Distributed Speculations: Providing Fault-tolerance and Improving Performance

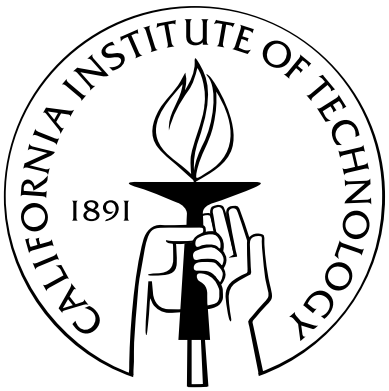
Thesis by

Cristian Ţăpuş

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

2006

(Submitted May 31, 2006)

To my son, Alexandru Ioan,
to my wife, Diana Mariela, and
to my parents, Mariana and Nicolae.

Acknowledgements

"La reconnaissance est la memoire du coeur."
(Gratitude is the memory of the heart)
Jean Baptiste Massieu

I would like to thank all the people who, through their valuable advice and support, made this work possible. First and foremost, I am grateful to my adviser, Professor Jason Hickey, for his distinguished mentor-ship, for the many insightful conversations that led to the development of the ideas in this thesis, for his constant support and his positive attitude, and for giving me the opportunity to work in the great research environment of the Mojave group at Caltech. I would also like to thank my colleagues in the Mojave group for their constructive input and feedback, and, in particular, Nathaniel Gray, David Noblet, Aleksey Nogin, and Justin Smith for their comments and contributions to the development of this work.

Many thanks and lots of love to my son, Alexandru Ioan, for being the best baby in the world. I thank Diana, my wife, for her love and for always being there for me. I thank my parents, Mariana and Nicolae, for their love and continuous support and encouragements. I would also like to thank my sister, Adriana, her husband, Yann, and Diana's mom, Diana, for taking care of Alexandru while I was busy writing this thesis.

Finally, "muchas gracias" to Ernie who brings joy to the entire Caltech community through his positive attitude and through his great Mexican, and not so Mexican food.

Abstract

This thesis introduces a new programming model based on *speculative execution* and it examines the use of *speculations*, a form of distributed transactions, for improving the performance, reliability and fault tolerance of distributed systems. A *speculation* is defined as a computation that is based on an assumption that is not validated before the computation is started. If the assumption is later invalidated the computation is *aborted* and the state of the program is rolled back; if the assumption is validated, the results of the computation are *committed*. The primary difference between a speculation and a transaction is that a speculation is not isolated—for example, a speculative computation may send and receive messages, and it may modify shared objects. As a result, processes that share those objects may be absorbed into a speculation.

The contributions presented in this thesis include:

- the introduction of a new programming model based on speculations,
- the definition of new speculative programming language constructs,
- the formal specification of the semantics of various speculative execution models, including message passing and shared objects,
- the implementation of speculations in the Linux kernel in a transparent manner, and
- the design and implementation of components of a distributed filesystem that supports speculations and guarantees sequential consistency of concurrent accesses to files.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Building safe and reliable programs is an important issue but a difficult endeavor. The challenge is even greater in the context of distributed environments, which may involve complex synchronization operations in the presence of process and network failures.

Transactions are one of the earliest and simplest abstractions for reliable concurrent programming [21]. They provide fault-isolation by guaranteeing the atomicity, the consistency and the durability of the actions performed as part of the transaction. Traditional transactions also provide isolation, which prevents the independent actions inside of a transaction from being visible to the rest of the world until the transaction either aborts or commits. This model has been ubiquitous in the database community but it has not been frequently incorporated in traditional programming languages.

This thesis considers the case where multiple processes may cooperate in a transaction using either message passing or distributed shared objects for communication. To achieve this requires relaxing the transactional isolation property. As it will be shown, this model improves performance and provides fault-tolerance for distributed applications. We call these transactions with relaxed isolation *speculations*. They are introduced as programming language primitives.

Performance is another important topic in the context of cooperative distributed computing, which is significantly affected by the choice of synchronization mechanisms used by programmers. Ideally, when a process reaches a synchronization point it should be possible to allow it to continue executing by *assuming* that the synchronization succeeded, even if the rest of the processes involved in the computation have not yet reached that state. We also call this kind of optimistic computation a *speculation*. Processes start to speculate as soon as they pass a synchronization point and, if it is later determined that a process in the computation has failed, they roll back to a common point and continue their execution.

This thesis presents the operational semantics of a new programming model that provides reli-

ability and fault tolerance using the notion of *speculative execution*. A *speculation*, or *speculative execution*, defines a computation that is based on an assumption that has not been verified yet. If the assumption is later verified, the speculation is *committed* and the execution of the program is continued as expected. If the assumption is invalidated, the speculation is *aborted*, the computation is rolled back, and the execution may continue on a different path.

The proposed speculative execution model defines the interaction between speculations in a distributed environment as well as the actions that are taken when *distributed speculations* are started, committed or aborted. Besides providing reliability and fault tolerance, speculations can also be used to improve the performance of distributed protocols and applications.

This thesis also presents an implementation of speculative primitives as part of the Linux kernel. In this implementation, speculations use lightweight checkpointing to be able to perform the rollback of processes. Speculative processes use a copy-on-write mechanism to backup their address space in memory, rather than saving the entire state of the system on stable storage. Similar mechanisms are used for speculative shared data.

We propose a new filesystem, MojaveFS, that provides speculative execution support for files. MojaveFS is a distributed filesystem with strong consistency guarantees. The design and implementation details of the filesystem are presented in Chapter 6.

The rest of this chapter presents the speculative primitives in an informal way, followed by a set of examples that highlight the advantages of speculative execution. Next we compare our speculative execution approach to improving performance and providing fault-tolerance to other existing mechanisms.

1.2 Speculative Execution

A *speculation* is defined as a computation based on an assumption whose verification may be delayed. We introduce three primitives for defining speculative execution. *Speculate* defines the entry point of a speculation. *Commit* marks the validation of the assumption on which the speculation is based and the program continues the execution as expected. *Abort* is used when the assumption on which the speculation is based is invalidated and the computation needs to be rolled back. In the latter case, the process is rolled back to the state it was in before it entered the speculation and a different path of execution may be taken.

The three functions described above have the following types associated with them, where we define the type *void* as the type of no arguments.

$$\begin{array}{lcl} \text{speculate} & : & \text{void} \rightarrow \text{int} \\ \text{abort} & : & \text{void} \rightarrow \bot \\ \text{commit} & : & \text{void} \rightarrow \text{void} \end{array}$$

```
Transfer (obj1, obj2, s) {
   // We want the transfer to be atomic
   if (sendto(main, REQUEST(obj1,obj2)) < 0)
      return failure
   if (val1=recv() < 0)
      return failure;
   if (val2=recv() < 0)
      return failure;
   update_data(val1,val2,new_val1,new_val2);
   if (sendto(main, UPDATE(obj1,new_v1)) < 0)
      return failure;
   if (sendto(main, UPDATE(obj2,new_v2)) < 0) {
     // Undo first update
     while (sendto(main, UPDATE(obj1, val1))) {
     // Unrecoverable error on write failure
     // Inconsistent state. Try again...
     }
     return failure;
   }
   return success;
}
```

```
Transfer (obj1, obj2, s) {
   if (speculate() == 0) {
      // Enter speculation
      if (sendto(main, REQUEST(obj1,obj2)) < 0)
         abort()
      if (val1=recv() < 0)
         abort();
      if (val2=recv() < 0)
         abort();
      update_data(val1,val2,new_v1,new_v2);
      if (sendto(main, UPDATE(obj1,new_v1)) < 0)
         abort();
      if (sendto(main, UPDATE(obj2,new_v2)) < 0)
         abort()
      commit(); // Speculation committed
      return success;
   } else { // Speculation aborted
      return failure;
   }
}
```

Figure 1.1: Using speculations for fault-tolerance. Left hand side code is written in the traditional programming model. Right hand side code uses the speculative model, which eliminates the error recovery code from the transfer operation.

The *speculate* function returns an integer. The integer is zero when the *speculate* function is first called or non-zero if the speculation is later rolled back. We call the *commit branch* the execution that follows after an initial *speculate* call until either an *abort* or a *commit* call is encountered. If the speculation is *aborted* during the speculative execution the program is rolled back to the *speculate* call and a non-zero value is returned. In this case, the program may take an alternate execution path. We call the code executed in this case the *abort branch*. The *abort* and the *commit* calls do not take any arguments, and conceptually, they don't return a value. Their main purpose is to guide the flow of the speculative program. Thus, speculations have dynamic scoping, based on when the *abort* or *commit* functions are called. This feature will be used in the examples presented in this section.

Speculations provide programs written in imperative languages (like C) with an exception mechanism similar to that found in pure functional languages (like Haskell).

1.3 Examples

We begin by presenting a series of examples that illustrates some of the properties of speculations, including fault-tolerance and improving performance in a very simple fashion.

Fault tolerance with speculations. Consider the traditional database example of writing an atomic function that implements a money transfer between two accounts. We represent it in our example as a transfer of data between two objects. The goal is to implement an atomic operation that

performs an update function on the values of two objects using message passing to read and write the values of the objects from a remote location. Figure 1.1 shows both a traditional implementation of such a transfer function, and a speculative version. Our assumption is that all the operations in the *commit branch* of our speculation will succeed. If any operation (*recv* or *sendto*) fails, the speculation is rolled back and the execution continues on the abort branch. In our example it simply returns failure. If all the operations are successful the execution of the function concludes and the speculation is committed.

It is important to notice that the speculative version *eliminates* the error recovery code from the implementation of the transfer operation. In contrast, in the traditional approach the error recovery code has to be in-line, obscuring the code and making the error recovery dependent on the execution path.

In this example we have included explicit calls to the speculative primitives *speculate*, *abort*, and *commit*. However, the code is simplified significantly if we treat speculations like an exceptions mechanism, where the send and the receive operations raise exceptions on failure and force the speculation to roll back.

The same mechanism can be used to prevent various types of software bugs from crashing applications. For example, in the case of a process crashing because of a buffer overflow bug, we can instrument the application using speculations to force it to roll back to the point where the memory allocation occurred and take a different path of execution (potentially allocating more memory and retrying), thus preventing the application from crashing. This would require minimal support from the operating system. A similar approach that uses traditional checkpoints is suggested in the Rx [47] system. Speculative execution, however, provides a significant advantage over traditional checkpoints in that it allows users to provide alternate paths of execution when the application is rolled back, based on how the precondition of the speculation is invalidated.

Speculations for scientific computing. The second example, shown in Figure 1.2, illustrates the use of speculations in a scientific computing application. The computation grid and the decomposition of the problem for parallel execution for a sample example are also presented in Figure 1.2. As shown, the computation domain of each computation node overlaps with that of its neighbors, allowing it to compute local information with only limited boundary information from its neighbors. The border information is exchanged using a message passing mechanism.

A simplified version of the main computing loop of the application is presented in Figure 1.2. A speculation is started at the beginning of the computation and after each data checkpoint. Data is saved regularly to stable storage, after a fixed number of iterations. This parameter can be adjusted to balance the overhead of speculations against the expected cost of fault recovery. At each computation step, each computing node retrieves the boundary information from its neighbors. If

```
speculate();
for(step = 1; step <= timesteps; step++) {
   /* Get boundary values from neighbors. */
   /* May have to rollback due to failure */
   err=get_borders(u,rows,cols,myid,step);
   if(err == MSG_ROLL)
       abort(1);
   /* Perform the computation. */
   do_computation(step, u, rows, cols);
   /* Save a checkpoint if it's time. */
   if((step % checkpoint_interval) == 0) {
       /* Save the data to stable storage */
       checkpoint_data(step);
       /* Save the current speculation */
       commit();
       /* Start a new speculation */
       speculate();
   }
}
```

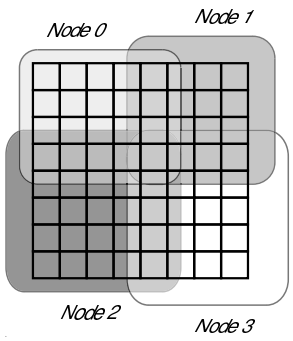


Figure 1.2: Simplified speculative main loop and the 2D domain decomposition for a scientific computation whose performance is improved by using speculations.

any of its neighbors fails, the local computation is rolled back to the previous speculation, and the boundary information is requested again. When a node fails the computation served by it is restarted on a different node and the data from the last saved data checkpoint for the failed node is used to restart that instance of the computation. The application relies on the existence of a reliable and distributed storage medium for a real fault-tolerant implementation. For the purpose of this example, an NFS mount point visible across the entire cluster provides the required functionality. The code presented in Figure 1.2 shows the clear distinction between the checkpointing and speculative code and the rest of the algorithm. The sample code we present can be easily used as a template for a large variety of scientific computing applications. The minimal annotation required through the use of specific language primitives is computation and architecture independent.

Improving performance with speculations. The third example presented in this section discusses the possible impact of speculations on improving the performance of distributed applications. Consider a total order protocol implemented over a reliable network that might reorder packets. Each message sent in the network is tagged with an identifier containing a sequence number. For the purpose of this example we consider sequence numbers to be real numbers, where gaps in the sequence of messages is inherent. When a message M is received, the message is held in a receive queue until all messages M' with smaller IDs have been delivered to the application, and only then is the message processed. We will further use the terms *received* and *delivered* as follows. A message is received when the message is passed from the network driver to our protocol. A message is delivered to the destination when the protocol makes it available to the application layer that uses our protocol. The decision of whether to deliver message M or not cannot always be taken at the time when M is received. We assume that there is a bound on the time between when the message is sent

```
1:  read message M from network;   t_0 ← time
2:  while time < t_0 + T ∧ ∃ m' . last(m') < M do
3:      process messages from network
4:  end while
5:  if ∃ m' . last(m') < M then
6:      enter speculation;   deliver M
7:      abort if receive message M' s.t. M' < M
8:      commit when ∀ m' . last(m') > M
9:  else
10:     deliver M
11: end if

    time is current system time
    last(m') is last message seen from machine m'
```

Figure 1.3: Algorithm for total-order communication using speculations

until it is received. This, together with the assumption that communication is reliable, provides an upper bound on how long a message would be stored in the receive queue.

To optimize performance, our algorithm uses speculations and a sliding window mechanism, similar to the TCP window. The recipient of a message uses speculations in the following way: if the first message, M, in the receive queue has not been delivered within time T, the recipient enters a new speculation that assumes M may be delivered at this time. Once the recipient enters the speculation, it delivers the message. The speculation can be committed once the recipient has seen messages from every other machine with IDs larger than that of M. If the recipient, however, receives a message, M', with ID smaller than M, then the recipient must abort the speculation and return to the state where M was at the head of the queue and waiting to be delivered. The new message, M' will be put at the head of the queue and the procedure is repeated. Figure 1.3 gives the pseudo-code for this algorithm.

Section 3.1 presents a mathematical model, what are the conditions for choosing the waiting window T, as a function of the overhead of entering, aborting, and committing a speculation, to improve the performance of the system.

The problems that can be optimized using speculations do not reduce to the class of total-order protocols. If a computation is based on a condition that is usually expensive to compute, but that most of the time returns an easy to estimate result, we can use idle computing units to concurrently execute the verification and the computation. The computation would be executed speculatively, assuming a certain return by the verification procedure. This mechanism can increase the overall performance of the system. Smart devices, like intelligent network cards and graphic cards, are becoming a commodity with high computation abilities that is idle most of the time. We envision that verification tasks could be shipped to such devices, while the computation speculatively executes

on the main CPU.

A Distributed Speculative Web Reservation System Another interesting example considers the interaction between different active entities (processes) in the system while they perform speculative operations. We use a reservation system to illustrate this. A client needs to reserve a plane ticket and a hotel room. She contacts a flight agent for the airfare, and a hotel for lodging that confirms availability and reserves a room for the requested dates. The programs using speculative constructs for the client and the agents are shown in Figure 1.4.

Client	Flight Agent	Hotel
request-flight	receive-request	receive-request
request-hotel	check-reservations	check-availability
speculate	speculate	if room then
get-quotes	send price-quote	send price-quote
if expensive then	check with airline	get payment
abort()	if unavailable then	else
else	*abort*()	send NO-ROOM
wait confirmation	else	
if all-OK then	send-confirmation	
pay for services	get payment	
commit()	*commit*()	
else *abort*()	$\oplus$	
$\oplus$	do nothing	
try different agents		

Figure 1.4: Speculative programs for reservation system.

Figure 1.5 shows the speculative dependencies for a successful reservation. The client requests quotes from both the flight agent and the hotel and speculates that the prices will be acceptable and that she will succeed with the reservation (speculation $s1$). The flight agent receives the request and computes a quote based on her local information, which might be inaccurate. It speculates that the reservation will be confirmed by the airline (speculation $s2$), and sends the quote to the client, pending a final confirmation from the airline. When the airline confirms the reservation the information is forwarded to the client and the speculation is committed on the flight agent's end. Meanwhile, the hotel receives the request, it successfully processes it and sends the quote to the client. When the client receives the speculative quote from the flight agent her speculation and the speculation started by the flight agent are merged ($s = merge(s1, s2)$). Furthermore, when its payment is received by the hotel it absorbs the hotel in the same speculation. The speculation is successfully committed only if both the client and the flight agent commit the speculation.

Figure 1.6 illustrates the behavior of the system in case of an aborted speculation. The first set of events is identical with the previous case. The flight agent runs the commit call because the reservation is successful on the airline's end. The speculation is not fully committed, since upon receiving the quote from the flight agent the client's speculation ($s1$) and the flight agent's

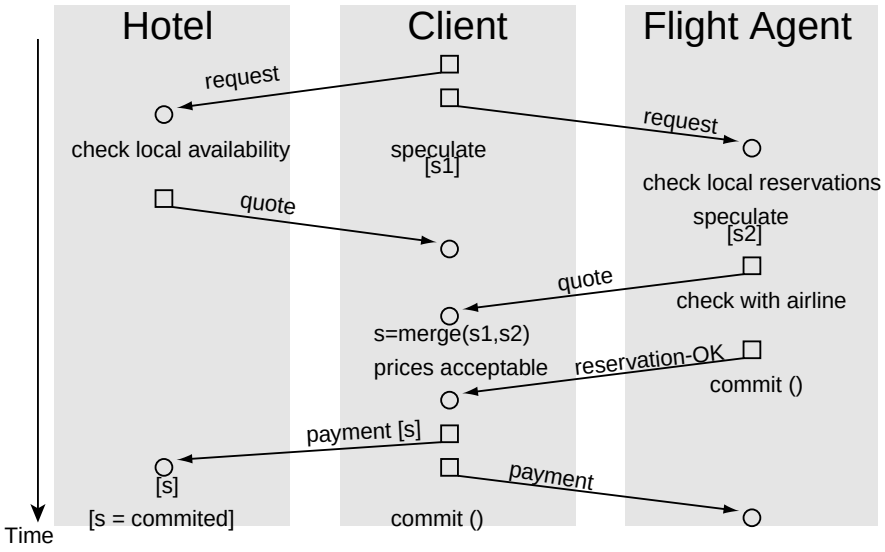


Figure 1.5: The reservation succeeds. Send operations are shown as squares, while receives are circles.

speculation ($s2$) have merged ($s = merge(s1, s2)$), and the client has not committed the speculation. After receiving the quotes from the flight agent and the hotel, the client decides the prices are too high and aborts its speculation. This triggers the rollback of both the client and the flight agent.

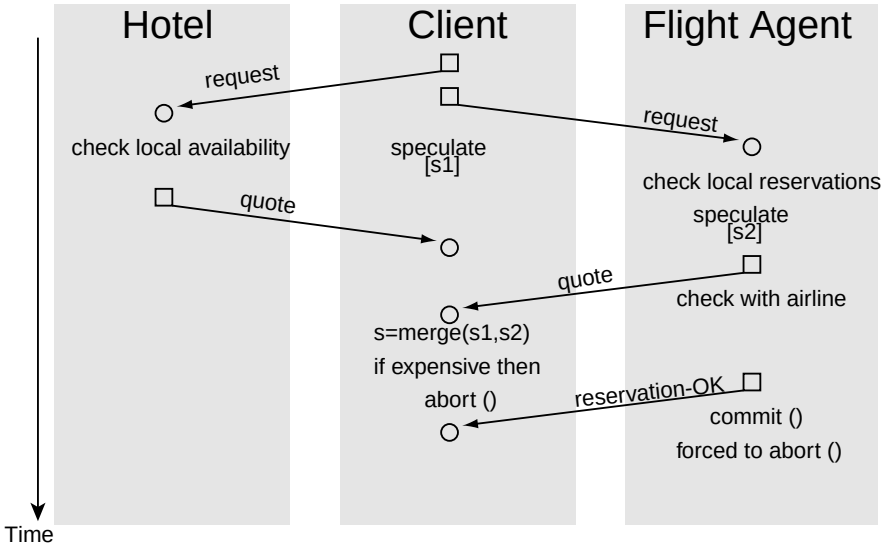


Figure 1.6: The reservation fails because prices are too high. Send operations are shown as squares, while receives are circles.

This example shows the ability of speculations to increase the parallelism and to restore the state of a distributed system to a consistent state upon an optimistic assumption being invalidated. The same type of problem has seen great exposure and has prompted the development of new transactional models in the database community [20].

1.4 Comparative Analysis

There are three main disjoint domains that could claim overlap with the speculative primitives that we introduce: databases, recoverability and exception handling in programming languages.

The field of databases uses transactions as primitives for reliable computing. The traditional database transactions are characterized by the ACID (Atomicity, Consistency, Isolation, Durability) properties. These allow transactions to be serializable and prevent inter-dependencies between transactions. Speculations can be thought of as a variant of transactions that lacks isolation. Speculation have dynamic scoping, which is different from how transactions work. This increases parallelism and, when used properly, could improve performance significantly.

Automatic recovery from faults is another area that shares similarities with our primitives for speculative execution. Recovery is usually performed using a checkpoint/recovery mechanism. Recovery of distributed applications may involve coordinated checkpoints and message logging. In this respect, the lightweight checkpoints required by speculations and the distributed rollback that is employed in case a speculation is aborted could be seen as a common point of the two mechanisms. Unlike the checkpoint and recovery mechanism, speculations allow processes to take a different path of execution upon rollback. This makes some of the algorithms used in checkpointing and recovery inapplicable to speculations. Furthermore, the ultimate goal is different. Checkpoint and recovery mechanisms are targeted at overcoming hardware failures with minimal computational overhead. Speculative execution, on the other hand, provides programming language primitives that eliminate the need for writing recovery code for soft failures in-line with the computation. If combined with existing checkpoint and recovery mechanisms, speculations could drastically improve performance and provide fault tolerance, as showed in the examples.

Exceptions and exception handling in programming languages is another area related to speculations. Exceptions are usually used

- to separate code error handling code from the computation,
- to propagate errors up the call stack,
- to group or differentiate errors.

Exceptions may also be classified by the safety level they provide, as follows:

- failure transparency, where the operations succeed even in the presence of failures,
- commit or rollback semantics, where operations can fail but failure is guaranteed not to have side-effects,
- data consistency semantics, where side-effects may occur, but the state invariants are preserved (data consistency is maintained),

- no exception safety, where no guarantees are made.

Speculations provide a strong "commit or rollback" semantic. While there are libraries [2, 1] that implement exception mechanisms for C, they only provide weaker guarantees, like data consistency semantics or no safety guarantees.

Since actions performed inside a speculation are not isolated, other processes can become implicitly speculative if they base their computation on a speculative message. This distributed component of speculations enforces the exceptions semantics on a distributed level, by forcing implicitly speculative processes to roll back along with the initiator of the speculation. This component is not found in traditional exception handling mechanisms.

1.5 Contributions

The contributions presented in this thesis include:

- the introduction of a new programming model based on speculations,
- the definition of new speculative programming language constructs,
- the formal specification of the semantics of various speculative execution models, including message passing and shared objects,
- the implementation of speculations in the Linux kernel in a transparent manner, and
- the design and implementation of components of a distributed filesystem that supports speculations and guarantees sequential consistency of concurrent accesses to files.

1.6 Outline of the Thesis

The thesis continues with a detailed discussion of related projects and how they are different than the programming model introduced by speculative execution (Chapter 2). A semi-formal representation of speculative primitives is presented in Chapter 3. A formal discussion of three operational semantics models and proof of equivalence with nonspeculative execution is provided in Chapter 4. The implementation details of speculations in the Linux kernel are presented in Chapter 5. This is followed by the design and implementation overview of the MojaveFS distributed filesystem in Chapter 6. The thesis is concluded with a summary of the material covered.

Chapter 2

Related Work

The related work can be broadly classified in three major categories. Firstly, there is a vast area of research that covers database transactions [21], distributed transactions and their adoption in compilers, operating systems, and hardware. Secondly, we can find significant work in using checkpointing and recovery as tools for fault-tolerance and software reliability. Thirdly, there is work on optimistic execution (or speculative execution) that focuses on how hardware and software systems can use short-lived speculative blocks to improve their performance. We discuss each of these categories below, providing specific examples and comparisons with our system.

2.1 Transactions

Database transactions, known for their ACID (Atomic, Consistent, Isolated, Durable) properties, are a powerful concept in providing reliability and fault-tolerance. They have been studied both from a theoretical point of view, and through application to various areas of computer science where reliability is desired, like programming languages, filesystems, and shared memory systems.

2.1.1 Theory of Transactions

While database transactions have been studied extensively [21], an operational semantics describing the behavior of speculations that could be applied to other domains has only been recently provided [46]. In their approach, Prinz and Thalheim discuss ACID transactions and do not take into consideration any relaxation of the properties.

Black et.al. [5] provide a very interesting equational theory of various types of transactions. They discuss lightweight transactions that deviate from traditional transaction by relaxing either the Consistency or the Durability property [5]. Their work is particularly interesting since they provide a theoretical analysis of how lightweight transactions may be nested, or composed through either parallel or sequential composition. The theory they provide is presented using an equational

calculus that has limited expressiveness, as it only analyzes actions and does not capture state. Furthermore, in their approach isolation is implicit.

Non-isolated transactions have been studied in the context of long-lived transactions that can hold on to database resources for long periods of time, delaying the termination of other transactions. Molina and Salem [20] introduced the concept of *saga*, which is a non-isolated, non-atomic transaction formed of a set of smaller isolated, atomic transactions and a set of compensating transactions that undo the actions of the smaller transactions if those have to be rolled back. The description of executing parallel sagas is limited. Relaxing isolation creates complex dependencies of transactions, which is why this has not seen the same level of exposure as other lightweight transactions.

Moss [40] introduced the notion of nested transactions in his Ph.D. thesis. His model is similar in many respects to the nested speculations model we presented. The main difference remains that speculations are not isolated, which allows them to create distributed dependencies in the system.

Our concept of speculations and traditional database transactions share many traits, but they are distinct in one significant way: speculations do not provide isolation. Thus, processes executing inside speculations expose their actions to the outside world and can absorb other processes in their speculation. While this might seem like opening a can of worms, we believe that it is a powerful mechanism that, if used properly, can have a significant positive impact on the performance of applications and on their reliability.

Furthermore, we push the concept of non-isolated transactions from databases to programming languages and distributed environments. This requires redefining its semantics to the new domain it is used in, which we do by specifying several models of speculative execution in the form of operational semantics.

2.1.2 Transactional Systems

We discuss applications of transactions to programming languages, filesystems and shared memory. Transactions are used in all these domains primarily for their ability to provide atomicity and isolation.

2.1.2.1 Transactional Shared Memory and Programming Languages

Software transactional memory was introduced in a seminal paper by Herlihy [26] in which he proposes a new methodology for constructing wait-free and non-blocking implementation of concurrent objects. This area has seen a significant amount of work. Recent research enables automatic conversion of correct sequential programs to concurrent code that does not use locking or other complex synchronization mechanisms that are prone to deadlocks and errors [37].

The Plurix operating system is implemented on top of a transactional distributed shared memory infrastructure [61]. It integrates transactions with optimistic synchronization mechanisms to

guarantee sequential consistency. The Plurix operating system is purely transactional, in that the running unit is not a process but a transaction. This makes the Plurix operating system behave like a distributed database system.

In the Venari project, Haines et.al. [22] implement a transaction mechanism as part of Standard ML, utilizing a mutation log produced by a generational garbage collector to implement undoability.

Harris and Keir provide conditional critical regions implementation in Java and integrate it with transactional memory and atomic execution blocks [24]. Their approach has been very successful in moving away from locks and condition variables in writing concurrent application. Unlike speculative execution, their transactional memory considers only isolated atomic blocks that could be evaluated in parallel.

2.1.2.2 Support for Transactions in Hardware

Software transactional memory and hardware implementations of similar atomic primitives have evolved in parallel. One of the early works describes an architecture with support for transactions [27]. The authors introduce transactional primitives for accessing memory, including the following: *load-transactional*, *load-transactional-exclusive*, *store-transactional*, *commit*, *abort*, and *validate*. They use a simulation environment and show the advantages that transactional memory has over traditional locking in terms of performance.

Other architectures with the same flavor have been suggested. Most of them set a limit on the size of the operations inside each transaction. Ananian et.al. [4] introduce an architecture for supporting unbounded transactional memory. Their approach addresses the issue of having transactions that are larger than the CPU cache, for which they devise special mechanisms to enable rollback. However, the life of transactions supported in hardware is a few orders of magnitudes shorter than that which we introduced through speculative execution. Furthermore, the traditional transactional memory support transactions with strict ACID properties. Hardware support for transactions does not consider distributed dependencies or distributed rollback.

Another interesting topic in the area of transactional memory support in hardware is building mechanisms that can identify, at runtime, lock-protected critical sections in programs and execute them without actually acquiring the lock [48].

The speculative execution in our system provides a programming model that extends optimistic computation to distributed environments.

2.2 Checkpoint and Recovery

Another area of research that is related to speculations is using checkpointing and rollback mechanisms to provide recoverability. Both theoretical and practical works have discussed various ap-

proaches and protocols that enable distributed applications to recover in a consistent state based on saved checkpoints. This is achieved by either optimistic or pessimistic message logging, and by coordinated or uncoordinated checkpoints. The goal of checkpoint and recovery algorithms is to provide distributed applications with mechanisms to survive failures and roll back their state to a previously globally consistent state. They usually assume the existence of stable storage that survives failures.

Strom and Yemini introduced in [55] the notion of optimistic recovery. They define it as a technique based on dependency tracking, which avoids the domino effect while allowing the computation, the checkpointing and the "committing" to proceed asynchronously. Their approach requires the analysis of existing checkpoints and computing a global safe recovery line. This can be done either centralized [32] or distributed [52]. Furthermore, their approach assumes that upon rollback the execution continues on the same path as before, hence messages that have been logged since the checkpoint can be replayed. This is true for most of the subsequent work based on optimistic logging [32, 41]. This approach is incompatible with speculative execution since processes may take a different execution path when the speculation is aborted.

The "Virtual Time" [31] paper introduces a mechanism for optimistic speculative execution in a distributed system. Processes exchange messages and assume that the messages they receive are in order. In case this assumption is violated the computation is rolled back, the messages in the receive queue are reordered and the computation continues by processing messages in the newly provided order. Processes are implicitly forced to checkpoint on regularly and the rollback mechanism can be cascading. The specification of the system requires the state of each process to be saved after each send or receive operation. On rollback, certain messages have to be "annihilated" using "anti-messages."

While speculations are similar to the concept of lookahead-rollback introduced by the Time-Warp [31] mechanism, we extend the concept by allowing both explicit and implicit speculations through programming languages extensions. We also introduce shared objects as part of the speculative model.

Other projects, like Condor [36], CRAK [62] or Score [56] support only heavyweight checkpointing and recovery mechanisms. Furthermore, none of these systems present a formalized operational semantics of their checkpoint/recovery mechanism.

The *Rx* [47] system uses checkpointing and rollback to enable applications to survive software bugs. It regularly saves checkpoints of running applications and, in case of process failure, it rolls back the process to one of the previously saved checkpoints. It performs various modifications to the environment in which the application is running, like padding newly allocated buffers to prevent buffer overflows, and it re-starts the program from the saved checkpoint. This mechanism proved to be efficient in combating failures due to certain race conditions and buffer overflows. The limitation of the Rx system is that it operates solely on isolated applications. A similar mechanism that would

enable the recovery of distributed applications could be implemented using speculations instead of the traditional checkpoint and rollback mechanism.

The main differences between this area of research and our approach are:

- speculations can provide programs with alternate execution paths upon rollback,
- speculations are lightweight checkpoints that are stored in memory and can be coupled with real checkpointing mechanism for increased reliability, and
- we expose speculations as programming language primitives that have a semantics closer to that of transactions than that of checkpoints.

In our implementation of speculation we do use mechanisms similar to those designed for checkpointing/rollback systems, like the protocol designed by Damani and Garg [14], to ensure safe recovery lines in case of distributed speculation rollback.

2.3 Speculative execution

Concepts of optimistic execution similar to speculations are used to address optimizations of I/O operations [11], fault-tolerant networking [60], shared memory systems [34] and also to increase the performance of processors [43]. By introducing programming language primitives we extend the usability of speculations to a wider range of applications.

One of the most recent systems, BlueFs [42] uses speculations to improve the performance of NFS. Its implementation uses a very similar kernel-level implementation of speculations and speculative actions based on internal primitives similar to those introduced by us in [59]. As mentioned before, our approach pushes speculative primitives to user level and provides an implementation that is able to handle distributed speculations and distributed rollback, making it more generic and more widely applicable. Furthermore, our system is based on a strong formal semantics which increases the confidence in our implementation.

The *angelic nondeterminism* [29] concept introduced by Hoare has a semantic that is similar to speculative execution. It defines nondeterminism ($P \sqcap Q$) as the "execution of both P and Q concurrently until the environment chooses an event which is possible for one but not the other." This implementation of nondeterminism has a high cost in terms of efficiency. In our speculative model we optimize the *angelic nondeterminism* implementation by setting a higher preference for one of the two execution branches, based on the assumption that we make. This permits a more efficient implementation. Furthermore, we consider communicating processes and the effects of rollback (abort) to the state of the entire distributed system.

Chapter 3

Semi-Formal Models

This chapter discusses the intuition behind speculative execution. It uses a semi-formal framework to present the behavior of speculative primitives.

It begins with a discussion of one of the previously introduced examples. The total order protocol example is used to introduce a probabilistic mathematical model. The model provides an analytical view of when speculations should be used to improve performance and how that decision depends on the overhead introduced by the speculative primitives.

Next, the chapter presents the effects of speculative primitives to the state of a distributed system. The discussion includes an overview on how the model captures the state of a distributed environment. It also shows how speculative primitives, in conjunction with communication primitives and with operations that access shared objects, interact with process and object states in such environments.

3.1 Improving Performance of a Total Order Protocol

Consider the total order protocol described in Section 1.3. The pseudo-code of the algorithm in reproduced in Figure 3.1 to facilitate references to it throughout this section.

Two possible outcomes of executing this protocol are illustrated in Figure 3.2. On the left hand side of the image the outcome of a successful speculative message delivery is shown. In this case the protocol waits for a given time T after the receipt of the message before it starts speculating. It speculates that M is deliverable and begins computing based on the value provided by M. This way useful computation is accrued. The confirmation that M is deliverable is received later on. In the speculative version the confirmation triggers the commit of the speculation and the earlier started computation continues. In the nonspeculative model, shown on the same graph, the useful computation based on message M is only started upon receiving the confirmation of M's deliverability.

The right hand side of the figure shows the case when the speculation that is based on the deliverability of the last received message is invalidated several times. However, as shown, at some

```
1:  read message M from network;   t0 ← time
2:  while time < t0 + T ∧ ∃ m' . last(m') < M do
3:      process messages from network
4:  end while
5:  if ∃ m' . last(m') < M then
6:      enter speculation;   deliver M
7:      abort if receive message M' s.t. M' < M
8:      commit when ∀ m' . last(m') > M
9:  else
10:     deliver M
11: end if

    time is current system time
    last(m') is last message seen from machine m'
```

Figure 3.1: Pseudo-code for the speculative algorithm used to implement a total-order communication protocol

point the message that should be delivered to the application arrives. In this case we fall back to the case shown on the left hand side of the figure, where speculating on the deliverability of that message is successful.

In the next section we present a probabilistic model that gives an analytical expression of the condition under which the speculative protocol described outperforms its nonspeculative counterpart.

3.1.1 Mathematical Model for Speculations

This mathematical model of speculations helps clarify the gains and losses of using them. The notation used in this section is first introduced. The notation $E[X]$ represents the expected value of random variable X. The notation $E[X]_{low}^{high}$ represents the expected value of random variable X for the range *low-high* of its domain. Thus, $E[X] = E[X]_0^\infty$ for a variable X defined on $[0, \infty)$

The goal of the model is to determine the latency L between the time message M is initially received and the time that M is successfully delivered to the application and compare it to the latency L_c from the non-speculative (classical) model. We begin by showing what the expected latency for the classical model is and then we describe the specifics of our model and derive the formula for its latency.

For each message M that is received we distinguish two cases: (1) the message could be delivered right away (immediate delivery case), or (2) the message needs to wait until some earlier messages are delivered (delayed delivery case). To represent this distinction we use a parameter p that gives the probability that message M could be delivered right away. We also consider two random variables, U and V that represent, for each case, the time from the reception of the message until the moment we can safely deliver it. Even for the immediate delivery case we might have to wait for a certain

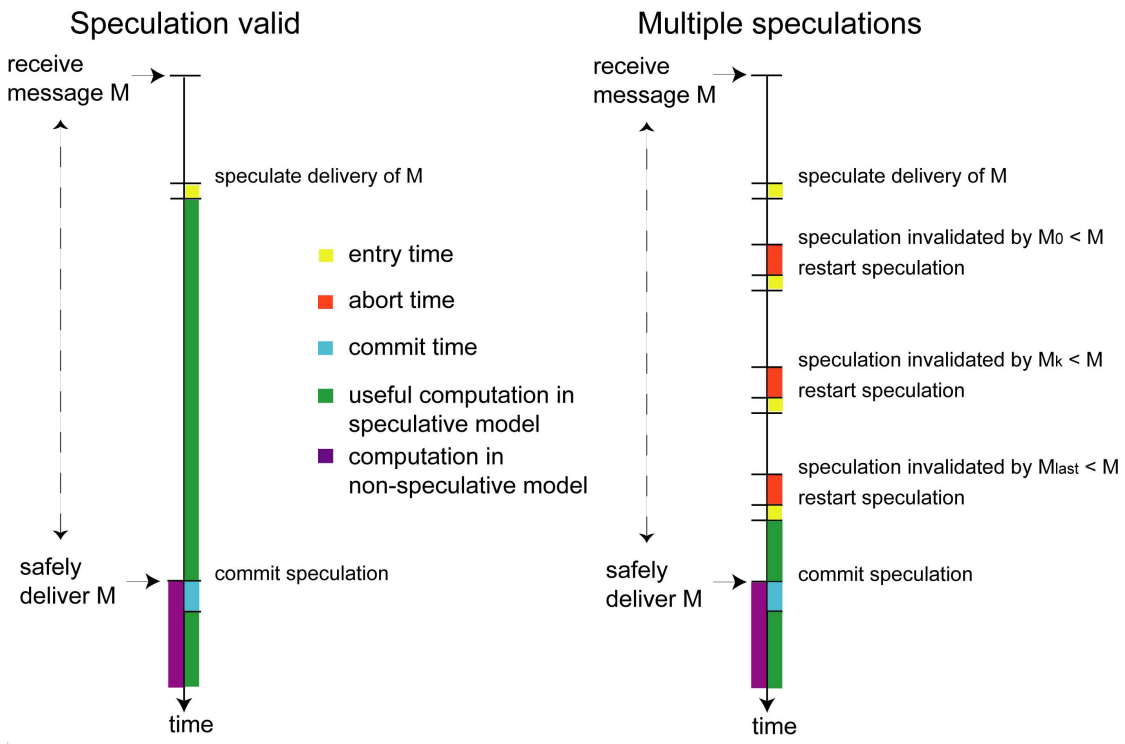


Figure 3.2: Two possible speculative executions of the speculative total order protocol.

period of time until we have the guarantee that it is safe to deliver it. This is the time modeled by random variable U.

The expected latency for the classical model is given by the following formula:

$$E[L_c] = pE[U] + (1-p)E[V] \tag{3.1}$$

To complete our model we also have to take into consideration the window size T and the arrival of messages with smaller IDs than the one we are trying to deliver. Let q be the probability that random variable U is greater than the window size T, and let r be the probability that random variable V is greater than T. The waiting window size T denotes the time we wait before we start speculating. However, if during time T we can deliver M we do it right away. Let W be a random variable representing the time from the reception of message M until the reception of the last of all messages with IDs smaller than that of M. We know that $W < V$ for the entire domain.

We can proceed to compute the expected latency for our model as follows. Consider the same two cases as for the classical model, described by probability p. For the immediate delivery case, with probability $(1-q)$ the latency is the expected time until we can safely deliver M (since with probability $(1-q)$ the delivery time is less than T) and with probability q the latency is the time we wait until we start speculating plus the time consumed for entering (T_e) and committing (T_c)

the speculation. In this case, we know the speculation succeeds without any doubt, so there is no abort time (T_a) involved.

For the delayed delivery case, with probability $(1 - r)$ the safe delivery time is less than T. With probability r we start speculating before delivering the message. However, the time spent speculating until the moment we receive the last message with an ID lower than M's is also part of the latency. When the last message with an ID lower than M is received we incur the abort time of the current speculation plus the time to enter (T_e) a new speculation, which is guaranteed to succeed, so we also add to it the commit time (T_c).

