Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Speculative Execution by using Software Transactional Memory

Emanuel Amaral Couto (25816)

Lisboa
(2009)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Speculative Execution by using Software Transactional Memory

Emanuel Amaral Couto (25816)

Orientador: Prof. João Manuel Santos Lourenço
Co-orientador: Prof. Nuno Manuel Ribeiro Preguiça

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do Grau de Mestre em Engenharia Informática.

Lisboa
(2009)

Acknowledgements

This work was supported by an investigation scholarship of FCT/MCTES, project # PTDC/EIA/74325/2006. I would like to thank both my advisors João Lourenço and Nuno Preguiça, for all the help and support they gave during the development of this dissertation. I also would like to thank Ricardo Dias for sharing the code of his transactional memory framework and atomic block converter and for the help he gave during the design of our solution. Finally, I would like to thank my colleagues, family and friends for their support.

Summary

Many programs sequentially execute operations that take a long time to complete. Some of these operations may return a highly predictable result. If this is the case, speculative execution can improve the overall performance of the program.

Speculative execution is the execution of code whose result may not be needed. Generally it is used as a performance optimization. Instead of waiting for the result of a costly operation, speculative execution can be used to speculate the operation most probable result and continue executing based in this speculation. If later the speculation is confirmed to be correct, time had been gained. Otherwise, if the speculation is incorrect, the execution based in the speculation must abort and re-execute with the correct result.

In this dissertation we propose the design of an abstract process to add speculative execution to a program by doing source-to-source transformation. This abstract process is used in the definition of a mechanism and methodology that enable programmer to add speculative execution to the source code of programs. The abstract process is also used in the design of an automatic source-to-source transformation process that adds speculative execution to existing programs without user intervention. Finally, we also evaluate the performance impact of introducing speculative execution in database clients.

Existing proposals for the design of mechanisms to add speculative execution lacked portability in favor of performance. Some were designed to be implemented at kernel or hardware level. The process and mechanisms we propose in this dissertation can add speculative execution to the source of program, independently of the kernel or hardware that is used.

From our experiments we have concluded that database clients can improve their performance by using speculative execution. There is nothing in the system we propose that limits in the scope of database clients. Although this was the scope of the case study, we strongly believe that other programs can benefit from the proposed process and mechanisms for introduction of speculative execution.

Keywords: Speculative execution, Futures, Automatic transformation, Database applications, Concurrency, Performance.

Sumário

Muitos programas executam operações sequencialmente que levam muito tempo a concluir. Algumas dessas operações podem retornar resultados altamente previsíveis. Se for esse o caso, execução especulativa pode melhorar a performance global do programa em causa.

Execução especulativa é a execução de código em que o resultado pode não ser necessário. Geralmente é utilizada como uma optimização de desempenho. Em vez de esperar pelo resultado de uma operação morosa, execução especulativa pode ser usada para especular o resultado mais provável da operação e continuar a execução baseando-se na especulação. Se mais tarde se confirmar que a especulação estava correcta, ganha-se tempo. Caso contrário, se a especulação estava incorrecta, a computação que executou baseando-se numa especulação tem de abortar e re-executar com o resultado correcto.

Nesta dissertação propõe-se o desenho de um processo abstracto para adicionar execução especulativa a um programa, através de transformação source-to-source. Este processo abstracto é usado na definição de um mecanismo e metodologia que permitem ao programador adicionar execução especulativa ao código fonte de programas. Este processo abstracto é também usado no desenho de um processo automático de transformação source-to-source que adiciona execução especulativa a programas, sem intervenção do utilizador. Por fim, avalia-se o impacto no desempenho de clientes de base de dados ao introduzir execução especulativa.

As propostas existentes para o desenho de mecanismos para adicionar execução especulativa favoreciam a performance em troca de portabilidade. Alguns dos mecanismos eram desenhados para serem implementados ao nível do núcleo do sistema ou de hardware. O processo e mecanismos propostos nesta dissertação permitem adicionar execução especulativa ao código fonte de um programa, independentemente do núcleo de sistema e hardware usado.

