incomparable levels in the lattice to execute one at a time (and not to mention the fact that this would be insecure due to sideways signaling channels). On the other hand with the aggressive scheme, we can poten-
tially have concurrent computations running at every level. This can happen if a computation is forked at the highest level in the lattice, and this is followed by consecutive fork requests where each request is at the next lower level and the lifetimes of these computations are long enough to overlap. One can always increase the degree of concurrency by exploiting intra-level concurrency. But conflicts at the same level can be easily handled by well-known concurrency control techniques. We do not explore this issue further in this thesis as it lies outside the scope of the execution model and scheduling protocols we present.

We now develop the notion of delay-degree as a metric for analyzing scheduling strategies. We demonstrate how by varying this metric, we can derive and admit a family of scheduling strategies offering varying degrees of concurrency, while guaranteeing confidentiality and serial correctness.

We begin with some definitions.

**Definition 4.2** A level is inactive if no computation is executing at the level.

**Definition 4.3** A level is active if there exists an executing computation at the level.

**Definition 4.4** We say a level $l$ is serial-execution enabled (or s-enabled for short), if there exists at least one forked computation $c$, at $l$, and there are no active or queued non-parent computations with smaller fork stamps than $c$, at level $l$ or below.

Intuitively, when a level is s-enabled, executing the next computation at the head of the queue at this level will not violate serial correctness. A computation that is denied execution by a scheduling scheme when its level becomes s-enabled is therefore experiencing an unnecessary delay. We build on this observation and extend it below to an entire security lattice in order to formulate a metric for analysis purpose.
Definition 4.5 A scheduling algorithm introduces an unnecessary delay whenever any level is s-enabled but remains inactive.

Definition 4.6 We say a chain of n security levels in a lattice is fully-enabled whenever every level in the chain is concurrently s-enabled.

Definition 4.7 We define a computation tree to be a full-enabler for a given security lattice, if it causes a longest maximal chain in the lattice to be fully-enabled.

Thus when a maximal chain in the lattice is fully-enabled, computations can be concurrently running at every level in the chain. However, when scheduling is governed by some scheme, it is only certain scenarios that can cause such chains to be fully-enabled. We characterize below the computation trees associated with such scenarios as realizers.

Definition 4.8 For a given scheduling algorithm and security lattice, we define a realizable full-enabler (or realizer for short) to be a full-enabler, which when scheduled by the algorithm, causes a longest maximal chain in the lattice to be fully-enabled.\(^5\)

Definition 4.9 We say a realizer has a delay-degree (d-degree) of k for some scheduling algorithm, if it causes k computations to experience unnecessary delays.

Definition 4.10 Given a security lattice (SC), a scheduling algorithm (A) is considered to have a delay-degree (d-degree) of k, where \(k = \max \{d\text{-degree of all realizers for SC under A}\}\).

\(^5\)It is important to note that our framework is not restricted to lattices; it applies equally well to maximal chains of partial orders.
Given a set of secure scheduling schemes, we can now use their d-degrees as a basis for comparison. We thus need to derive the d-degree for any given scheduling scheme. To do this, we consider all the realizable full-enablers (realizers) and observe the maximum number of computations, excepting the root, that are denied immediate execution, on being forked. This number would give us the d-degree.

As an illustration, consider the full-enabler trees in figure 4.7 for a lattice with a longest maximal chain of three levels U, C, and S (where U < C < S). For the aggressive scheme, we see that both trees are realizers and in either cases no computation would be unnecessarily delayed. For the conservative scheme, only the tree in 4.7(a) is a realizer and we see that computations 2(S) and 3(C) would be unnecessarily delayed. For a further illustration, consider all the full-enabler trees for four levels U, C, S, and TS, as shown in figures 4.8(a) through 4.8(e). All the trees are realizers for the aggressive scheme, and in each case no computation would be unnecessarily delayed. However, only the tree in figure 4.8(a) is a realizer for the conservative scheme and the computations 2(TS), 3(S), and 4(C) would be unnecessarily delayed.