The exact mathematical formula is given below:

$$E[L] = p((1-q)E[U]_0^T + q(T + T_e + T_c)) + (1-p)((1-r)E[V]_0^T + r(E[W]_T^\infty + T_a + T_e + T_c)) \quad (3.2)$$

To find the condition for which the latency of our system is less than that of the classical model we require that $E[L] \leq E[L_c]$. This is satisfied if the following inequality holds:

$$pqE[U]_T^\infty + (1-p)r(E[V]_T^\infty - E[W]_T^\infty) \geq pq(T + T_e + T_c) + (1-p)r(T_a + T_e + T_c) \quad (3.3)$$

We used the following formula in rewriting the classical model expected latency:

$$E[X] = p(X < Y)E[X]_{-\infty}^Y + (1 - p(X < Y))E[X]_Y^\infty \quad (3.4)$$

Inequality 3.3 gives us the requirements our system must satisfy to perform better than the classical model. We distinguish again between the two cases. First, for the immediate delivery case, we improve only if the expected delivery time is greater than the size of the waiting window T plus the time spent to enter and commit the speculation. Second, for the delayed delivery case, we improve only if the time difference between the safe delivery time and the time the last message with lower ID is received is greater than the time spent to abort a speculation plus the time to enter and commit the final speculation.

The same framework may be used to build similar probabilistic models for other speculative protocols and algorithms. We believe that applying such models to applications of speculations in distributed filesystem design and other related areas would increase their performance significantly.

		Construct	Description
e	::=	$speculate(e_1 \oplus e_2)$	Speculate call
		$commit()$	Commit call
		$abort()$	Abort call
		$sendto(p_i, m)$	Send a message to process p_i
		$let\ v = recv()\ in\ e$	Receive a message from the receive queue.
		$let\ v = read(o_j)\ in\ e[v]$	Read the value of object o_j
		$write(o_j, x)$	Write value x to object o_j
		$e\ ;\ e$	Sequencing

Figure 3.3: Syntactically valid terms

3.2 Speculative Behavior

3.2.1 Semantics of Speculative Behavior

This section presents the semantics of our speculative primitives in the context of a distributed environment. The semantics is presented as a set of rules that capture the transition of the distributed system state when speculative actions occur in the system. For clarity, a diagram representation of the semantics is used, rather than a purely algebraic form.

3.2.1.1 Syntax

Our model considers a set of processes running concurrently in a distributed environment. The processes communicate with each other either by exchanging messages or through reads and writes performed on shared objects.

Figure 3.3 defines the syntax of the discussed primitives. The $speculate(e_1 \oplus e_2)$ primitive defines a speculation. The program e_1, called the "commit" branch, is executed while the assumption that the speculation is based upon is assumed to be valid. If the assumption is invalidated, then the program executes the $abort()$ call, the speculation is aborted and the process rolls back and executes e_2, the "abort" branch. If the assumption is validated (the $commit()$ call is executed) the process continues execution inside e_1.

The *sendto* and *recv* primitives define point-to-point message passing. The *read* and *write* primitives are used to access shared objects. The *recv* and the *read* constructs are shown using a functional syntax.

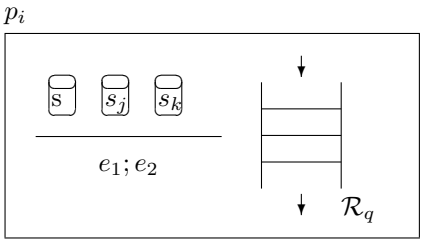
3.2.1.2 Terminology and Notation

The model requires capturing the state of the distributed environment. The transition triggered by the speculative action of one process may reflect upon other processes and objects in the system. For

this purpose, we define the state of the distributed system as having two components: the individual states of all the processes in the system and the states of all the shared objects in the system.

The state of a process p_i, graphically represented by a square box, has three components:

- a checkpoint list (each checkpoint being labeled with a speculation identifier: s, s_j, s_k),
- the program that the process executes ($e_1; e_2$), and
- a receive queue ($\mathcal{R}_q$).

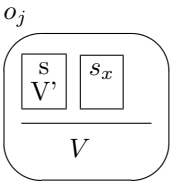


The checkpoint list contains saved checkpoints, the most recent one being represented as the leftmost one. A checkpoint, graphically represented as a cylinder, is created each time a process starts a new speculation. It contains the state of the process and is labeled with the speculation on which it depends. The checkpoint is used to roll back the state of the process to the state it was in before entering the speculation if the speculation is aborted.

All messages received by a process are enqueued in a single receive queue ($\mathcal{R}_q$). Messages are tagged with a speculation id (in case the sender is involved in a speculation). The receive queue is accessed through two operations: *enqueue* and *dequeue*.

The state of an object o_j, graphically represented as a square box with rounded corners, has two components:

- a checkpoint list (s, s_x), and
- the current value of the object.



Each checkpoint in the checkpoint list contains the identifier of the speculation on which the checkpoint depends and the value of the object before it entered the speculation. The checkpoints are used in case speculations are aborted and the state of the objects needs to be rolled back.

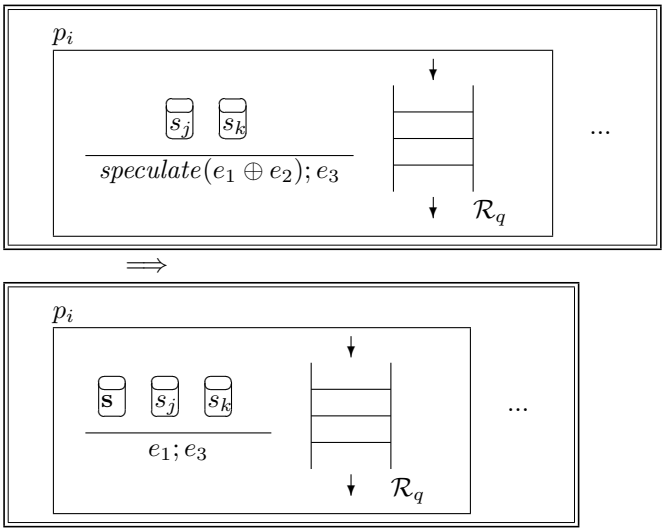
3.2.1.3 Semantics of Speculations

This section presents a set of state transition rules that provide an overview of the semantics of the speculative model. Each of the rules is accompanied by a brief discussion of the intuition behind it, followed by the graphical representation of the state change.

Process execution may be arbitrarily interleaved. In the diagrams, the identifier p_i represents the choice of an arbitrary process.

Starting a speculation. When a process starts a new speculation by invoking the *speculate* primitive, it has to create a local checkpoint of its state. This checkpoint is used in case of a rollback. The speculation is associated a new speculation identifier, call it s. The checkpoint depends on the new speculation s, which is graphically represented by printing the identifier of the speculation in the block represented by the checkpoint. The checkpoint also keeps the *abort branch* of the speculation. The process continues to execute inside the commit branch of the speculation, as shown. The states of other processes and objects in the system are not affected directly by this primitive. Rule SPECULATE below illustrates this behavior.

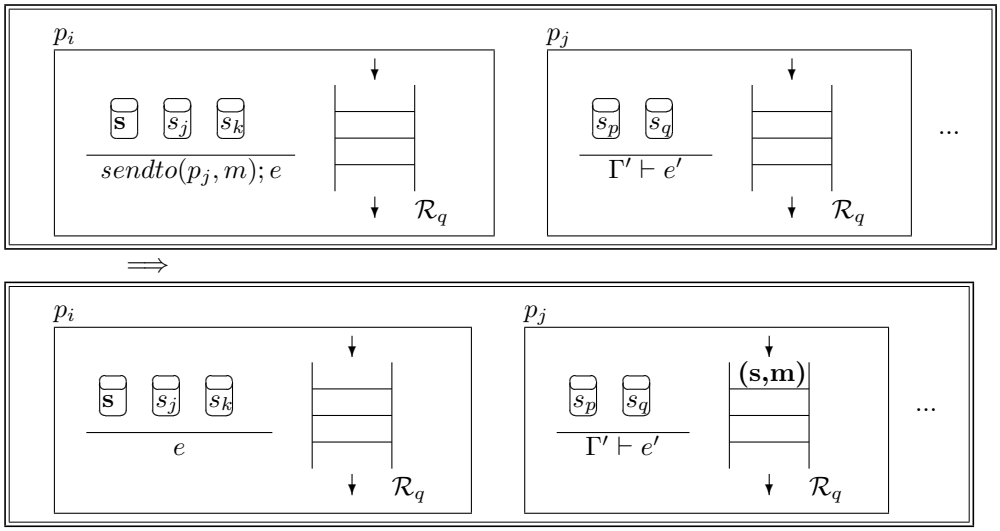
Speculate



Speculative message passing. The behavior of the message passing primitives are presented only in the context of speculations. In the absence of speculations the behavior of these primitives is clearly understood and does not present any interest to the topic at hand.

The SENDTO-SPECULATIVE-MESSAGE rule shows the effects of the sendto operation when the sender of a message is speculative. For simplicity, we ignore network latency in this informal model and assume that the message instantaneously arrives at its destination. The local state of the receiving process is modified because the message is appended to its receive queue. The sender of the message continues its execution with the next statement in its program. As long as the message is not processed by the receiving process there are no other changes in the state of the system.

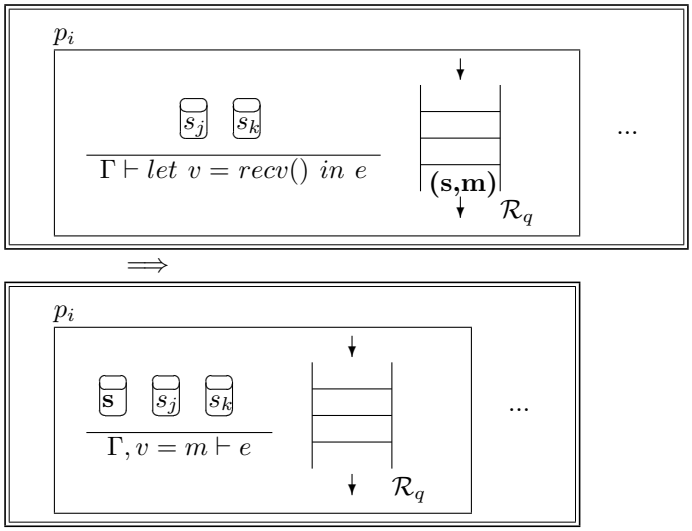
SendTo-Speculative-Message



When a speculative message is processed by its receiver, the receiver is absorbed in the speculation the sender belonged to when the message was sent. By receiving a speculative message the computation of the receiver becomes dependent on speculative data. Since the act of sending the speculative message might be rolled back, if the assumption on which the speculation is based is invalidated, the receive operation should also take into consideration a rollback operation. This is guaranteed by forcing the receiver to become part of the same speculation as the one of the message it receives, and implicitly the one the sender belongs to.

The receiving process creates a checkpoint, as in the case described by the SPECULATE rule above, with the following differences. The process is prevented to decide the outcome of the speculation through explicit *abort*() or *commit*() statements. This is ensured by setting a flag in the saved checkpoint (not shown in our graphical representation). This is the desired behavior since processes that are implicitly absorbed in speculations due to speculative messages should not be aware of the speculation. Implicit speculations are transparent. The checkpoint is also tagged with the speculation on which the message depends (s). The RECEIVE-SPECULATIVE-MESSAGE rule presents this behavior.

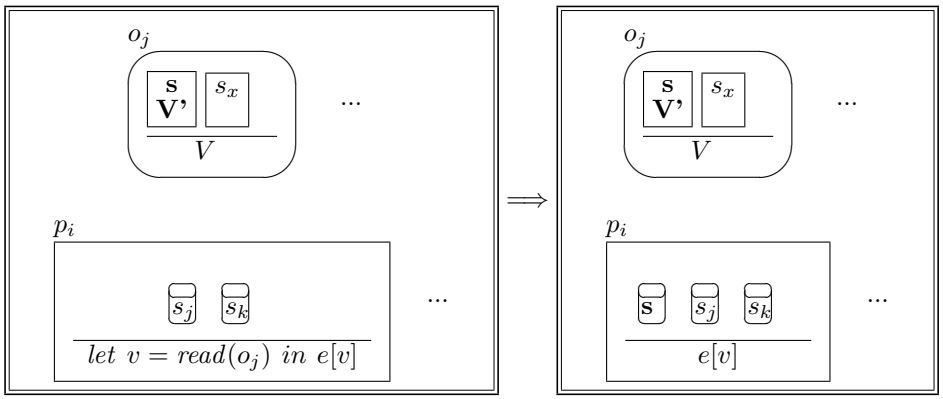
Receive-Speculative-Message



Reading from / Writing to a Shared Object. Shared objects are another vehicle that help the propagation of speculations in distributed systems. Again, we restrict the discussion to actions that are speculative. In the absence of speculations the effect of a *read* operation is reflected only upon the state of the process that issues the command and the value of the object is assigned to a local variable. A purely nonspecualtive *write* has effects on the object on which is performed and it overwrites the value of the object with the one provided to the *write* operation.

However, in the presence of speculations side-actions need to be performed, as described next. If a process reads data from a speculative object that belongs to a different speculation its state depends on speculative information. Therefore, the process is absorbed in the object's speculation. As usual, a checkpoint of the process is created to allow it to rollback if the speculation is later aborted, as shown in rule Read-Speculative-Object.

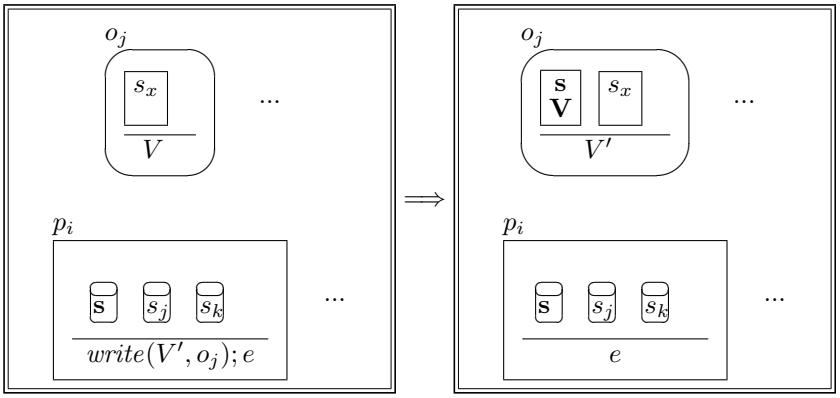
Read-Speculative-Object



When a speculative process writes a value to a shared object that is either nonspeculative or that

belongs to a different speculation it forces the object in the speculation. The object's value depends on speculative data and before being absorbed in the speculation it needs to create a checkpoint. The checkpoint stores the value the object had before becoming part of the speculation, which allows it to rollback if the speculation is aborted. The speculation id is also stored as part of the checkpoint. From this moment, the object will absorb processes that read its value into the same speculation.

Write-Speculative-Process



Aborting a speculation. The next three rules describe the behavior of the system when speculations are aborted. The first one, ABORT-SPECULATION, describes the system's state transition when the owner of a speculation aborts the speculation it started.

In this case we distinguish three parallel actions:

- the owner process rolls back to the point where the speculation was started and continues execution on the abort branch.
- if the owner of the speculation has started any other speculations since it started the one that it explicitly aborted it needs to abort those speculations as well, as described by the two next steps.
- other processes in the system that depend on the aborted speculation(s) will see a system-wide checkpoint invalidation step. A checkpoint is invalidated if it was created when the process entered a speculation that has now been aborted. We graphically represent the invalidation by crossing out the checkpoint in the process's state.
- objects in the system will see a similar invalidation step. Checkpoints that depend on aborted speculation are invalidated. Graphically, we use the same crossing representation as in the case of processes.

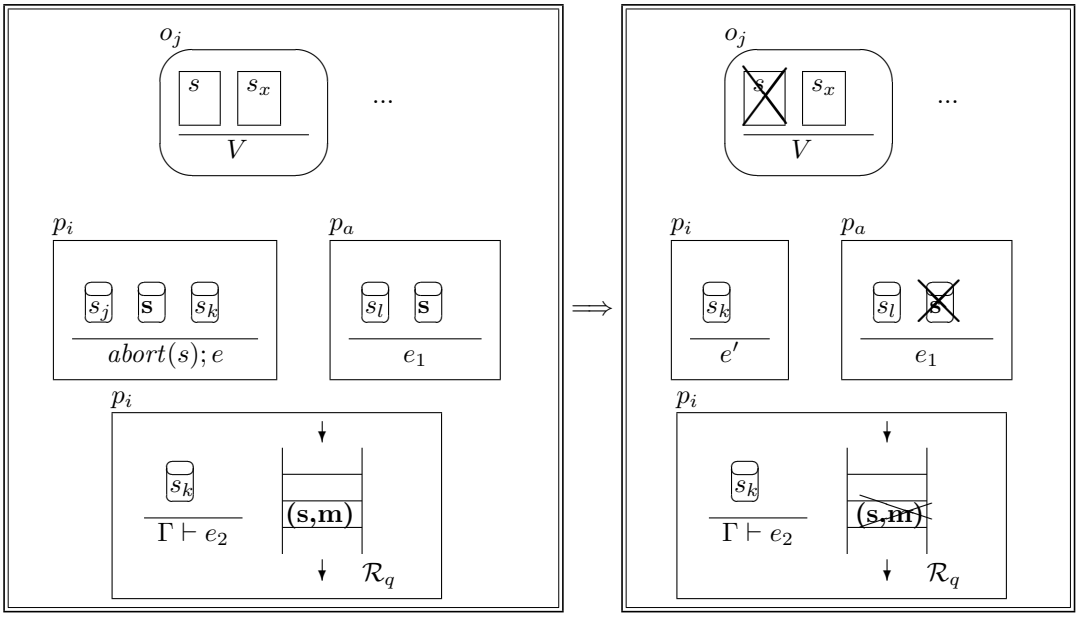
Rule ABORT-SPECULATION below shows how this behavior maps to the system state. Consider a process p_i that aborts one of the speculations it started: s, in this case. The state transition

shows the effects of the rollback on the process's state. The information stored in the checkpoint depending on speculation s is used (e') and all the other checkpoints up to that one are discarded.

If another process, p_a, has a checkpoint that depends on speculation s, then its checkpoint will be marked as part of the invalidation step. The same is true for objects (see object o_j). Furthermore, messages that were sent as part of the speculation are also invalidated during this stage. If a process's receive queue contains a message labeled with speculation s that has not been yet processed, the message is discarded from the receive queue. This prevents the speculation to absorb other processes after it has been aborted.

Assuming that our initial process p_i was also the owner of speculation s_j, whose checkpoint is discarded as part of the rollback, then the same invalidation step will occur system wide for checkpoints that depend on that speculation as well.

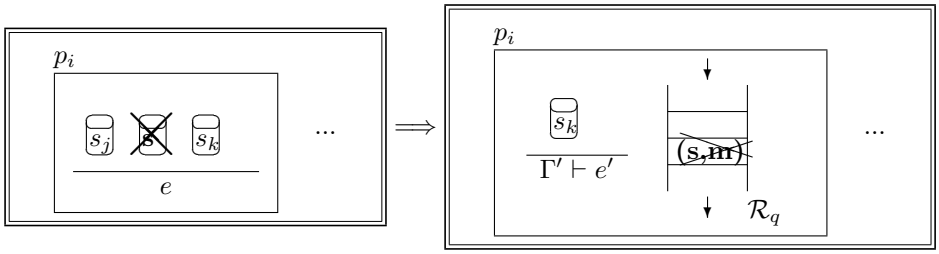
Abort-Speculation



All processes that depend on a speculation that has been aborted must be rolled back eventually to the point before they were absorbed in the speculation. If a process has a checkpoint that has been invalidated, then it does belong to a speculation that has been aborted. In this case it needs to roll back to the point where it became part of that speculation. It can continue execution after it restores the state saved in the checkpoint. When state is restored all messages from the restored receive queue that depend on the aborted speculation should be removed. The rule ABORTED-PROCESS-CHECKPOINT presents this behavior.

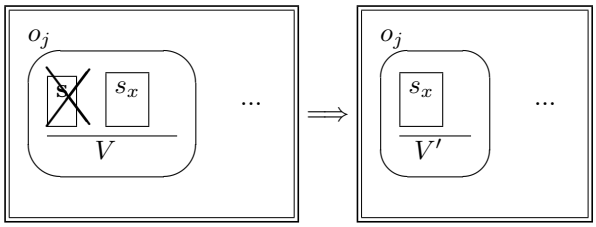
Again, as in the case described in rule ABORT-SPECULATION, if the process has started a speculation since it has been absorbed in the speculation that is now aborted it needs to abort its own speculations. This involves the invalidation step described above.

Aborted-Process-Checkpoint



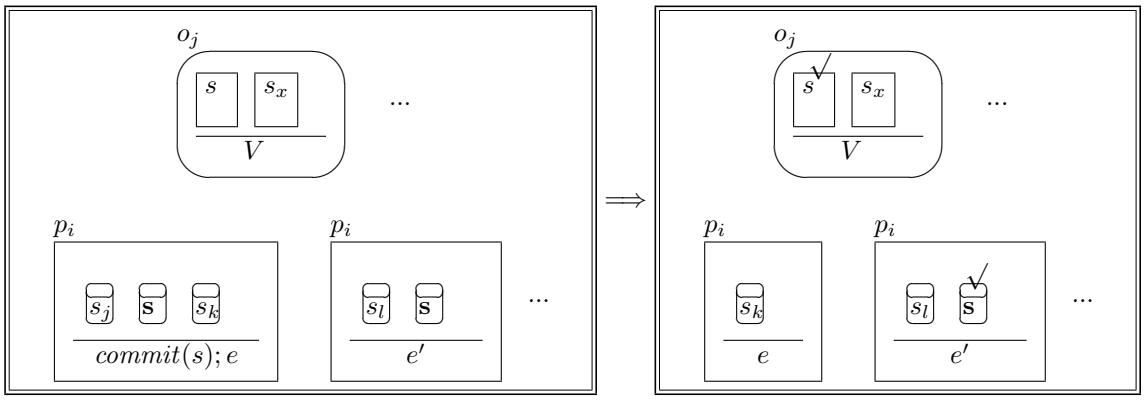
Objects that depend on speculations that have been aborted have to roll back their state to what it was before they became part of the speculation. This is shown in rule Aborted-Object-Checkpoint.

Aborted-Object-Checkpoint



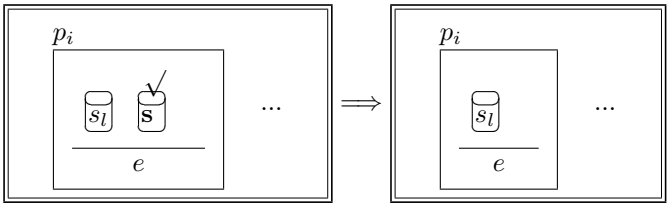
Committing a speculation. Finally, the next three rules describe the required actions when speculations are committed. A speculation can only be committed by the process that started it. If the owner of a speculation commits it, the owner discards the checkpoint that depends on that speculation and it notifies other processes and objects that were absorbed in the speculation that they can discard their checkpoint associated with the speculation as well. This is illustrated by the next rule. We use a check mark to flag that a checkpoint depends on a committed speculation.

Commit-Speculation



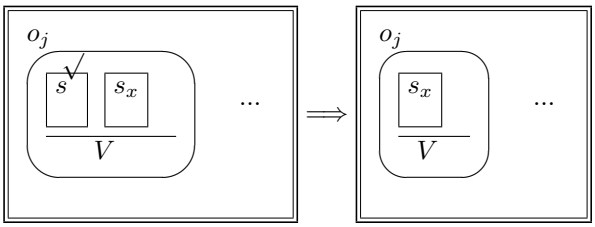
If a speculation has been committed by its owner, and a running process has a checkpoint that depends on that speculation, then it discards the checkpoint and it continues its execution.

Committed-Process-Checkpoint



The same is true for objects that have a checkpoint that depends on a committed speculation.

Committed-Object-Checkpoint



This concludes our semi-formal representation of speculative execution. This representation is used as a guideline for the formal operational semantics presented in Chapter 4 and for the implementation discussed in Chapter 5.

3.2.2 Sample informal proof of a safety property

To illustrate the benefits of specifying a formal model of the speculative behavior we present an informal proof of the following property: "After a speculation is fully aborted the system is in a consistent state in which there are no messages whose corresponding send operation was rolled back."

The informal proof by contradiction is the following. Assume there is a message m that depends on speculation s, whose corresponding send was rolled back when speculation s was aborted. The message could be in the receive queue of a process. However, rules Abort-Speculation and Aborted-Process-Checkpoint show the message would have been invalidated and removed from the receive queue of a process during the abort of the speculation, so we have a contradiction.

Assume that a process processed the speculative message but its computation was not rolled back. If so, its execution followed the Receive-Speculative-Message rule, in which case it should have saved a checkpoint that contains its state before processing the message. When speculation s was aborted the invalidation of the checkpoint should have occured (see rule Abort-Speculation). According to rule Aborted-Process-Checkpoint, the process rolled back to the state saved in the checkpoint, and it removed the message m that absorbed it in the speculation from its receive queue. Therefore, the computation was rolled back and could no longer depend on the value of message m. This contradicts our initial assumption and concludes our proof.

Chapter 4

Formal Speculative Models

Implementing distributed systems, where the interaction between distributed processes can be very complex, is prone to errors that can be hard to debug and reproduce. Providing an operational semantics, on the other hand, can make it easier and it may increase the confidence in the implementation, as long as the semantics of each of the possible actions that influence the state of the system is clearly specified.

Defining a formal specification of speculative execution is important because

- it provides a clear semantics of the speculative model,
- it allows us to reason about the correctness of the speculative programs, and
- it increases the confidence in the implementation if it closely follows the formal model.

We created a model that encapsulates the state of the distributed system in a way that makes the specification of speculative actions easy. The operational semantics that we describe has the advantages that:

- it can easily be mapped to a real implementation;
- it can be used in either a model checker or a theorem prover to further reason about the correctness of distributed speculative programs; and
- it is easy to use in a paper proof to show the equivalence between the speculative model we propose and a traditional non-speculative execution model similar to non-deterministic Turing machines. [57]

In this chapter we present three formal models for speculative execution in the form of operational semantics. The first one presents speculative message passing when there is at most one active speculation per process. The second model describes a speculative distributed objects system under the same assumption of having at most one active speculation per process or per object. The third model is a refinement of the speculative distributed objects system and discusses nested speculations.

Each of the three models is presented with a complete description of the syntax of the speculative primitives and of the terminology used to describe the rules. The models are different from each other in significant ways, either in terms of the syntax or in terms of the representation of the rules, to warrant a full description of each rather than a comparative discussion.

The decision to design and formalize several models for speculative execution has been prompted by the fact that certain applications are better suited for a model that uses message passing as communication primitives, other applications better fit the distributed shared objects paradigm, while others may need to use both. The examples presented in Section 1.3 demonstrate the need of these various models.

The chapter concludes with a proof of equivalence between the model for the speculative distributed objects system and a nonspeculative, nondeterministic model. This equivalence proof enables the use of existing tools, like model checkers and theorem provers to reason about speculative programs, rather than building new such tools that understand the speculative constructs that we introduce.

4.1 Speculative Message Passing Model

This section presents the semantics of our speculative primitives in the context of a distributed environment. The semantics is presented as a set of operational rules that capture the transition of the distributed system state when speculative actions occur in the system. For clarity, we use diagrams to represent the semantics, rather than a purely algebraic specification.

In this model, a process can be executing within at most one speculation at any given time. In other words, a process cannot start a speculation while it executes inside another speculation.

Our model considers a set of processes running concurrently in a distributed environment. The processes communicate with each other by exchanging messages.

4.1.1 Syntax

The syntax of the primitives is shown in Figure 4.1. We assume that the operational syntax extends over any language that incorporates the speculative primitives. The base language ($\mathcal{L}$) can be any language that does not involve message passing. We extend it with speculative constructs and with two calls for sending and receiving messages.

The *speculate*$(e_1 \oplus e_2)$ primitive defines a speculation. The program represented by e_1, called the "commit" branch, is executed while the assumption that the speculation is based upon is assumed to be valid. If the assumption is invalidated, then the program executes the *abort*() call, the speculation is aborted and the process rolls back and executes e_2, the "abort" branch. If the assumption is

		Construct	Description
e	::=	$\mathcal{L}$	The base language
		$speculate(e_1 \oplus e_2)$	Speculate call
		$commit()$	Commit call
		$abort()$	Abort call
		$sendto(p_i, m)$	Send a message to process p_i
		$let\ v = recv()\ in\ e$	Receive a message from the receive queue.
		$e\ ;\ e$	Sequencing

Figure 4.1: Syntactically valid terms

MsgPass-Sample-Rule

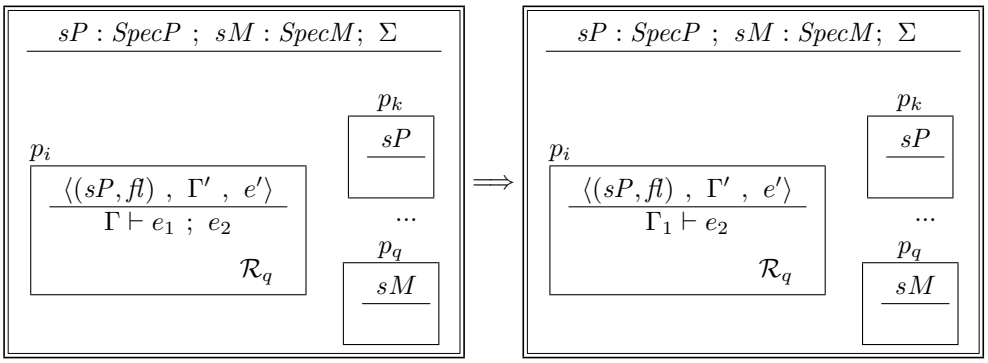


Figure 4.2: Notation used in the operational semantics

validated (the *commit*() call is executed) while the process executes inside e_1, the process continues its execution and it will never roll back to take the e_2 branch.

The *sendto* and *recv* primitives define point-to-point message passing. The *sendto* call takes as parameters the destination process and the message to be delivered. The *let* $v = recv()$ *in* e construct returns the first message in the receive queue and assigns the data to variable v. This variable can further be used in the rest of the program specified by e. For simplicity, we consider one receive queue for each process that queues all the messages sent by various senders. It is straight-forward to extend the model with multiple per-process receive queues.

4.1.2 Terminology and Notation

The model requires capturing the state of the distributed environment in order to represent the speculative actions as state transitions. The transition triggered by the speculative action of one process is reflected upon the entire distributed state of the system. For this purpose, we define the state of the distributed system as having two components: a speculations environment (Σ) and a set

of the individual states of all the processes in the system. Figure 4.2 shows the notation we use in specifying our operational semantics in the form of a sample operational semantics rule. The details are described below.

The state of a process p_i, graphically represented in Figure 4.2 by a square box, has four components:

- a checkpoint (the upper half of the box),
- the local environment (or state) of the process (Γ),
- the program that the process executes ($e_1; e_2$), and
- a receive queue ($\mathcal{R}_q$).

The checkpoint, graphically represented in Figure 4.2 as the part above the horizontal line in the process state, is created when the process becomes speculative and itself has three components:

- the information identifying the speculation (s, fl),
- the local environment (state) of the process when it entered the speculation (Γ'), and
- the program to be executed if the speculation is rolled back (e').

A speculation identifier (s, fl) has two components. The first is a system-wide unique name (s) assigned when the speculation is created. The second one is a flag (fl) whose values can be any of: *own*, *peer*, or *client*, to be explained later.

The final component of a process state is the optional receive queue ($\mathcal{R}_q$). All messages received by a process are enqueued in a single receive queue. Messages are tagged with an optional speculation id (in case the sender is involved in a speculation). The receive queue is accessed through two operations: *enqueue* and *dequeue*. The *enqueue* operation modifies the receive queue from $\mathcal{R}_q$ to $\mathcal{R}_q :: (s, m_{last})$. The *dequeue* operation is observed as a state change of the receive queue from $(s, m_{first}) :: \mathcal{R}_{q-rest}$ to $\mathcal{R}_{q-rest}$.

The speculation environment (Σ), represented graphically at the top of the system-wide state, contains information about the active speculations in the system. Each speculation has an entry of the form $s : \mathcal{S}$, where s is the name of the speculation and the set $\mathcal{S}$ contains the co-owners of the speculation. These are the processes that can explicitly abort or commit the speculation.

In the sample rule (Figure 4.2) we show the evolution of the system-wide state when process p_i executes one instruction (represented by e_1). Its local environment may change as a consequence of e_1 (thus using Γ_1 in the new state of the system). One process's execution might affect the state of other nodes in the system as well. The sample rule also show that other processes in the system could be speculative and that they could belong to either the same speculation as p_i (sP) or to other speculations, as is the case with p_q.

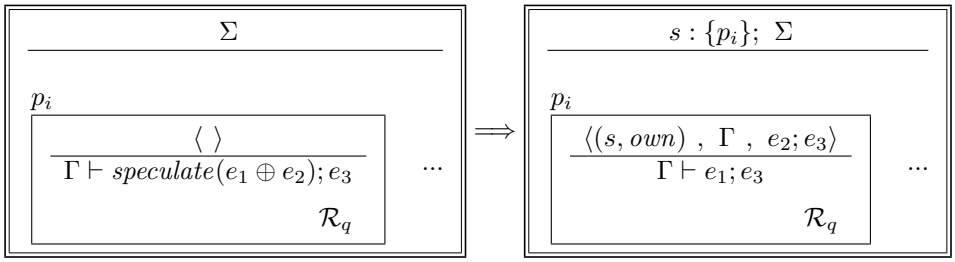
4.1.3 Operational Semantics Rules

In this section we present a subset of the operational semantics rules that define our model. For each of the rules we present the intuition behind it, followed by the graphical representation of the state change.

Process execution may be arbitrarily interleaved. In the diagrams, the identifier p_i represents an arbitrary process.

Starting a speculation. We begin by presenting the operational semantics rule for the *speculate* primitive. When a process starts a new speculation it creates a local checkpoint, which is used in case of a rollback. The checkpoint contains a new speculation identifier (s), and the flag *own* to mark that it owns the speculation. It also contains the local environment of the process as it was just before calling the *speculate* primitive and the program it needs to execute in case of a rollback. The program is composed of the *abort* branch (e_2) of the speculation and the rest of the program (e_3). Graphically, this is expressed by the MsgPass-Spec rule below. The process continues to execute inside the commit branch, as shown.

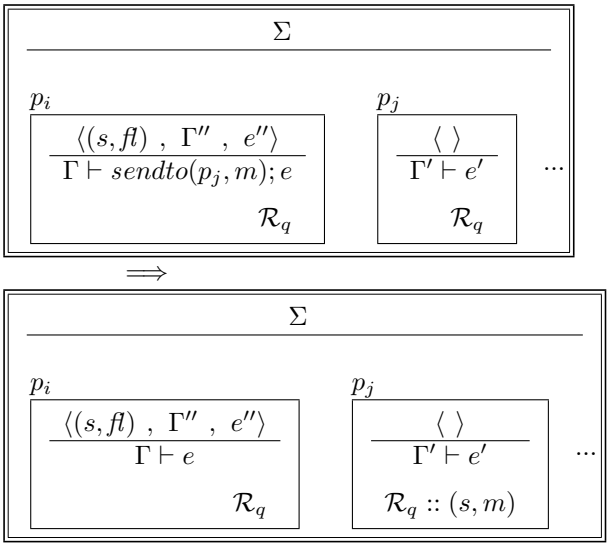
MsgPass-Spec



Speculative message passing. There are three interesting cases involving message passing and speculative processes. The main assumptions that we make in our model with respect to messages and message passing are as follows. We ignore network latencies and assume that when a process sends a message it instantaneously appears in the receive queue of its destination. This assumption does not affect the correctness of our model, as its sole purpose is to describe speculative behavior and not model the network. Speculative messages are tagged with the id of the speculation the sender belongs to when the send operation occurs.

The MsgPass-SendTo-Spec rule shows the effects of the *sendto* operation when the sender is speculative. The state of the entire system is modified in a straight-forward manner. The sender of the message continues its execution with the next statement in its program, and the message is appended to the receive queue of the destination process. As long as the message is not processed by the receiving process nothing else changes.

MsgPass-SendTo-Spec



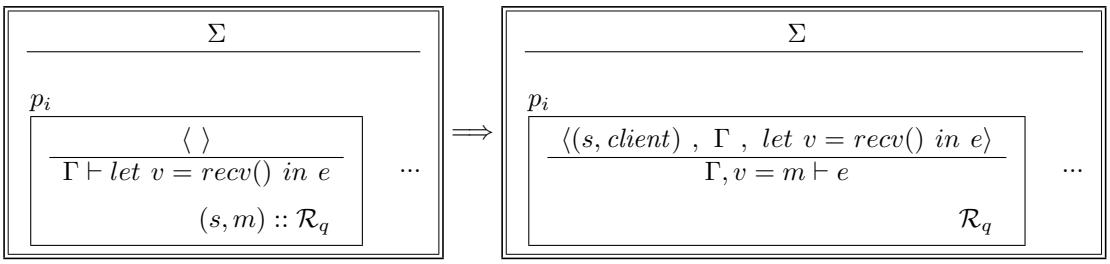
On the receive side we distinguish two cases that involve speculative messages. The first one assumes that the following conditions are met:

- The receiving process is non-speculative.
- The receiving process executes a $let\ v = recv()\ in\ e$ statement.
- The first message in the receiver's receive queue is speculative.

In this case the receiver of the message is absorbed in the speculation that the sender of the message belongs to.

The receiving process creates a checkpoint, as in the case described by the MsgPass-Spec rule above, with the following differences, also shown in the rule below.

MsgPass-Recv-Spec



The checkpoint is tagged with the speculation id of the message (s) and the flag is set to *client*. Furthermore, the abort branch of the speculation is the same as the commit branch, since the speculation is implicit and the process can't specify an alternate execution path. The *client* flag prevents the process to decide the outcome of the speculation through explicit *abort*() or *commit*() statements. This is assured by not having any specific rule that involves *abort*() or *commit*() statements

executed by processes that do not own the speculation. This is the desired behavior since processes that are implicitly absorbed in speculations due to speculative messages should not be aware of the speculation.

The second case, and probably the most interesting, is found when a speculative process receives a speculative message that belongs to a different speculation. Since the current model supports only one active speculation per process at any given time we chose to merge the two speculations and to make the owners of each individual speculation co-owners of the speculation.

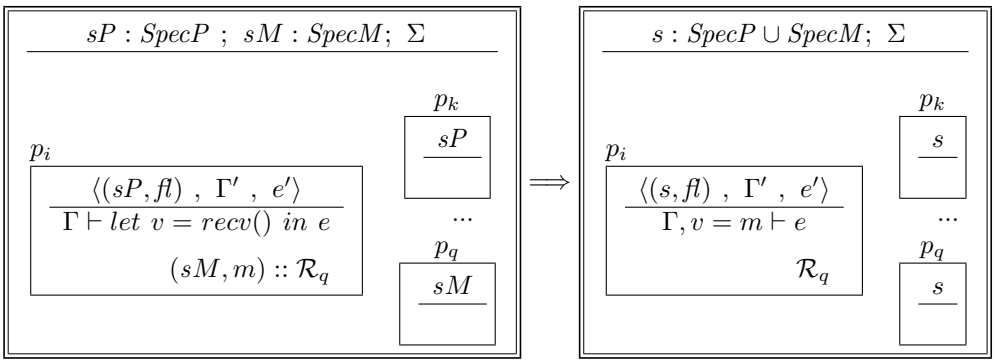
The MsgPass-Recv-Spec-Merge rule below shows the actions involved in a speculation merger. The description of the initial state of the system ensures that the transition only occurs if :

- the two speculations are still active, and
- the receiver is currently executing inside a speculation that is different from that of the message.

When the merger occurs, the list of owners of the two initial speculations are merged and all processes that belonged to either of the two speculations are announced of the merger. The owners of the initial two speculations share ownership of the one emerging from the merger. All co-owners of a speculation are allowed to *commit* or *abort* it, as they would the one they initially created. Formally, this is expressed through the union of the two ownership lists and through the substitution operation presented in the rule. The substitution operation of the speculation identifiers from sP and sM to s could be implemented using a publish/subscribe mechanism as follows.

- When a process starts a new speculation it creates a new publish/subscribe channel.
- When a process becomes part of a speculation (implicitly) it is automatically subscribed to the channel corresponding to that speculation.
- All the important control messages (like the merger) are published by the owners of the speculation to the appropriate publish/subscribe channel.

MsgPass-Recv-Spec-Merge

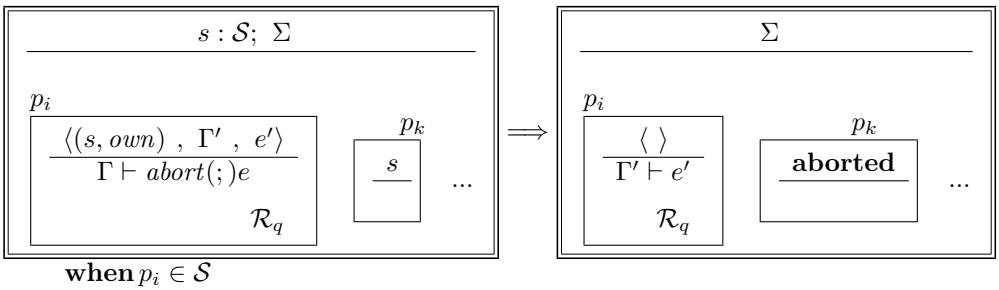


The above rule describes the behavior of the system for this special case and leads to an efficient implementation, as we will see in Section 5.2.