Através das nossas experiências concluimos que clientes de bases de dados podem melhorar a sua performance ao utilizar execução especulativa. Não existe nada no sistema que propomos que limite a abrangência a sistemas de bases de dados. Apesar de este ter sido o nosso case de estudo, acreditamos que outros programas podem beneficiar do processo e mecanismos propostos para introdução de execução especulativa.

Palavras-chave: Execução especulativa, Futuros, Transformação automática, Aplicações de base de dados, Concorrência, Performance.

Contents

List of Figures

List of Tables

1

Introduction

Many programs take a longer time than desired to complete. Frequently, these programs execute operations in sequence, waiting for an operation to complete before proceeding to the next one. Some of these operations that execute sequentially may have a highly predictable result. If this is the case, the program can resort to speculative execution to improve its performance.

Speculative execution is the execution of code whose result may not be needed and can be used as a performance optimization. A program with sequential operations can be optimized by using speculative execution, if some of its operations have a highly predictable result. Instead of waiting for the results of these operations, they can be immediately speculated to the most probable result and continue the computation based in the speculated values as a speculative computation. Later, if the speculation was correct, time had been gained. Otherwise, if the speculation was incorrect, the speculative computation must be aborted and re-executed, replacing the speculated value with the correct result.

The operations in which their results are to be speculated, are usually executed concurrently with their following computations. However, this means speculative execution is only beneficial if the hardware where it is used has sufficient processing power to run the operation and its subsequent computations simultaneously in different processors. Modern computers have multi-core processors. With such computers it is possible to run the operation and its subsequent computation in separate cores. Moreover, the tendency is for the number of cores increase in the future, reducing further the cost of speculative execution.

An example of programs that can benefit from speculative execution are database clients. Database clients are computer programs that access a database by sending database operation requests to a database system. The time a database client takes to process and reply to a database may be long, depending on the database itself and the latency of the communications between the database client and the database system. Some of the results of requesting

database operations are highly predictable. Namely, the result from requesting a write-only database operation is usually highly predictable (e.g., true if the write-only database operation succeeds or false otherwise). Speculative execution can be used to speculate the result of such operations.

This dissertation has two main goals. The first goal is to study and develop mechanisms that enable to integrate speculative execution to specific operations of a program. The second goal is to study the performance improvement programs can have when they incorporate speculative execution.

1.1 Problem Statement and Work Goals

The objective of this dissertation is to study and design mechanisms to explore speculative execution in programs.

Speculative execution is used in many systems to improve the performance of some aspects of them. Some systems [8, 10] implement speculative execution for specific operations to serve a unique purpose. Other systems [12, 15, 21] use a mechanism that enables a user to explicitly add speculative execution to some of the operations. We call such mechanisms as speculative execution mechanisms. The latter solution is clearly the best if one wants to add speculative execution to various operations. However, the design of these mechanisms lacked portability in favor of performance.

There are few proposals that have contributed to the design of software speculative execution mechanisms, and are essentially implemented at kernel level [12,21]. Any implementation at kernel level requires the use of a specific operating system. One of the main goals of this dissertation is to design speculative execution mechanisms to enable the introduction of speculative execution to the source code of programs, independently of the operating system and hardware that is used. Another of the main goals of this dissertation is to study the effects in the performance of database clients when integrating speculative execution to their source code. We believe database clients are good candidates to increase their performance through the use of speculative execution.

1.2 Proposed Solution

Our solution has four main steps. The first step is to study and design an abstract process to add speculative execution to the source code of a program. This process should be generic enough to allow the integration of speculative execution into the source code of any program, independently from its nature, area of application or specific code. Our proposal is for the process to introduce concurrent execution to a set of sequential computations with highly predictable results. If another computation depends on the result from one of the transformed computations and the result is not yet known, the computation may continue executing by speculating that the result will be the most probable. If this speculation is correct time is gained. Otherwise, if the speculation is incorrect, the speculative computation must abort and re-execute.

This process is going to be the base of all speculative execution mechanisms we develop in this dissertation. We call this process the *transformation process*.

The second step is to is design and implement a speculative execution mechanism for the programmer to easily add speculative execution to the source code of a program. Our proposal is to use futures for this. Futures are a mechanism that enable to concurrently execute a computation and later get its result when necessary. Using only futures a programmer can manually apply a simpler version of our proposed *transformation process* to the source code of programs.