In both of the examples above, we see that the aggressive scheduling scheme would have a d-degree of zero (0), while the conservative scheme would have a d-degree of \( n - 1 \) for a lattice with a longest maximal chain of \( n \) elements. These results are general and not specific to these two examples. To be more precise, the d-degree, say \( k \), of a scheduling scheme holds true for any lattice with a longest maximal chain of \( n \) elements, as long as \( k \leq n \). Also, it follows that for any scheduling scheme with a d-degree of 0, a level is inactive only if it is not s-enabled.

Now are there other scheduling schemes that have d-degrees between the extreme values of 0 and \( n - 1 \)? To answer this question, let us look at the hybrid (variant of the level-by-level) scheduling scheme discussed earlier. Recall that with the level-by-level scheme, computations are executed one level at a time. Thus at
any given time, there is a current-level at which computations are de-queued and executed. While our variant would also require that computations be de-queued and executed one level at a time, it would in addition permit the execution of all the immediate child computations of any active computation at the current-level. To derive the d-degree of this variant, consider again the full-enabler trees in figures 4.7 and 4.8. Both trees in figures 4.7(a) and 4.7(b) are realizers with d-degrees of 0 and $n - 2$ respectively, and thus giving a d-degree of $n - 2$ for this variant (i.e., $\max\{0, n - 2\}$). In figure 4.8 the trees (a), (c), (d), and (e) are realizers with d-degrees 0, $n - 2$, $n - 3$, and $n - 3$ respectively, giving again a d-degree of $n - 2$ for this variant. It thus introduces fewer delays, due to increased concurrency, than the conservative scheme with a d-degree of $n - 1$. We conjecture that by varying the metric d-degree, one could derive several scheduling schemes.
Figure 4.4: Progressive execution under conservative scheduling
Figure 4.5: Progressive execution under aggressive scheduling
Figure 4.6: Progressive execution under hybrid scheduling
Figure 4.7: Full-enablers for a longest maximal chain of 3 elements \( n = 3 \)

\[
\text{(a)} \quad \begin{aligned}
1(C) & \quad \text{Con = n-1} \\
2(S) & \quad \text{Agg = 0} \\
3(S) & \quad \text{Var = 0}
\end{aligned} \\
\text{(b)} \quad \begin{aligned}
1(U) & \quad \text{Con = NR} \\
2(C) & \quad \text{Agg = 0} \\
3(S) & \quad \text{Var = n-2}
\end{aligned}
\]

Figure 4.8: Full-enablers for a longest maximal chain of 4 elements \( n = 4 \)

\[
\text{(a)} \quad \begin{aligned}
1(C) & \quad \text{Con = n-1} \\
2(TS) & \quad \text{Agg = 0} \\
3(S) & \quad \text{Var = 0} \\
4(C) & \quad \text{Var = NR}
\end{aligned} \\
\text{(b)} \quad \begin{aligned}
1(U) & \quad \text{Con = NR} \\
2(C) & \quad \text{Agg = 0} \\
3(S) & \quad \text{Var = n-2}
\end{aligned} \\
\text{(c)} \quad \begin{aligned}
1(U) & \quad \text{Con = NR} \\
2(S) & \quad \text{Agg = 0} \\
3(TS) & \quad \text{Var = n-3}
\end{aligned} \\
\text{(d)} \quad \begin{aligned}
1(U) & \quad \text{Con = NR} \\
2(TS) & \quad \text{Agg = 0} \\
3(C) & \quad \text{Var = n-3}
\end{aligned} \\
\text{(e)} \quad \begin{aligned}
1(U) & \quad \text{Con = NR} \\
2(TS) & \quad \text{Agg = 0} \\
3(C) & \quad \text{Var = n-3}
\end{aligned}
\]