Aborting a speculation. The next two rules describe the behavior of the system when speculations are aborted. The first one (MsgPass-Ab-Owner), describes the transition of the system's state when the owner (or co-owner) of a speculation aborts it. In this case, the process rolls back to the point where the speculation was started and continues execution on the abort branch. Since there might be other processes in the system that depend on this speculation there is also a substitution involved, which updates the information system-wide. This ensures that processes absorbed in this speculation will roll back their states as well, as per the MsgPass-Ab-Proc rule below.

Furthermore, messages that were sent as part of the speculation are also invalidated during this stage. If a process's receive queue contains a message labeled with speculation s that has not been processed, the message is discarded from the receive queue. This prevents the speculation to absorb other processes after it has been aborted.

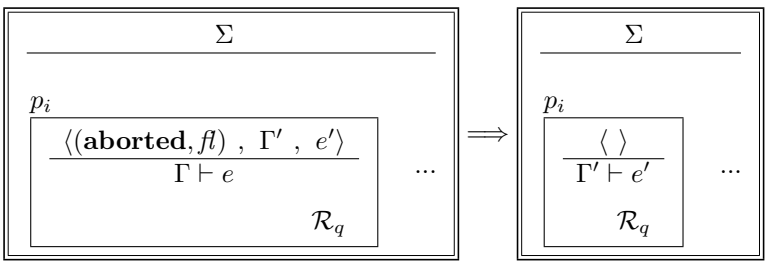
MsgPass-Ab-Owner



when $p_i \in \mathcal{S}$

If a process belongs to a speculation that has been aborted it needs to roll back to the point where it became part of the speculation. It can continue execution after it restores the information saved in the checkpoint. This is presented below.

MsgPass-Ab-Proc

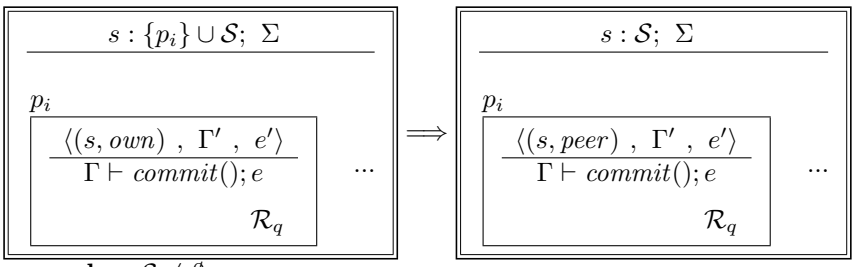


Committing a speculation. Finally, the next three rules describe the required actions when speculations are committed. A speculation can only be committed if all its co-owners commit it.

If at least one co-owner aborts the speculation, then the speculation is aborted, as described in rule MsgPass-Ab-Owner. Note that a speculation can have several co-owners, as a result of one or more mergers (as described by rule MsgPass-Recv-Spec-Merge).

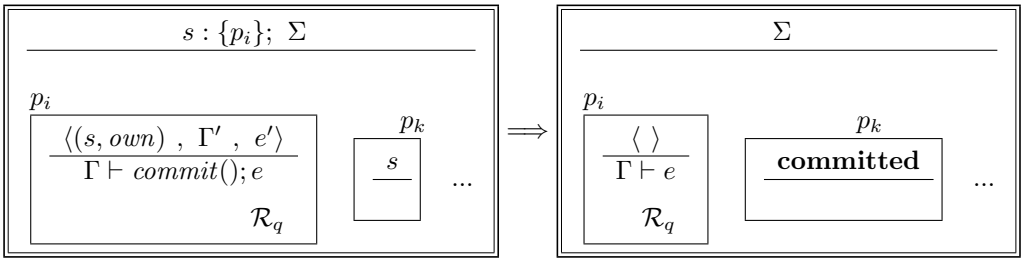
If a co-owner of a speculations commits it while its peers are still executing inside the speculation then it is blocked until every one of its peers executes a commit call or one of them executes an abort call. The reduction rule below illustrates this behavior. The flag associated with the speculation changes from *own* to *peer* so that there are no matching rules except for MsgPass-Comm-Proc or MsgPass-Ab-Proc that the process can further execute.

MsgPass-Comm-Peer



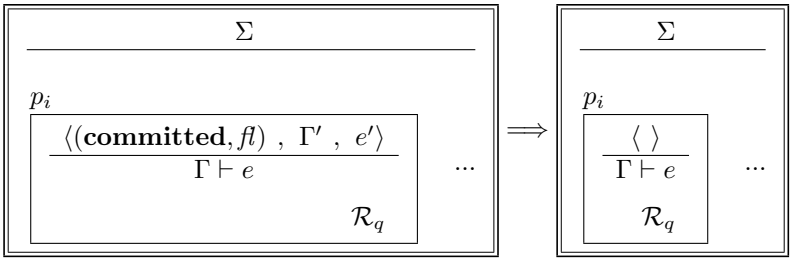
If a process is the only owner of a speculation, or if it is the last co-owner to commit it, then it substitutes the id of the speculation with the **committed** special constant. The checkpoint is discarded, and the speculation is erased from the speculations environment.

MsgPass-Comm-Owner



The following rule (MsgPass-Comm-Proc) applies to the co-owner of a fully committed speculation or to a process that was absorbed in a speculation. If the speculation has been fully committed by its co-owners the process can only continue its execution as described by the following rule, which discards the saved checkpoint and continues the execution of the process outside any speculation.

MsgPass-Comm-Proc



This concludes the presentation of the operational semantics of the speculative message passing model. The operational rules presented in this section are reflected in the implementation of the speculative message passing interface and are referenced in the description of the implementation presented in Chapter 5.

4.2 Model for a Speculative Distributed Objects System

We present a speculative distributed objects system model consisting of processes and shared objects. Shared objects store values that can be accessed (read or written) by any process in the system. The objects may be accessed by any process. They will be the vehicles to propagate speculations in the system.

Processes execute programs and start speculations by executing the *speculate* call. After a speculation is started, we say that the speculation is *active* until a *commit* or an *abort* call is executed. We say that a process is *executing inside* a speculation if the process's program is executed as part of a speculative computation. An object becomes part of a speculation, or it is involved in a speculation, if a process that is inside a speculation accesses the object. An object becomes *absorbed* in a speculation if it reads data from a speculative objects. The *merger* of two speculations is defined as the operation by which two speculations initiated by different processes, change their speculation identifier to a shared, common one. From that point on the new identifier is used to refer to either of the two initial speculations.

A process can be executing within at most one speculation at any given time. In other words, a process cannot start a speculation while it executes inside another speculation. A process is allowed to access multiple objects and an object can be accessed by multiple processes.

We say that a process or an object *belongs* to a speculation if it started that speculation or if it was absorbed in the speculation. A shared object belongs to at most one speculation at any given time.

4.2.1 Overview of the Language

The terms of the language are defined in Figure 4.4. The base language ($\mathcal{L}$) can be any language that does not have a read/write interface for shared objects. We extend it with speculative constructs and with a special call for accessing shared objects.

		Construct	Description
e	::=	$\mathcal{L}$	The base language
		$speculate(e_1 \oplus e_2)$	Speculate call
		$commit()$	Commit call
		$abort()$	Abort call
		$let\ v = read(o_j)\ in\ e[v]$	Read the value of object o_j
		$write(o_j, x)$	Write value x to object o_j
		$e\ ;\ e$	Sequencing

Figure 4.3: Syntactically valid terms

The speculative construct $speculate(e_1 \oplus e_2)$ defines a speculation. In the speculative mode, the program executes e_1. If it executes an $abort()$, the speculation is aborted and the process rolls back and executes e_2. If a $commit()$ is encountered in e_1, the process will never roll back its state to take the e_2 branch. We refer to e_1 as the "commit" branch, and to e_2 as the "abort" branch.

The $let\ v = read(o_j)\ in\ e[v]$ construct assigns the value of shared object o_j to variable v, which is bound in e. The write operation is represented by $write(o_j, x)$. It stores the value x in shared object o_j.

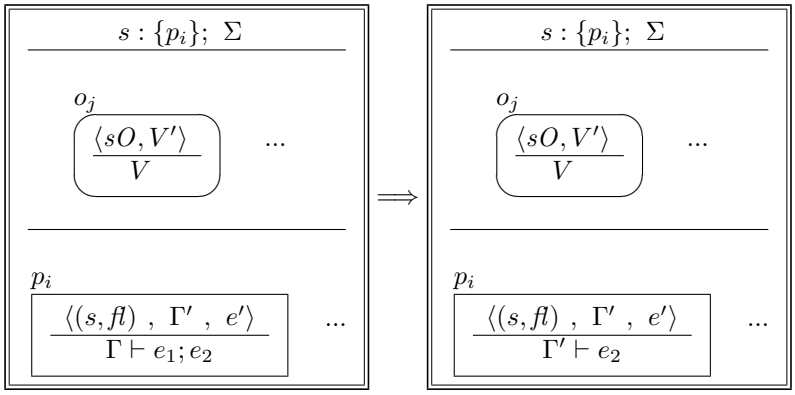
The syntactically valid terms presented above can be sequenced using the *";"* separator.

4.2.2 Terminology and Notation

The notation used in this section is illustrated through a set of sample rules similar to those used in the operational semantics.

The operational semantics defines a reduction system that operates on states. If a distributed system in state Δ_i reduces in one step to state Δ_i' we write $\Delta_i \Longrightarrow \Delta_i'$. The meaning of $\Delta_i \Longrightarrow \Delta_i'$ is that the state of one, and only one, process reduces as part of the reduction step. Formally, this is written as follows.

Simple-Sample-Rule



The state of a distributed system has three components.

- The speculations environment (Σ) is a set definitions of the form $s : \mathcal{S}$ where $\mathcal{S}$ is a set of process ids (p_i) of processes co-owning speculation s.
- The state of shared objects in the system (Θ).
- The state of processes in the system (Π).

The state (P_i) of a speculative process (p_i) is defined by three components:

- an optional checkpoint (c) (present in the top half of the box),
- a local environment (Γ), and
- the instructions of the program it executes ($e_1; e_2$).

The checkpoint of a process has three components.

- The unique id of the speculation (s) that generated it, along with a flag (fl) that evaluates to *own* if the process is the "owner" of the speculation, to *peer* if the process is a co-owner of the speculation and has committed it (see Rule Comm-Peer below) and to *client* if the process was *absorbed* in the speculation due to a read/write operation.
- The local environment of the process at the time the process became part of the speculation.
- The program to be executed in case of rollback, the "abort" branch of the speculation.

A process is the owner of a speculation if it started that speculation.

The state (O_j) of a shared object (o_j) is characterized by the value it stores (V) and by an optional checkpoint, which stores the speculation id (s) along with the value (V') the object had before entering the speculation. The checkpoint, presented in the upper half of the object's state box of our graphical representation, is empty if the object is not part of any speculation.

When a new speculation is started it gets added to the speculations environment. When two speculations merge, they are erased from Σ and the speculation representing the merger is added. When a speculation aborts or commits it is erased from Σ. The state of a distributed system (Δ) is well formed if all the speculations that appear in Π and Θ have definitions in Σ. We only consider well formed states of the distributed system.

Additional notation used by the operational semantics is shown in Table 4.1.

Π	$::=$	$p_1 : P_1 \ldots p_n : P_n$	Set of states for processes $p_1 \ldots p_m$
Θ	$::=$	$o_1 : O_1 \ ; \ \ldots \ ; \ o_m : O_m$	Set of states for objects $o_1 \ldots o_m$
$s : \mathcal{S}$	$::=$	$s : \{p_{l_1} \ldots p_{l_q}\}$	List of peer processes co-owning speculation s
Σ	$::=$	$s_0 : \mathcal{S}_0; \ldots ; s_k : \mathcal{S}_k$	Speculations environment

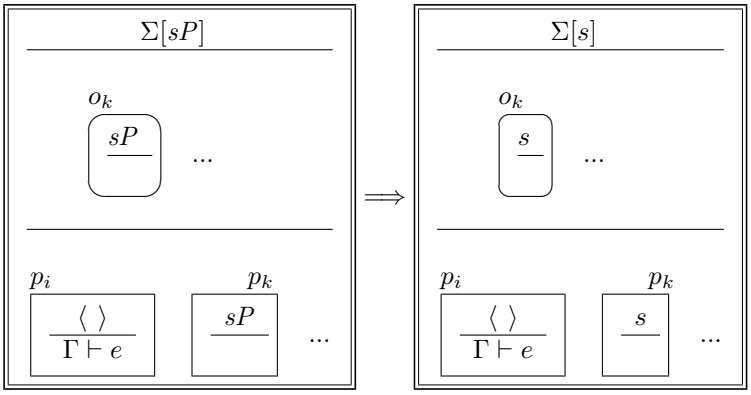
Table 4.1: Notation for speculative processes

The next sample rule illustrates one of the key operations performed in our rules: the substitution of speculation ids. The substitution operation involves changes throughout the state of the distributed system, as shown below. If the outcome of a speculation is decided or if the speculation changes its identifier all objects and processes that have checkpoints depending on it have to be notified. For example, if speculation sP is aborted, committed or replaced with the new identifier s (as a consequence of a merger) then the new state of the system reflects the change by substituting all occurrences of sP with either **aborted**, **committed**, or s as needed.

The notation $\Sigma[sP]$ is used if the speculation id sP occurs in the speculation environment. The substitution of sP with s is represented by $\Sigma[s]$. For simplicity, the reduction rules use the notation for bound speculation ids only in conjunction with the substitution operation. In all the other cases where substitution does not occur the above notation is omitted.

The substitution is also reflected in the states of processes and objects, as shown in the sample rule Substitution-Sample-Rule below.

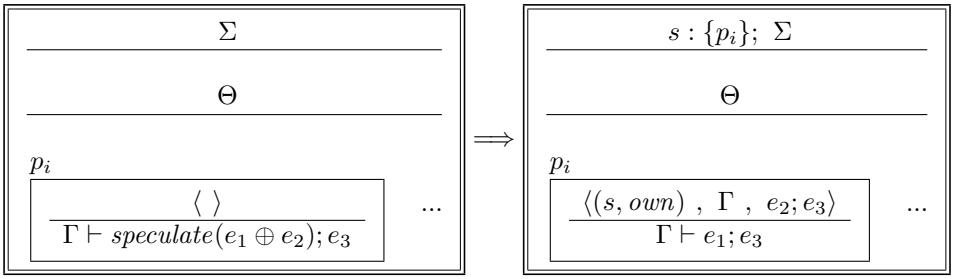
Substitution-Sample-Rule



4.2.3 Speculate

A process outside any speculation successfully starts a new speculation by using the *speculate*$(e_1 \oplus e_2)$ construct. A new speculation is created and added to the speculations environment, and a checkpoint of the process is taken. The checkpoint becomes part of that process's state and the process advances with the execution to the next instruction in its program.

Spec



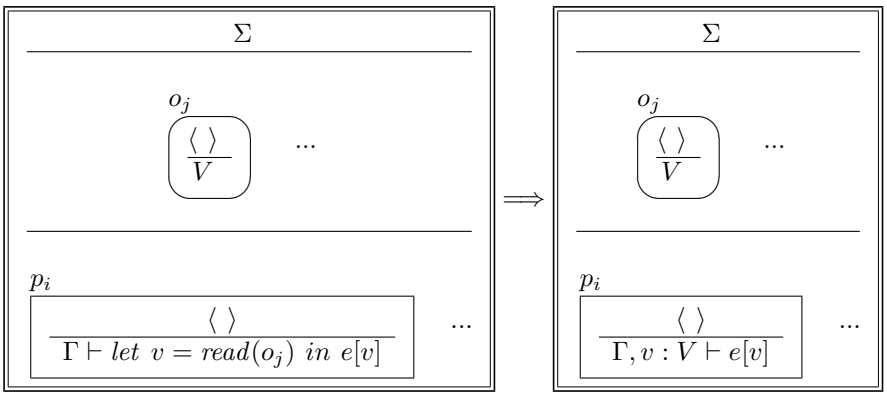
4.2.4 Reading from a Shared Object

The reduction rules for the read operation ($let\ v = read(o)\ in\ e[v]$) take into consideration whether the object and the process belong to a speculation or not. The local variable v is assigned the value read from the object and becomes part of the process's local environment.

4.2.4.1 Both the process and the object are outside speculations

Reading the value of a shared object when neither the object nor the process is part of any speculation is illustrated by the following reduction rule.

Read-NoSpec

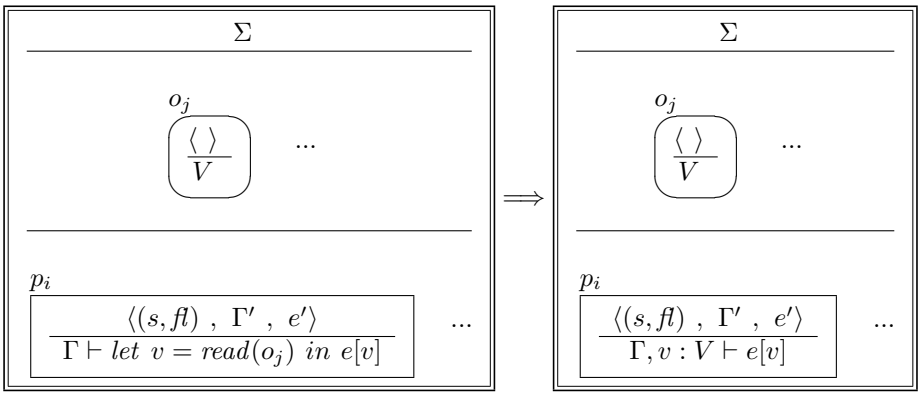


The process continues the execution with the next instruction in its program.

4.2.4.2 The process is inside a speculation and the object is outside any speculation

When a speculative process reads the value of a shared object that is not part of any speculation it does not absorb the object in its speculation. The behavior of the system is defined as such to prevent the needless propagation of speculations in the distributed system.

Read-Spec-Proc

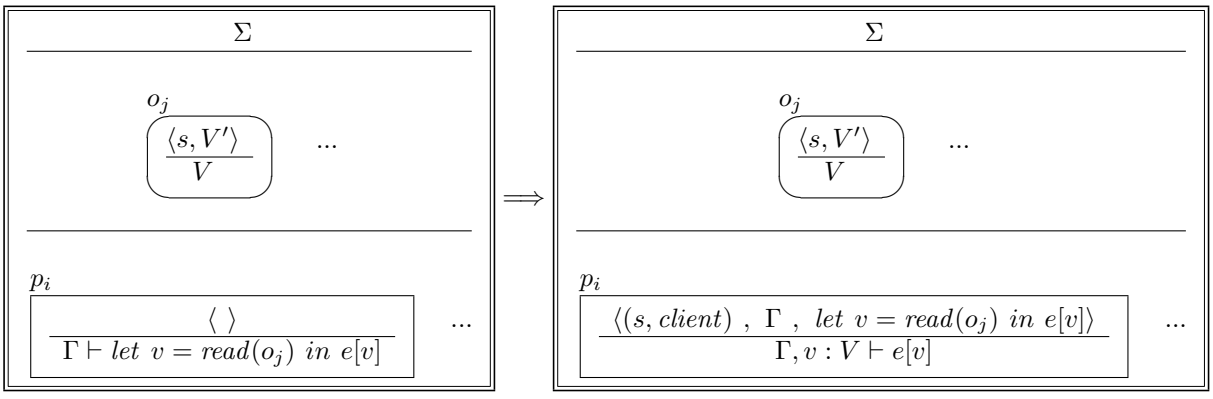


The process continues the execution with the next instruction in its program.

4.2.4.3 The process is outside any speculation and the object is inside a speculation

After executing the read operation the process's state depends on speculative information, so the process is absorbed in the object's speculation. A checkpoint of the process is created to allow rollback if the speculation is later aborted.

Read-Spec-Obj

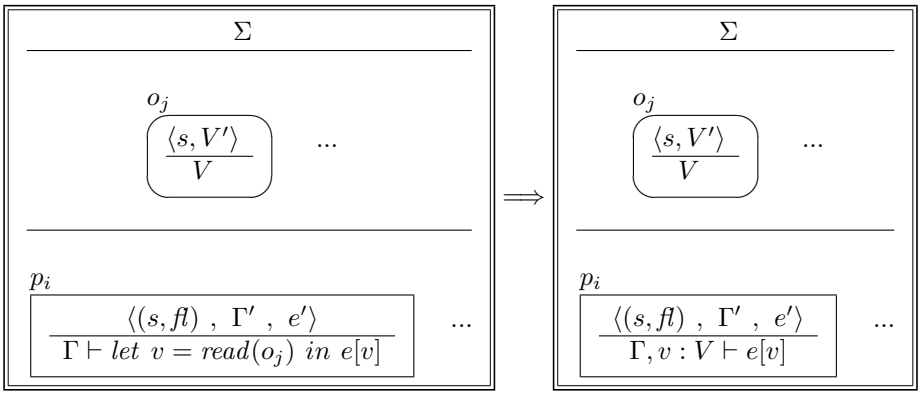


4.2.4.4 Both the process and the object are inside the same speculation

The reduction rule is similar to the case when neither the process nor the object were part of any speculation (Rule READ-NOSPEC). Only the internal environment of the process changes and the

program continues the execution with the next instruction.

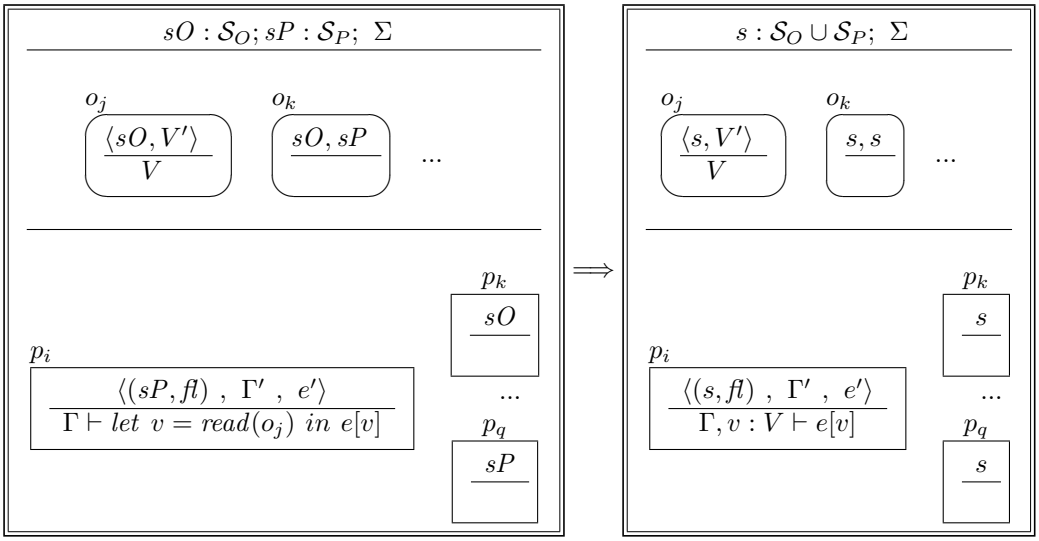
Read-Spec-Same



4.2.4.5 The process and the object are inside different speculations; speculations are merged

The most interesting case for reading the value of a shared object is when both the process and the object are speculative and they belong to different speculations (sP and sO, respectively). After the read operation is performed the state of the process depends on the speculative data stored in the shared object at the time of the access. Since we don't consider nested speculations in this model, the only way to guarantee the process is rolled back if the object's speculation is aborted is to merge the two speculations. The merger of the two speculations, sP and sO, is represented in the operational semantics rule by a substitution operation of sP and sO with s, which is the "new" speculation id created from the merger of the initial two speculations. The objects and processes that belong to either of the two speculations become aware of the merger. This operation guarantees that the outcome of the "new" speculation will be broadcast to all interested parties, as described by the *abort* and the *commit* rules below.

Read-Spec-Merge



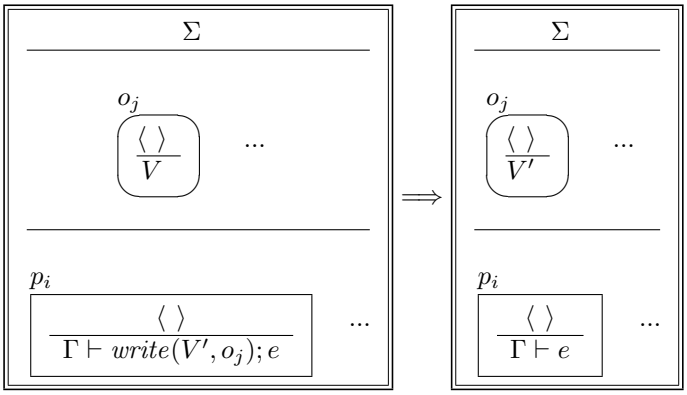
4.2.5 Writing Data to a Shared Object

The operational semantics defines different reduction rules to describe the action of writing data to shared objects, depending on whether processes and objects are inside or outside speculations. After a process executes the first instruction defined by the $write(V, o_j); e$ program the value of object o_j is updated with value V and execution of the program continues with e.

4.2.5.1 Both the process and the object are outside speculations

Writing a value to a shared object, when neither the object nor the process accessing it are part of any speculation is illustrated by the following reduction rule.

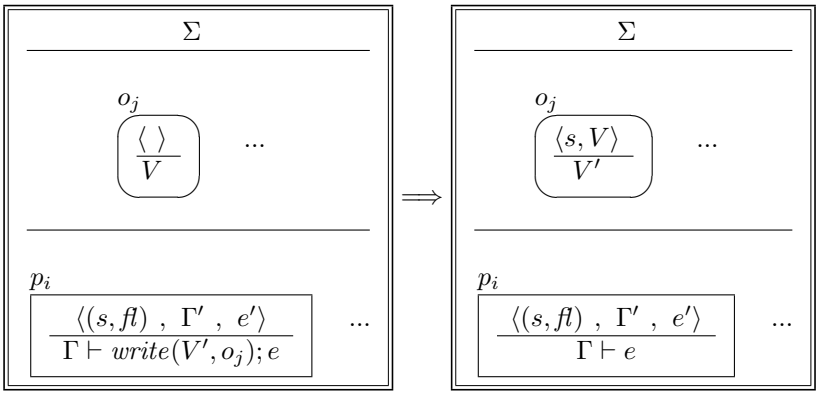
Write-NoSpec



4.2.5.2 The process is inside a speculation and the object is outside any speculation

When a process that speculates writes a value to a shared object that is not part of any speculation the object is absorbed in the process's speculation. This behavior is different from the one seen for the read operation because the value stored in the shared object is speculative and the object has to become speculative itself. The system creates a checkpoint for the object. The checkpoint stores the value the object had before becoming part of the speculation, which allows it to rollback if the speculation is aborted. The speculation id is also stored as part of the checkpoint. From this moment, the object will absorb processes that read its value into the same speculation.

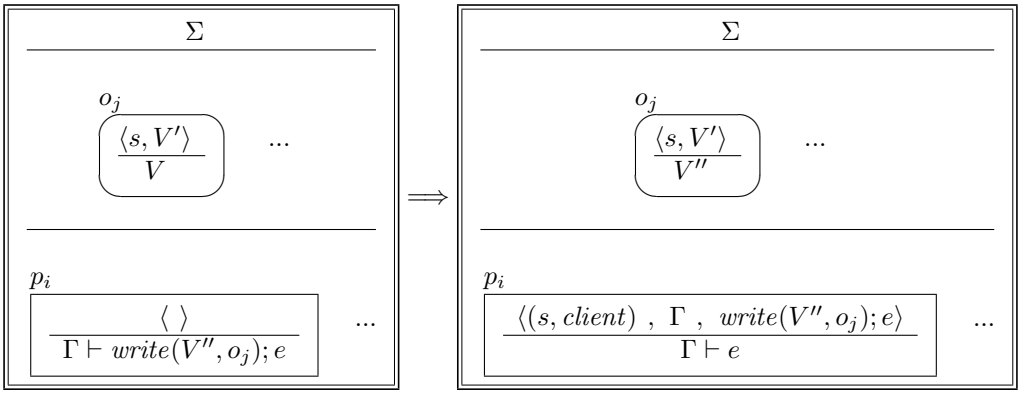
Write-Spec-Proc



4.2.5.3 The process is outside any speculation and the object is inside a speculation

The process writing to an object that is absorbed in a speculation will itself be absorbed in that same speculation. While it could be argued that this allows speculations to spread even though it might not always be needed, this behavior is desired for the following reasons. When a process that is not speculating successfully performs a write operation the object's value should only be changed by a subsequent write operation performed by one of the processes in the system. If the process were not absorbed in the object's speculation then a rollback of the speculation would change the value of the object, canceling the effects of the write operation. With the current rule in place the process rolls back along with the object and it is allowed to retry the write operation.

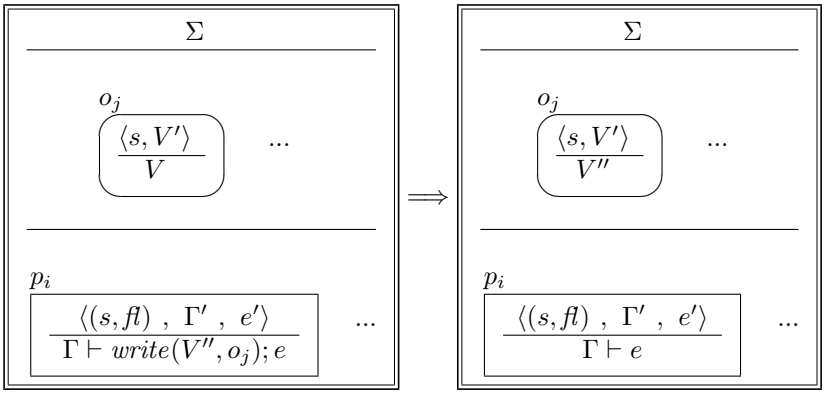
Write-Spec-Obj



4.2.5.4 Both the process and the object are inside the same speculation

The reduction rule is similar to the case where neither the process nor the object were part of any speculation (Rule WRITE-NOSPEC).

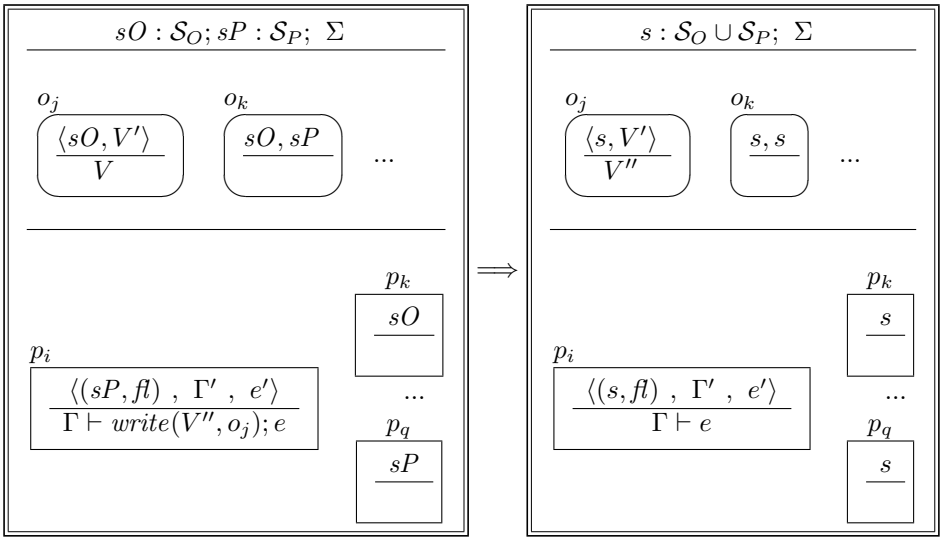
Write-Spec-Same



4.2.5.5 The process and the object are inside different speculations; speculations are merged

The most interesting case for writing an object is when the process and the object are inside different speculations. The two speculations, sP and sO, merge and are substituted in all the states of objects and processes composing the distributed system with with s, the "new" speculation id created from the merger of the initial two speculations.

Write-Spec-Merge



Again, as in the case of the read operation (READ-SPEC-MERGE) the merger is known to all the involved processes and objects.

4.2.6 Abort a Speculation

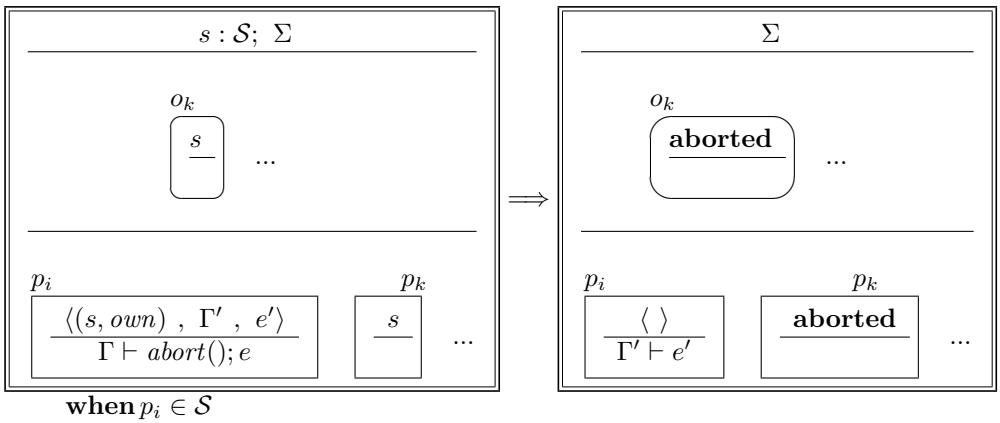
A speculation can be aborted by the initiating process when the assumption it was based on turns out to be false. Speculations also abort if their initiating process fails due to external factors.

The reduction rules presented in this section are constructed such that they guarantee that when a process or an object becomes aware that the speculation they belong to has been aborted they will roll back their state. Furthermore, they would not be allowed to continue execution inside the aborted speculation. This is ensured by the following. If a reduction rule contains a speculation identifier, like s, sP, or sO, it means that the identifier cannot be a constant, like **aborted**.

4.2.6.1 Processes and Aborted Speculations

A process that is inside a speculation that it owns is allowed to abort it. The reduction rule for the *abort*() call substitutes the id of the aborted speculation with the **aborted** special constant and rolls back the process to the state it was before entering the speculation. Also, the speculation id is erased from the speculations environment. The substitution operation guarantees that once a speculation has been aborted no other process or object can be absorbed in that speculation, since its id has been replaced by the special constant **aborted**. The state change is presented below.

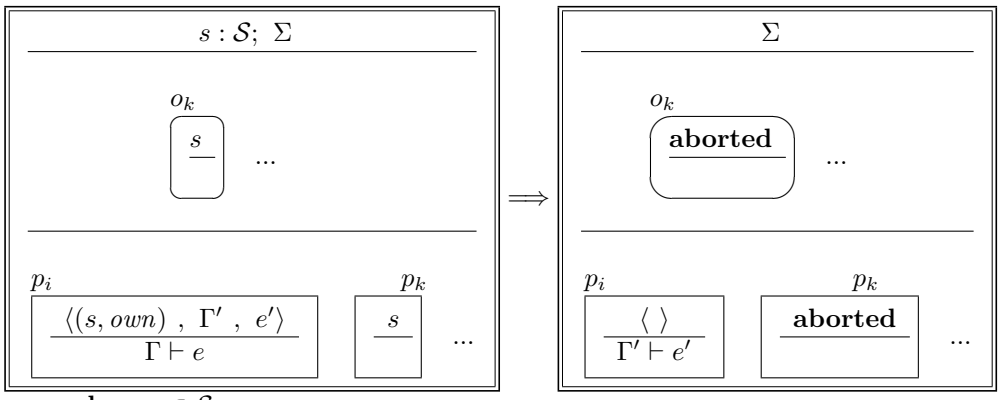
Ab-Owner



when $p_i \in \mathcal{S}$

Processes can also fail at any time during their execution due to external factors. When a process fails while it executes inside a speculation that it owns, the system aborts the speculation, and the process rolls back and executes the abort branch.

Ab-Fail

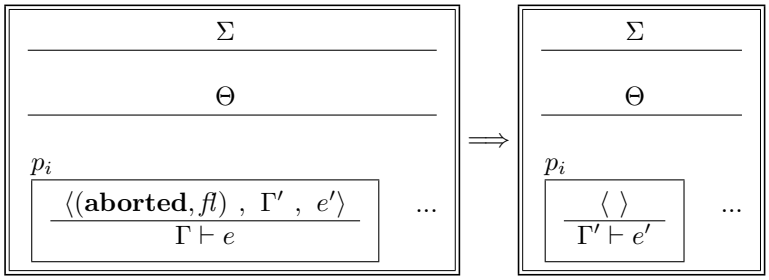


when $p_i \in \mathcal{S}$

If a process is inside a speculation that has been aborted by another process it rolls back and restarts the computation from the point where it was absorbed in the speculation. If the process is a co-owner of the speculation it rolls back to where it started its own original speculation.

This behavior is illustrated by the next operational semantics rule.

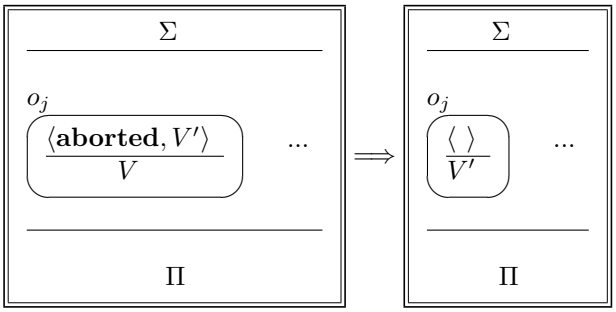
Ab-Proc



4.2.6.2 Objects and Aborted Speculations

If an object is inside an aborted speculation it rolls back its state to the state saved in the checkpoint.

Ab-Object

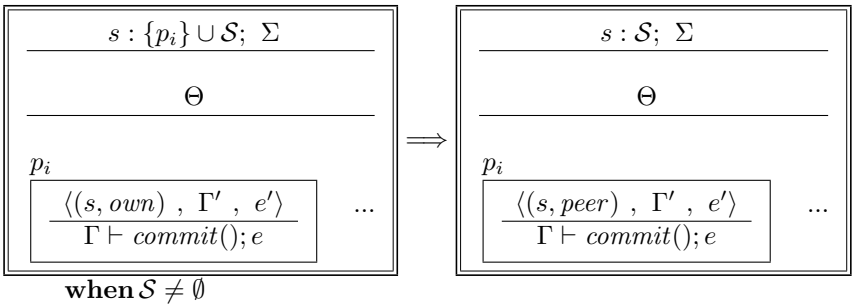


4.2.7 Commit a Speculation

4.2.7.1 Processes and Committed Speculations

Only processes that own (or co-own) a speculation can commit it. If a co-owner of a speculations commits it while its peers are still executing inside the speculation then it is blocked until every one of its peers commits the speculation or one of them executes an abort call. The reduction rule below illustrates this behavior. The flag associated with the speculation changes from *own* to *peer* so that there are no matching rules except for Comm-Proc, Ab-Proc, or Ab-Proc-Peer that the process can further execute to make progress.

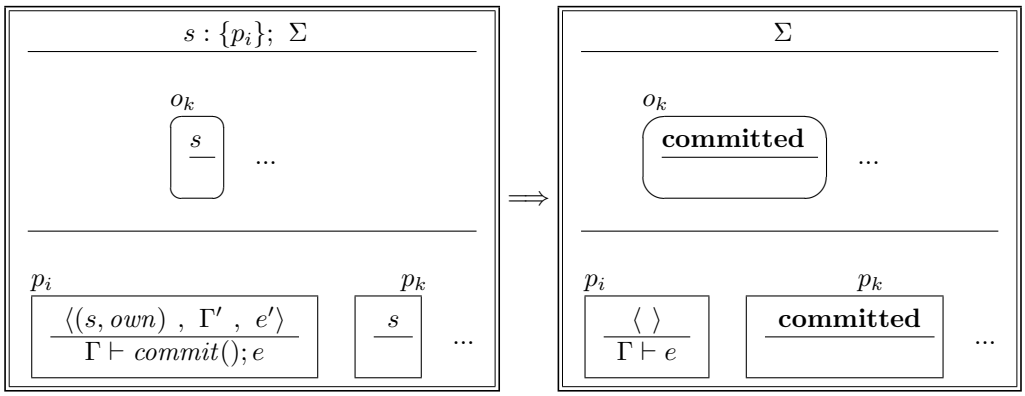
Comm-Peer



when $\mathcal{S} \neq \emptyset$

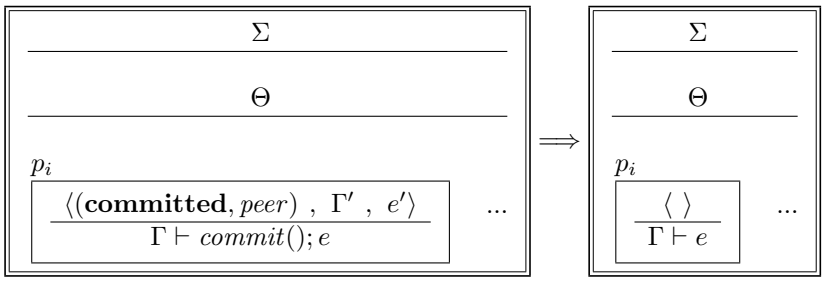
If a process is the only owner of a speculation, or if it is the last co-owner to commit it, then it substitutes the id of the speculation with the **committed** special constant. The checkpoint is discarded, and the speculation is erased from the speculations environment. The operational semantics rule showing the state change is the following.