The third step is to design and implement an automatic speculative execution mechanism to automatically add speculative execution to the source code of a program. This mechanism should enable integration of speculative execution to a program with minimum possible user intervention. Our proposal is for the mechanism to be able to apply our proposed *transformation process* to the source code of a program with source-to-source transformation. We also propose the use of futures to concurrently execute computations and transactional memory [11] to abort and re-execute computations based in an incorrect speculation.

The final step is to evaluate the performance improvements database clients can have by using speculative execution. We believe database clients are good candidates to improve the performance through the use of speculative execution.

1.3 Contributions

This dissertation provides the following contributions. The design of an abstract process to add speculative execution to the source code of a program. The design of a speculative execution mechanisms for the programmer to easily add speculative execution to the source code of a program. The design of an automatic speculative execution mechanism for the user to automatically add speculative execution to the source code of a program. The study of the effects in the performance of database clients when speculative execution is integrated into its source code.

1.4 Document Outline

The remainder of this dissertation is organized has follows. Chapter 2 describes the related work. Chapter 3 describes an abstract process to add speculative execution to the source code of a program. Chapter 4 describes a speculative execution mechanism for the programmer to easily add speculative execution to the source code of a program. Chapter 5 describes an automatic speculative execution mechanism for the user to automatically add speculative execution to the source code of a program. In Chapter 6 we test our developed mechanisms in a transaction processing benchmark we have developed ourselves. Finally, in Chapter 7 we present the conclusions of this dissertation and a list of tasks to achieve in future work.

2

Related Work

2.1 Introduction

Understanding our proposal implies having basic knowledge about concurrency control, database systems, transactional memory and speculative execution.

This chapter starts by presenting the fundamentals about concurrency control. Concurrency control is the basis for the remainder of this dissertation and for that reason it is very important to have basic knowledge about this topic. Next, we present the fundamentals about database systems and transactions. Having basic knowledge about database systems and transactions is not only important because one of our objective is to improve the performance of database clients, but also because this topic is good basis to understand transactional memory. Next, we present the fundamentals about transactional memory. One of our proposals is to use transactional memory as a mechanism that enables to abort speculative computations, hence the importance of this topic. Afterwards, we present the definition of speculative execution and a range of existing systems that use speculative execution, systems of which we believe to be helpful to better understand our proposal. Finally, since this project is part of the Byzantium project, we present the latest version of the Byzantium system. In the long run, the objective of this dissertation is to add speculative execution by using transactional memory to the Byzantium system.

2.2 Concurrency Control

Concurrency is a property of systems in which several processes or subtasks are making progress at the same time, and potentially interacting with each other [18]. These subtasks might be running truly simultaneously on separate physical processors, or time-sharing on a single pro-

cessor.

Because processes can interact with each other, the number of possible execution paths can be very large and the resulting behavior extremely complex. Race conditions may occur in concurrent computing and result in unpredictable system behavior. A race condition is a flaw in a system or process where the output and/or result is unexpectedly and critically dependent on the sequence or timing of other events. Concurrency control mechanisms are used to control concurrent execution of operations and prevent race conditions but their use can lead to other problems such as deadlock [5].

Concurrency control ensures that concurrent operations generate correct results. It is used in many fields of computer science and most commonly in computer programming, operating systems, databases and multiprocessors. In traditional computer programming, processes are coordinated by explicit synchronization mechanisms. In systems such as database systems, concurrency control is implicit.

The most usual way of enforcing concurrency control is by using lock synchronization mechanisms. Most locking designs block the execution of an entity requesting a lock until it is allowed to access the locked resource. Careless use of locks may lead to many problems such as the ones listed next:

- *Deadlock*. Two processes are each waiting for the other to release a resource, or more than two processes are waiting for resources in a circular chain. The result is that none of the processes involved in the deadlock progresses, since they stay blocked waiting until one of them forcefully releases its lock. Neither the processes individually nor the overall system exhibits any progress.

- *Livelock*. Similar to a deadlock, except that the states of the processes involved in the livelock constantly change in response to the changes of the other processes. Individual processes exhibit progress (they are not blocked) but the overall system does not progress.

- *Priority inversion*. An high priority process is forced to wait until a low priority process that is holding a common lock releases it.