Comm-Owner



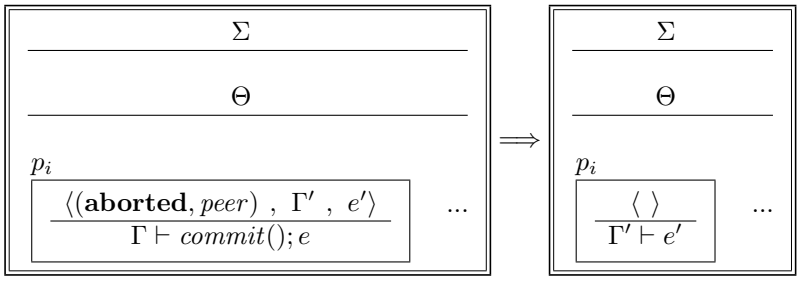
The co-owner of a speculation that has committed the speculation can make progress if the speculation has been committed by all the other co-owners, as shown in the rule below.

Comm-Proc



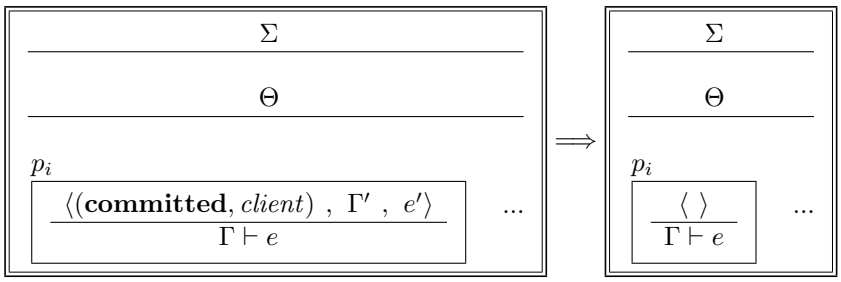
Another situation in which the co-owner of a speculation that has committed the speculation can make progress is if the speculation has been aborted by another co-owner. This case is presented below.

Abort-Proc-Peer



If a process was absorbed in a speculation that has been fully committed by its co-owners it can only continue its execution by the following rule, which discards the saved checkpoint and continues the execution of the process outside any speculation.

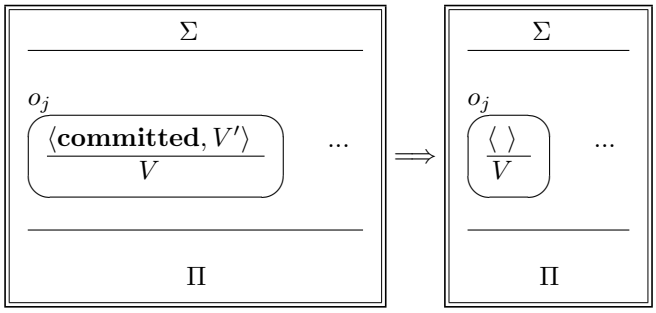
Comm-Client



4.2.7.2 Objects and Committed Speculations

When a speculation is committed, the objects absorbed in the speculation have to discard their saved checkpoint. The following reduction rule illustrates this behavior.

Comm-Obj



4.3 Model for Nested Speculations in a Distributed Shared Objects System

Nested speculations, where processes are allowed to start speculations while they are executing inside speculations, are a refinement of the model. Nested speculations allow for finer grained speculative execution and provide, in certain cases, rollback to a more recent state than the single speculation model. The semantics of speculations does not change in the presence of nested speculations. However, the complexity of the model increases significantly.

Processes execute programs and can start speculations by executing the *speculate* call. After a speculation is started, we say that the speculation is *active* until a commit or an abort call is executed. We say that a process is *executing inside* a speculation if the process's program is executed as part of a speculative computation. An object becomes part of a speculation, or is involved in a speculation if a process that is inside a speculation accesses the object. A process is *absorbed* in a speculation if it reads data from an object that belongs to that speculation.

		Construct	Description
e	::=	$\mathcal{L}$	The base language
		$speculate(e_1 \oplus e_2)$	Speculate call
		$commit()$	Commit call
		$abort()$	Abort call
		$let\ v = read(o_j)\ in\ e[v]$	Read the value of object o_j
		$write(o_j, x)$	Write value x to object o_j
		$e\ ;\ e$	Sequencing

Figure 4.4: Syntactically valid terms

In this model each process can be involved in multiple, nested speculations at any given time. A process is allowed to access multiple objects and an object can be accessed by multiple processes. Each object may be involved in one or more speculations speculation at any given time. We say a process or an object belongs to a speculation if it started that speculation or if it was absorbed in the speculation.

4.3.1 Syntax of the Primitives

The terms of the language that we consider are defined in Figure 4.4. The base language ($\mathcal{L}$) can be any language that does not posses operations for reading from and writing to shared objects. Church-Rosser language [38]. constructs and with a special call for accessing shared objects.

The speculative construct $speculate(e_1 \oplus e_2)$ defines a speculation. In the speculative mode, the program executes e_1. If it executes an $abort()$, the speculation is aborted and the process rolls back and executes e_2. If a $commit()$ is encountered in e_1, the process will never roll back its state to take the e_2 branch. We refer to e_1 as the "commit" branch, and to e_2 as the "abort" branch.

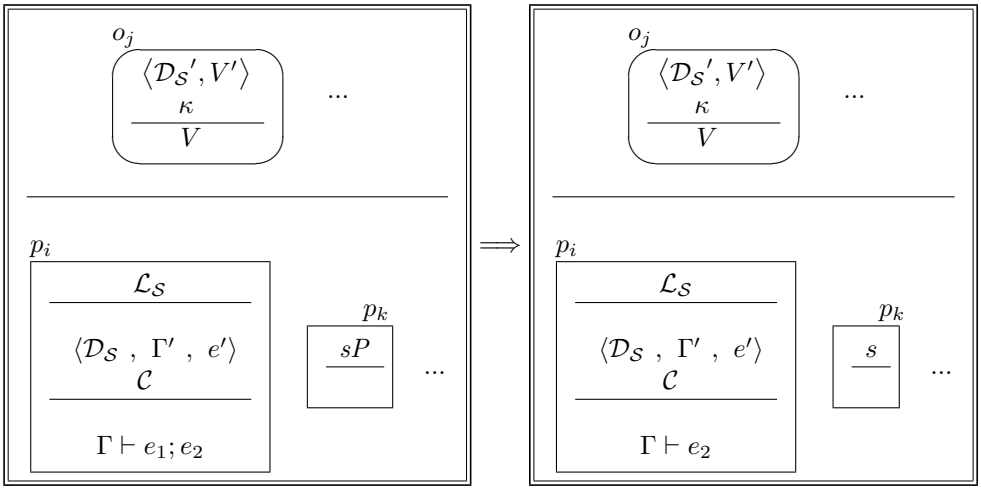
The $let\ v = read(o_j)\ in\ e[v]$ construct assigns the value of shared object o_j to variable v, which is bound in e. The write operation is represented by $write(o_j, x)$. It stores the value x in shared object o_j.

The syntactically valid terms presented above can be sequenced using the *";"* separator.

4.3.2 Terminology and Notation

The speculative operational semantics presented in this section uses the notation shown in rule NESTED-SAMPLE-RULE. The operational semantics defines a reduction system that operates on states. If a distributed system in state Δ_i reduces in one step to state Δ'_i we write $\Delta_i \rightarrow \Delta'_i$. The meaning of $\Delta_i \rightarrow \Delta'_i$ is that the state of one, and only one, process changes as part of the reduction step. Formally, this is written as follows.

Nested-Sample-Rule



In this model the state of the distributed system is composed of two entities: the states of the objects in the system Θ, and the states of the processes running in the system Π. Graphically, they split the system state in two parts.

The state of a shared object (O_j) is characterized by the value it stores (V) and by an optional checkpoint stack. The optional checkpoint stack stores a dependency list ($\mathcal{D}_\mathcal{S}$) and the value (V') the object had before entering the speculation. The checkpoint stack is empty if the object is not part of any speculation.

The state of a speculative process (p_i) is defined by three components:

- The list of speculations that the process started ($\mathcal{L}_\mathcal{S}$).
- An optional checkpoint stack, where the most recent checkpoint lives at the top of the stack, followed by the rest of the checkpoint stack (c).
- An environment (Γ), and the instructions of the program it executes (e). The environment, (Γ), contains variable definitions.

The checkpoint of a process has three components:

- The dependency list ($\mathcal{D}_\mathcal{S}$) represents the list of speculations on which the checkpoints depends. The order of speculation identifiers in the list is relevant (the list is ordered).
- The local environment of the process (Γ'), at the time the process became part of the speculation.
- The expression to be executed in case of rollback (e'), the "abort" branch of the speculation.

The dependency list saved in a checkpoint is defined as an ordered list of speculation identifiers, represented as follows: $\{s_0, s_1, ...s_q, ...s_k\}$. The order of the speculation identifiers in the dependency

list gives the order in which speculations were entered by the current process or by other processes or objects whose speculative data was read at some point by the current process. We define the concatenation, or merger, operator ($\overleftrightarrow{\cup}$) on dependency lists, as follows:

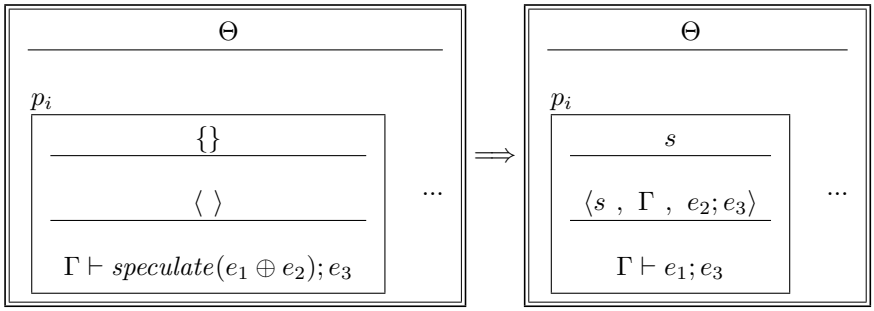
If $\mathcal{D}_{\mathcal{S}} = \{s_{i_0}, s_{i_1}, ...s_{i_p}\}$ and $\mathcal{D}_{\mathcal{S}}{}' = \{s_{j_0}, s_{j_1}, ...s_{j_q}\}$, then $\mathcal{D}_{\mathcal{S}} \overleftrightarrow{\cup} \mathcal{D}_{\mathcal{S}}{}' = \{s_{i_0}, s_{i_1}, ...s_{i_p}, s_{j_0}, s_{j_1}, ...s_{j_q}\}$.

4.3.3 Speculate

A process outside any speculation successfully starts a new speculation by using the *speculate*($\oplus$) construct. A new speculation is created and added to the list of speculations created by the process. A checkpoint of the process is also taken. The checkpoint becomes part of that process's state and is added to the top of the checkpoints stack. The process advances with the execution to the next instruction in its program.

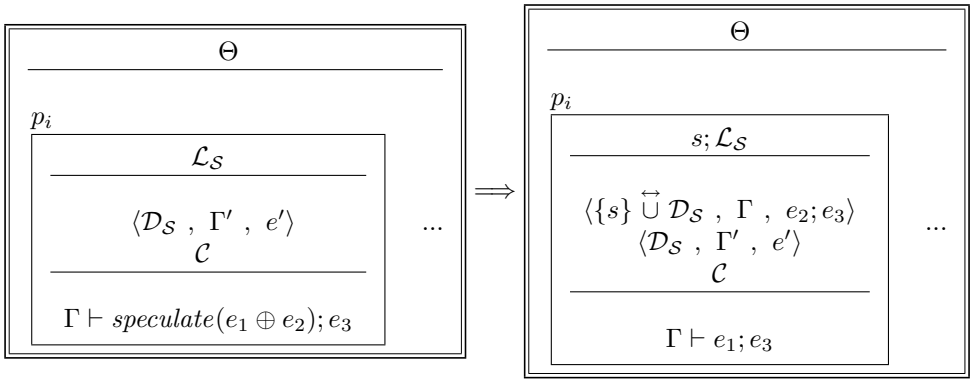
We distinguish two cases. The first one, shown below, describes the actions performed when a nonspeculative process starts a speculation.

Nested-SpecSimple



The second case shows an already speculative process starting a new speculation. The speculation identifier is added to the list of speculations started by the process. A new checkpoint is saved in the checkpoint stack. The speculation dependency list associated with the checkpoint is computed by adding the speculation identifier to the front of the dependency list of the previous checkpoint. By adding the speculation s to the front of the dependency list $\mathcal{D}_{\mathcal{S}}$, we force the speculation to depend on all the previous speculations the process is involved in. The rule describing these actions is shown next.

Nested-Spec



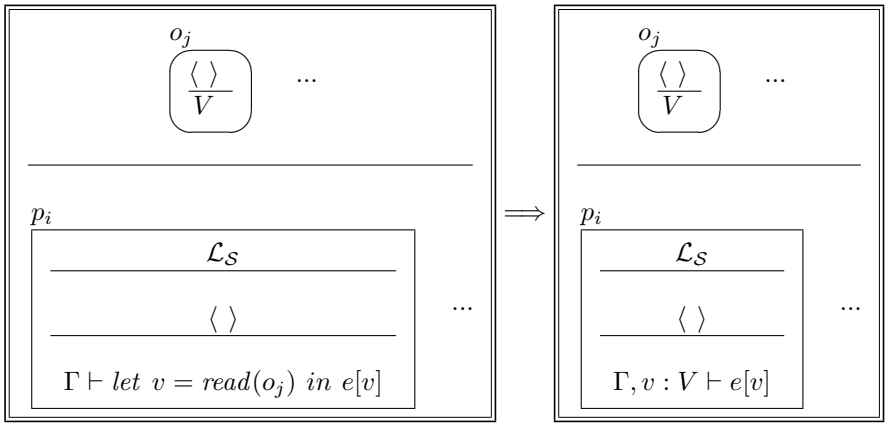
4.3.4 Reading from a Shared Object

The reduction rules for the read operation take into consideration whether the object and the process are speculative or not.

4.3.4.1 Both the process and the object are outside speculations

Reading the value of a shared object when neither the object nor the process are part of any speculation is illustrated by the following reduction rule.

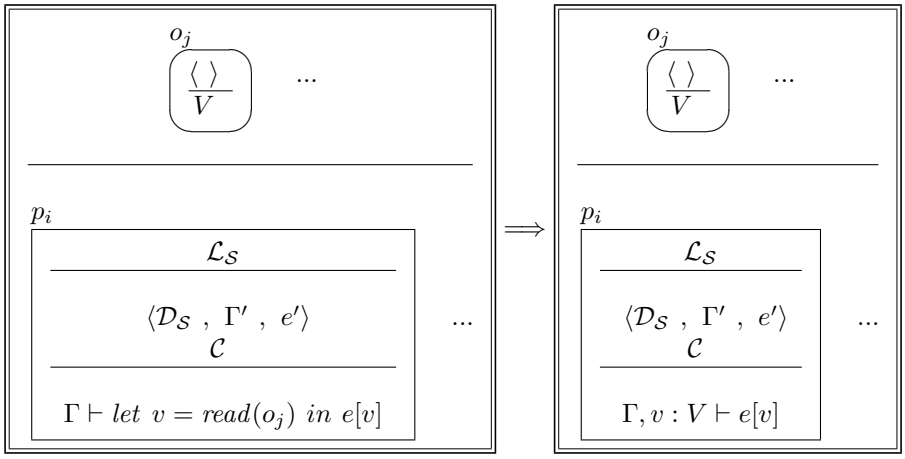
Nested-Read-NoSpec



4.3.4.2 The process is inside a speculation and the object is outside any speculation

When a speculative process reads the value of a nonspeculative object it does not absorb the object in its speculation. This prevents unnecessary propagation of speculations.

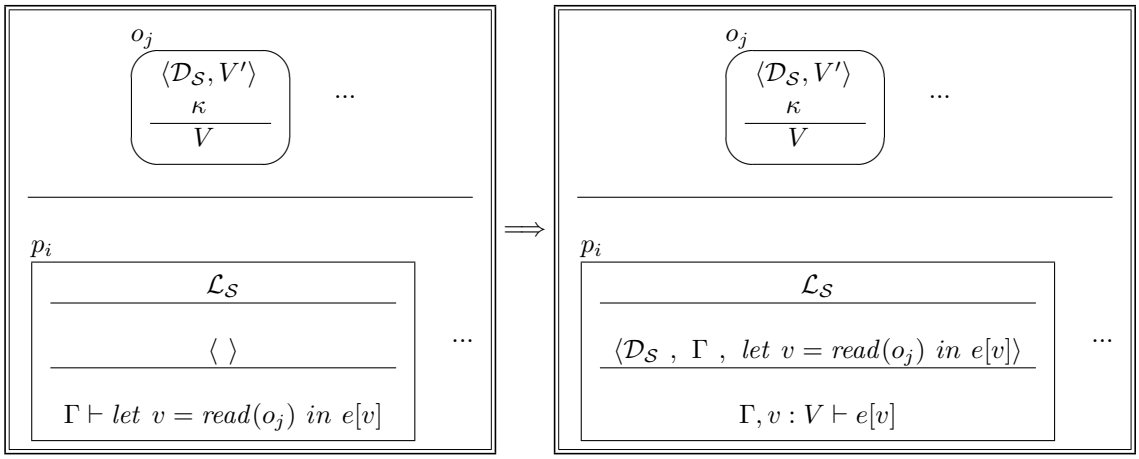
Nested-Read-Spec-Proc



4.3.4.3 The process is outside any speculation and the object is inside a speculation

A nonspeculative process executing a read from a speculative object becomes itself speculative. The process's state depends on speculative information, so the process is absorbed in the object's speculation. A checkpoint of the process is created to allow rollback if the speculation is later aborted.

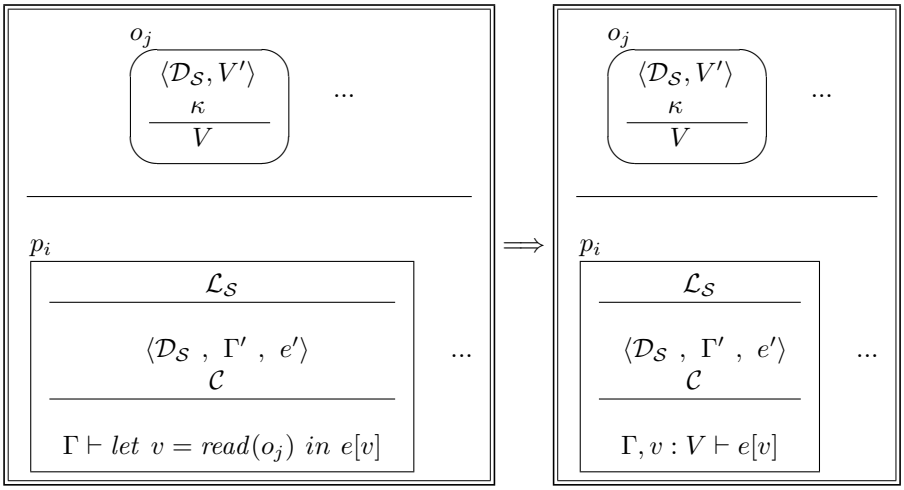
Nested-Read-Spec-Obj



4.3.4.4 Both the process and the object are inside the same speculation

The reduction rule is similar to the one presented in the case when neither the process nor the object were speculative (Rule NESTED-READ-NOSPEC). Only the internal environment of the process changes and the program continues with the next instruction.

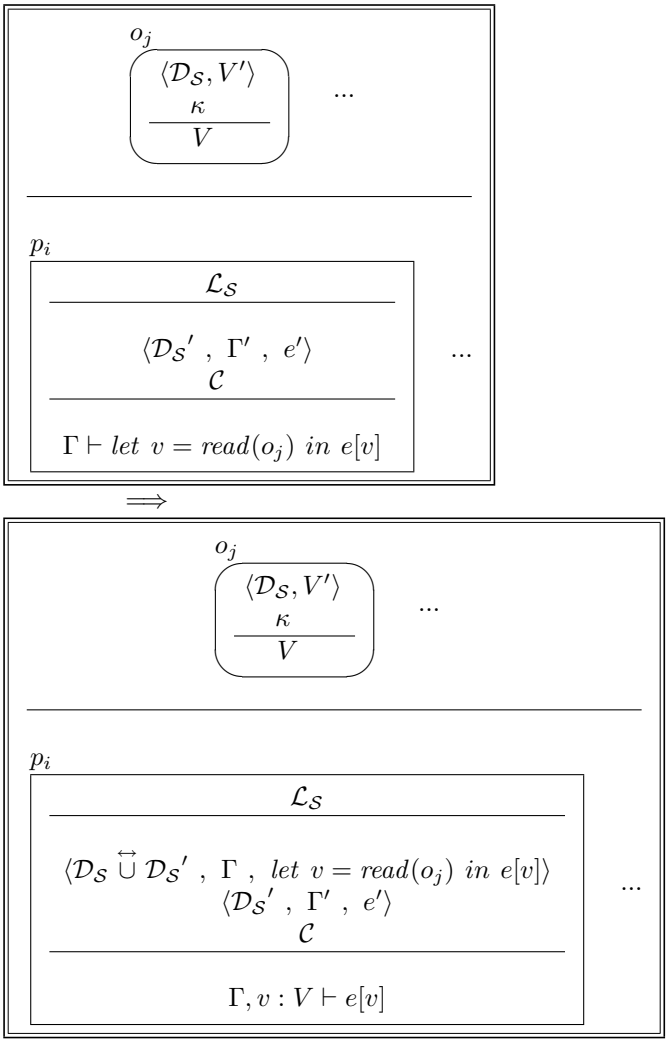
Nested-Read-Spec-Same



4.3.4.5 The process and the object are inside different speculations

The most interesting case for reading the value of a shared object is when the process and the object belong to different speculations. After the read operation is performed the state of the process depends on the speculative data stored in the shared object at the time of the access. The process saves a checkpoint that depends both on the speculation list its current state depends on and the speculation list the object's state depends on. We use the $\mathcal{D}_\mathcal{S} \overset{\leftrightarrow}{\cup} \mathcal{D}_\mathcal{S}{}'$ notation to merge the two dependency lists. The following rule provides the graphical representation of the state change.

Nested-Read-Spec-Nested



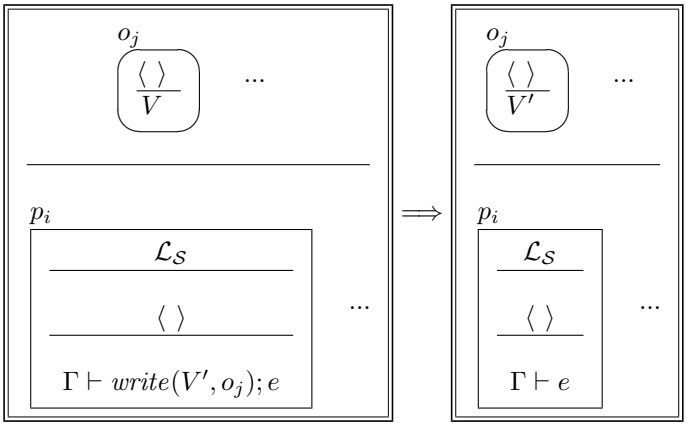
4.3.5 Writing Data to a Shared Object

The operational semantics defines different reduction rules to describe the action of writing data to shared objects, depending on whether processes and objects are speculative or not. After executing the $write(V, o_j); e$ the value of object o_j is updated with value V and execution of the program is resumed to e.

4.3.5.1 Both the process and the object are outside speculations

Writing a value to a shared object, when neither the object nor the process accessing it are speculative is illustrated by the following reduction rule.

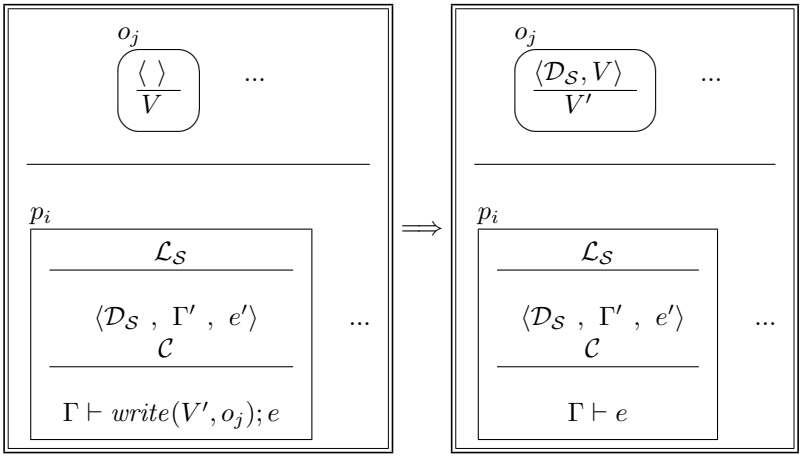
Nested-Write-NoSpec



4.3.5.2 The process is inside a speculation and the object is outside any speculation

When a process that speculates writes a value to a shared object that is not part of any speculation the object is absorbed in the process's speculation. This behavior is different from the one seen for the read operation because the value stored in the shared object is speculative and the object has to become speculative itself. The system creates a checkpoint for the object, storing the value it had before becoming part of the speculation, which allows it to roll back if the speculation is aborted. The speculation id is also stored as part of the checkpoint.

Nested-Write-Spec-Proc

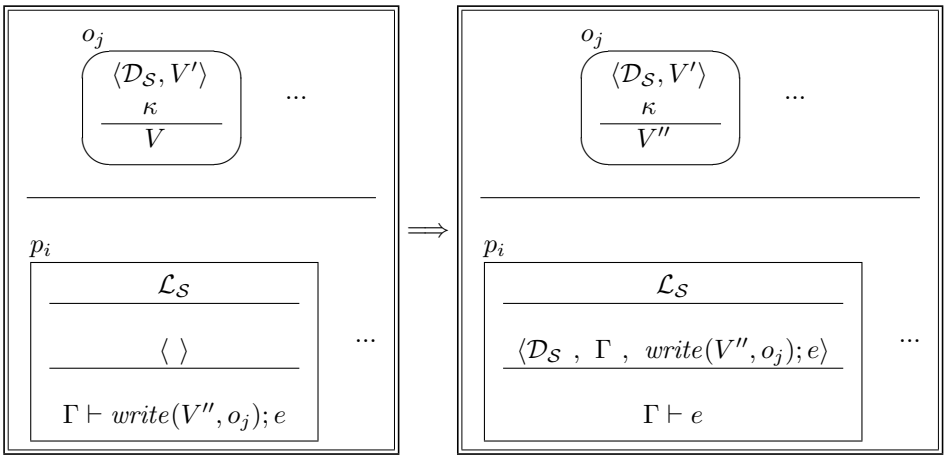


4.3.5.3 The process is outside any speculation and the object is inside a speculation

The process writing to the object is absorbed in the object's speculation. The abort branch of the speculation is the same as the initial commit branch, since in case of a rollback the execution has to start from the same point. This implicit speculation is desired because a write followed by a read,

when no other actions are performed on an object should return the value written. However, if the object rolls back between the write and the read and the process is not involved in the speculation we could get an unexpected result. It can also be argued that the rollback operation can be thought of as an external write, but we prefer the behavior presented in the rule below.

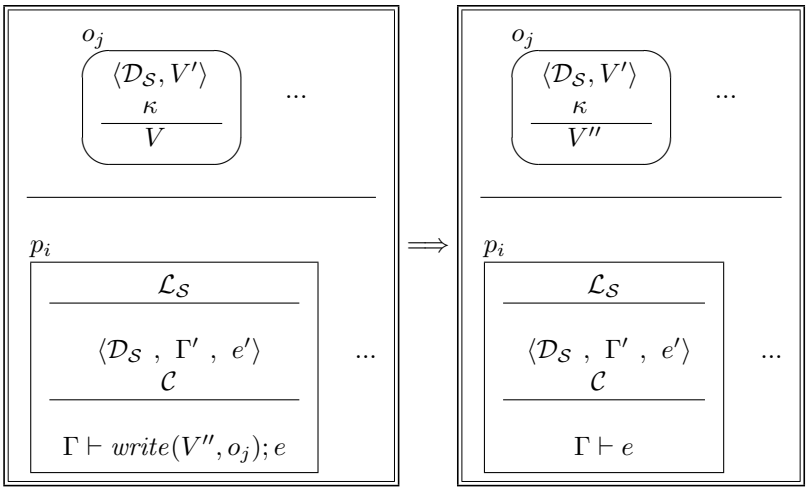
Nested-Write-Spec-Obj



4.3.5.4 Both the process and the object are inside the same speculation

The reduction rule is similar to the case when neither the process nor the object where part of any speculation (Rule NESTED-WRITE-NOSPEC).

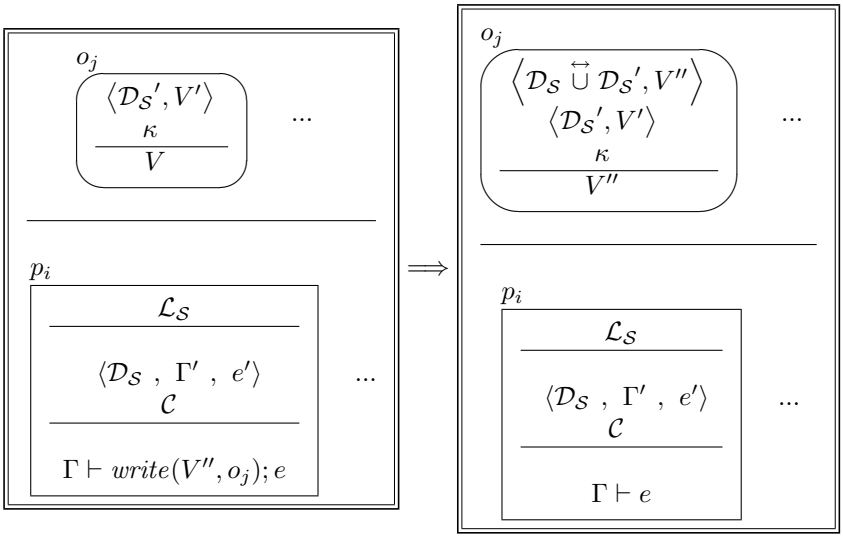
Nested-Write-Spec-Same



4.3.5.5 The process and the object are inside different speculations; speculations are merged

The most interesting case for writing an object is when the process and the object are inside different speculations. In this case the object becomes speculative and a new checkpoint is created. Its checkpoint depends on both the speculations in the object's dependency list as well as the process's dependency list. Again, we use the $\mathcal{D}_{\mathcal{S}} \overset{\leftrightarrow}{\cup} \mathcal{D}_{\mathcal{S}}{}'$ notation to merge the two dependency lists.

Nested-Write-Spec-Nested



4.3.6 Aborting a Speculation

A speculation can be aborted by the initiating process when the assumption it was based on is invalidated. We present several operational semantics rules describing the actions taken when speculations are aborted.

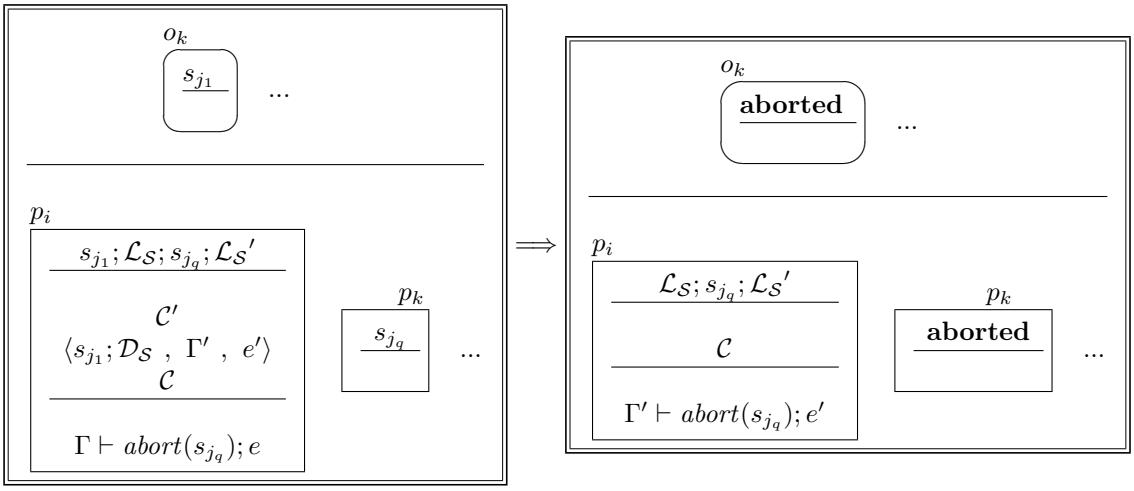
4.3.6.1 Processes and Aborted Speculations

A process that is inside a speculation that it owns is allowed to abort it. The reduction rule for the *abort*() call substitutes the id of the aborted speculation with the **aborted** special constant and rolls back the process to the state it was before entering the speculation. Also, the speculation id is erased from the speculations environment. The substitution operation guarantees that once a speculation has been aborted no other process or object can be absorbed in that speculation, since its id has been replaced by the special constant **aborted**.

The operation has in fact two steps in our operational semantics, as described below. This separation is not necessary in the implementation, its sole purpose being to clearly identify the actions that have to be performed for the system to behave correctly.

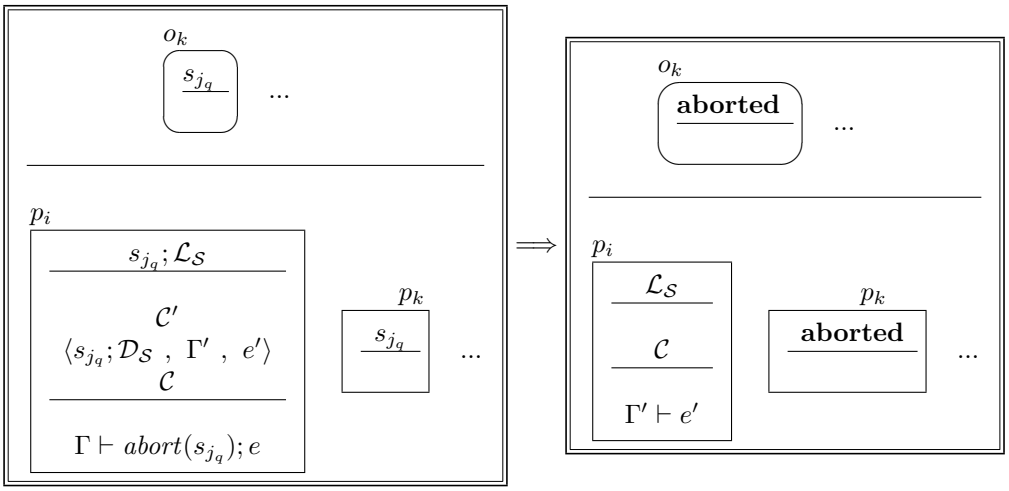
First, the process that aborts the speculation needs to abort all the speculations that it started since it entered the speculation that it wants to explicitly abort. Step by step, by repeatedly applying rule NESTED-AB-OWNER it aborts them and rolls back its state. The program does not change for the process during this stage.

Nested-Ab-Owner



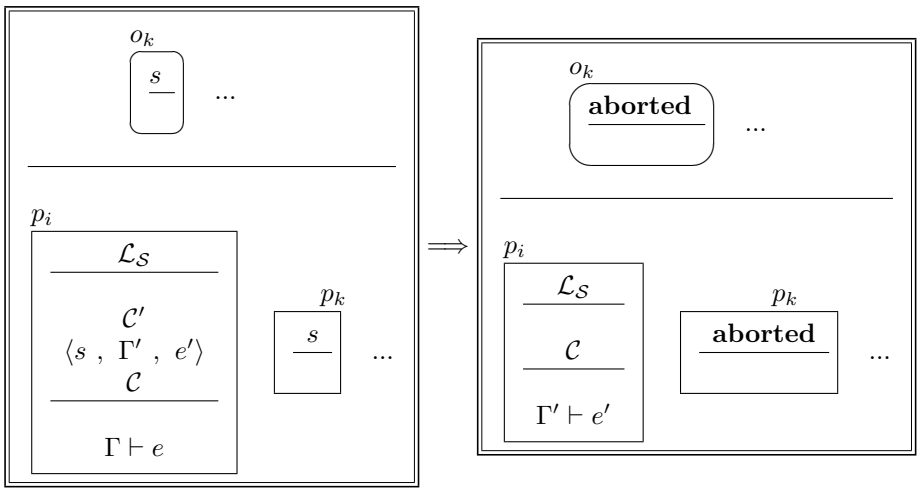
The second step is reached when the speculation that it wants to abort is the last one in its list $\mathcal{L}_\mathcal{S}$. At this point it aborts the speculation and it continues execution with e.

Nested-Ab-Owner-Last



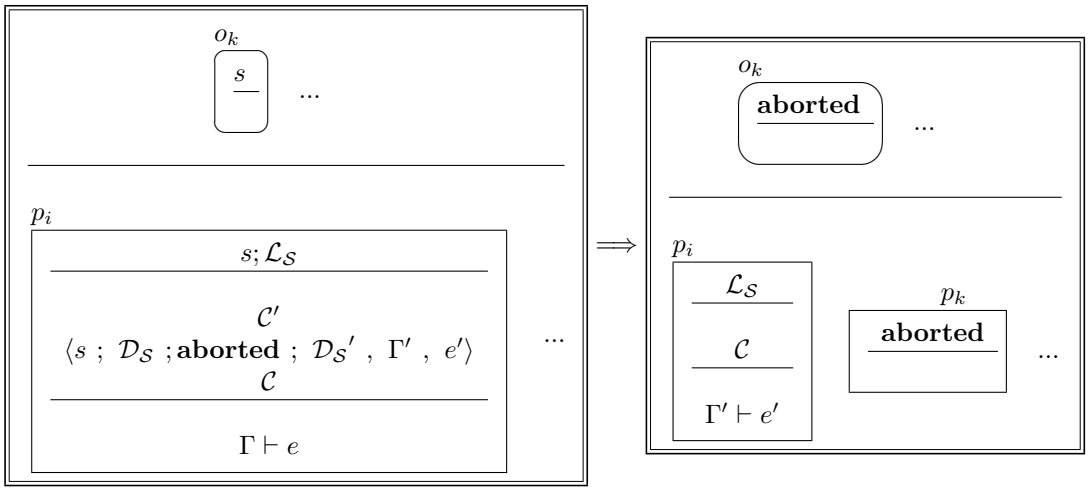
Processes can also fail at any time during their execution due to external factors. When a process fails while it executes inside a speculation that it owns, the system aborts the speculation, the process rolls back and starts executing the abort branch.

Nested-Ab-Fail



If the execution of a process depends on a speculation that has been aborted by another process, it needs to roll back and restart the computation from the point where it was absorbed in the speculation. If the process has itself started speculations during the aborted computation, it needs to abort them as well. Again, we designed a two-step process to handle this. First, the process aborts all the speculations that it has started and rolls back, step by step, to the last speculation it started inside the remotely aborted one. For this we need to apply rule NESTED-AB-FORCE-AB repeatedly.

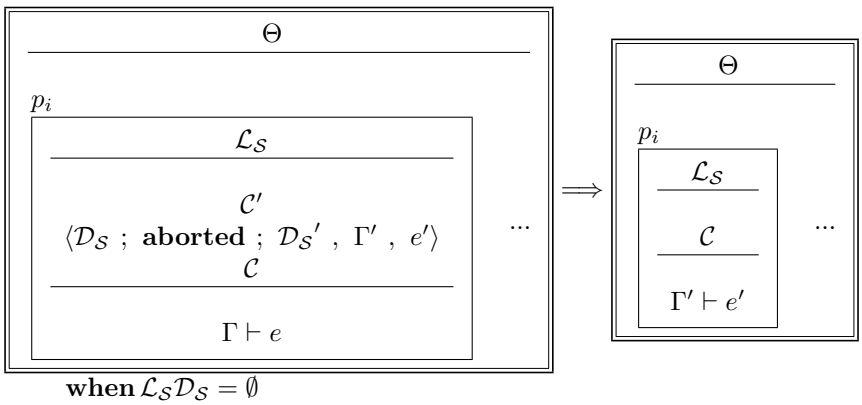
Nested-Ab-Force-Ab



When the process is inside the remotely aborted speculation and has no active speculations that it started itself, then it rolls back to the state it had before becoming part of that speculation. Note that the rule below may need to be applied repeatedly until there are no aborted speculations. The reason for this is that, the way it is written, the rule does not guarantee that we rolled back to the earliest point. This is a small caveat of our representation, but it makes it easy to code the semantics

in an automated theorem prover.

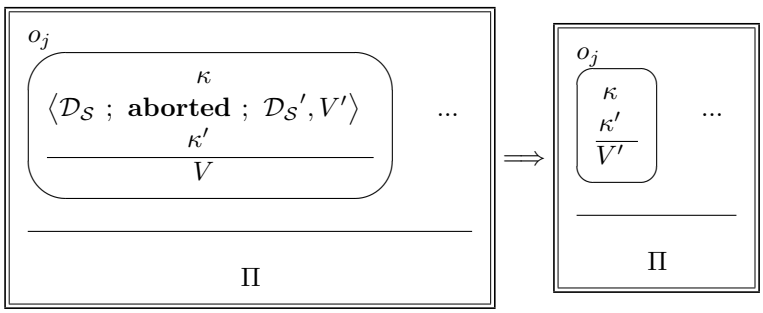
Nested-Ab-Proc



4.3.6.2 Objects and Aborted Speculations

If an object is inside an aborted speculation it rolls back its state to the state saved in the checkpoint associated with the speculation.

Nested-Ab-Object



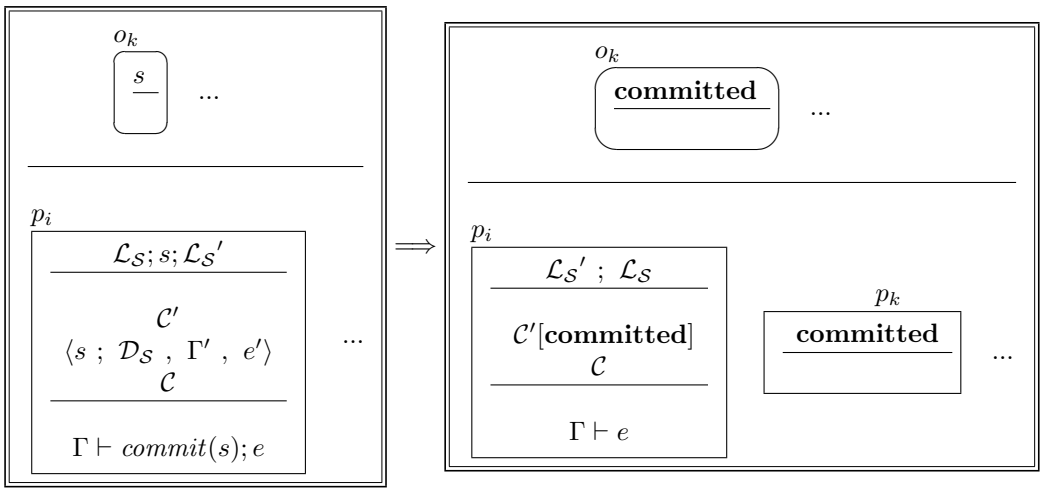
The reduction rules for aborted speculation are constructed such that they guarantee that a process or an object that is inside a speculation that has been aborted is not allowed to be involved in any other operation other than rolling back its state.

4.3.7 Commit a Speculation

4.3.7.1 Processes and Committed Speculations

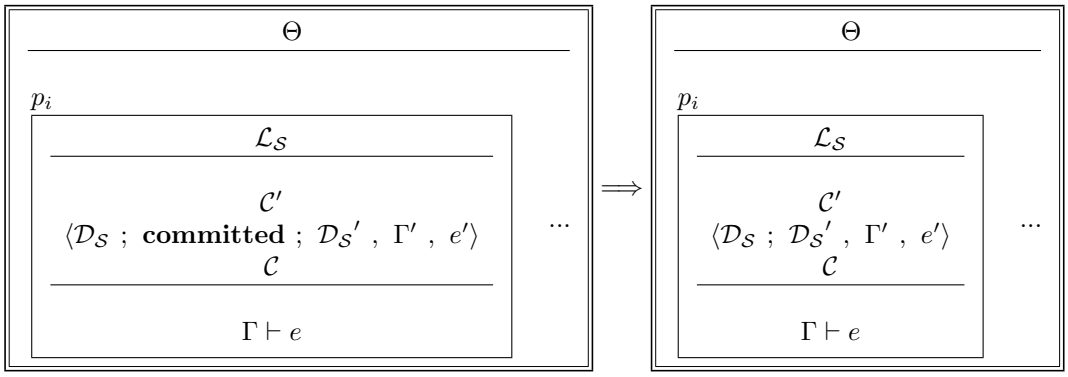
Only processes that own a speculation may commit it. The checkpoint associated with the speculation is discarded and the speculation is marked as committed. This prevents other processes to become dependent on the speculation. The reduction rule below illustrates this behavior.

Nested-Comm-Owner



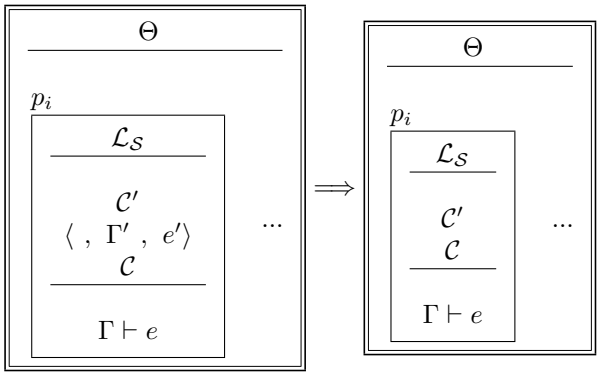
If a speculation has been aborted its identifier can be eliminated from the dependency list associated with the checkpoints that depended on it. The operational semantics rule is the following.

Nested-Comm-Peer



Finally, if a checkpoint does not depend on any speculations it can be completely eliminated. The next rule shows this action.

Nested-Comm-Remove

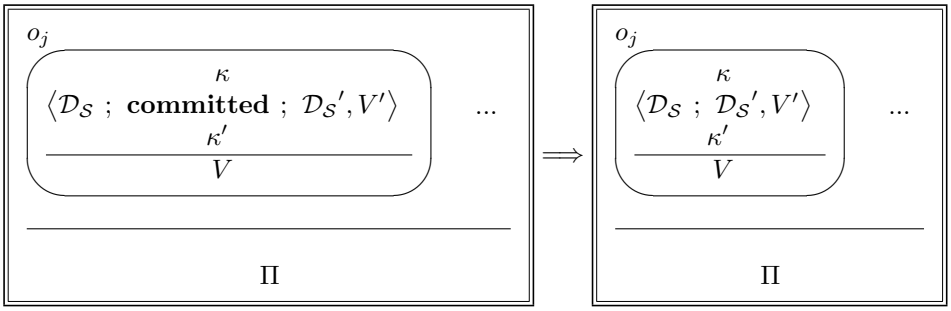


4.3.7.2 Objects and Committed Speculations

When speculations are committed by processes, the objects inside the speculation may have to discard their saved checkpoint. The following reduction rules are used to model this behavior.

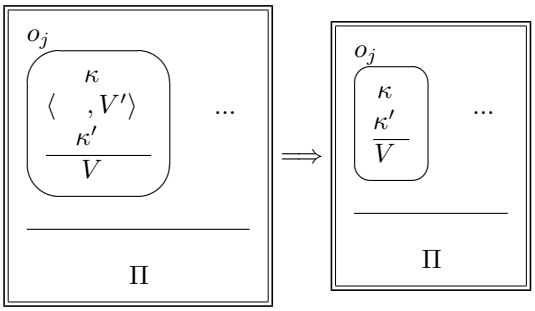
First, if a speculation is committed its identifier can be eliminated from the dependency list of all the checkpoints that depended on it.

Nested-Comm-Obj



If a checkpoint does not depend on any speculation anymore it can be discarded.

Nested-Comm-Obj-Remove



4.4 Nonspeculative model

In this section we define a nonspeculative model for the speculative constructs introduced in Section 4.2.1. We show how the execution of speculative programs is equivalent to the execution of programs that use this nonspeculative, nondeterministic model. This may allow us to reason about properties of speculative programs by using existing tools that can reason about the nonspeculative model. We use the speculative model in Section 4.2 for the equivalence proof that is presented in Section 4.5.

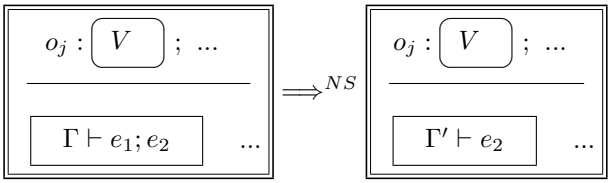
The nonspeculative model discussed in this section uses processes and shared objects in a way similar to that of the speculative model. An object is characterized by the value V it stores. Processes execute programs. A program is defined by its current environment Γ, and the expression e to be

reduced.

The expression e is a sequence of statements. Statements include the speculative programming constructs presented in Section 4.2.1. The state of the distributed system is composed of the state of all processes (Π^{NS}) and the state of all objects (Θ^{NS}) in the system. We use a diagram representation of the system state, where objects are shown in the upper half, and processes in the lower half of the block corresponding to the state.

We use a graphical representation of the model similar to the one used in the speculative model. A sample rule, showing the two components of the distributed system state is illustrated below.

NS-Sample

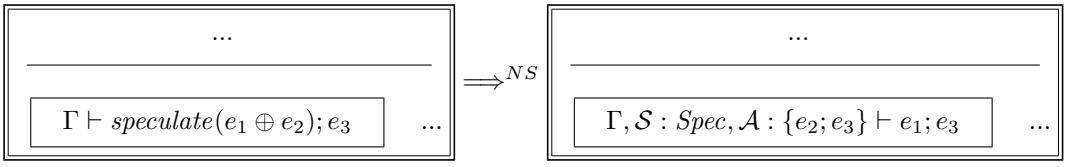


4.4.1 Nonspeculative operational semantics

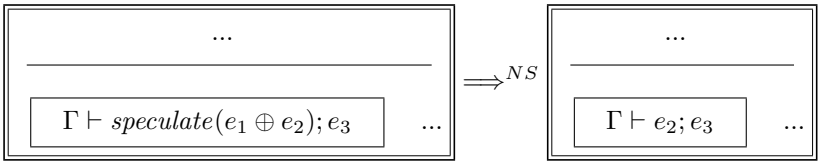
When the process encounters a speculate call, it may choose nondeterministically to take either the commit or the abort branch. This behavior is illustrated by Rules NS-Spec-C-br and NS-Spec-A-br below.

For the purpose of proving the equivalence of this model to the speculative model presented in Section 4.2, we introduce two history variables that are local to each process's environment. These history variables are only used by the NS-Spec-C-br reduce rule. There is one history variable per process, $\mathcal{S}$, that refers to the fact that we are currently inside a speculation. The second history variable, $\mathcal{A}$, of each process refers to the abort branch that is ignored by the program. These history variables are not accessible to programs.

NS-Spec-C-br

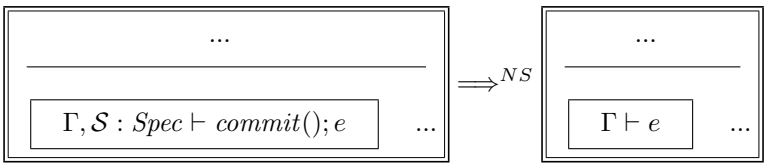


NS-Spec-A-br



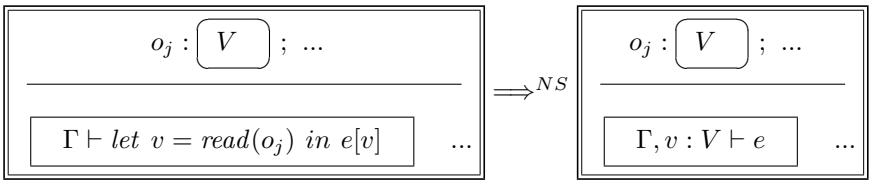
The commit operation does not have any effect on a nonspeculative process. The process continues the execution, as shown in the following reduction rule.

NS-Commit



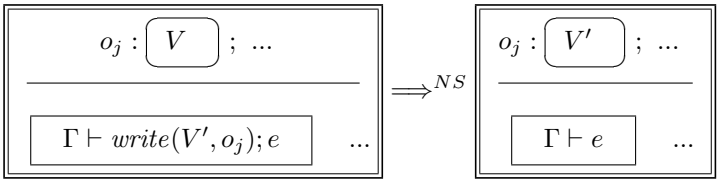
Reading the value of a shared object is defined by the following reduction rule. Local variable v is assigned the value of object o_j and becomes part of the local environment of the process.

NS-Read



Writing a value to a shared object is defined by the following reduction rule. The value V' is the value of object o_j after the write operation is performed.

NS-Write



There is no rule for the abort operation, since rollback is not defined in the nonspeculative model. If a nonspeculative process tries to execute an abort operation it is considered to have taken an invalid execution branch and is said to be "stuck."

4.5 Equivalence of the Speculative and Nonspeculative Versions of the Distributed Objects System Model

4.5.1 Algebraic Representation of the Operational Semantics Rules

For the purpose of a clear and concise presentation of the definitions, theorems and proofs in this section we resort to an algebraic representation of the operational semantics rules presented in Sections 4.2 and 4.4.

Figure 4.5 presents the three different states that we encounter in our speculative model and their equivalent algebraic representations. The state of a process, p_i, will be referred to using the symbol P_i, while the state of an object, o_j is represented by O_j. The set of processes in the system that are not explicitly shown in any given rule are referred to using the symbol Π, while the equivalent symbol for objects is Θ.

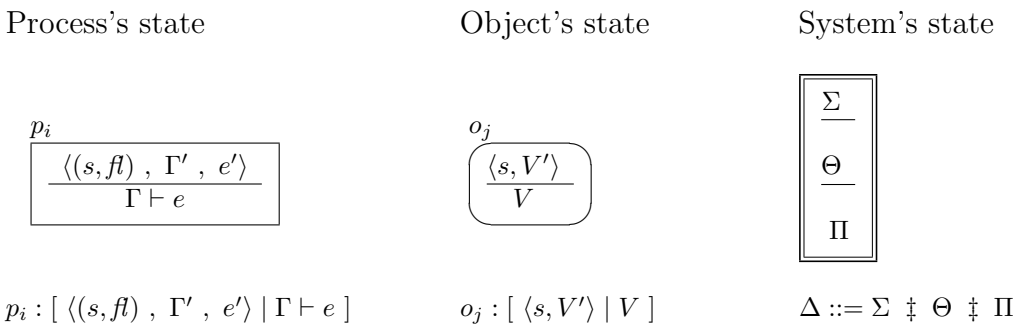


Figure 4.5: Graphical and algebraic representation of speculative states.

Figure 4.6 presents the three different types of states we encounter in the nonspeculative model and their equivalent algebraic representations. A similar notation as in the case of the speculative model is used to refer to the states without explicitly showing their components. The state of a process will be referred to using notation P^{NS}, while the state of an object is represented by O^{NS}. The set of processes in the system that are not shown in any given rule are referred to using the symbol Π^{NS}, while the equivalent symbol for object is Θ^{NS}.

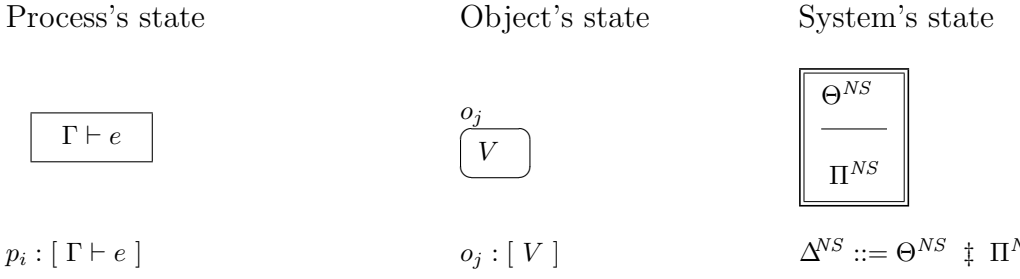


Figure 4.6: Graphical and algebraic representation of nonspeculative states.

4.5.2 Definitions and Abstractions

Definition 4.5.1. The *reduction trace* of a distributed system is defined as the sequence of distributed system states given by the reduction rules that are applied during the execution of the programs.

$$\overrightarrow{\Delta} ::= \Delta_0 \ \Rightarrow \ \ldots \ \Rightarrow \ \Delta_n$$

The definition of the reduction trace refers only to single reduction steps. We also introduce the notion for one or more reduction steps, with notation $\Rightarrow^*$. Definition 4.5.1 can also be written as:

$$\overrightarrow{\Delta} ::= \Delta_0 \Rightarrow^* \Delta_n$$

Definition 4.5.2. An oracle, $\mathcal{O}$, is defined as a mapping from speculations (ids) to either *commit* or *abort*, and provides the outcome of each speculation.

The oracles, as defined above, allow speculations to be mapped to *commit* or *abort* regardless of the actual outcome of the speculation. We introduce a relation between oracles and reduction traces that guarantees that oracles do not contradict the reduction rules composing a trace. For example, if a trace contains the Rule Ab-Owner involving speculation s, then the oracle should not predict the outcome of speculation s as commit.

Definition 4.5.3. The set of *valid oracles*, $\Omega(\overrightarrow{\Delta})$, for a given reduction trace, $\overrightarrow{\Delta}$, is defined as follows:

$$\Omega(\overrightarrow{\Delta}) =$$
$$\Big\{\mathcal{O} \in ((\cup_{i=0}^{n} \textit{ active speculations of state } i) \Rightarrow \{a, c\}) \ \Big| \ \forall \Delta_{i-1}, \Delta_i \in \overrightarrow{\Delta},$$

$$\left.\begin{array}{lll} \textit{if } \Delta_{i-1} \overset{\textit{merge } s_1,s_2 \textit{ into } s_3}{\Longrightarrow} \Delta_i & \textit{then} & \mathcal{O}(s_1) = \mathcal{O}(s_2) = \mathcal{O}(s_3) \\ \textit{if } \Delta_{i-1} \overset{\textit{abort } s}{\Longrightarrow} \Delta_i & \textit{then} & \mathcal{O}(s) = a \\ \textit{if } \Delta_{i-1} \overset{\textit{commit } s}{\Longrightarrow} \Delta_i & \textit{then} & \mathcal{O}(s) = c \end{array}\right\}$$

As a reduction trace grows, the set of valid oracles shrinks, since new reduction rules can impose restrictions on the outcome of speculations. This is formally expressed by the following lemma.

Lemma 4.5.4. The set of valid oracles for a reduction trace is a subset of the set of valid oracles for any prefix of that reduction trace.

Proof. This lemma immediately follows from Definition 4.5.3. □

There are two special constants used for committed or aborted speculations. Any valid oracle extends over these two constants.

Definition 4.5.5. The definition of a valid oracle extends over constants **aborted** and **committed** as follows:

$$\forall \mathcal{O} \in \Omega(\overrightarrow{\Delta}), \mathcal{O}(\textbf{aborted}) = a \text{ and } \mathcal{O}(\textbf{committed}) = c$$

Next we define a conversion from speculative states into nonspeculative ones. For clarity we use both a graphical representation and its equivalent algebraic notation throughout this section.

Definition 4.5.6. The action of lowering ($\Downarrow ()$) a speculative state to a nonspeculative state, given an oracle $\mathcal{O}$, is a function from speculative states to nonspeculative states defined as follows:

$$\Downarrow (O) = \Downarrow ([\ \langle\ \rangle \mid V\ ]) = [\ V\ ]$$

$$\Downarrow (O) = \Downarrow ([\ \langle s, V' \rangle \mid V\ ]) = \begin{cases} [\ V\ ] \ \textit{if}\ \mathcal{O}(s) = c \\ [\ V'\ ] \ \textit{if}\ \mathcal{O}(s) = a \end{cases}$$

$$\Downarrow (P) = \Downarrow (p_i : [\ \langle\ \rangle \mid \Gamma \vdash e\ ]) = [\ \Gamma \vdash e\ ]$$

$$\Downarrow (P) = \Downarrow (p_i : [\ \langle (s, \not{n}) \ ,\ \Gamma' \ ,\ e' \rangle \mid \Gamma \vdash e\ ]) = \begin{cases} [\ \Gamma \vdash e\ ] \ \textit{if}\ \mathcal{O}(s) = c \\ [\ \Gamma' \vdash e'\ ] \ \textit{if}\ \mathcal{O}(s) = a \end{cases}$$

$$\Downarrow (\Theta) = \Downarrow (O_1\ ;\ \ldots\ ;\ O_m) = \Downarrow (O_1)\ ;\ \ldots\ ;\ \Downarrow (O_m)$$

$$\Downarrow (\Pi) = \Downarrow (P_1 \ldots P_n) = \Downarrow (P_1) \ldots \Downarrow (P_n)$$

$$\Downarrow (\Delta) = \Downarrow (\Sigma\ \ddagger\ \Theta\ \ddagger\ \Pi) = \Downarrow (\Theta)\ \ddagger\ \Downarrow (\Pi)$$

The lowering operation strips the checkpoint of a speculative state in order to create an equivalent nonspeculative state. The result of the lowering operation depends on the chosen oracle.

Lemma 4.5.7. Given any valid oracle, $\mathcal{O}$, the result of applying $\Downarrow ()$ to any well-formed speculative state, $O, P, \Theta, \Pi,$ or Δ, is a well-formed nonspeculative state, $O^{NS}, P^{NS}, \Theta^{NS}, \Pi^{NS},$ or Δ^{NS}, respectively.

Proof. The lemma follows directly from Definition 4.5.6. □

We can also show that the operation of substituting the special constants **committed** and **aborted** for any speculation s, as defined in Section 4.2, is consistent with the definition of lowering a speculative state.

Lemma 4.5.8. For any valid oracle, $\mathcal{O}$, the following properties hold.

$$\Downarrow(\Theta[s]) = \begin{cases} \Downarrow(\Theta[\textbf{committed}]) \ \textit{if} \ \mathcal{O}(s) = c \\ \Downarrow(\Theta[\textbf{aborted}]) \ \textit{if} \ \mathcal{O}(s) = a \end{cases}$$

$$\Downarrow(\Pi[s]) = \begin{cases} \Downarrow(\Pi[\textbf{committed}]) \ \textit{if} \ \mathcal{O}(s) = c \\ \Downarrow(\Pi[\textbf{aborted}]) \ \textit{if} \ \mathcal{O}(s) = a \end{cases}$$

Proof. This lemma immediately follows from the definition of $\Downarrow(\ )$ (Definition 4.5.6), the extended definition of oracles (Definition 4.5.5), and the definition of substitution presented in Section 4.2. □

We also define the inverse of the lowering function, which we call lifting ($\Uparrow()$) a nonspeculative state to a speculative state. The operation mainly adds empty checkpoints to the nonspeculative states as described in the following definition.

Definition 4.5.9. Lifting a nonspeculative state to a speculative state is a function from nonspeculative states to speculative states defined as follows:

$$\Uparrow(O^{NS}) = \Uparrow([\ V\ ]) = [\ \langle\ \rangle \mid V\ ]$$

$$\Uparrow(P^{NS}) = \Uparrow([\ \Gamma \vdash e\ ]) = p_i : [\ \langle\ \rangle \mid \Gamma \vdash e\ ]$$

$$\Uparrow(P^{NS}) = \Uparrow([\ \Gamma, \mathcal{S} : \mathit{Spec}, \mathcal{A} : \{e'\} \vdash e\ ]) = p_i : [\ \langle(\mathcal{S}, \mathit{own})\ ,\ \Gamma\ ,\ e'\rangle \mid \Gamma \vdash e\ ]$$

$$\Uparrow(\Theta^{NS}) = \Uparrow(O_1^{NS}\ ;\ \ldots\ ;\ O_m^{NS}) = \Uparrow(O_1^{NS})\ ;\ \ldots\ ;\ \Uparrow(O_m^{NS})$$

$$\Uparrow(\Pi^{NS}) = \Uparrow(P_1^{NS} \ldots P_n^{NS}) = \Uparrow(P_1^{NS}) \ldots \Uparrow(P_n^{NS})$$

$$\Uparrow(\Delta^{NS}) = \Uparrow(\Theta^{NS}\ \ddagger\ \Pi^{NS}) = \quad \ddagger\ \Uparrow(\Theta^{NS})\ \ddagger\ \Uparrow(\Pi^{NS})$$

In certain cases, when the history variables exist in the local environment of a process the lifting function creates a non-empty checkpoint in the speculative process state. This is required to keep track of whether or not a process is still inside a speculation or not. More details on when this lifting rule is applied will be discussed in the proofs of the equivalence theorems.

Lemma 4.5.10. Given any valid oracle, $\mathcal{O}$, the result of applying $\Uparrow()$ to any well-formed nonspeculative state, $O^{NS}, P^{NS}, \Theta^{NS}, \Pi^{NS}$, or Δ^{NS}, is a well-formed speculative state, O, P, Θ, Π, or Δ respectively.

Proof. The lemma follows directly from Definition 4.5.9. □

4.5.3 Equivalence Theorems

The proof of equivalence between the two models could also be done using a simulation relation. However, showing that the speculative model has a non-speculative reduction trace requires, in case of an abort, discarding states and replacing them with no-ops in the simulation. This is not natural and can be avoided by using oracles.

Theorem 4.5.11. For any speculative reduction rule $\Delta_{i-1} \Rightarrow \Delta_i$ and for any valid oracle $\mathcal{O}$, there is a sequence of nonspeculative reduction rules such that

$$\Downarrow(\Delta_{i-1}) \Rightarrow^{*NS} \Downarrow(\Delta_i).$$

Proof. The proof of this theorem is done by examining each possible rule defined in the speculative operational semantics.

Rule SPEC**.**

$$\begin{aligned}
\Downarrow(\Delta_{i-1}) &= \Downarrow(\Sigma \ \ddagger\ \Theta \ \ddagger\ [\ \langle\ \rangle \mid \Gamma \vdash \mathit{speculate}(e_1 \oplus e_2); e_3\ ]\ ;\ \Pi) = \\
&= \Downarrow(\Theta) \ddagger \Downarrow([\ \langle\ \rangle \mid \Gamma \vdash \mathit{speculate}(e_1 \oplus e_2); e_3\ ]\ ;\ \Pi) = \\
&= \Theta^{NS} \ \ddagger\ [\ \Gamma \vdash \mathit{speculate}(e_1 \oplus e_2); e_3\ ]\ ;\ \Pi^{NS}
\end{aligned}$$

$$\begin{aligned}
\Downarrow(\Delta_i) &= \Downarrow(\Sigma \ \ddagger\ \Theta \ \ddagger\ [\ \langle(s, \mathit{own})\ ,\ \Gamma\ ,\ e_2; e_3\rangle \mid \Gamma \vdash e_1; e_3\ ]\ ;\ \Pi) = \\
&= \Downarrow(\Theta) \ddagger \Downarrow([\ \langle(s, \mathit{own})\ ,\ \Gamma\ ,\ e_2; e_3\rangle \mid \Gamma \vdash e_1; e_3\ ]\ ;\ \Pi)
\end{aligned}$$

Now, given the outcome of speculation s the evaluation of $\Downarrow(\Delta_i)$ is as follows.

- if $\mathcal{O}(s) = c$ then
  $\Downarrow(\Delta_i) = \Theta^{NS} \ \ddagger\ [\ \Gamma \vdash e_1; e_3\ ]\ ;\ \Pi^{NS}$

- if $\mathcal{O}(s) = a$ then
  $\Downarrow(\Delta_i) = \Theta^{NS} \ \ddagger\ [\ \Gamma \vdash e_2; e_3\ ]\ ;\ \Pi^{NS}$

This means that reduction rule SPEC is equivalent to either NS-SPEC-C-BR or NS-SPEC-A-BR reduction rules from the nonspeculative operational semantics, depending on the chosen oracle.

Rule READ-NOSPEC**.**

$$\begin{aligned}
\Downarrow(\Delta_{i-1}) &= o_j : [\ V\ ]\ ;\ \Theta^{NS} \ \ddagger\ [\ \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v]\ ]\ ;\ \Pi^{NS} \\
\Downarrow(\Delta_i) &= o_j : [\ V\ ]\ ;\ \Theta^{NS} \ \ddagger\ [\ \Gamma, v : V \vdash e[v]\ ]\ ;\ \Pi^{NS}
\end{aligned}$$

This reduction rule is equivalent to NS-READ.

Rule READ-SPEC-PROC**.**

Since this rule involves speculations, the lowering process has to take into account the outcome of the speculation, as illustrated in Definition 4.5.6.

- if $\mathcal{O}(s) = c$ then the oracle predicts that the speculation commits, in which case the process will not rollback its state.

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v][v]\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma, v : V \vdash e[v]\ ]\ ;\ \Pi^{NS} \end{aligned}$$

  In this case, the Read-Spec-Proc reduction rule is equivalent to the NS-Read reduction rule from the nonspeculative operational semantics.

- if $\mathcal{O}(s) = a$ then the oracle predicts that the speculation aborts, so the operations performed inside this speculation will be rolled back.

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \end{aligned}$$

In this case the Read-Spec-Proc reduction rule is equivalent to a no-op, as far as the nonspeculative model is concerned, because the speculative program will rollback its state and this operation is treated as if it never happened.

Rule Read-Spec-Obj**.**

- if $\mathcal{O}(s) = c$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v]\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma, v : V \vdash e[v]\ ]\ ;\ \Pi^{NS} \end{aligned}$$

- if $\mathcal{O}(s) = a$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v]\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v]\ ]\ ;\ \Pi^{NS} \end{aligned}$$

The Read-Spec-Obj reduction rule is equivalent to either the NS-Read nonspeculative rule or with a no-op, depending on the outcome of speculation s.

Rule Read-Spec-Same**.**

- if $\mathcal{O}(s) = c$ then

$$\begin{array}{lcl} \Downarrow(\Delta_{i-1}) & = & o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v]\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) & = & o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma, v : V \vdash e[v]\ ]\ ;\ \Pi^{NS} \end{array}$$

- if $\mathcal{O}(s) = a$ then

$$\begin{array}{lcl} \Downarrow(\Delta_{i-1}) & = & o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) & = & o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \end{array}$$

The Read-Spec-Same reduction rule is equivalent to either the NS-Read nonspeculative rule or with a no-op, depending on the outcome of speculation s.

Rule Read-Spec-Merge**.**

The two speculations, sP and sO, merge and a new speculation, s, is substituted for them. Since the oracle,$\mathcal{O}()$, is valid it must predict the same outcome for all three speculations.

- if $\mathcal{O}(s) = \mathcal{O}(sP) = \mathcal{O}(sO) = c$ then

$$\begin{array}{lcl} \Downarrow(\Delta_{i-1}) & = & o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v]\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) & = & o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma, v : V \vdash e[v]\ ]\ ;\ \Pi^{NS} \end{array}$$

The Read-Spec-Merge reduction rule is equivalent to the NS-Read nonspeculative rule.

- if $\mathcal{O}(s) = \mathcal{O}(sP) = \mathcal{O}(sO) = a$ then

$$\begin{array}{lcl} \Downarrow(\Delta_{i-1}) & = & o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) & = & o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \end{array}$$

In this case, the Read-Spec-Merge reduction rule is equivalent to a no-op, since the speculations will be aborted and all the actions performed since entering them will be equivalent to null.

Rule Write-NoSpec**.**

$$\begin{array}{lcl} \Downarrow(\Delta_{i-1}) & = o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{write}(V', o_j); e\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) & = o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash e\ ]\ ;\ \Pi^{NS} \end{array}$$

As expected, when no speculation is involved, this reduction rule is equivalent to NS-Write.

Rule Write-Spec-Proc**.** Since this rule involves speculations, we have to analyze two cases, which depend on the oracle we choose.

- if $\mathcal{O}(s) = c$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{write}(V', o_j); e\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash e\ ]\ ;\ \Pi^{NS} \end{aligned}$$

  In this case, the WRITE-SPEC-PROC reduction rule is equivalent to the NS-WRITE nonspeculative rule.

- if $\mathcal{O}(s) = a$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi \end{aligned}$$

In this case, the WRITE-SPEC-PROC reduction rule is equivalent to a no-op.

Rule WRITE-SPEC-OBJ**.**

- if $\mathcal{O}(s) = c$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{write}(V'', o_j); e\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V''\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash e\ ]\ ;\ \Pi^{NS} \end{aligned}$$

- if $\mathcal{O}(s) = a$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{write}(V'', o_j); e\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{write}(V'', o_j); e\ ]\ ;\ \Pi^{NS} \end{aligned}$$

The WRITE-SPEC-OBJ reduction rule is equivalent to either the NS-WRITE nonspeculative rule or with a no-op, depending on the outcome of speculation s.

Rule WRITE-SPEC-SAME**.**

- if $\mathcal{O}(s) = c$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash \mathit{write}(V'', o_j); e\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V''\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash e\ ]\ ;\ \Pi^{NS} \end{aligned}$$

- if $\mathcal{O}(s) = a$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \end{aligned}$$

The WRITE-SPEC-SAME reduction rule is equivalent to either the NS-WRITE nonspeculative rule or with a no-op, depending on the outcome of speculation s.

Rule WRITE-SPEC-MERGE**.** According to the definition of valid oracles, the oracle that we choose will provide the same outcome for the three speculations present in this rule, $s, sP,$ and sO.

- if $\mathcal{O}(s) = \mathcal{O}(sP) = \mathcal{O}(sO) = c$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash write(V'', o_j); e\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V''\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma \vdash e\ ]\ ;\ \Pi^{NS} \end{aligned}$$

- if $\mathcal{O}(s) = \mathcal{O}(sP) = \mathcal{O}(sO) = a$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \end{aligned}$$

The WRITE-SPEC-MERGE reduction rule is equivalent to either the NS-WRITE nonspeculative rule or with a no-op, depending on the outcome of speculation s.

Rules AB-OWNER**,** AB-PROC**, and** AB-FAIL**.**

All three reduction rules,AB-OWNER, AB-PROC, and AB-FAIL, are equivalent to no-ops when the $\Downarrow()$ is applied to the left and right hand side of the rules, since the oracle can only predict that the speculation has aborted, in which case we obtain the following.

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= \Theta^{NS}\ \ddagger\ [\ \Gamma' \vdash e'\ ]\ ;\ \Pi^{NS} \end{aligned}$$

This is expected, since the action of aborting a speculation cannot be translated to the nonspeculative operational semantics.

Rule AB-OBJECT**.**

Because the speculation that the object belongs to has been aborted the reduction rule is equivalent to a no-op.

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V'\ ]\ ;\ \Theta^{NS}\ \ddagger\ \Pi^{NS} \end{aligned}$$

Rule COMM-PEER**.**

This rule is equivalent to a no-op since the program executed by the process illustrated in the rule doesn't really advance to the next statement. The rule only changes the flag associated with the speculation. By applying the *lowers* to both sides of the rule the following nonspeculative rules are obtained, depending on the final outcome of the speculation.

- if $\mathcal{O}(s) = c$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= \Theta^{NS} \ddagger [\ \Gamma \vdash \mathit{commit}(); e\ ] \ ; \ \Pi^{NS} \\ \Downarrow(\Delta_i) &= \Theta^{NS} \ddagger [\ \Gamma \vdash \mathit{commit}(); e\ ] \ ; \ \Pi^{NS} \end{aligned}$$

- if $\mathcal{O}(s) = a$ then

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= \Theta^{NS} \ddagger [\ \Gamma' \vdash e'\ ] \ ; \ \Pi^{NS} \\ \Downarrow(\Delta_i) &= \Theta^{NS} \ddagger [\ \Gamma' \vdash e'\ ] \ ; \ \Pi^{NS} \end{aligned}$$

Rules Comm-Owner **and** Comm-Proc**.**

For both rules Comm-Owner and Comm-Proc their equivalents is rule NS-Commit, since the oracle can only predict that the speculation has committed, in which case we obtain the following.

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= \Theta^{NS} \ddagger [\ \Gamma \vdash \mathit{commit}(); e\ ] \ ; \ \Pi^{NS} \\ \Downarrow(\Delta_i) &= \Theta^{NS} \ddagger [\ \Gamma \vdash e\ ] \ ; \ \Pi^{NS} \end{aligned}$$

Rule Comm-Client**.**

The nonspeculative equivalent of this rule is a no-op. The lowering function simply discards the checkpoint belonging to a committed speculation.

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= \Theta^{NS} \ddagger [\ \Gamma \vdash e\ ] \ ; \ \Pi^{NS} \\ \Downarrow(\Delta_i) &= \Theta^{NS} \ddagger [\ \Gamma \vdash e\ ] \ ; \ \Pi^{NS} \end{aligned}$$

Rule Comm-Obj**.**

Because the speculation the object is part of has been committed this reduction rule is equivalent to a no-op.

$$\begin{aligned} \Downarrow(\Delta_{i-1}) &= o_j : [\ V\ ] \ ; \ \Theta^{NS} \ddagger \Pi^{NS} \\ \Downarrow(\Delta_i) &= o_j : [\ V\ ] \ ; \ \Theta^{NS} \ddagger \Pi^{NS} \end{aligned}$$

□

The next theorem relies on the definition of lifting a nonspeculative state a speculative state (Definition 4.5.9).

Theorem 4.5.12. For any nonspeculative reduction rule such that

$$\Delta^{NS}_{i-1} \Rightarrow^{NS} \Delta^{NS}_i,$$

there is a sequence of speculative reduction rules such that

$$\Uparrow(\Delta^{NS}_{i-1}) \Rightarrow^{*} \Uparrow(\Delta^{NS}_i)$$

Proof. The proof is done in a similar manner with the proof for Theorem 4.5.11, by considering each of the reduction rules in the nonspeculative operational semantics and showing that there exists

an appropriate sequence of transformations in the speculative operational semantics that satisfy the condition stated in the theorem.

Rule NS-SPEC-C-BR**.**

$$\begin{aligned} \Uparrow(\Delta^{NS}_{i-1}) &= \Uparrow(\Theta^{NS} \ \ddagger\ [\ \Gamma \vdash \mathit{speculate}(e_1 \oplus e_2); e_3\ ]\ ;\ \Pi^{NS}) = \\ &= \quad \ddagger\ \Theta\ \ddagger\ p_i : [\ \langle\ \rangle \mid \Gamma \vdash \mathit{speculate}(e_1 \oplus e_2); e_3\ ]\ ;\ \Pi \end{aligned}$$

$$\begin{aligned} \Uparrow(\Delta^{NS}_{i}) &= \Uparrow(\Theta^{NS} \ \ddagger\ [\ \Gamma, \mathcal{S} : \mathit{Spec}, \mathcal{A} : \{e_2; e_3\} \vdash e_1; e_3\ ]\ ;\ \Pi^{NS}) = \\ &= \quad \ddagger\ \mathcal{S}\ \ddagger\ \Theta p_i : [\ \langle(\mathcal{S}, \mathit{own})\ ,\ \Gamma\ ,\ e_2; e_3\rangle \mid \Gamma \vdash e_1; e_3\ ]\ ;\ \Pi \end{aligned}$$

This rule is equivalent to the SPEC reduction rule from the operational semantics of the speculative model.

Rule NS-SPEC-A-BR**.**

$$\begin{aligned} \Uparrow(\Delta^{NS}_{i-1}) &= \Uparrow(\Theta^{NS} \ \ddagger\ [\ \Gamma \vdash \mathit{speculate}(e_1 \oplus e_2); e_3\ ]\ ;\ \Pi^{NS}) = \\ &= \quad \ddagger\ \Theta\ \ddagger\ p_i : [\ \langle\ \rangle \mid \Gamma \vdash \mathit{speculate}(e_1 \oplus e_2); e_3\ ]\ ;\ \Pi \end{aligned}$$

$$\begin{aligned} \Uparrow(\Delta^{NS}_{i}) &= \Uparrow(\Theta^{NS} \ \ddagger\ [\ \Gamma \vdash e_2; e_3\ ]\ ;\ \Pi^{NS}) = \\ &= \quad \ddagger\ \Theta[\mathbf{aborted}]\ \ddagger\ p_i : [\ \langle\ \rangle \mid \Gamma \vdash e_2; e_3\ ]\ ;\ \Pi[\mathbf{aborted}] \end{aligned}$$

Lifting the right and the left hand side of this rule presents us with an interesting case. There is no rule that perfectly matches $\Uparrow(\Delta^{NS}_{i-1}) \Rightarrow \Uparrow(\Delta^{NS}_{i})$.

However, if we carefully look at the operational semantics for the speculative model we observe that $\Uparrow(\Delta^{NS}_{i-1})$ matches the left side of the SPEC reduction rule.

After applying the SPEC rule, we get the following state of the system:

$$\ddagger\ \Theta\ \ddagger\ p_i : [\ \langle(s, \mathit{own})\ ,\ \Gamma\ ,\ e_2; e_3\rangle \mid \Gamma \vdash e_1; e_3\ ]\ ;\ \Pi$$

To obtain $\Uparrow(\Delta^{NS}_{i})$ from the above state, the speculation has to be aborted, so applying the AB-OWNER reduction rule would take us to the desired state.

In conclusion, this nonspeculative reduction rule is equivalent to a reduction trace composed of two rules.

$\Uparrow(\Delta^{NS}_{i-1}) \overset{\text{SPEC}}{\Longrightarrow} \Delta \overset{\text{AB-OWNER}}{\Longrightarrow} \Uparrow(\Delta^{NS}_{i})$.