- *Convoying*. If a thread holding a lock is descheduled due to a time-slice exhaustion or a page fault, all other threads willing to acquire will not be allowed to make use of their time-slices.

- *Composition*. Composition is the property of combining simple entities to form a complex one. For example, consider a function to delete an item from a table and another function to insert an item to a table. In sequential programming, one could combine both functions to create a function that moves an item from a table to another table. This cannot be done with explicit lock synchronization mechanisms since they are not composable [11].

There are alternative mechanisms that avoid locks completely, such as non-blocking synchronization mechanisms [9] and transactional memory [11].

2.3 Database Systems and Transactions

Database Management Systems (DBMS) typically use implicit concurrency control schemes to ensure that database transactions perform correctly and do not violate the data integrity of the database. Additionally, database systems use recovery schemes to restore the database to a consistent state in the presence of failures.

Due to their similarities, database transactions are a good study basis for Transactional Memory (TM) (described in Section 2.4).

2.3.1 Database Transaction Concept

A database transaction is a collection of operations that forms a single logical unit of work, treated in a coherent and reliable way independent of other transactions. Usually a transaction is written in a high-level data-manipulation or programming language and it is delimited by statements in the form "begin transaction" and "end transaction".

To ensure the integrity of data, database systems maintain the ACID properties of the transactions [20]:

- *Atomicity*. Either all operations of the transaction are reflected properly in the database, or none are. Atomicity is responsibility of the database system transaction-management component.
- *Consistency*. Guarantee that the execution of a transaction in isolation preserves the consistency of the database. Ensuring consistency of an individual transaction is responsibility of the application programmer who codes the transaction.
- *Isolation*. For every two concurrent transactions, `A` and `B`, it appears to `A` the `B` occurred either before or after `A`, and vice-versa. Isolation is responsibility of the database system concurrency-control component.
- *Durability*. Guarantee that once a transaction completes successfully its modifications to the database persist, even in the presence of system failures. Durability is responsibility of the database system recovery-management component.

A transaction that completes successfully commits. Otherwise the transaction aborts and must have no effects in the database to ensure the atomicity property. The effects of a committed transaction are permanent and cannot be undone by aborting it. Only a compensating transaction (another transaction that when executed compensates the effects of a previously committed transaction) can "undo" the effects of a committed transaction.

2.3.2 Database Concurrency Control

Usually transaction-processing systems allow multiple transactions to run concurrently. Despite being much easier to ensure consistency when transactions run sequentially (that is, one

at time), allowing concurrent transactions increases throughput and resource utilization, and reduces execution delays [20].

The concept of correct concurrent execution on database systems can be explained through schedules. A schedule represents the chronological order on which instructions are executed in a system. A schedule for a set of transactions is serial if their instructions are not interleaved. Otherwise the schedule is concurrent. Since instructions are interleaved in concurrent schedules, some do not preserve the isolation property and could leave the database in an inconsistent state. Leaving control of execution entirely to the operating system does not guarantee that any schedule that gets executed will leave the database in a consistent state. So, it is the database system concurrency control manager that must guarantee it.

Serializable schedules have the serializability property. That is, they produce the same effect as a serial schedule. Consequently, database consistency can be ensured if only serializable schedules are executed. The database system ensures the isolation property by controlling the interaction among concurrent transactions through concurrency control schemes. There are many concurrency control schemes such as lock based protocols (e.g., two phase locking protocol), graph based protocols (e.g., tree protocol), timestamp based protocols (e.g., timestamp ordering protocol), validation based protocols and multiversion schemes (e.g., multiversion timestamp ordering and multiversion two phase locking protocols). Most of them guarantee schedules that are conflict serializable (a type of serializable schedules), unless relaxed forms of serializability are allowed for better performance [20].

The described concurrency control schemes can be categorized as optimistic or pessimistic. Pessimistic concurrency control schemes block or rollback whenever a conflict is detected, even if there is a chance that the schedule may be conflict serializable (e.g., locked and timestamp ordering based protocols). On the other hand, optimistic concurrency control schemes optimistically assume they will be able to finish execution and validate in the end of the transaction (e.g., validation based protocols).