Rule NS-COMMIT**.**

$$\begin{aligned}
\Uparrow(\Delta^{NS}_{i-1}) &= \Uparrow(\Theta^{NS} \ \ddagger\ [\ \Gamma, \mathcal{S}: \mathit{Spec}, \mathcal{A}: \{e'\} \vdash \mathit{commit}(); e\ ]\ ;\ \Pi^{NS}) = \\
&= \mathcal{S}: \{p_i\} \ \ddagger\ \Theta \ \ddagger\ p_i : [\ \langle (\mathcal{S}, \mathit{own})\ ,\ \Gamma\ ,\ e' \rangle \mid \Gamma \vdash \mathit{commit}(); e\ ]\ ;\ \Pi
\end{aligned}$$

$$\begin{aligned}
\Uparrow(\Delta^{NS}_{i}) &= \Uparrow(\Theta^{NS} \ \ddagger\ [\ \Gamma \vdash e\ ]\ ;\ \Pi^{NS}) = \\
&= \quad \ddagger\ \Theta \ \ddagger\ p_i : [\ \langle\ \rangle \mid \Gamma \vdash e\ ]\ ;\ \Pi
\end{aligned}$$

This rule is equivalent to the COMM-OWNER reduction rule from the operational semantics of the speculative model.

Rule NS-READ**.**

$$\begin{aligned}
\Uparrow(\Delta^{NS}_{i-1}) &= \Uparrow(o_j : [\ V\ ]\ ;\ \Theta^{NS} \ \ddagger\ [\ \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v]\ ]\ ;\ \Pi^{NS}) = \\
&= \quad \ddagger\ o_j\,[\ \langle\ \rangle \mid V\ ]\ ;\ \Theta \ \ddagger\ p_i : [\ \langle\ \rangle \mid \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v]\ ]\ ;\ \Pi
\end{aligned}$$

$$\begin{aligned}
\Uparrow(\Delta^{NS}_{i}) &= \Uparrow(o_j : [\ V\ ]\ ;\ \Theta^{NS} \ \ddagger\ [\ \Gamma, v: V \vdash e\ ]\ ;\ \Pi^{NS}) = \\
&= \quad \ddagger\ o_j\,[\ \langle\ \rangle \mid V\ ]\ ;\ \Theta \ \ddagger\ p_i : [\ \langle\ \rangle \mid \Gamma, v: V \vdash e\ ]\ ;\ \Pi
\end{aligned}$$

This rule is equivalent to the READ-NOSPEC reduction rule from the operational semantics of the speculative model.

Rule NS-WRITE**.**

$$\begin{aligned}
\Uparrow(\Delta^{NS}_{i-1}) &= \Uparrow(o_j : [\ V\ ]\ ;\ \Theta^{NS} \ \ddagger\ [\ \Gamma \vdash \mathit{write}(V', o_j); e\ ]\ ;\ \Pi^{NS}) = \\
&= \quad \ddagger\ o_j\,[\ \langle\ \rangle \mid V\ ]\ ;\ \Theta \ \ddagger\ p_i : [\ \langle\ \rangle \mid \Gamma \vdash \mathit{write}(V', o_j); e\ ]\ ;\ \Pi
\end{aligned}$$

$$\begin{aligned}
\Uparrow(\Delta^{NS}_{i}) &= \Uparrow(o_j : [\ V'\ ]\ ;\ \Theta^{NS} \ \ddagger\ [\ \Gamma \vdash e\ ]\ ;\ \Pi^{NS}) = \\
&= \quad \ddagger\ o_j\,[\ \langle\ \rangle \mid V'\ ]\ ;\ \Theta \ \ddagger\ p_i : [\ \langle\ \rangle \mid \Gamma \vdash e\ ]\ ;\ \Pi
\end{aligned}$$

This rule is equivalent to the WRITE-NOSPEC reduction rule from the operational semantics of the speculative model.

□

To conclude the equivalence proof, we need to show next that for any distributed system, composed of processes and shared objects, as defined in Section 4.2.1, the speculative and the nonspeculative operational semantics defined in Section 4.2 and Section 4.4 are equivalent.

To show the equivalence of the two operational semantics we need to show first that for any *reduction trace* that takes a distributed system from an initial speculative state Δ_0 to a state Δ_n, by only applying reduction rules in the speculative operational semantics, there is an equivalent nonspeculative *reduction trace* that takes the nonspeculative state Δ^{NS}_0 to state Δ^{NS}_m, using only

reduction rules from the nonspeculative operational semantics. Furthermore, the initial and final states have to be equivalent under the lowering and lifting operations.

The reciprocal of the above statement concludes the proof, and it states that for any nonspeculative *reduction trace* there is an equivalent speculative *reduction trace* such that the initial and the final states are equivalent under the lifting and lowering operations.

Formally, this is summarized by the following theorem.

Theorem 4.5.13. The speculative and the nonspeculative operational semantics are equivalent under the lowering and lifting operations, as follows:

$$\forall \overrightarrow{\Delta} ::= \Delta_0 \;\Rightarrow\; \ldots \;\Rightarrow\; \Delta_n, \quad \forall \mathcal{O} \in \Omega(\overrightarrow{\Delta}),$$
$$\exists \overrightarrow{\Delta}_{NS} ::= \Delta_0^{NS} \;\Rightarrow^{NS}\; \ldots \;\Rightarrow^{NS}\; \Delta_m^{NS}, \quad \text{such that}$$

$$\Downarrow(\Delta_0) = \Delta_0^{NS} \;\wedge\; \Downarrow(\Delta_n) = \Delta_m^{NS},$$

and

$$\forall \overrightarrow{\Delta}_{NS} ::= \Delta_0^{NS} \;\Rightarrow^{NS}\; \ldots \;\Rightarrow^{NS}\; \Delta_m^{NS},$$
$$\exists \overrightarrow{\Delta} ::= \Delta_0 \;\Rightarrow\; \ldots \;\Rightarrow\; \Delta_n, \quad \text{such that}$$

$$\Uparrow(\Delta_0^{NS}) = \Delta_0 \;\wedge\; \Uparrow(\Delta_m^{NS}) = \Delta_n$$

Proof. The proof of this theorem relies on the statement made in Lemma 4.5.4 and it follows from applying Theorem 4.5.12 and Theorem 4.5.11 to each reduction rule in the speculative and nonspeculative *reduction traces*, respectively.

□

Chapter 5

Implementation

This chapter describes our implementation of the primitives for distributed speculative execution and our efforts to provide a transparent local and distributed rollback as part of the implementation.

We begin by discussing our initial work in implementing speculative execution as part of a compiler and we continue by presenting a kernel level implementation that is more general and efficient than our initial approach.

5.1 Background

The formal description of the speculative primitives makes them suitable to be implemented as extensions to existing programming languages. This approach calls for integrating them with existing compilers to provide the appropriate conversion to machine code.

There are two issues to consider when discussing this approach.

- What is the right programming language to extend with speculative primitives?
- Is there a compiler for that language that can naturally support the primitives used for speculative execution?

In our initial approach we answered these questions as follows:

- Use several languages, both imperative (C and Java) and functional (ML).
- Use an open source compiler collection that we developed as part of a previous project, called the Mojave Compiler Collection (MCC) [53].

5.1.1 MCC Overview

MCC is a multi-language compiler that compiles several languages: C, Java, Pascal, ML. It was used as a testbed to implement primitives for process migration and checkpointing. A compiler, and MCC in particular, is a perfect choice for testing new language primitives because:

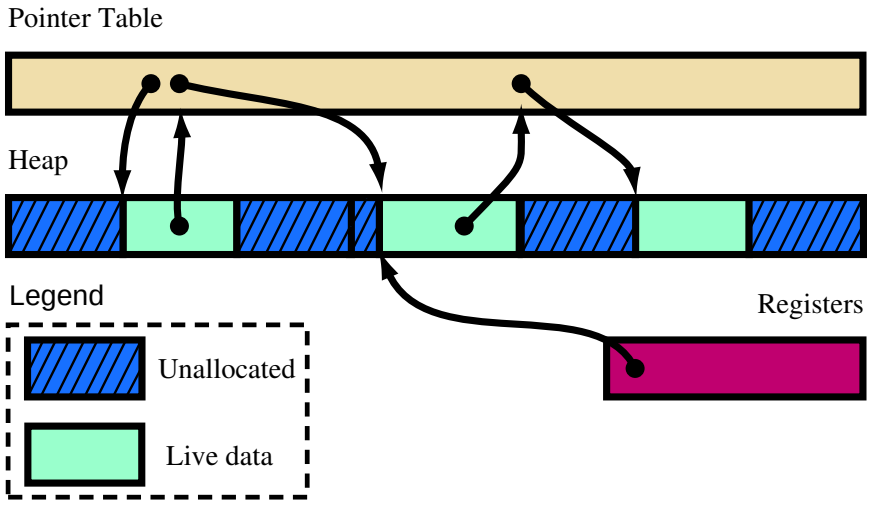


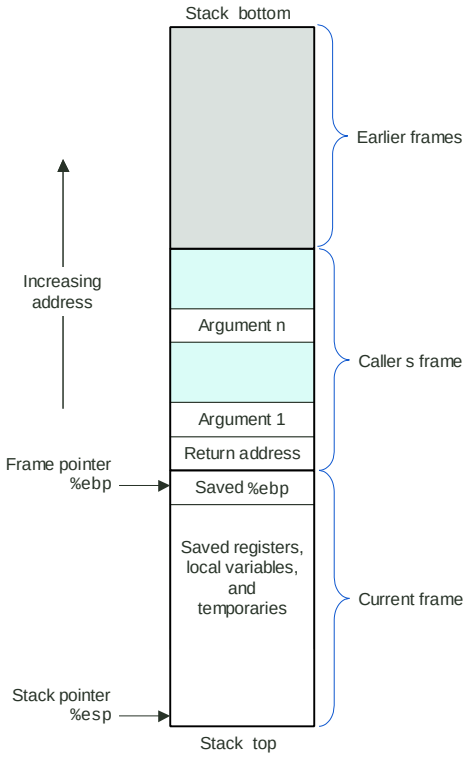
Figure 5.1: Pointer table representation

- it determines the way a process's state is organized,
- it can automatically generate code to manage the state of the process as required by the tested primitives,
- it requires minimal knowledge from the user.

Therefore, it seemed natural to extend the functionality of MCC with speculative primitives. MCC uses a common functional intermediate representation (FIR) [28, 53] to represent internally the various programming languages it compiles. This enables us to make the speculative primitives available to multiple programming languages by implementing support for them at the FIR level. Another advantage of MCC is its heap design that allows for easy support of speculative execution models, as described below.

5.1.2 Speculative Support in MCC

MCC's runtime provides the support needed by the speculative primitives. The runtime integrates speculative execution support with the garbage collector used by other parts of the compiler; the interaction with the garbage collector is is discussed elsewhere [53, Chapter 5]. The runtime also provides an architecture-independent representation of the state of the program. All the memory structures associated with the process's state are stored in a *heap*. Data blocks in the heap are tracked by a *pointer table*. These structures are shown in detail in Figure 5.1.



- The stack grows down in memory.
- A function call increases the stack by pushing the arguments of the function and the frame pointer (as depicted on the left).
- Execution inside a called function increases the stack by creating a new frame where local variables are saved.
- Returning from a function call pops the entries from the stack.
- Normal execution of a program may modify the stack in its entirety.
- To save the state of the stack a full copy of it is necessary.

Figure 5.2: The representation of a process in a stack-based compiler

5.1.2.1 Heap versus stack-based compiler

The choice to implement speculative execution in a heap-based, continuation passing style compiler, like MCC, versus a stack-based compiler is discussed next. Speculations require the ability to roll back the state of the process to a previous state. This requires taking a lightweight (in memory) checkpoint of the process. A stack-based compiler is one that represents the state of the process as a stack (see Figure 5.2). While this representation is simpler than using a heap it also reduces the kind of operations that can be performed on the state of a process. The operations that can be performed on a stack (push and pop) prohibit the efficient implementation of the speculative primitives. In a stack-based compiler the lightweight checkpointing needed by speculations requires copying the entire stack frame of a process to a different location in order to save its state. The full copy is needed because continuing the execution of the process could alter the entire contents of the stack.

By contrast, in a heap-based compiler the state of the process (represented by a heap) can be randomly accessed. This allows marking it as read-only at the time of speculation and use a copy-on-write mechanism when data in the heap is mutated. Further modifications can be represented as a new generation of the heap, as explained below.

Variable	Properties	*Description*
spec_next	$spec\_next \geq 0$	*Current speculation level*
base[*l*]	$l \in \{0..spec\_next\}$	*Base of heap corresponding to level l*
limit[*l*]	$l \in \{0..spec\_next\}$	*Limit of heap corresponding to level l*
ptr_diff[*l*]	$l \in \{0..spec\_next - 1\}$	*Pointer differential tables*
limit	$limit = limit[spec\_next]$	*Upper bound of the heap*
current	$current \geq base[spec\_next]$	*Current allocation point*

Figure 5.3: Speculation variables

5.1.2.2 Heap Support for Speculations in MCC

Speculation requires several state variables, described below and summarized in Figure 5.3. The heap layout and its relation to the state variables are illustrated in Figure 5.4. The base of the heap is at the bottom of the figure and the limit is at the top.

Our MCC implementation of speculative execution supports a simple model with nested speculations. Speculation uses the following state. The current level of speculation of a program is defined by *spec_next*. If $spec\_next = 0$, then there are no speculations active. Each speculation level $l \in \{0..spec\_next\}$ is delimited in the heap by a base pointer $base[l]$ and a limit pointer $limit[l]$. The youngest speculation level is maintained by *limit*, which is equal to $limit[l]$ and to $base[l+1]$ for that level. Each speculation level corresponds to a generation in the garbage collector, and the levels are strictly ordered, with $base[l] \leq base[l+1]$ for $l < spec\_next$.

In addition to the $base[l]$ and $limit[l]$ bounds, each speculation level, except the youngest, has a pointer difference table $ptr\_diff[l]$ that is used to restore the pointer table if speculation level $l+1$ is aborted. To minimize storage requirements, $ptr\_diff[l]$ is stored as a set of differences with the current pointer table *ptr_table*. When a block is copied due to a copy-on-write fault, the original block is added to the difference table $ptr\_diff[spec\_next - 1]$ and *ptr_table* is updated to point at the new copy.

The current allocation point is in *current*. At all times,

$$base[spec\_next] \leq current \leq limit.$$

5.1.3 Limitations of MCC's Support for Speculations

While the implementation of speculations in MCC was clean and the integration with MCC's internal representation of data was natural, the full support for distributed speculative execution required significant support from the operating system. According to the semantics we presented, the propagation of speculations in a distributed environment is done through either message passing or shared objects (or files). The following are required for a complete support of speculative execution:

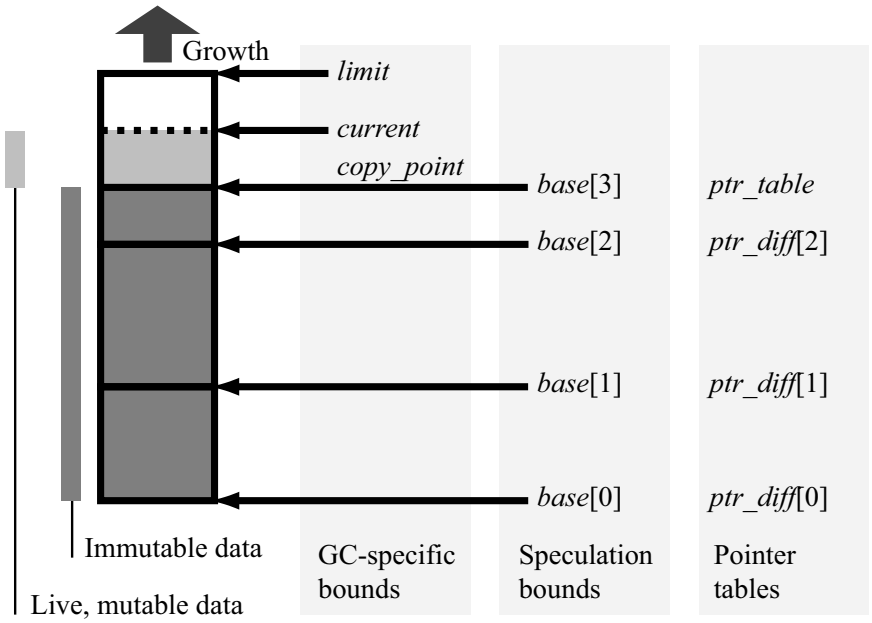


Figure 5.4: Heap data with multiple speculation levels

- Support for speculative message passing can not be fully implemented at the compiler level in a generic manner. This feature requires support from the kernel network stack.
- Speculative file system accesses need support for rollback logs in case speculations are aborted. The operating system needs to provide this functionality for an efficient implementation.
- A process may create or terminate the execution of child processes while it executes inside a speculation. In case the speculation is aborted these actions have to be undone. Supporting this functionality at the compiler level is hard and inefficient. Operating system support is again required for efficiency reasons.
- Propagating the outcome of speculations to all interested processes needs a communication infrastructure that compilers cannot provide.

Furthermore, as MCC is a research compiler it cannot compete with popular compilers, like *the GNU compiler collection*, in terms of speed of compilation and optimizations of the generated code. This makes our approach less appealing to production environments, making it harder to push our new programming model.

5.2 Kernel-level Implementation Overview

To address the limitations of MCC we have implemented speculations as an extension to the Linux kernel (version 2.6). This approach provides an increase in performance and it is more generic than

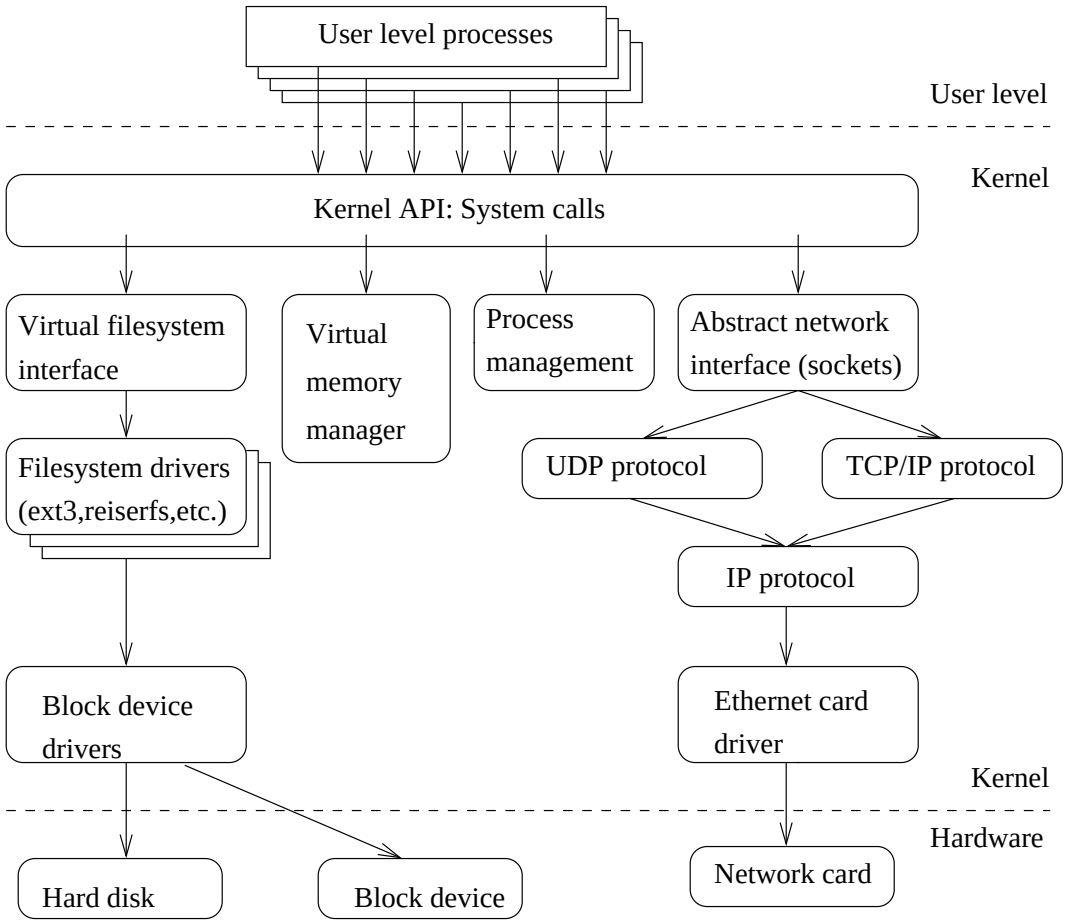


Figure 5.5: Interaction between user level processes and the Linux kernel.

the alternative compiler implementation discussed in [53].

Before presenting the details of the implementation performed as part of the Linux kernel, we introduce the architecture of the kernel, and the interface between user level applications and the kernel. A graphical representation is shown in Figure 5.5.

The core of the Linux operating system is the kernel. The function of the kernel is to manage user programs and to mediate access to the hardware. User programs , also called processes, run in user space and interact with the kernel through system calls, a pre-defined set of functions that provide access to the hardware and to certain data internal to the kernel. When a system call is executed by a process the control is passed from the user program to the kernel and execution is continued in kernel-space, in the context of the process. The kernel may also execute if a hardware interrupt is generated, in which case execution is dine in interrupt context.

The processes in the system are stored by the kernel in a doubly linked list called the task list. Each process in the task list is characterized by a data structure called the task struct, whose relevant components are shown in Figure 5.6. The *Process management* component of the kernel

(see Figure 5.5) handles the information associated with processes. It manages the creation and termination of processes, the parent-child relation between processes, and it schedules processes in the system for the use of the CPU. Its behavior is strongly intertwined with that of the *Virtual memory manager* component, which refers to the part of the kernel that handles functions like memory allocation, the copy-on-write memory mechanisms, and the page faults generated by process execution.

Figure 5.5 also presents a simplified overview of the network component and the filesystem component of the Linux kernel. The *Abstract network interface* component refers to the functionality provided by various system calls associated with networking. Some of these system calls translate in send and receive operations that use the underlying network transport protocols, like UDP and TCP. As data makes its way to the physical layer (the actual network), it may pass through code that implements the IP layer.

The *Virtual filesystem interface* provides an abstraction of the filesystem operations. It requires that various underlying filesystems, like ext3, or reiserfs, export a given interface, which standardizes the addition of new filesystems in the kernel. Each of the filesystem interacts with the hardware through the block device drivers.

The rest of this section describes the implementation by providing a parallel with the operational semantics rules described in Section 4.1.

5.2.1 Assumptions

In our prototype implementation we consider process execution and message passing to be speculative, but we do not include other external operations like printing documents, controlling external devices, or other operations that are difficult to roll back.

As a technicality, the kernel-to-kernel communication uses the IP multicast protocol. This requires that multicast is allowed by the routers serving the communication between the nodes that compose our system.

5.2.2 Implementation Details

5.2.2.1 Speculative primitives

The speculative primitive *speculate*, *abort*, and *commit* are implemented as three new system calls: *speculate()*, *spec_abort()*, and *spec_commit()*. The new system calls have been added through the standard procedure of updating the system calls table (arch/i386/kernel/entry.S).

A new data structure was added to the kernel to keeps track of the active speculations. This takes the form of a hash table called *speculations_hash*. The *task_struct* data structure, which describes the process control block of each process in the system, was updated with a new field called *speculation*.

The relevant parts of the *task_struct* are shown in Figure 5.6 along with the type of the new field. The *speculation* field keeps the information about the active speculation that the process belongs to. It includes the speculation id, the flags associated with the speculation, and accounting information related to the speculation, as described below.

The type of the *speculation* field, represented by the *spec_info* data structure, has the following data fields:

- pointers to two task_struct structures. The first one is used by the commit branch of the computation (the main process) to keep track of the abort branch that it creates inside the speculation call. The second one is used by the abort branch if it becomes active (as part of a speculation rollback), and it points to the main process that started the speculation.
- a wait condition used during the abort phase,
- a *flag* variable used to mark the ownership of the speculation or whether the speculation has been aborted or not, and
- a speculation header (*shdr*) that is used to tag outgoing speculative messages.

The type of the *shdr* field is a new record data structure that was introduced to abstract out the interaction with the communication layer. The structure of the new record type *spec_header* is shown in more detail in Figure 5.7. The fields composing it are as follows:

- the number of the IP option used to mark the speculative header.
- the length of the IP option. The first two fields are used by the IP networking layer to add the speculation header information to the outgoing network packets sent by the speculative process.
- the speculation id.
- the multicast address used for the Publish/Subscribe mechanism for control messages.

5.2.2.2 Local Management of Speculations

The local management of speculations is performed by a new kernel thread that was added to the system. The purpose of this thread is to manage local speculations and to process incoming $ABORT/COMMIT/MERGE$ messages and act upon each according to the behavior specified by the operational semantics. The kernel thread runs in a loop listening for incoming messages on a certain number of Publish/Subscribe channels. Upon receiving a message associated with a speculation it knows of, it forces local processes involved in that speculation to abort or commit, or it performs the merger of two existing speculations, as needed. To implement the Publish/Subscribe mechanism

```
struct task_struct {
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        ...
        struct list_head tasks;
        ...
/* task state */
        ...
        pid_t pid;
        pid_t tgid;
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->parent->pid)
         */
        struct task_struct *real_parent; /* real parent process (when being debugged) */
        struct task_struct *parent;     /* parent process */
        /*
         * children/sibling forms the list of my children plus the
         * tasks I'm ptracing.
         */
        struct list_head children;  /* list of my children */
        struct list_head sibling;   /* linkage in my parent's children list */
        struct task_struct *group_leader;       /* threadgroup leader */

        /* PID/PID hash table linkage. */
        struct pid pids[PIDTYPE_MAX];
        ...
/* signal handlers */
        ...
        struct sigpending pending;
        ...
#ifdef CONFIG_SPEC_EXEC /* State related to speculations */
        struct spec_info speculation;
#endif
};

struct spec_info {
        /* pointer to the abort branch */
        struct task_struct *abort_thread;
        /* pointer to the commit branch */
        struct task_struct *commit_thread;
        /* wait for the completion of the initial thread in case of an abort */
        /* also used to prevent exiting if speculation's outcome is undecided*/
        struct completion *onabort;
        /* flags describing the properties of the speculation */
        unsigned long flags;
        /* speculation header */
        struct spec_header shdr;
};
```

Figure 5.6: Relevant information from the process control block (task_struct) and the new data structures introduce to keep track of speculation information.

```
/* the speculation header that is sent along with various messages */
struct spec_header {
   char ipopt_header; //=0x19;
   char ipopt_len; //=sizeof_spec_ipoption;
   struct spec_id id;
   struct in_addr mcast_addr;
};

/* the speculation id */
struct spec_id {
   pid_t pid; /* the pid of the process initiating the speculation */
   unsigned long jiffies;
};
```

Figure 5.7: The speculation header sent as part of the new introduced IP option and the speculation id data structures

mentioned in Section 4.1 for the control messages we use IP Multicast. The kernel thread uses a kernel-level socket to monitor the control messages. The socket is subscribed to the multicast addresses corresponding to the speculations in which local processes are currently involved. More details on how the multicast groups are created are presented in Section 5.2.2.3

5.2.2.3 Starting a Speculation

When a user process enters a new speculation by calling the *speculate()* call, it needs to create a checkpoint, as described by the SPEC rule. A new user process is created using a modified *do_fork* system function and the copy-on-write mechanism associated with it. Figure 5.8 shows the creation of the new process. The new process corresponds to the abort branch of the speculation and it represents the *lightweight checkpoint*.

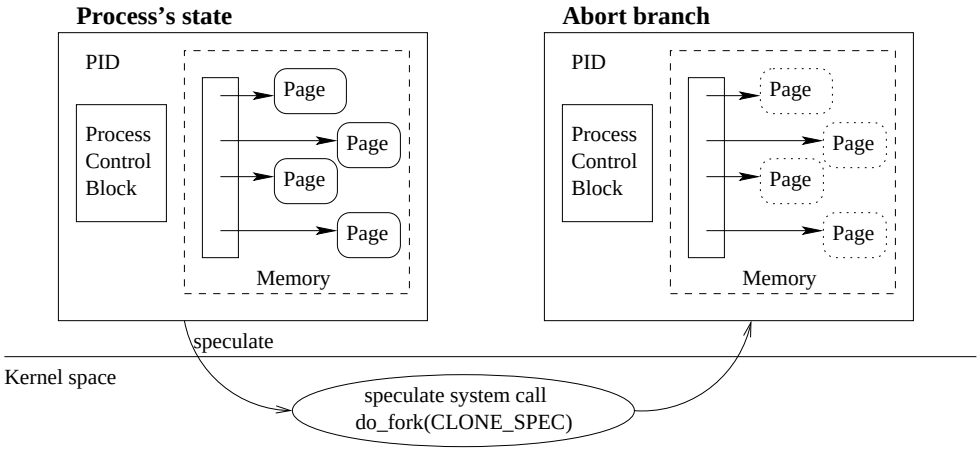


Figure 5.8: The speculate call uses the do_fork() function to create the abort branch of the speculation. The abort branch is used only if the speculation is aborted.

The copy-on-write mechanism associated with the do_fork() function provides the means to save the *lightweight checkpoint*. Instead of copying the entire state of the system, which could be very costly, it copies memory pages only when the original is modified. This prevents unnecessary copying and preserves the original state of the process in the abort branch (see Figure 5.9).

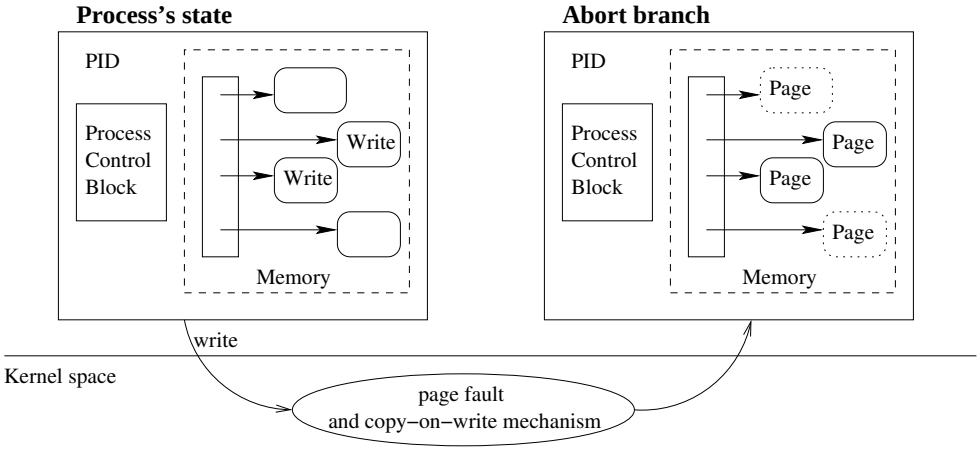


Figure 5.9: The copy-on-write mechanism associated with the do_fork() function copies memory pages only when the original is modified. This prevents unnecessary copying and preserves the original state of the process in the abort branch.

The abort branch is not put on the run queue, as is the case with processes created using the standard *do_fork* function. The abort branch is blocked and its handler (the task_struct associated with it) is stored in an internal queue of the main process. It is only used if the speculation is aborted, as described in Section 5.2.2.5.

The *do_fork* function (located in kernel/fork.c) is called by all system calls that need to spawn a new process (like fork, vfork, clone). Its behavior is different based on the parameters that are passed to it. The function was enhanced with actions to support new parameters specific to speculative execution.

The standard *do_fork* function creates a new task_struct (include/linux/sched.h) and populates it by calling the *copy_process* function (kernel/fork.c). The *copy_process* function is responsible for copying the various components of the *task_struct* associated with the main process (like the page table entries, etc.) to the process control block defining the newly created process. It also sets the point where the execution of the newly created process is resumed and what value is returned when it resumes execution. The return point is traditionally set to the *ret_from_fork* label in the arch/i386/kernel/entry.S file.

The modifications to the *do_fork* function include updating the *speculation* fields in the *task_struct* associated with both the initiator of the speculation and the newly created abort branch. The update for the initiator contains the new speculation id, and the flag *SPEC_OWNER*.

As mentioned before, IP multicast is used for kernel-to-kernel communication. The *ABORT/*

COMMIT/MERGE control messages associated with a speculation are sent using this method. A set of unused (unassigned) multicast addresses was reserved for this purpose. When a speculation is created the speculation identifier is hashed to one of the multicast addresses. The socket of the kernel thread that manages local speculations is subscribed to the multicast group and will receive future control messages. The hashing function is presumed unique across the distributed system. This enables processes that are absorbed in speculations to subscribe to the same multicast group as the initiator of the speculation. Since the volume of messages is very small, it is feasible to have the one socket subscribed to several multicast groups and perform the required bookkeeping at the same time.

5.2.2.4 Committing a Speculation

When a speculative process that has started a speculation calls the *commit()* system call it triggers two actions, as described by the operational semantics. First, it might have to release the local checkpoint. Second, it announces that it committed the speculation.

If the process is the only owner of the speculation it can fully commit the speculation. It discards its locally saved checkpoint, as described in the COMM-OWNER operational semantics rule. If the process co-owns the speculation and it is not the last to decide on its outcome, then the *commit()* system call blocks the process until the outcome is decided by the other co-owners (COMM-PEER rule).

The checkpoint is discarded by releasing the process (*abort branch*) that was created upon speculating. The procedure to correctly discard it follows the following steps:

- Add a kill signal to the pending signals of the abort branch.
- Change the status of the abort branch from STOPPED to RUNNING.
- Schedule the abort branch to be executed after the commit branch is done with its time slot.
- Remove the speculation from the local hash.

The announcement of the commit operation is done by sending a multicast *COMMIT* message on the multicast address associated with the speculation. Upon receiving this message, the processes that were absorbed in the speculation, or waiting for the outcome of the speculation, discard their checkpoint and continue execution. This models the behavior described by the COMM-PROC rule.

5.2.2.5 Aborting a Speculation

When the initiator of a speculation decides to abort it it broadcasts the *ABORT* message to the corresponding multicast group and it rolls back its state to what it was just before the speculate

call was executed (see AB-OWNER). Upon receiving the ABORT message other co-owners of the speculation and process that were absorbed in the speculation will also have to roll back their states.

The rollback is done by reviving the child process (*abort branch*) created during the speculate call and forcing the current process to exit. We also need to preserve the PID of the initial process.

The actions required to roll back are summarized next.

- Reparent the child processes of the running process to the abort branch. Child processes that were created during the speculation are terminated. Child processes that were alive when the speculation was started but tried to exit during the commit branch are restored to the state they had before the speculation was entered.
- Reparent the abort branch to have the same parent as the running process.
- Wake up the abort branch and schedule it for execution.
- Wait for the abort branch to become a real running process and then force the current process to exit.
- The abort branch does the initial start-up, notifies the main process when that is done, and then waits for the main process to reach the point where it is about to release its PID.
- The abort branch "switches" PIDs with the main process and they both continue execution. The main process finishes its exit procedure, while the abort branch becomes the new main thread of execution for the process.
- Release and relink entries in the *proc* filesystem.

Execution of the abort branch resumes at the *speculate* call. A different value is returned than in the case of the initial invocation and execution may continue on a different branch.

5.2.2.6 Speculative Communication

Before describing the mechanisms involved in supporting this implementation we give a brief overview of the network stack and its implementation in the Linux kernel.

The Open Systems Interconnection Reference Model (OSI Model for short) is a layered abstract description for communications and computer network protocol design. Its seven layer design is illustrated in Figure 5.10. This model is reflected in the implementation of the Linux kernel network stack as shown in Figure 5.10. The kernel implements layers 2 (Data Link Layer) through 5 (Session Layer). Figure 5.10 also shows the parallel between the OSI layers and the layers from the Linux kernel network stack. The protocols shown are those of interest to our implementation.

Each network layer encapsulates the data of the above layer and it attaches to it its own header information, as presented in Figure 5.11.

Layer	Function	Parallel with Linux network stack
7	Application	
6	Presentation	
5	Session	TCP
4	Transport	TCP or UDP
3	Network	IP (Internet Protocol)
2	Data Link	Ethernet
1	Physical	

Figure 5.10: The OSI Model and the Linux network stack

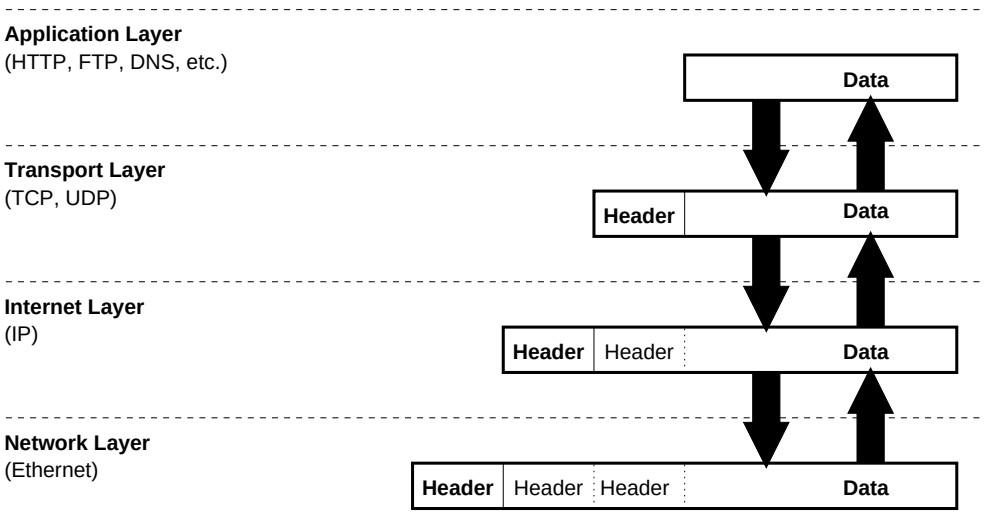


Figure 5.11: Data encapsulation between various network layers

To make the implementation of speculative messages totally transparent to the user level application and to allow it to co-exist with non-speculative communication we needed to find a non-intrusive way to tag speculative communication.

After analyzing the data encapsulation implemented by each protocol in the kernel network stack, we decided that the best place to introduce the speculative information was inside the IP layer. This enables support for both the TCP and the UDP transport protocols without requiring significant modifications to the kernel code that implements them.

The format of the IP datagram is shown in Figure 5.12. The only possibility to attach extra information to the transmitted information at the IP layer is through the IP options. We introduce a new IP option (*IPOPT_SPEC*) that is attached to all outgoing messages. This approach is non-invasive as far as the sending and receiving of messages are concerned, and it is general enough to

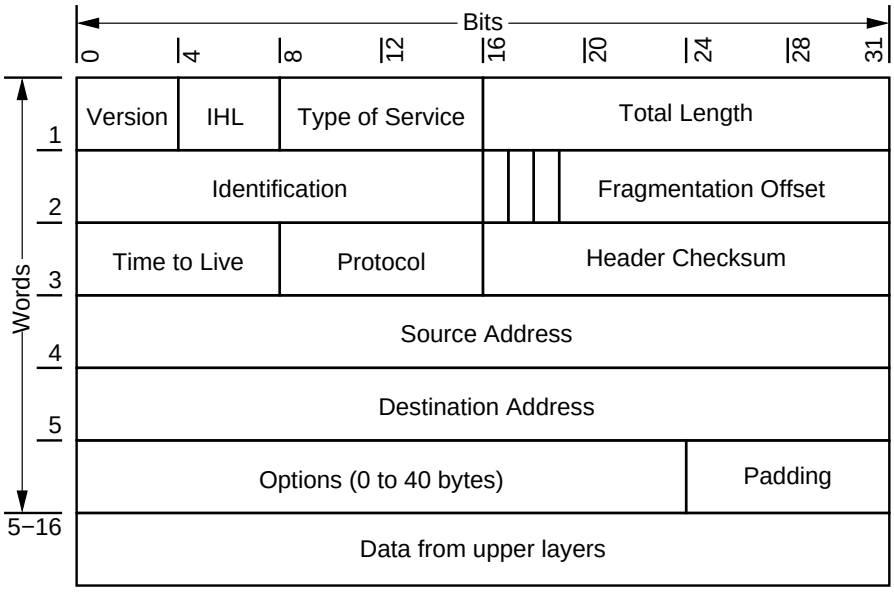


Figure 5.12: The layout of the IP header encapsulated by IP datagrams

allow speculative processes to communicate to non-speculative aware entities if they knowingly and willfully choose to do so. If a router or an end-node receives messages containing IP options that are unknown to them they usually ignore the option and still deliver the message to the destination. The changes required to support a new IP option (*IPOPT_SPEC*) at the kernel level is presented next.

5.2.2.7 Sending a Speculative Message

When a speculative process sends a message to a remote location the message needs to be tagged as speculative. This requires attaching the speculation information to it.

The kernel modifications required to support this mechanism are as follows:

- Create a new IP option (*IPOPT_SPEC*) and add it to the file containing the definitions of the other IP options (*include/linux/ip.h*).

- When a speculative process sends a message using the standard *sendto/send* interface it uses the *ip_setsockopt* to enable the IPOPT_SPEC option for the socket. This forces the addition of the IPOPT_SPEC option to all outgoing messages. The *ip_setsockopt* (net/ipv4/ip_sockglue.c) function is modified to recognize the new option and to act upon being passed that as a parameter.

- The *ip_options_compile* (net/ipv4/ip_options.c) function is modified to recognize and process the new IP option. The processing involves adding the speculative header defined in the process control block to outgoing IP datagrams sent by the process.

5.2.2.8 Receiving a Speculative Message

Receiving and processing speculative messages requires several modifications in the kernel network stack as well. The receipt of the message from the network and the delivery of message to the user level process are two separate actions. They may occur at different moments in time, with the receipt preceding the delivery. Also, the code that implements these actions in the network stack is different and lives at different layers.

When a message is received from the data layer it passed to the Network Layer. The Ethernet header is stripped off and the rest of the data is passed to the Internet Layer, as shown in Figure 5.10. At the Internet Layer the IP header is stripped off, the IP options are processed and the rest of the message is sent up to the Transport Layer. The data is stored at the Transport Layer until a *recv/recvfrom* system call is performed. When that happens the data is delivered to the application. It is only when data is delivered to the application that the application should become speculative if the message was speculative.

Since data and the various headers part at different layers of the network stack both the Transport layer and the Internet layer required modifications to support speculative execution. At the IP layer the information associated with the IPOPT_SPEC option needs to be preserved and paired with the data until the data is delivered to the application. This information is used to force the receiving process in a speculation when the data contained in the message is passed on to the user-level application.

The modifications required at the level of the transport layer (both UDP and TCP) are performed on the receive message functions of each of the two protocols: *udp_recvmsg* and *tcp_recvmsg*, respectively. Upon delivering a message that has speculative information associated with it the recvmsg function invokes the code required to absorb the receiver into the speculation. This code is shared with the *speculate* function. The *flag* associated with the speculation in the process control block marks the fact that the speculation is implicit, rather than owned by the process.

5.2.2.9 Implicit Speculations

From the point of view of a process that is absorbed in a speculation, the acts of entering, committing or aborting the speculations must be completely transparent. A process is absorbed in a speculation if it receives a speculative message. If the speculation is committed by its initiator, then the process discards the checkpoint that it created when it was absorbed. If the speculation is aborted by its initiator it forces the absorbed process to roll back its state as well. Unlike in the case of explicit speculations, when the execution of the program continues by returning from the speculate call with a different value, the absorbed process has to retry the receive call that absorbed it in the speculation.

```
x=malloc(size_of_array);
if (test_with_nonspec_accesses)
   assign_values(x, percentage);
if (test_nonspec) {
   assign_values(x, percentage);
   exit(0);
}
if (speculate()==0) {
   /* the commit branch */
   if (test_commit) {
      assign_values(x, percentage);
      spec_commit();
      exit(0);
   } else
      spec_abort();
} else {
   /* the abort branch */
   assign_values(x, percentage);
}
```

Figure 5.13: Skeleton of the benchmark program

When the kernel thread that manages local speculations receives notification that a certain speculation has been aborted or committed it needs to force processes that were involved in the speculation to automatically roll back their state or commit it. To achieve this it modifies the flag associated with the speculation in the control block of the processes and it sends each of them a signal ($SIGURG$) that forces them to re-evaluate the flag. The choice for the SIGURG signal was made since the signal was declared unused and ignored in the kernel code. The signal handler processes the signal the next time the process executes and, depending on the outcome of the speculation, it performs the actions required by the abort or the commit operation.

As discussed above, in the case of implicit speculations the processes are absorbed during the recvfrom system call. The kernel thread instruments the code of the program so that the system call is repeated when the process rolls back. The instrumentation is done when the abort branch is created and it involves setting a signal that, when processed, forces the kernel to retry the system call.

5.3 Synthetic Experimental Results

The set of experiments presented in this section are aimed at measuring the maximum overhead incurred for executing inside speculations. It is meant to be a guideline for programmers who want to include speculative execution in their software, rather than an absolute benchmark. We feel that the real overhead of speculations depends strongly on the application. As it will become clear from describing our testing framework, the numbers we present measure the maximum overhead of

Nonspeculative

```
x=malloc(size_of_array);
assign_values(x, percentage);
```

Commit branch

```
x=malloc(size_of_array);
if (speculate()==0) {
   /* the commit branch */
   assign_values(x, percentage);
   spec_commit();
}
```

Abort branch

```
x=malloc(size_of_array);
if (speculate()==0) {
   /* the commit branch */
      spec_abort();
} else {
   /* the abort branch */
      assign_values(x, percentage);
}
```

Figure 5.14: Skeleton of the benchmark program

speculation.

5.3.1 The Testing Setup

The setup used for the tests presented in this section is as follows. We use one Pentium 4 machine with a 2.2 Ghz CPU and 512Mb of RAM.

The skeleton of the testing program is presented in Figure 5.13. The program manipulates the information in an array (x) of a given size. The program is passed a series of flags that determine the execution flow of the program. The meaning of each flag is as follows:

- *test_with_nonspec_accesses* is set to true when we measure the overhead of speculations provided that some entries in the array are accessed before entering the speculation.
- *test_nonspec* is set to true when we test a pure non-speculative program.
- *test_commit* is set to true when we test a speculative program that does work on the commit branch and then successfully commits the speculation.
- *test_abort* is set to true when we test a speculative program that aborts the speculation and does work on the abort branch.

The *assign_values* function assigns different values to a specified percentage of the entries in the array and it also reads the information that was written. The parameter taken by the function marks the *mutation percentile* of data. We used two versions of the function in separate sets of tests. In the first version the entries were chosen in a random manner, while the second version modified entries sequentially, starting from a random position in the array. The results observed in both cases were consistent. The graphs presented in this section display only the results obtained from using the first version of the function.

It is important to keep in mind that due to the totally random access patterns in the *assign_values* function, the actual percentage of the memory pages associated with array x that are touched during the test is greater than the mutation percentile passed as a parameter to the program.

5.3.2 Overhead of Executing Inside a Speculation

The first test measures the overhead of executing inside each branch of a speculation in isolation versus a non-speculative execution. The three instances of the program that we compare are shown in Figure 5.14. The test compares the time spent in the *assign_values* function for three different cases. Each case is represented by passing a different set of parameters to the benchmark program presented in Figure 5.13. The *test_with_nonspec_accesses* parameter is set to false in all cases. We average the running time over 100 executions. The program is executed on a freshly started operating system. to prevent caching contamination.

The first instance of the test program measures the time for a traditional non-speculative execution. The *test_nonspec* parameter was set to true, while all the others were set to false. The second instance measures the time inside the *commit branch*, and it only had the *test_commit* parameter set to true. The third instance measures the time for executing on the *abort branch* (*test_abort* was set to true).

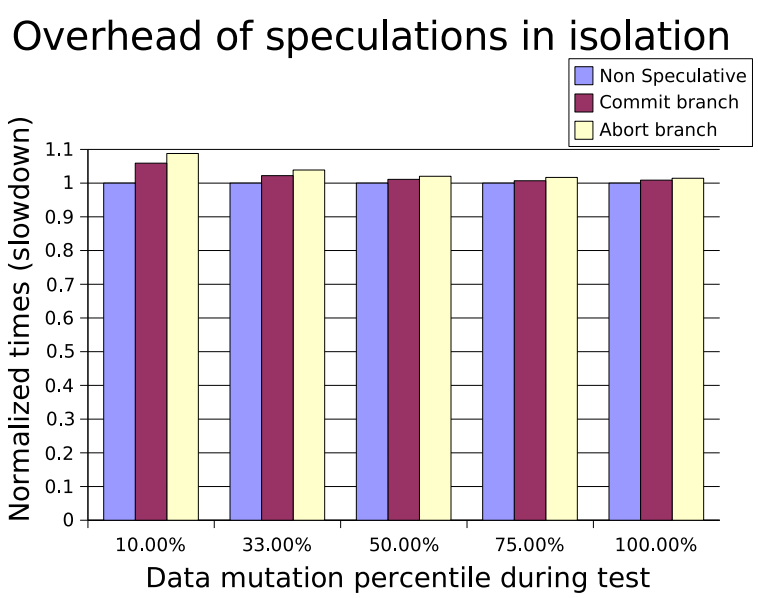


Figure 5.15: The overhead of randomly accessing the entries of an array of 128Mb for the first-time using various mutation percentiles. The three cases are: nonspeculative, inside the commit branch, inside the abort branch.

The results obtained for an array size of 128Mb and for various mutation percentiles of the data are presented in Figure 5.15. The running times are normalized using as a reference the measured time of the non-speculative program. The times measured by the test includes the time to create the accessed memory pages of the array in all cases. For the *commit* branch there is also the cost of the copy-on-write mechanism used by the *fork* system call. However, since the entries are initially empty the overhead is minimal. The results show a less than 10% overhead of executing inside speculations

Nonspeculative

```
x=malloc(size_of_array);
assign_values(x, percentage);
assign_values(x, percentage);
```

Before and on commit branch

```
x=malloc(size_of_array);
assign_values(x, percentage);
if (speculate()==0) {
   /* the commit branch */
   assign_values(x, percentage);
   spec_commit();
}
```

Before and on abort branch

```
x=malloc(size_of_array);
assign_values(x, percentage);
if (speculate()==0) {
   /* the commit branch */
      spec_abort();
} else {
   /* the abort branch */
      assign_values(x, percentage);
}
```

Figure 5.16: Skeleton of the benchmark program

for the 10% mutation percentile. This overhead is amortized as more entries are modified. This is mainly due to the random nature of our access pattern that may touch a larger number of memory pages than the specified percentile.

We also observed an anomaly in that the running times recorded for the accesses performed on the abort branch were slightly higher than those performed on the commit branch. We believe this is a consequence of the way *fork*'s copy-on-write mechanism interacts with the virtual memory page-faulting mechanism for accessing pages that have not been allocated before.

5.3.3 Speculation Overhead with Initial Nonspeculative Accesses

In our second round of testing we combine the data accesses inside each branch of a speculation with a set of accesses performed before the speculation is started. This takes away some of the cost of allocating a memory page for the first time, but it puts more stress on the copy-on-write mechanism which has to perform backups of the pages modified inside the speculation. For this testing round the *test_with_nonspec_accesses* parameter was set to true in all cases. The other parameters were set in the same manner as in the first set of tests. The skeletons of the test programs are shown in Figure 5.16.

Again, we use an array size of 128Mb and we run the test for various mutation percentiles. In this setup we measure the total running time of the program, rather than the time spent inside the *assign_values* function. We introduce a new mutation percentile of 1% to illustrate an exceptional case where the distribution of the random accesses makes it such that a significantly larger percent of the memory pages is accessed and the overhead is slightly increased. The complete results are shown in Figure 5.17. The maximum slowdown observed was 21%.

5.3.4 Cost of Speculative System Calls

The last results measure the cost of each of the speculative primitives: entering the speculation, performing the commit operation and performing the abort operation. The numbers were averaged over 100 runs and were measured for a program where data is created and accessed before the speculation is started, data is modified in its entirety inside the speculation, and the speculation is

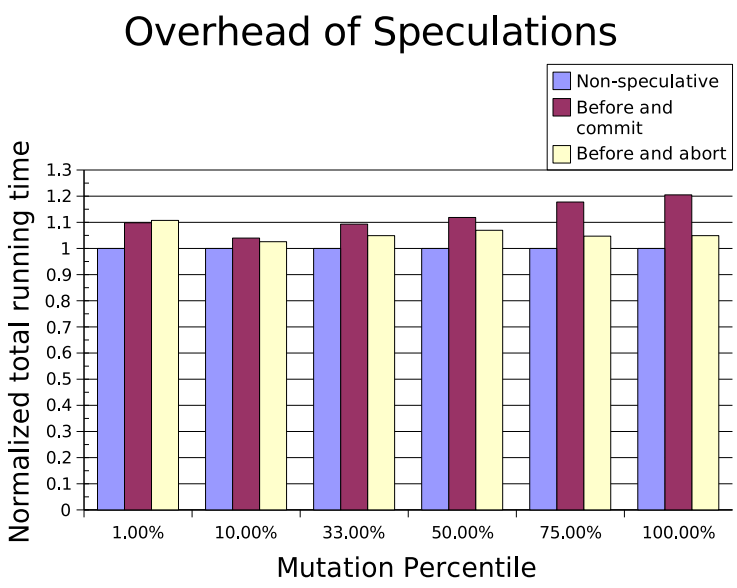


Figure 5.17: The overhead of randomly modifying various percentages of an array of 128Mb. The array is modified before the speculation is started in all cases. The array is also modified outside the speculation, on the commit branch and on the abort branch, respectively.

either committed or aborted. The results are shown in Figure 5.18.

Op. \ Size	100kb	32Mb	64Mb	128Mb
Speculate	708	751	857	1727
Commit	412	704	1334	1733
Abort	950	1786	1807	1866

Figure 5.18: Cost of speculative operations (in μs).

For comparison we present in Figure 5.19 sample times for several system calls as well as the context switch time for various numbers of processes of the same size running in parallel. The results were collected using the LMbench [39] tool.

System call	getpid()	stat()	open()/close()
Time	0.2063	23.0667	28.6804

Context switch				
# proc \ Size	10kb	32Mb	64Mb	128Mb
2	2.05	17949	35958	72029
5	2.55	17989	35972	///////
10	3.30	17989	///////	///////

Figure 5.19: Cost of other system calls and that of context switch time (in μs). The grayed-out entries were eliminated because they were significantly affected by swapping.

Chapter 6

Support for Speculations in a Distributed Filesystem

Complete support for speculative execution requires both speculative communication and speculative I/O. To complement the speculative message passing implementation presented in Chapter 5, we chose to support speculations as part of a distributed filesystem. The distributed filesystem implementation maps to the speculative distributed objects system model presented in Section 4.2, with files being mapped to shared objects.

Most of the commonly used filesystems for distributed environments, like NFS [45], do not provide the desired level of reliability, fast access to remote data, transparency or guarantees with respect to data consistency in the presence of concurrent access.

At the least, this roadblock unnecessarily complicates the development process for highly distributed applications. Frequently, this adds to the complexity of development by requiring programmers to employ complex synchronization and communication mechanisms, leading to the introduction of subtle bugs in the software that are difficult to track down and eliminate.

We propose a new distributed filesystem, MojaveFS, that supports speculative execution and addresses the issues of reliability and scalability and that provides location transparency and sequential consistency with respect to data access. Support for speculative execution is provided by a generational logging mechanism similar to that implemented in MCC (see Figure 5.4). In order to address the issue of reliability, our filesystem introduces a degree of redundancy in the distribution of data, replicating the data for each file among multiple physical servers. For performance and scalability, we use a variant of well-known [13, 10] filename hashing schemes to distribute the load over a set of data servers; we enhance the traditional approach by replacing single servers with virtual server groups that communicate using a variant of totally-ordered group communication [58]. In order to support location transparency, MojaveFS uses a global namespace for files such that each file and directory in the filesystem has a globally-unique identifier. The filesystem uses a group communication protocol that guarantees a causal order of messages in order to provide sequential

consistency.

The next section describes the design of our distributed filesystem. Section 6.2 discusses the architecture of the system and some of the implementation-specific details.

6.1 MojaveFS Design

6.1.1 Speculation Support in MojaveFS

The *speculate* system call signals to MojaveFS that the process is entering a new speculation. On this call, MojaveFS iterates over all file descriptors owned by the process, and enters an atomic speculation on each file that is on a MojaveFS partition. MojaveFS may return a list of descriptors on which it *cannot* enter a speculation, like files on non-MojaveFS partitions. It is up to the user or a user level library, like MCC's runtime library, to filter I/O on these "uncontrolled" descriptors. MojaveFS will log all actions on the remaining descriptors until the speculation is committed or rolled back.

The mechanism for supporting nested speculations is the following. For each speculation level a separate log is maintained, similar to the generations of the heap presented in Figure 5.4 in Section 5.1. Until the speculation level created by the *speculate* call is committed or rolled back, MojaveFS must intercept any call to open a new file on MojaveFS partitions and make sure that I/O operations on the file are logged until the speculation is completed. Also, MojaveFS defers any call to close a file descriptor until the speculation is completed. This requires MojaveFS to keep track of some state information specific to the process in addition to the logs maintained for each descriptor.

The *commit_spec* system call signals to MojaveFS that all logs associated with a particular speculation level can be folded into the parent level. If the speculation is the outermost speculation, then all logs for that level can be committed to the file system. Otherwise, the changes are applied to the parent level's logs. The *commit_spec* call should close any descriptors that were closed within the speculation but deferred by MojaveFS.

The *abort_spec* system call signals to MojaveFS that all logs associated with a particular speculation level, and all later levels, should be discarded. The MCC runtime indicates which level should be rolled back. Since MojaveFS does not modify the original files in place, it can simply delete all logs associated with the indicated level and all later levels.

6.1.2 The Distribution Component of MojaveFS

We assume that our distributed system contains at least the following two types of nodes. First, there are *data clients*; these are the consumers (and possibly producers) of data. Second, there are *data*

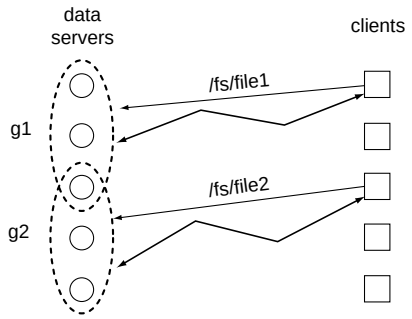


Figure 6.1: Each filename maps to a virtual data server group.

servers; these nodes store and maintain the data and communicate it to the clients. Data servers are organized into *virtual server groups.* [1] When clients access files, we employ a hashing mechanism to map filenames to virtual server groups. The high-level architecture is shown in Figure 6.1.

A virtual server group may contain several physical servers, and any physical server may belong to more than one virtual server (physical servers can be added to or removed from these virtual servers dynamically). Since the physical servers that belong to a virtual group may, in general, overlap with the servers of another group, the number of virtual servers can increase dramatically. This can significantly decrease the effect of hash collisions used by our hashing mechanism.

Also, the ability to configure virtual server groups dynamically is particularly attractive from a system administrator's perspective. An individual machine can be hardened against failures by using redundant network cards, power supplies, RAID arrays, and so on, but if an operating system update needs to be applied then that machine will still have to be taken offline. With the ability to add or remove machines at will, however, taking a system offline to perform maintenance is not an issue. In addition, the need to harden individual machines against failures is also reduced, enabling the use of inexpensive off-the-shelf components.

All members of a virtual server group replicate the entire set of data that is mapped by our hashing function to that virtual server. Once one decides to distribute redundant copies of data objects among multiple servers, load balancing among those servers becomes a possibility. A simple way of distributing the load in our system is to program clients such that, when a client resolves a data object to a virtual server group, a client chooses a random server from that group to interact with. This simple approach helps to exploit the redundancy inherent in the system. However, more can be done.

If a given data object becomes a "hot spot," the virtual server that serves it can dynamically recruit new machines to help carry the load. Once the load decreases, servers can resign from the group to reduce the synchronization overhead. Moreover, when a particular usage pattern results in one data group being used more than others, it can be split into several smaller groups. In some

[1] We use the terms virtual server group, virtual server and virtual group interchangeably throughout the text. They all refer to the same concept.

applications it may be beneficial to take the network structure into account, migrating data objects to locations that are closer to the clients that access them most frequently; in some extreme cases it may even be beneficial to allow some of the clients to become data servers *themselves* in order to reduce latency. Note that the distinction we make between data servers and clients is merely conceptual; in practice, there is nothing in our approach that precludes a node from being both a client and a server simultaneously.

Given that our design makes use of replication in order to provide increased reliability and performance, it becomes necessary to consider a consistency model in order to ensure that the data remains uniform among the replicas in the system. However, consistency, replication, and performance are mutual antagonists. The issue is that, even though the data is replicated and multiple servers may be able to respond to a request, the consistency of a data item must be ensured before the request can be processed. In the worst case, this requires that a server contacts the replica holders to obtain an updated data item before responding to a request. Furthermore, while failures may be infrequent, servers must be prepared to respond and recover from faults at any time.

To address this we use in our approach an efficient group communication protocol to enforce sequential consistency of replicated data items inside each virtual server. It is widely accepted that the simplest programming model for data consistency is sequential consistency [35], which preserves the order of accesses specified by each of the programs that concurrently access the shared data. In practice, the simple sequential consistency model has been reluctantly adopted due to concerns about its performance. However, recent work in compilers for parallel languages [33] has shown that sequential consistency is a feasible alternative to relaxed consistency models.

6.1.3 Support for Sequential Consistency in MojaveFS

Computing systems have evolved from single, static computation nodes to dynamic distributed environments. This evolution has raised new issues for distributed objects systems, such as maintaining consistency in distributed filesystems. Several group communication models have been designed [12, 15], but few address the stricter and more intuitive consistency models required to facilitate such complex communication schemes.

In this section we present a decentralized protocol for ensuring the sequential consistency of accesses to objects in a loosely coupled distributed environment, with specific application to distributed filesystems.

First, we present the problem statement. We then give an informal overview of a protocol that guarantees sequential consistency of access in the distributed filesystem. Next we present a mathematical model and use it to prove the basic properties of our protocol.

6.1.3.1 Problem statement

Consider a distributed objects system consisting of processes and data objects. Objects are to be stored in a distributed fashion by subsets of the set of all processes in the system; the membership in these subsets is dynamic. Processes access object data through read and write operations. Each process may be accessing several different objects at a time. The goal of the protocol is to provide a guarantee that the object accesses are sequentially consistent. A system implements a sequentially consistent model if the result of any execution is the same as if the operations of all the processes were executed on a "master copy" in a sequential order, and, furthermore, the operations issued at each process appear in the same sequence as specified in the program.

The protocol we propose makes the following assumptions:

1. The only communication between processes is through read and write operations to the data of the shared objects.

2. The number of processes that access each individual object is small compared to the total number of processes in the system. In particular, achieving a consensus between all processes accessing a certain object at the time is considered practical, while achieving consensus between all processes in the system is considered impractical.

3. The network transport protocol is reliable. If a message is sent by a process, it is eventually delivered.

4. Nodes are fail-stop. When a node fails, it never comes back again.

6.1.3.2 Protocol Overview

We implement access to individual objects via group communication. Processes are organized in groups; each group is assigned to an object. Optionally, opening and closing an object could be mapped to join and leave operations for the corresponding group.

We assume that each group uses a *group total order* communication mechanism. Each access to object data (i.e. a read or a write operation) is implemented by sending an appropriate message to the group associated with an object.

When a write message is delivered, the local copy of the object data, if present, is updated accordingly. The result of a read request is based on the local data at the time the corresponding read message is *delivered back* to the initiating process.

The presence of the *explicit read messages*, the group total order, together with some simple constraints on the way individual processes send out their messages guarantee sequential consistency, as explained in the next section.

6.1.3.3 Sequential consistency

As stated in Section 6.1.3.1, we assume that the only communication mechanism between the applications is through the shared object data. The protocol implements each read or write access as a message in the system. The problem of sequential consistency reduces to the problem of imposing an order on these messages.

Definition 6.1.1. We will call an ordering O on messages *consistent* if it satisfies the following conditions:

1. On messages originating at the same process, the order O is consistent with the order in which the messages were sent by the process.
2. On messages sent to the same group, the order O is consistent with the group total order imposed by the group communication mechanism.

Next, we show that if there exists a consistent total order of messages, as defined above, the sequence of messages exchanged by the processes taking part in the protocol is sequentially consistent.

Lemma 6.1.2. If all messages sent in a particular run of the protocol can be arranged in a *consistent total order*, then this run of the protocol is sequentially consistent.

Proof. Suppose $m_0, m_1, \ldots, m_n$ is a consistent *total* ordering of messages. We can make the execution sequential as follows. First, run the application that originated the message m_0 until the point where it originated m_0. Then perform the read/write operation specified by m_0. Next, run the application that originated m_1 until the point where it originated m_1. Furthermore, perform the read/write operation specified by m_1, and so forth. Due to condition 1 in Definition 6.1.1, the ordering of messages is consistent with the execution order of each specific process, so the sequential schedule is feasible. Because of the condition 2, the operations on each object are performed in the order specified by the group ordering, which means that they are performed in the same order as in each local copy in the parallel execution. This guarantees that the data returned by the read operations in the sequential schedule is the same as it was in the parallelized execution. □

Note that we do not require that all the messages in the system be delivered in the same order. It is acceptable for certain messages (sent to different groups) to be delivered "out of order" by a process in certain cases.

Example 6.1.3. Figure 6.2 presents a situation where messages sent by processes P_3 and P_4 could be seen in different orders by P_1 and P_2. Assume P_3 and P_4 each send one write message, call them w_{3x} and w_{4y}. In the case that neither of the processes P_1 and P_2 was actively accessing objects X and Y at the time, they can deliver messages w_{3x} and w_{4y} in any order. This does not break

sequential consistency because the applications running at P_1 and P_2 are not allowed to see the effects of these write operations until they send *an explicit read message*.

Assume now that P_1 is active and it sees that w_{3x} happens before w_{4y}. In other words, it sends two read messages — first r_{1x} and then r_{1y}. It delivers r_{1x} after w_{3x}, while delivering r_{1y} before w_{4y}. Now the condition 1 in Definition 6.1.1 guarantees that any consistent order will contain r_{1x} before r_{1y} and the condition 2 guarantees that any consistent order will contain w_{3x} before r_{1x} and r_{1y} before w_{4y}. This means that every consistent order must place w_{3x} before w_{4y}. However, if P_2 is not actively accessing X and Y, it can still receive messages "out of order" without affecting sequential consistency.

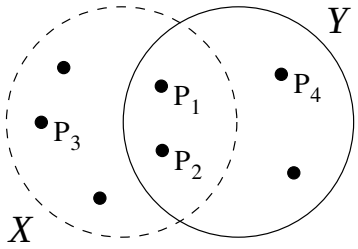


Figure 6.2: An "out of order" interleaving of messages from P_3 and P_4 is allowed during the passive periods of P_1 and P_2. Once P_1 becomes active it enforces its own thumbprint on the order of messages through the read and write operations it performs.

6.1.3.4 Achieving sequential consistency

We approach the problem of guaranteeing sequential consistency in our implementation by picking a specific (*postfactum*) total ordering of messages O. We provide an implementation that makes sure that O will always be consistent (in the sense of Definition 6.1.1).

There are different ways of picking an appropriate O. We believe that during the protocol execution, the earlier the messages get assigned to order O, the less intrusive the corresponding protocol adjustments will be. Since O must be consistent with the group total order (see condition 2 in Definition 6.1.1), unless we adjust the group communication protocol to be aware of O, the earliest a message can be assigned to O is when the group communication protocol assigns the message to a group total order.

Therefore, we pick O as follows — the messages in O are ordered according to the order (from the point of view of an external global clock) in which they are assigned a "group sequence number" by the group total order communication protocol. This is expressed by the following lemma.

Lemma 6.1.4. The protocol described in Section 6.1.3.2 is sequentially consistent if the following conditions are met:

- The group communication protocol will order messages sent by the same process to the same group consistently with the order in which they were sent.
- When a process attempts to send a message to a different group than it sent its previous message to, the send operation is *blocked* and is *not performed* until all the messages already sent by this process get assigned to the group total order by the group communication protocol.

By "*assigned to the group total order*" we mean that the group communication protocol commits to putting the message right after a specific message (which is already assigned to the group total order) in the group total order.

Proof. Let us define the relation O on messages as "*message* 1 *was assigned to the group total order before message* 2 *(according to the external global clock).*" We claim that under the conditions above, relation O is a consistent global total order.

The relation O is obviously a total order (without the loss of generality, we can assume that no two events are ever perfectly simultaneous). By construction, O agrees with the group total order (this follows from the way we have defined the notion of "*assigned to the group total order*").

We are left to show that on messages sent by the same process, O agrees with the process send order. If both messages were sent to the same group, then condition 6.1.4 of the lemma ensures that O orders them correctly. If the messages are sent to different groups, then condition 6.1.4 of the lemma ensures that the sending of the second message will be delayed until it can be guaranteed that O will order them correctly.

Therefore, O is indeed a consistent global total order on messages and by Lemma 6.1.2 we are guaranteed sequential consistency. □

6.1.3.5 Mathematical model

This section presents a mathematical model for the problem presented in Section 6.1.3.1 based on the protocol presented in Section 6.1.3.2 and provides a set of requirements for a possible implementation.

Definitions. First, we define the terms used throughout this section. Consider the set of processes $\mathbb{P} = \{p_i \mid i \in P\}$, where P is a set of indices identifying the processes. We also define a set of groups: $\mathcal{G} = \{g_j \mid j \in G\}$, where G is a set of indices identifying the object associated with each group (group g_i is associated with object i).

Let K be a totally-ordered set of unique labels (or sequence numbers) used by processes to mark the order of the messages they send. Let L be a totally ordered set of unique labels (or sequence numbers) the group total order mechanism assigns to messages sent to each group.

Now we can define the set $\mathbb{M}$ of all messages[2] that can potentially be sent in the system as

[2]Strictly speaking these are not messages, but rather message "headers" that are assigned *post-factum*. However, in this model this distinction can be safely ignored.

$\left\{m_{ij}^{kl} \mid i \in P, j \in G, k \in K, l \in L, m \in \{read, write\}\right\}$, where m_{ij}^{kl} represents a message from process p_i with process label k, sent to group g_j with group label l.

We also define two relations $<_p$ and $<_g$ on messages in $\mathbb{M}$ as follows:

- $<_g$: $m_{ab}^{cd} <_g m_{a'b'}^{c'd'}$ *iff* $d < d' \wedge b = b'$
- $<_p$: $m_{ab}^{cd} <_p m_{a'b'}^{c'd'}$ *iff* $c < c' \wedge a = a'$

The $<_g$ defines a relation on messages sent in each group, and $<_p$ defines a relation on messages originating at each process. To make the analogy with the protocol description in Section 6.1.3.3, the $<_g$ relation is given by the order imposed on messages by the group communication mechanism used inside each group. The $<_p$ relation is the order imposed on messages by the run trace of each process. If a send operation on message mx occurs in the run before the send operation of message my, and both messages originate at the same process, then $mx <_p my$.

Now we will consider the set of messages sent in the system for a specific run.

Definition 6.1.5. We will call a subset M of the set $\mathbb{M}$ *feasible* if for any two messages $m_{ab}^{cd}, m_{a'b'}^{c'd'} \in M$ we have:

1. Uniqueness:

   $(a = a' \wedge c = c') \Leftrightarrow (b = b' \wedge d = d')$.

   This condition states that there can be at most one message in M that was sent by a specific process a, and labeled with a specific process label c. It also states that there can be at most one message sent to a specific group labeled by a specific group label.

2. Ordering consistency: $(a = a' \wedge b = b') \Rightarrow (c < c' \Leftrightarrow d < d')$.

   For messages sent by the same process to the same group, the ordering of messages by the originating process agrees with the ordering of the same messages by the destination group.

Definition 6.1.6. Finally, we define a relation $\prec \subseteq M \times M$ as $\prec := <_g \cup <_p$. This relation is the minimal requirement for consistency (see Definition 6.1.1) – an ordering on messages is consistent *iff* it agrees with $\prec$.

Sequential consistency. Consider relation $\prec$ of Definition 6.1.6 on a feasible set of messages M. We present an example that illustrates how we can obtain a cyclic dependency of messages that is not sequentially consistent. The setup in Figure 6.2 can also be used to show a simple example of an object access that could lead to a cyclic dependency. Consider the two processes, P_1 and P_2, that access objects X and Y. Each process issues two operations, one on each object. Without loss of generality, assume process P_1 performs a read operation on data object X and a write operation on data object Y. Similarly, process P_2 performs a read operation on data object Y and a write

P_1		P_2	
read(X)	m_{1x}^{ab}		
		m_{2y}^{cd}	read(Y)
write(Y,1)	m_{1y}^{ef}		
		m_{2x}^{gh}	write(X,1)

Table 6.1: An interleaving of read and write operations leading to non-sequentially consistent outcome.

operation on data object X. We can order messages according to the order relation $\prec$ as follows. The read operation, represented in message notation as m_{1x}^{ab}, is performed by process P_1 before the write operation mapped to message m_{1y}^{cd}. Similarly, the read operation performed by process P_2, m_{2y}^{ef}, is performed before the write operation m_{2x}^{gh}. An illustration of this can be found in Table 6.1.

If we assume that the initial values of both X and Y are 0, and that the read initiated by process P_2 returned value 1, then the read operation must have happened after the write performed by P_1. We extend the relation to include messages m_{1y}^{gh} and m_{2y}^{ef}, giving us the following:

$$m_{1\ x}^{a\ b} <_p m_{1\ y}^{e\ f} <_g m_{2\ y}^{c\ d} <_p m_{2\ x}^{g\ h}$$

According to the definition of relation $\prec$ and subject only to the feasibility Definition 6.1.5, we can also include the pair of messages $m_{2\ x}^{g\ h}$ and $m_{1\ x}^{a\ b}$ in our relation $\prec$. This means that the read(X) operation performed by P_1 was performed after P_1. By a simple examination on labels a, b, c, d, e, f, and g and the relations deduced from the definition of $\prec$, no constraint or assumption has been violated. If that is the case, we have the following order:

$$m_{1\ x}^{a\ b} <_p m_{1\ y}^{e\ f} <_g m_{2\ y}^{c\ d} <_p m_{2\ x}^{g\ h} <_g m_{1\ x}^{a\ b}$$

This enforces the return value of read(X) in process P_1 to be 1. It is easy to show now that with these return values from the two read operations, there is no possible sequentially consistent order of the four operations, subject to the conditions presented in Section 6.1.3.4.

We must therefore break the kind of cycles we have seen above.

First we define a new relation on message in M.

Definition 6.1.7. Consider the relation $<_d$ defined as: $m_{ab}^{cd} <_d m_{a'b'}^{c'd'}$ iff $d < d'$.

Next, we show how to ensure that this relation is acyclic.

Lemma 6.1.8. Suppose a feasible set M satisfies an additional *strict ordering* property: $a = a' \Rightarrow (c < c' \Leftrightarrow d < d')$. In other words, messages originating at a node a are related under $<_p$ if and only if they are also labeled with group labels that are similarly related under the numbers less than relation. Then the relation $<_d$ is acyclic on M.

Proof. From the initial assumption it follows immediately that L is totally ordered by $<_d$. □

Next, we show that both $<_g$ and $<_p$ relations are subsets of $<_d$. From there we will be able to show that relation $\prec$, which is the union of the two relations, is also a subset of relation $<_d$. It directly follows that $\prec$ is an acyclic relation.

Lemma 6.1.9. $<_g \subseteq <_d$.

Proof. By the definition of $<_g$, all messages related by $<_g$ are also related by $<_d$. □

Lemma 6.1.10. Under the conditions of Lemma 6.1.8, $<_p \subseteq <_d$.

Proof. By the definition of $<_p$: $m_{ab}^{cd} <_p m_{a'b'}^{c'd'}$ iff $a = a' \wedge c < c'$. According to the strict ordering condition $a = a' \Rightarrow (c < c' \Leftrightarrow d < d')$. This means that $d < d'$, so $m_{ab}^{cd} <_d m_{a'b'}^{c'd'}$. □

Theorem 6.1.11. Under the conditions of Lemma 6.1.8, relation $<_d$ is a consistent total order on M (as defined in Definition 6.1.1).

Proof. From Lemma 6.1.9 and Lemma 6.1.10 it immediately follows that $<_d$ is consistent. By definition and Lemma 6.1.8, $<_d$ is total order on M. □

From this theorem, we conclude that any implementation that guarantees the strict ordering condition of Lemma 6.1.8, will also guarantee sequential consistency. If we take d to always be the timestamp (according to the external global clock) of the moment each message gets assigned a group serial number by the group communication protocol, we will arrive at exactly the same implementation as outlined in Section 6.2.4.

6.2 Implementation Overview of MojaveFS

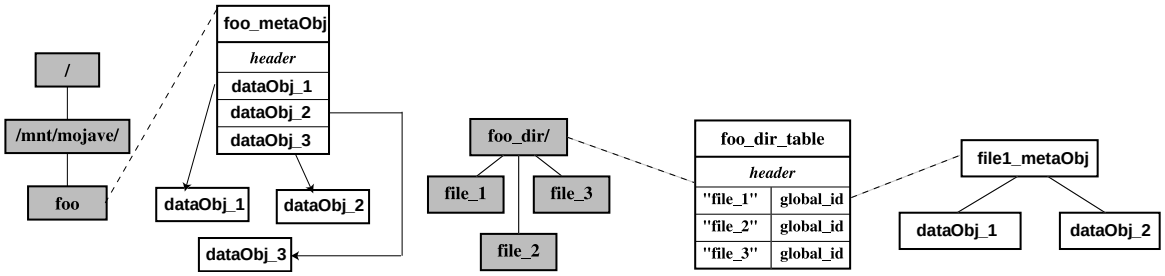


Figure 6.3: Representation of a file and of a directory in MojaveFS. Only the shaded areas are visible to the user and they represent the user's perspective.

MojaveFS has two components. The first one lives in kernel space and acts as a layer that intercepts and interprets filesystem calls before passing them to the second component, a user-level

daemon. The user-level daemon uses the existing underlying filesystem to store the data. Due to its open source nature, we chose Linux as the implementation platform for our filesystem (thus providing us with easy access to the internals of the operating system).

The internals of MojaveFS implement an effective mechanism that allows files to be replicated and distributed in an efficient manner. Normally, only part of a file will be used by any single process at a given time. Other parts of the file may be used by other processes in other locations. MojaveFS uses a novel data storage strategy where files are split into smaller *objects*. These objects are the indivisible data unit of our system and they are mapped to virtual data server groups through a hashing function. This scheme increases the availability of the data, improves file utilization, and decreases the access time to very large files.

Each user-visible file corresponds to a set of objects that collectively contain all of the data associated with the file. The metadata for each file is also contained in a separate object, and is comprised of a header describing properties of the entire file and a list of the objects of which the file is composed.

In Figure 6.3, we present the internal representation of a file in MojaveFS, named *foo*. The metadata object for *foo*, named *foo_metaObj*, contains the identifiers of the objects composing file *foo*: *dataObj_1*,*dataObj_2*, and *dataObj_3*.

A directory in MojaveFS is represented as a metadata object containing entries for each file that resides inside the directory. To access a file inside a given directory, the actual file can be resolved by looking up its identifier inside of the metadata object of the directory. This is illustrated on the right hand side of Figure 6.3. Here we have one directory, named *foo_dir*, containing the three files *file_1*, *file_2*, and *file_3*.

We have opted for a modular design for the implementation of MojaveFS, consisting of the following layers: Cap, Indirect I/O, Direct I/O, and Group Communication. The model we chose has an internal asynchronous behavior. Our design uses function calls to propagate requests and information from the top to the bottom layers, and events/callback functions to propagate information up the stack as it becomes available. Figure 6.4 shows the API exported by each layer on the left hand side of the layer block and the events/callback functions on the right hand side. The functionality of each of the layers is described in more detail below.

The Cap layer is a wrapper that provides applications with the standard filesystem interface. The exported API of the Cap layer, shown on the left side in Figure 6.4 includes calls for creating a new file (*create*), and for accessing data from a file (*read*,*write*). While the internal communication between the various layers of MojaveFS is asynchronous, the interface that the Cap layer exports to the application layer uses the traditional synchronous model.

The Indirect I/O Layer (IIO) handles data replication, and naming and localization of objects. It provides a one-to-one mapping to the API exported by the Cap layer, but its functionality is

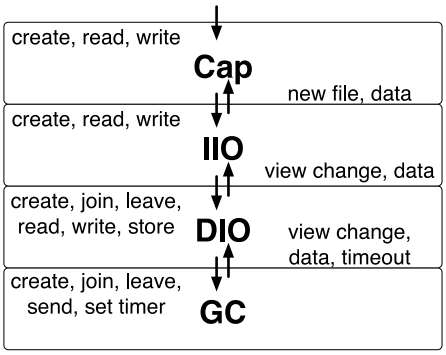


Figure 6.4: The layered architecture of MojaveFS.

asynchronous. It is only when data is available that it uses the callback function provided by the Cap layer to send the *new file*, and *data* events.

The Direct I/O (DIO) layer handles data consistency across replicas, saves information to stable storage, and deals with certain fault-tolerance issues. Its API is closer to that of the bottom layer, the Group Communication (GC) layer. The events that the DIO layer propagates to the IIO layer are *view change* (when the membership of the data servers changes), *data* (when data becomes available for a previous request by IIO), and *timeout* (when an IIO expected event fails to occur in the specified time window).

Finally, the purpose of the Group Communication layer is to provide access to a network communication mechanism that enforces the sequential consistency guarantees that the upper layers rely upon.

6.2.1 The Indirect I/O layer

The IIO layer, which is the most complex layer in our stack, provides the following services: replication, naming and localization.

6.2.1.1 Reliability and Replication

In any filesystem it is important to keep the data sound. We propose to achieve data reliability through replication. We implement a simple k-copy replication mechanism in our prototype. As explained below, data replication helps improving the performance of the system by increasing its throughput.

One of the traditional approaches to distributed data allocation is to use a hash function to map some property of a file (such as its file name or full path) to the server which provides the data associated with the file [13]. We call the data associated with the same server after applying the hashing function a *data group*.

In a traditional hash-based distribution scheme, the hash function for a file's path indicates

which server has the file. Adding a new server requires a change to the hash function. Since the hash function determines the physical location of each file, a change to the hash function will generally cause large amounts of metadata to be moved among the servers, which is a very expensive operation. Various approaches exist to try to amortize this cost over longer periods of time, but the expense cannot be avoided entirely.

In our approach, however, the hash function produces a virtual server group identifier. Each client has a look-up table that maps these identifiers to sets of physical servers, which is updated during the normal course of client-server interaction. This table acts as an extra layer of indirection between the hash value of a path and its physical location on the network, so adding a machine to a group does not require any change to the hash function. From the client's perspective, adding a new server to a group is transparent. Removing a server from a group can, in the worst case, cause a client to encounter a network timeout; the client will then proceed to the next server in the group.

A virtual data server acts as the replication group for its corresponding data group. Every node in a virtual data server possesses an up-to-date copy of the data corresponding to the data group that it serves. Note that we do not limit the number of different data groups that a data server can serve. Furthermore, each physical server can belong to an arbitrary set of virtual data servers without restriction.