2.3.3 Database Isolation levels

Serializability ensures that if every transaction maintains consistency when executed alone, the concurrent execution of transactions will also maintain consistency. However, protocols that ensure serializability may allow little concurrency for certain applications. In these cases, weaker levels of consistency are used. The SQL standard allows a transaction to specify that it may be executed in a non serializable manner with respect to other transactions.

The isolation levels specified by the SQL-92 are the following [20]:

- Serializable: Transactions produce the same effect as if transactions had executed serially, one at time.

- Repeatable read: Allows only committed records to be read, and that between two reads of a record by a transaction, no other transaction is allowed to update the record.

- Read committed: Allows only committed records to be read, but does not require repeatable reads.
- Read uncommitted: Allows uncommitted records to be read.

When using weaker isolation levels the following phenomenas may occur [2]:

- Dirty Read: Transaction T_i modifies a data item that is later read by another transaction T_j before T_i performs a commit or rollback. If T_i then performs a rollback, T_j has read a data item that was never committed.
- Non Repeatable Read or Fuzzy Read: Transaction T_i reads a data item that is later modified or deleted by another transaction T_j that then commits. Afterwards, if T_i attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted.
- Phantom read: Transaction T_i reads a set of data items satisfying some condition. Transaction T_j then creates a data items that satisfy the previous condition and commits. If T_i then repeats its read with the same condition, it gets a different set of data items from before.

Table 2.1 shows which phenomenas may occur when using a correspondent isolation level. Besides the ANSI SQL-92 isolation levels, database systems implement other isolation levels, of which Snapshot Isolation (SI) is one of the most relevant [2]. SI is a multiversion isolation type. In SI, transactions appear to operate in a personal snapshot of the database, taken at the start of the transaction. A transaction successfully commits only if the data items it updated have not been modified by any other concurrent transaction since the snapshot was taken. Otherwise the transaction aborts. Note that in this way readers never get blocked by writers.

Isolation Level	Dirty Read	Fuzzy Read	Phantom
Serializable	Not possible	Not possible	Not possible
Repeatable Read	Not possible	Not possible	Possible
Read Committed	Not possible	Possible	Possible
Read Uncommitted	Possible	Possible	Possible

Table 2.1: ANSI SQL 92 Isolation Levels possible phenomenas

Although SI does not suffer from the above phenomenas, it is not serializable [2]. However, it can be made serializable, as shown in [7].

2.3.4 Database Recovery System

There are many types of failures that can occur in a database system, each of which needs to be dealt in a different manner. The simplest are the ones where there is no loss of information, and the hardest, the ones where there is. Transaction failures can happen because of logical errors (internal error such as bad input, data not found, overflow or resource limit exceeded) or

system errors (undesirable state such as deadlock). Either way the transaction cannot continue and can only be restarted. Computer systems can also suffer from system crashes or disk failures. If a system crash occurs volatile storage is lost, but it is usually assumed that non volatile storage is not [20]. This assumption is called the fail-stop assumption. To recover from disk failures, copies to other disks or backups are used.

Database systems use recovery schemes to restore the database to a consistent state after a crash. One example of such recovery scheme is the log based recovery. In a log based recovery database modifications are stored in a log as a sequence of log records. There are several types of log records, such as the update log record where a single database write is described. Whenever a log transaction performs a write operation, a log record is created before the database is modified. This allows to undo or redo the modifications if necessary. Additionally, the log must reside in stable storage to be useful for recovery from system or disk failures. To ensure transaction atomicity despite failures, two techniques are used with the log: deferred and immediate modification techniques.

In the deferred modification technique all database modifications are stored in the log, and the write operations deferred until the transaction partially commits (partially commit state is the transaction state where the final action as been executed, but the transaction has not yet committed). If the system crashes or the transaction aborts before it completes, the information in the log can be ignored. If a system crashes after the transaction completes, the system can redo the transaction modifications by using the log information, ensuring the durability property.

In the immediate modification technique, modifications are allowed to be output to the database during the transaction execution. However, the modifications are only allowed after being stored in the log to ensure atomicity. This way, in case of a system crash or transaction abort, the system can undo the modifications by using the log. Moreover, like in the deferred case, in presence of a system crash, the system can redo the operations that completed by using the log.

2.4 Transactional Memory

For many years the most used computers had a single processor in its core. However, around 2004, exponential performance growth of these computers ended [14]. It became increasingly difficult to improve their performance.

The Industry's response to these changes was the introduction of multi-core computers. These computers have a parallel architecture: a single chip with two or more processors connected through shared memory. Unlike single-core computers, where the only option is to improve the performance of a single processor, the architecture of multi-core computers allows to add extra processors. As the number of processors increase so does the number of instructions executed per second.

Performance increase by adding processors is only possible if the operating system and software are designed to take advantage of it. Unfortunately, parallel programs are much

more difficult to design, write and debug than their sequential equivalents, and often perform poorly [11].

One source of difficulty in parallel programming is the lack of effective synchronization mechanisms for abstraction and composition. Abstraction is the process of generalization of an entity, by capturing features that are essential to understand and manipulate it for a particular purpose. Composition is the process of combining simple entities to form abstract composite entities.

Low level parallel programming models, such as threads and explicit synchronization, are unsuitable for constructing abstractions because explicit synchronization is not composable. A program that uses an abstraction containing explicit synchronization must be aware of its details, to avoid causing races or deadlocks.

For instance, consider a hash table that supports thread safe `Insert` and `Delete` operations. In a sequential program, each of these can be an abstraction. However, if we want to construct a new operation `Move` in a parallel program, which deletes an item from a hash table and inserts it into another, by simply composing the `Insert` and `Delete` operations, we cannot hide the intermediate state where neither hash table contains the element. This is due to the fact that the operations `Insert` and `Delete` lock the table only for the duration of each individual operation. Fixing this problem requires new methods, such as methods that lock and unlock both hash tables. However, that breaks the hash table abstraction by exposing implementation detail. Moreover, the hash tables must be locked and unlocked in a global consistent order to prevent a deadlock.

Abstraction and composition make programming far easier since they open the possibility to reuse components by depending on their interfaces, rather than having to understand their implementation. One of the major difficulties in concurrent programming by using explicit synchronization mechanisms is the fact that they are not composable [11].

2.4.1 Transactional Memory Concept

While parallelism has been a difficult problem for general programming, database systems have successfully exploited concurrency. Databases achieve good performance by running several transactions concurrently and on multiple processors whenever possible. Furthermore, since concurrency control is implicit, transaction programmers do not have to worry about concurrency issues.

Database transactions offer a proven abstraction mechanism for constructing reusable parallel computations [11]. A computation executed in a transaction need not expose the data items it accesses or the order in which these accesses occur. Transactions can be composed by executing sub-transactions in the scope of a surrounding transaction.

With the advent of the multi-core computers an old idea came in interest: incorporate transactions into a programming model used to write parallel programs. While programming language transactions share some similarities with database transactions, the implementation and execution environments differ greatly, as databases typically store data in stable memory and

programs store data in volatile memory. Because of this difference a new name was given to this abstraction, *Transactional Memory* (TM).

The following differences between transactional memory and database systems are among the most important [11]:

- Database data resides on disk rather than in memory. Databases can freely trade disk access with computations since disk access is considerably slower in comparison. On the other hand, transactional memory accesses the main memory and a memory access does not give opportunity for a transaction to do much computation. Consequently, hardware support is more attractive for TM than for database systems.
- Transactional memory is not durable since data in memory does not survive program termination. Consequently, the durability property of the ACID properties is not important in TM. This simplifies the implementation of TM, since the need to record data permanently on disk before a transaction commits considerably complicates a database system.
- Transactional memory is a retrofit into a complex world of existing programming languages, programming paradigms, libraries, programs, and operating systems. For TM to be successful, there must be coexistence with the existing infrastructure, even if a long-term goal may be to supplant parts of this infrastructure with transactions. Programmers will find it difficult to adopt otherwise.

2.4.2 Transactional Memory Concurrency Control Mechanisms

Transactional memory provides concurrency control mechanisms for mutual exclusion and coordination.