Unlike traditional replication mechanisms where complete replicas are made of the entire information store from one physical node, our replication mechanism is more nimble. Specifically, we can exploit the relatively small cost of allocating additional virtual data servers in order to tune the breadth of the responsibilities of each of the virtual data servers in our system.

6.2.1.2 Naming and localization

The naming service provides the following functionality. It generates system-wide unique object identifiers and it provides a global look-up service. The service distinguishes between two types of objects: data objects and metadata objects. The former type stores data contained in user files, while the latter type contains data pertaining to directories or file properties, as well as pointers to the data objects composing the files.

In our current design we use a hybrid hash-based and tree-based look-up mechanism for objects. This, combined with an efficient caching scheme for access privileges, provides fast access to files in the system.

The system uses a function that maps each potential file in the system to a unique virtual data server. The look-up mechanism for a file works as follows. First, the system identifies, using the mapping function, the virtual data server that is responsible for the metadata object associated with the file that is being looked up. Next, it contacts the virtual data server and retrieves the metadata object that contains information about the identifiers of the data objects composing the file.

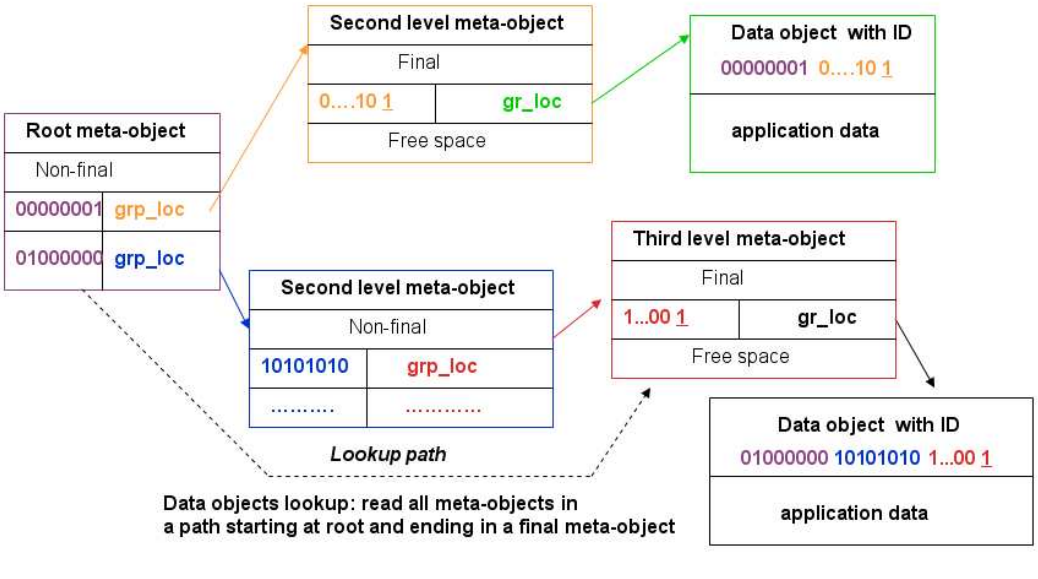
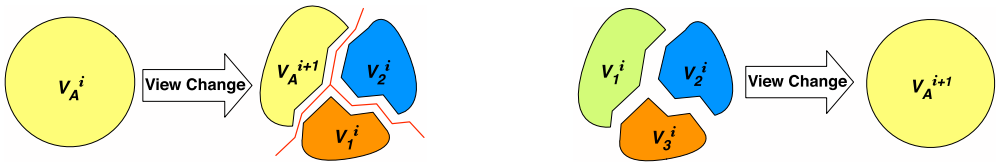


Figure 6.5: Object look-up mechanism.

In order to conserve resources it may be desirable for the system to consolidate the responsibilities of the virtual data servers corresponding to all of the files in some subtree of the filesystem directory structure. In this case, the look-up mechanism would perform a binary search in order to determine the most specific virtual data server that is responsible for the metadata of the file to look up. For example, suppose the directory "a/b/c" is responsible for all of the files in the subtree rooted at that directory and that a client is trying to locate the file "a/b/c/d/e.txt". In this case, the client will initially attempt to contact the virtual server corresponding to the file "a/b/c/d/e.txt". However, given that this virtual server is nonexistent, the look-up mechanism will then look up directory "a/b/c". Since this virtual server does exist, the client will then try to look up directory "a/b/c/d" in order to find a more specific virtual data server responsible for the file. However, since "a/b/c/d" is also nonexistent, this would conclude the search and the request would be directed towards the virtual data server serving "a/b/c". The advantage of this mechanism is that we don't need to create all possible virtual servers up front and, if we couple this with a caching mechanism, the scheme can perform similarly to the non-consolidated version.

While the mapping scheme is fast for file look-up, it is still convenient to preserve the hierarchical structure of the filesystem in order to determine access permissions. For example, given this structure, it is possible to traverse the resulting hierarchy, starting from the root, to resolve the access permissions for a particular file or directory. Unfortunately, these tree traversals can be costly. Thus, we implement a need-based caching mechanism to allow the look-up mechanism to avoid performing these tree traversals unless the cache of the object has been invalidated.

Figure 6.5 shows how the metadata objects preserve the hierarchical directory structure utilized



(a) A three-way split of a virtual server group due to a network failure.

(b) Three non-authoritative views merge to create a new authoritative view.

Figure 6.6: Views split and merge, changing authoritative status.

by the tree traversal mechanism for determining file access permissions.

6.2.2 The Direct I/O layer

The primary function of the Direct I/O (DIO) layer is to handle the propagation of data and metadata to stable storage. When a node receives new data to be written to an object that it has stored locally, or if the IIO layer requires the creation of a new object, the DIO layer saves the new data and metadata associated with the object to stable storage by interacting directly with the underlying local filesystem.

However, in order to maintain consistency of this data in the presence of failures, it is necessary for the DIO layer to perform some additional bookkeeping. For example, due to network or node failures there could be a split of a virtual data server such that only some subset of the original members of the virtual server group are still able to communicate with each other. In this case, it is the responsibility of the DIO layer to decide which subset of the members of the original virtual server can continue to make progress and which nodes are prohibited from making progress (blocked) until the failure is resolved.

To illustrate the need for this bookkeeping, consider the scenario highlighted in Figure 6.6(a). In this case, a network failure causes a three-way partition of the nodes that comprise one of the virtual servers in the filesystem. Each of the nodes in the three fragments of the original group are capable of communicating with other nodes within its respective partition, but are unable to communicate with any of the nodes in the other partitions. If the data were not mutable, this might not be a problem. However, it is clear that allowing each of the three fragments to operate normally (i.e., handle both reads and writes) could lead to data inconsistency among the replicas.

In order to handle such scenarios as might lead to data inconsistency, the DIO layer introduces the notion of an *authoritative* view of a virtual data server. Intuitively, a view, $\mathcal{V}$, is authoritative when $\mathcal{V}$ corresponds to the initial membership of a newly created virtual server or when $\mathcal{V}$ contains a majority of the membership of the previous authoritative view. To keep track of the order of authoritative views in the system, the DIO layer associates an *epoch* with every view in the system; this value is monotonically increasing and is incremented every time a view experiences a change in membership that results in the view attaining (or maintaining) authoritative status. We denote the

value of the epoch, e, of a view, $\mathcal{V}$, using the superscript notation $\mathcal{V}^e$ (we use the convention that 0 is the epoch of a view corresponding to a newly created virtual server). More formally, the following two conditions govern the authoritative status of a view.

1. The initial view, $\mathcal{V}^0$, associated with a newly created virtual server is authoritative.
2. The view, $\mathcal{V}^i$, is authoritative if there existed an authoritative view $\mathcal{V}^{i-1}$ such that $\mathcal{V}^i \cap \mathcal{V}^{i-1} \subseteq_m \mathcal{V}^{i-1}$, where $A \subseteq_m B$ denotes that A is a majority subset of B.

Thus, the condition for a view to attain authoritative status ensures that at most one authoritative view of a virtual data server exists in the entire system at any given time. Given this property of authoritative status, it is safe for the DIO layer to allow only authoritative views of virtual data servers to make progress.

Note, however, that the condition for authoritative status does permit scenarios where all the views of a virtual data server become non-authoritative. For example, consider Figure 6.6(b) and suppose that none of the three fragments contains a majority of the nodes of the initial virtual server group. In such a case, no view of the virtual data server can make progress. This necessitates the adoption of some sort of recovery policy that permits one to "rebuild" an authoritative view out of non-authoritative views. The specific recovery policy we have chosen for our implementation is embodied by the second condition presented above. This condition implies that two or more non-authoritative views could merge to form an authoritative view (provided that the membership of the merged view contains a majority of the nodes that were members of the last authoritative view).

Given the need for these types of recovery policies, the DIO layer must keep track of certain information, both to facilitate the reconstruction of authoritative views and to prevent more than one authoritative view of a particular virtual data server from being created. In particular, our simple recovery policy requires that the DIO layer maintain the membership of the last authoritative view of which a node was a member along with the epoch of that authoritative view. Furthermore, on a split, a resulting non-authoritative view must also keep track of all the messages the view has received that the members cannot confirm have been delivered to the nodes on the other side of the partition. This is needed to ensure that, if two or more non-authoritative views merge to create an authoritative view, a resulting authoritative view has the most recent set of messages (reads and writes) sent to the view before the split.

It is salient to note that different policies governing authoritative status could be implemented. However, it is important that such a policy enforce the invariant that at most one view is permitted to be authoritative at any time. In general, policies that allow more automatic failure recovery tend to require more bookkeeping. For example, one might envision a policy that would allow non-authoritative views with different epochs to merge, provided that the data has not changed between these epochs. In this case, the DIO layer would have to keep track of extra information, such as the

entire history of authoritative view membership for a node (as opposed to the membership of only the previous authoritative view of which the node was a member). Without this extra information it would not be possible for the DIO layer to determine if the membership of a view resulting from a merge of two or more non-authoritative views contains a majority of the possible nodes that could combine to form an authoritative view (thus ensuring that there could be no other merger involving a disjoint set of nodes that would result in a view being considered authoritative). Though, it may be possible to maintain less information if a metric other than the majority were used.

6.2.3 Implementation of the Lower Layer of the Group Communication Protocol

This subsection presents the implementation of the lower layer of our group communication protocol that guarantees the total order of messages inside a group. We have shown the correctness of the upper layer of our protocol and proved the guarantees that it makes [58]. We focus on the correctness of the implementation of the lower layer and present how its distributed approach makes it robust in the presence of failures.

6.2.3.1 Overview

The system is composed of a set of processes. Processes have a unique identifier that is persistent across failures. We define a *group* to be a set of processes. A *view* of a group is defined as a subset of the group membership.

Processes can belong to several groups at the same time. The upper layer of the protocol relies on the existence of a total order of messages sent within each group that is provided by the lower layer of the protocol.

Each group can have several views and the views can overlap. Views are local to processes and represent their image of the membership of the group. In each group we implement the Virtual Synchrony [19] model. When a request to join a group is received by a process or when a process leaves or is detected to have failed a view change is triggered in the group. The view change is a synchronization point for the processes that survive it. It guarantees that membership changes within a process group are observed in the same order by all the group members that remain in the view. Moreover, view changes are totally ordered with respect to all regular messages that are sent in the system. This means that every two processes that observe the same two consecutive view changes, receive the same set of regular messages between the two view changes. Each message sent in a view is characterized by an epoch corresponding to the view in which the message was sent, and by a group sequence number. The epochs and the group sequence numbers are monotonically increasing during the life of a process, but they are not persistent across failures.

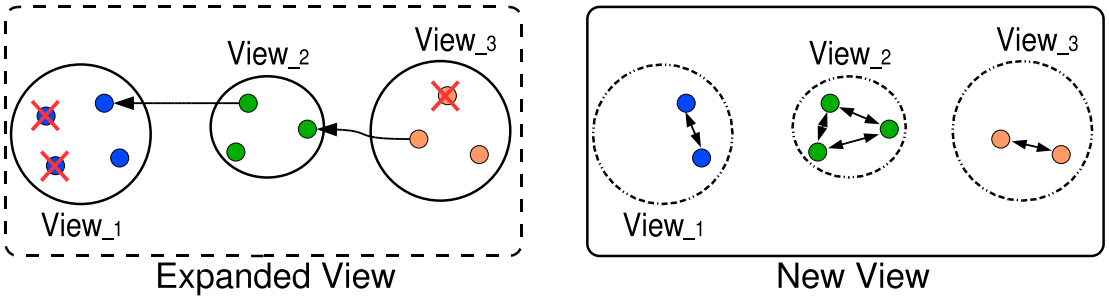


Figure 6.7: The view change event.

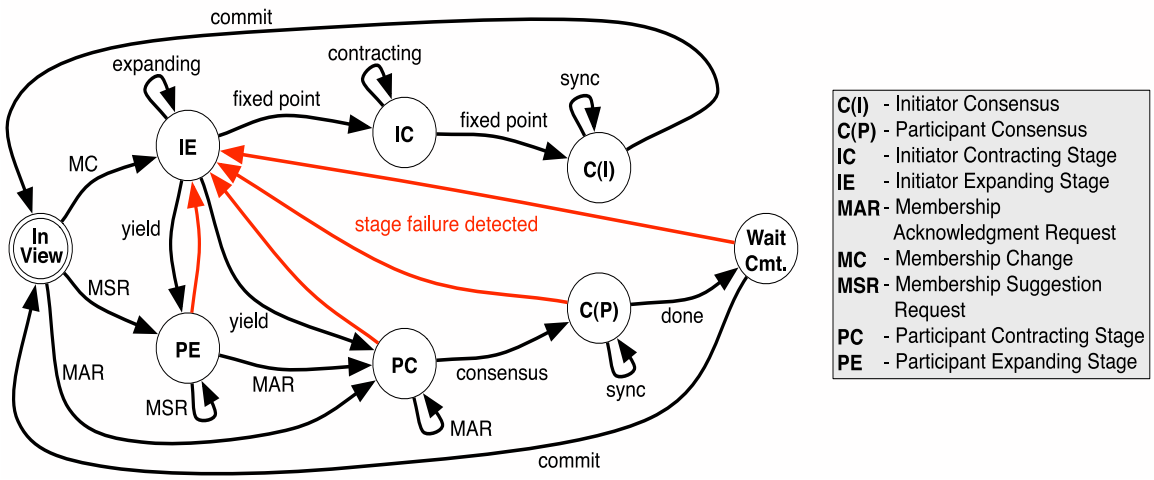


Figure 6.8: The state machine for our protocol.

Each process that wants to join a group starts as a singleton view. It then contacts a directory service that provides hints on the current membership of the group. These hints are used by the process to join other processes that belong to the same group. At this layer of the protocol we do not discriminate between different views of a group. It is only at the upper layers that we treat separate views of the same group differently based on criteria that we deem appropriate. For example, we might want to allow only the members of a given view (like the one with highest cardinality) of a group to send or receive messages.

6.2.3.2 The View Change

A view change is triggered by one of two events: a join request or a leave request. A join request is sent by the member of one view to members of another view of the same group in an effort to merge the two views. We do not allow views to split at will. If a process wants to leave a group it sends a leave request to the members of its current view. If a process is detected to have failed, then one or more members of the view initiate a view change and try to exclude the process from the view.

Upon receiving a join or a leave request a process initiates the view change procedure. We employ a resolution mechanism for concurrent view changes in the same group involving an overlapping set of processes. This resolution mechanism allows only one of the initiators to complete the view change successfully. However, we do permit concurrent view changes in disjoint views to evolve in parallel. One major advantage of our protocol is that it allows changes in view membership to include, at the same time, multiple processes joining and leaving the view. For example, Figure 6.7 illustrates how, in one step, three views merge and certain nodes from each view will be excluded as they fail during the view change process.

The view change is done in several stages: the *expanding stage*, followed by the *contracting stage*, the *consensus stage* and finally the *commit stage*. Figure 6.8 illustrates the states that a process can be in during the view change process, along with the events that trigger the transitions from one state to another. The initial state is the *In View* state. On detecting a membership change (MC) a process becomes an initiator of a view change and initiates the expanding stage (IE). The other nodes involved in the view change would be participants in the expanding stage (PE). If the stage ends successfully the initiator becomes the initiator of a new view as part of the contracting stage (IC), while the other nodes become participants in the new view during the contracting stage (PC). The consensus stage is represented by two states, depending on the previous state of the process (C(I) and C(P)); the participants then wait for the final commit stage. Each stage is detailed below.

The purpose of the first stage of the view change process is to collect suggestions from the current members of the view and, from these suggestions, ascertain what the new membership of the view should be. This stage is repeated until a fixed point is reached (nodes are only added to the membership with each round of suggestions). In the example presented in Figure 6.7 this is shown in the top drawing. At the end of the expanding stage, the expanded view contains all the members of the initial three views.

During the contracting stage, processes that have failed or that want to leave the group are removed from the maximal fixed point view reached during the previous stage. The goal of this stage is to reduce the membership of the view to the current set of active processes. It is important to note that between a process's interest in joining a group and the commit of the view change that process could fail or there might be a network partition, in which case more than one process might need to be excluded from the view. In Figure 6.7 the *new view* illustrates the membership after the contracting stage, where failed nodes have been evicted from the view.

The consensus stage is critical for preserving the properties of the Virtual Synchrony model. During this stage processes that have survived so far agree on what messages they need to deliver to the application layer to guarantee that all members of the view that survive the view change have delivered the same set of messages in the previous view. The consensus stage is illustrated by the arrows in the second drawing of Figure 6.7, where each surviving process synchronizes with all the

other processes in its previous view.

In the last stage, the new view to be installed is broadcast to all of its members and is locally installed by each member. The view change initiator sets the epoch of the new view to be larger than the largest epoch involved in the view change. Also, the sequence number is reset to 0 and a ViewChange message with the new epoch and the new sequence number is broadcast to all members of the new view. Upon receiving the ViewChange message each process delivers it to the application (the upper layer running on top of the group communication).

Surviving failures. To detect network or process failures we introduce a set of failure detection mechanisms that dynamically adapt to the environment. We expect acknowledgments for the messages sent in each group and we use heartbeat messages to detect failures during times of network inactivity. Our failure detectors can be dynamically changed to report on what is considered to be an acceptable latency or a tolerated loss rate before the processes composing the system are evicted from views.

On the failure of a process the appropriate action is taken by the process that detects the failure. This depends on the current state of the detector process. Figure 6.8 shows in red (light color) the actions triggered by our failure detection mechanism. When a critical node fails during the various stages of a view change it prompt a process to initiate a new view change. Generic node failures are reported during the contracting stage.

Providing sequential consistency. The existence of shared data and of concurrent processes that can access it prompts the need for using a data consistency model. Using our two-layered group communication protocol, we can easily provide sequential access to shared data. We organize processes in groups based on the shared data they are interested in accessing. Thus, each group will be associated with a piece of shared data. For simplicity, we call this shared data an object. We map the "opening" and "closing" operations on objects to the join and the leave operations in the corresponding group. Thus, to access the information of a shared object, processes dispatch read and write messages to the appropriate group. One of the key features of our protocol is the presence of *explicit read messages*. The read messages act as synchronization points for the objects and guarantee that when they are processed all previous changes to the object have been seen.

6.2.3.3 Experimental results

This section presents a set of preliminary experimental results that focus on the lower layer of the group communication protocol. We have run our protocol in an emulated environment where we started 64 identical processes. We monitored the processes as they were trying to form one single view of the same group. Figure 6.9 shows the number of view changes that occurred from the

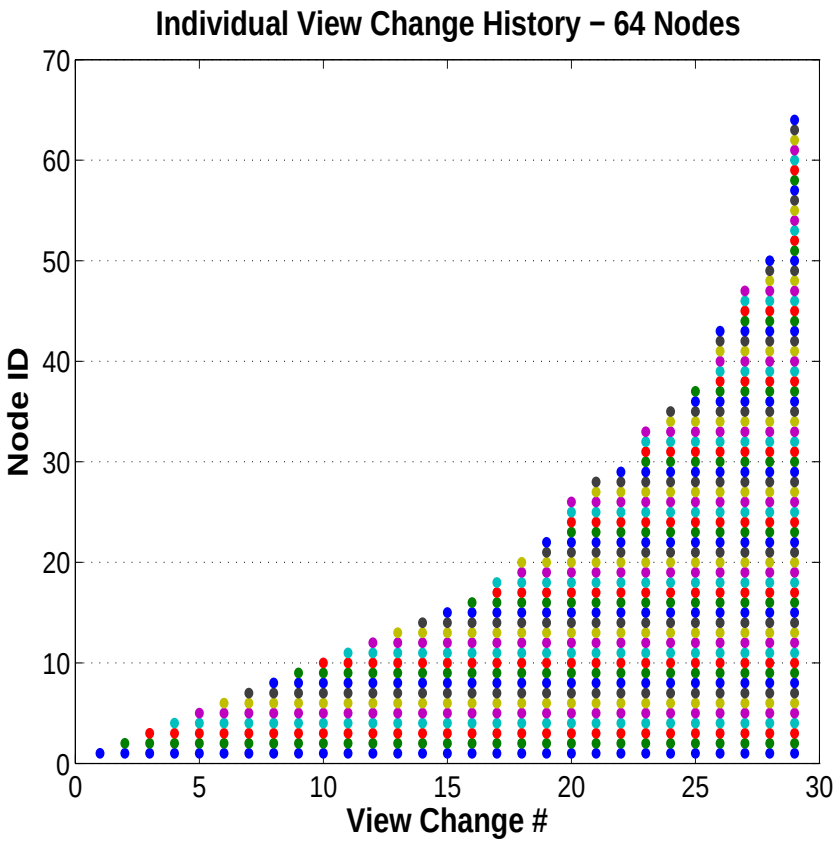


Figure 6.9: The membership of the view changes from the perspective of one node during the deployment of 64 nodes.

perspective of one of the processes and the membership of each of the view. This graph shows us that while this node was taking part in view changes there were parallel view changes involving other nodes that eventually merged into a single view. The second graph, presented in Figure 6.10, shows the number of view changes occurring in each one-second interval from when the first process was started until the final view change, comprised of all processes, was installed. It is important to keep in mind that our emulation environment has a few shortcomings. For example, it restricts the parallelism of the execution due to limited shared resources and it delays the start of a few of the processes until near the end of the experiment.

6.2.3.4 Related Work

The idea of using formal methods to prove the correctness of distributed protocols is not new. Ensemble [25], and its predecessor Horus [49], used the PRL [54] theorem prover to show the correctness of the group communication protocols provided by it. The long and tedious process of formalizing the protocols also required a formalization of a significant subset of the language used to implement it, ML [23]. While this mechanism worked for Ensemble, where the high level of modularity in the

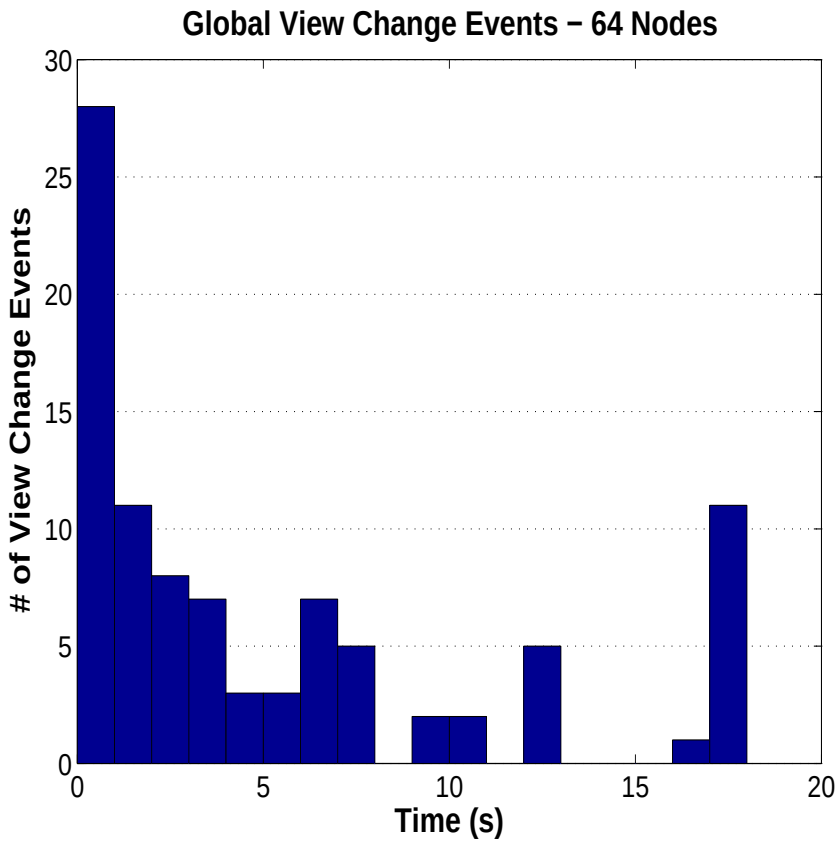


Figure 6.10: The number of view changes occurring in each 1s interval during the deployment of 64 nodes.

system and the choice of the implementation language played a large role, this would not easily apply to protocols implemented in languages such as C or Java. Furthermore, more complex systems would be even harder to formalize. Also, our group communication protocol has taken on a few of the challenges neglected by Ensemble, like handling multiple join and leave events simultaneously and allowing merges of overlapping views.

Spread [3], another group communication system, tries to address the problem of communication in the context of wide area networks. While Spread is a popular toolkit, its implementation is not very close to the abstract formal model discussed in the design paper. Over time, this has led to some degree of confusion.

Newtop [18] provides a solid mathematical model for proving the correctness of the group communication protocol that it uses. However, implementations have been slow to be developed.

Finally, we want to mention the Message Passing Interface (MPI), the "de-facto" standard for communication APIs in GRID systems. One of the problems of MPI stands in that the specification of the behavior and API of the system is too loose, which has led to various interpretations that mapped into sometimes incompatible implementations.

6.2.4 Implementation Overview of the Top Layer Protocol

Lemma 6.1.4 provides most of the information needed to describe our top layer protocol implementation. In order to achieve sequential consistency, we rely on a group total order communication protocol, which makes sure that the messages sent by the same process to the same group are ordered consistently with the order in which they were sent. We discussed its implementation in the previous section.

One of the key elements of our approach is the way we guarantee sequential consistency of messages in two groups that have common processes. When a process sends multiple messages to the same group, it is allowed to send the request for a sequence number and then continue with its execution until the sequence number is granted. In the meantime, the process may send multiple sequence number requests to the same group. However, when a process switches the group it sends the message to, it has to wait for all its previously requested sequence numbers to be granted before sending a request for a sequence number in the second group.

According to Lemma 6.1.4, this guarantees sequential consistency.

6.3 Optimization

One important issue in designing a distributed filesystem is to understand the difference between the data and the metadata. Metadata differs from file data in several ways. Metadata is usually smaller than file data. Loss or corruption of metadata can be catastrophic, leading to a disproportionately large loss of data. The frequency of metadata operations often exceeds the frequency of operations on file data. In many applications, 50-80% of all filesystem accesses are to metadata [10]. Metadata services are prone to hot spots where, as a computation shifts to a new dataset, a burst of operations are focused on a single file or on the files in a single directory or subdirectory. In general, metadata operations must be serializable—that is, they must be sequentially consistent and atomic.

We propose a separation of metadata servers from data servers, where the former would implement a strict consistency model, while the latter would allow various relaxed consistency models, as specified by users. Figure 6.11 illustrates this optimization. We define metadata broadly to include any data in which sequential consistency and/or reliability are principal concerns. This includes, for example, filesystem metadata such as directory and security information where loss or corruption of the metadata can be severe; as well as synchronization operations like locking, where sequential consistency is critical.

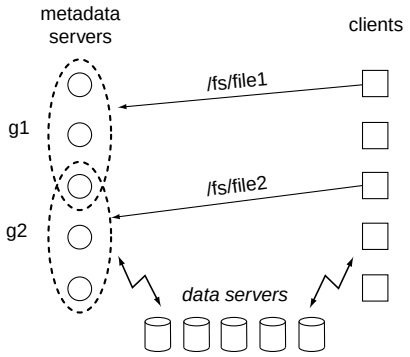


Figure 6.11: Each filename maps to a virtual metadata server group. While not required, the metadata and data servers that are shown here are in separate server farms.

6.4 Related Work

The area of distributed system and distributed filesystems has been an active research area for more than two decades. We discuss a few of the systems that shaped research in this area. Our work attempts to incorporate many of the ideas of these systems.

One of the first distributed filesystems used on a large scale was the Andrew Filesystem (AFS) [30]. Developed at Carnegie Mellon University in the early eighties, AFS was very popular due to its scalability, and its security which was provided using Kerberos. We are addressing some of the problems that researchers and users alike put forward with respect to AFS. AFS offers session semantics rather than UNIX semantics, which makes real-time concurrent file sharing almost impossible. In our system, we provide the user with UNIX semantics. Another possible drawback of AFS is availability. In the initial version, entire files were transferred to destinations on the open call. Our system is more selective in sending data. This gives our system higher availability and lower overhead on open calls.

CODA [6] is one AFS's descendants. Developed at Carnegie Mellon as well, CODA was designed as a distributed file system that would provide access to files even while machines were disconnected. The filesystem was designed for portable machines and did not enforce strict consistency of data, requiring user input to solve conflicts. Another issue is that CODA "hoards" files on the local machine to make them available while off-line, which makes CODA impractical to use on low-end machines with scarce resources. Our system currently does not address the availability of data while a mobile computer is off-line.

The latest filesystem in the series initiated by AFS is InterMezzo [9]. Although it is similar to MojaveFS in the sense that it was built as a filter between the VFS and the local filesystems, InterMezzo has a client-server architecture and it follows the direction taken by CODA in providing access to data while the computer is off-line.

SPRITE [44] is similar in many ways with MojaveFS. It is part of a distributed operating system

that was designed to transparently provide a distributed filesystem and process migration [16]. SPRITE also provides UNIX semantics. One major difference between the two systems is the fact that SPRITE was designed using a traditional client-server paradigm, which makes it less adaptive to failures and changes in system configuration than MojaveFS.

Lately, there have been other attempts at developing distributed filesystems. For example, there is the zFS filesystem [50] (the successor to the DSF project [17]) that makes use of transactions and a lease-based scheme to provide consistency in the presence of concurrent file access. The primary difference between zFS and our system is that our approach uses a group communication protocol to guarantee consistency. As a result, MojaveFS is also more decentralized as it does not rely on a transaction server (such as the one that zFS uses).

Also, following a recent trend towards the development of specialized metadata management services, several projects, like LH [10], NFS [45], Lustre [7], Vesta [13], InterMezzo [8], and Coda [51] have incorporated specialized metadata handlers into their filesystems. Still, most of these existing projects have taken a relatively static approach to the dissemination of their respective metadata services. All of the approaches listed above make use of a fixed set of dedicated metadata servers with little or no replication of the metadata. One of the deficiencies of these current mechanisms is that they do not have the flexibility to shift resources around dynamically in an efficient manner to match the specific access patterns of their clients. A more fine-grained replication policy based on groups of interest would be a more general solution (with the capacity for better performance characteristics). At this time we are not aware of any other project that has taken this type of highly distributed approach.

Chapter 7

Conclusion

This thesis introduced a new programming model based on speculations. A speculations defines an optimistic computation that is based on an assumption whose verification is done in parallel with the computation. If the assumption is invalidated the computation is rolled back to the point where it made the assumption and may take a different path of execution. If the assumption is validated the computation continues. Speculative behavior is introduced through three primitives: *speculate*, *commit*, and *abort*. The first one marks the beginning of a speculation, while the last two primitives are used when the assumption is validated or invalidated, respectively.

While speculations and traditional database transactions share properties, like atomicity, they are different in two major respects: speculations are not isolated, and they have dynamic scoping. The isolation relaxation of traditional transactions allows speculations to increase parallelism and to propagate in a distributed environment. Dynamic scoping, as opposed to static "speculative" or transactional blocks, enables more flexibility in writing speculative applications.

Speculations have the following properties and advantages.

- They can eliminate the need for error handling code in writing programs.
- They provide a mechanism similar to the concept of exceptions from programming languages that extends to parallel and distributed applications, and enables parallel applications to do synchronized rollbacks.
- They provide a safe recovery line in distributed applications.
- They may improve performance of distributed applications by allowing optimistic execution.

Speculations have been analyzed both from a theoretical and practical (implementation) point of view. The theoretical models presented three detailed operational semantics of speculative primitives, both considering nested and non-nested speculations, as well as speculative message passing and speculative shared objects. In the same theoretical framework it was defined a nonspeculative

model similar to nondeterministic Turing machines. The model has been shown to be equivalent in terms of operational semantics with its speculative counterpart.

An implementation of speculations inside the Linux kernel was presented in detail, while showing a parallel between the theoretical models and the actual implementation. Full support for speculations requires the accommodation of speculative execution at the filesystem level as well. In this thesis we also introduced the design of MojaveFS, a distributed filesystem with support for speculations. MojaveFS implements a strict consistency model for files and uses a decentralized approach to provide reliability. The sequential consistency model used by MojaveFS is supported by a new group communication protocol. The protocol implementation was presented along with mathematical proofs that attest to the correctness of the protocol.

Finally, the thesis presented a set of experimental results that measured the maximum overhead that one might encounter when incorporating speculative primitives in one's programs. These results show that the overhead of speculations is low, and that, depending on the time spent by the nonspeculative version at various synchronization points, the speculative version of a program may outperform the nonspeculative version.

Bibliography

[1] Exception handling for c. http://www.nicemice.net/cexcept/. 1.4

[2] General exception-handling facility for the c programming language. http://home.rochester.rr.com/bigbyofrocny/GEF/. 1.4

[3] Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Jonathan Stanton. The spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, John Hopkins University, 2004. 6.2.3.4

[4] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Franscisco, California, February 2005. 2.1.2.2

[5] Andrew P. Black, Vincent Cremet, Rachid Guerraoui, and Martin Odersky. An equational theory for transactions. In *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science*, pages 38–49. Australian Computer Society, Inc., 2003. 2.1.1

[6] Peter J. Braam. The coda distributed file system. *Linux Journal*, June 1998. 6.4

[7] Peter J. Braam. Lustre: A scalable, high-performance file system, 2002. 6.4

[8] Peter J. Braam, M. Callahan, and P. Schwan. The InterMezzo filesystem, 1999. 6.4

[9] Peter J. Braam, Michael Callahan, and Phil Schwan. The intermezzo filesystem. 1999. 6.4

[10] S. Brandt, L. Xue, E. Miller, and D. Long. Efficient metadata management in large distributed file systems, 2003. 6, 6.3, 6.4

[11] Fay Chang and Garth A. Gibson. Automatic i/o hint generation through speculative execution. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation.* 2.3

[12] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001. 6.1.3

[13] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 285–308. IEEE Computer Society Press and Wiley, New York, NY, 2001. 6, 6.2.1.1, 6.4

[14] Om P. Damani and Vijay K. Garg. How to recover efficiently and asynchronously when optimism fails. In *International Conference on Distributed Computing Systems*, pages 108–115, 1996. 2.2

[15] Xavier Defago, André Schiper, and Peter Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. Technical Report 200356, École Polytechnique Fédérale de Lausanne, September 2003. 6.1.3

[16] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software-Practice and Experience*, 21(8), August 1991. 6.4

[17] Zvi Dubitzky, Israel Gold, Ealan Henis, Julian Satran, and Dafna Scheinwald. Dsf - data sharing facility. technical report. Technical report, IBM Labs in Israel, Haifa University, Mount Carmel, http://www.haifa.il.ibm.com/pro jects/systems/dsf.html, 2000. 6.4

[18] Paul D. Ezhilchelvan, Raimundo A. Macêdo, and Santosh K. Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, page 296. IEEE Computer Society, 1995. 6.2.3.4

[19] Roy Friedman. Using virtual synchrony to develop efficient fault tolerant distributed shared memories. Technical report, Ithaca, NY, USA, 1995. 6.2.3.1

[20] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259, New York, NY, USA, 1987. ACM Press. 1.3, 2.1.1

[21] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1994. 1.1, 2, 2.1.1

[22] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, November 1994. Short Communication. 2.1.2.1

[23] Robert Harper, D.B. MacQueen, and R. Milner. Standard ML. Technical Report TR ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986. 6.2.3.4

[24] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. Oct 2003. 2.1.2.1

[25] Mark Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, 1998. 6.2.3.4

[26] M. Herlihy. A methodology for implementing highly concurrent data structures. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 197–206, New York, NY, USA, 1990. ACM Press. 2.1.2.1

[27] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993. 2.1.2.2

[28] Jason Hickey, Justin D. Smith, Brian Aydemir, Nathaniel Gray, Adam Granicz, and Cristian Ţăpuş. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR 2002.007, California Institute of Technology, Computer Science, July 2002. 5.1.1

[29] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. 2.3

[30] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988. 6.4

[31] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985. 2.2

[32] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *PODC*, pages 171–181, 1988. 2.2

[33] Amir Ashraf Kamil, Jimmy Zhigang Su, and Katherine Anne Yelick. Making sequential consistency practical in titanium. In *The International Conference for High Performance Computing and Communications, SuperComputing 2005*, 2005. 6.1.2

[34] An-Chow Lai and Babak Falsafi. Memory sharing predictor: the key to a speculative coherent dsm. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 172–183. IEEE Computer Society Press, 1999. 2.3

[35] Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. Comput. C*, 28(9):690–691, 1979. 6.1.2

[36] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, Computer Sciences Department, University of Wisconsin, 1997. 2.2

[37] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May 2005. 2.1.2.1

[38] Robert McNaughton, Paliath Narendran, and Friedrich Otto. Church-rosser thue systems and formal languages. *J. ACM*, 35(2):324–344, 1988. 4.3.1

[39] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. *Usenix*, 1996. 5.3.4

[40] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Cambridge, MA, USA, 1981. 2.1.1

[41] Nuno Neves, Miguel Castro, and Paulo Guedes. A checkpoint protocol for an entry consistent shared memory system. In *PODC*, pages 121–129, 1994. 2.2

[42] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 191–205, New York, NY, USA, 2005. ACM Press. 2.3

[43] Jeffrey Oplinger et al. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical report, Stanford, CA, USA, 1997. 2.3

[44] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The sprite network operating system. *IEEE Computer*, 21(2):23–26, February 1988. 6.4

[45] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. In *Proceedings of the 2^{nd} international system administration and networking conference (SANE2000)*, page 94, 2000. 6, 6.4

[46] Andreas Prinz and Bernhard Thalheim. Operational semantics of transactions. In *CRPITS'17: Proceedings of the Fourteenth Australasian database conference on Database technologies 2003*, pages 169–179. Australian Computer Society, Inc., 2003. 2.1.1

[47] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248, New York, NY, USA, 2005. ACM Press. 1.3, 2.2

[48] Ravi Rajwar and Philip A. Bernstein. Atomic transactional execution in hardware: A new high-performance abstraction for databases. In *Position paper for the 10th International Workshop on High Performance Transaction Systems*, Oct 2003. 2.1.2.2

[49] Robbert Van Renesse, Kenneth P. Birman, Bradford B. Glade, Katie Guo, Mark Hayden, Takako Hickey, Dalia Malki, Alex Vaysburd, and Werner Vogels. Horus: A flexible group communications system. Technical report, Ithaca, NY, USA, 1995. 6.2.3.4

[50] Ohad Rodeh and Avi Teperman. zfs - a scalable distributed file system using object disks. In *IEEE Symposium on Mass Storage Systems*, pages 207–218, Asilomar Conference Center, Pacific Grove, U.S., 2003. ACM Press. 6.4

[51] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990. 6.4

[52] A. Prasad Sistla and Jennifer L. Welch. Efficient distributed recovery using message logging. In *PODC*, pages 223–238, 1989. 2.2

[53] Justin D. Smith. Fault tolerance using whole-process migration and speculative execution. Master's thesis, California Institute of Technology, Department of Computer Science, 2003. 5.1, 5.1.1, 5.1.2, 5.2

[54] The Nuprl Staff. PRL: Proof refinement logic programmer's manual (Lambda PRL, VAX version). Cornell University, Department of Computer Science, 1983. 6.2.3.4

[55] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985. 2.2

[56] Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada, and Yutaka Ishikawa. Pm2: High performance communication middleware for heterogeneous network environments. In *Proceedings of the IEEE/ACM SC2000 Conference*, 2000. 2.2

[57] Cristian Ţăpuş and Jason Hickey. Extended operational semantics for simple distributed speculative execution. Technical Report caltechCSTR 2005.002, California Institute of Technology, Computer Science, January 2005. 4

[58] Cristian Ţăpuş, Aleksey Nogin, Jason Hickey, and Jerome White. A simple serializability mechanism for a distributed objects system. In David A. Bader and Ashfaq A. Khokhar, editors, *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems (PDCS-2004)*. International Society for Computers and Their Applications (ISCA), 2004. 6, 6.2.3

[59] Cristian Ţăpuş, Justin D. Smith, and Jason Hickey. Kernel level speculative DSM. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2003), Tokyo, Japan*, 2003. Workshop on Distributed Shared Memory (DSM). 2.3

[60] Douglas Thain and Miron Livny. The ethernet approach to grid computing. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*. 2.3

[61] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, and P. Schulthess. Optimistic synchronization and transactional consistency. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 331, Washington, DC, USA, 2002. IEEE Computer Society. 2.1.2.1

[62] Hua Zhong and Jason Nieh. Crak: Linux checkpoint / restart as a kernel module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, 2002. 2.2

Index