The atomic statement delimits a block of code that should execute in a transaction. Figure 2.1 shows an example of such atomic block. The transaction executes method `foo` from object `x` if it is not null and changes the value of boolean `y` to the value true. The resulting transaction is dynamically scoped — it encompasses all code executed while control is in the atomic block. So the code in method `foo` also executes transactionally.

```
1  atomic {
2    if (x != null) x.foo();
3    y = true;
4  }
```

Figure 2.1: Example of an atomic block

Like database transactions, memory transactions must obey to an isolation criteria. In fact, many implementations are based on the isolation criteria from database systems. That is, serializability (see section 2.3.2). This model however is not fully applicable because of the differences between database-manipulating and multi-threaded programs. The ACI properties only apply to legal interactions of atomic blocks [11]. For example, it says nothing about interactions of atomic blocks with code outside of a transaction.

Within a single process, transactions often need to coordinate execution order. For instance, when a transaction produces a value used by another transaction. The `retry` statement can be used to coordinate transactions. A transaction that executes a `retry` statement aborts and re-executes and usually does so in hope of producing a different result. Figure 2.2 shows an example of a transaction using the `retry` statement. The transaction waits until a buffer contains an item before continuing execution.

```
1  atomic {
2    if (buffer.isEmpty()) retry;
3    Object x = buffer.getElement();
4    ...
5  }
```

Figure 2.2: Example program of a retry statement

The `orElse` statement is another mechanism to coordinate the execution of two transactions. Consider the transactions T_1 and T_2. T_1 `orElse` T_2 starts by executing transaction T_1:

- If T_1 commits or explicitly aborts, the `orElse` statement terminates without executing T_2;
- If T_1 executes a `retry` statement, the `orElse` statement starts executing T_2;
- If T_2 commits or explicitly aborts, the `orElse` statement finishes its execution;
- If T_2 executes a `retry` statement, the `orElse` statement waits until a location read by either transaction changes, and then re-executes itself in the same manner.

For example, the objective of the program in Figure 2.3 is to read a value from one of two transactional queues. The method `getElement()` is a blocking operation that will retry if the queue is empty, to wait until there is a value to return. In the execution of the program, the value from `Q1` is returned if possible, otherwise the same is tried with `Q2` and only blocks if both are empty.

```
1  atomic {
2    { x = Q1.getElement(); }
3    orElse
4    { x = Q2.getElement(); }
5  }
```

Figure 2.3: Example program of an `orElse` statement

2.4.3 Transaction Memory Design

In TM systems, there are many alternatives in the semantics of a transaction.

A system may have weak or strong isolation (commonly referred as weak and strong atomicity). A system with weak isolation only guarantees transactional semantics among transactions, which means that non-transactional code may conflict with a transaction. The non-transactional code may not follow the protocols of the TM system and, consequently, disrupt correct execution of the transaction. On the other hand, a system with strong isolation guarantees transactional semantics between transactions and non-transactional code. Strong isolation automatically converts all operations outside an atomic block into individual transactional operations. That way, it replicates a database model in which all accesses to a shared state execute in transactions.

A nested transaction is a transaction whose execution is properly contained in the dynamic extent of another transaction. The inner transaction sees the modifications to program state made by the outer transaction. In TM, there are different types of nested transactions, each of which, behaves differently:

- If transactions are flattened, aborting the inner transaction causes the outer transaction to abort, and committing the inner transaction has no effect until the outer transaction commits;
- If transactions are closed, aborting a transaction does not terminate the parent transaction. When a closed inner transaction aborts or commits, control passes to its surrounding transaction. However, nonsurrounding transactions do not see these changes until the outermost surrounding transaction commits;
- If transactions are open, committing a transaction makes its changes visible to all other transactions in the system, even if the surrounding transaction is still executing. Moreover, even if the parent transaction aborts, the results of nested, open transactions will remain committed.

Despite being easy to implement, flattened transaction are a poor programming abstraction that subverts program composition, since an abort terminates all surrounding transactions. Closed transactions solve the problem of the flattened transactions but they have higher overhead. Open transactions can improve program performance but may also subvert the ACI properties of transactions.

2.4.4 Transaction Memory Implementation

Implementation differences in transactional memory can affect a system's programming model and performance.

Transaction granularity is the unit of storage over which a TM system detects conflicts. Most software transactional memory (STM) systems extend an object-based language and implement object granularity. Systems with object granularity detect conflicting access to an object even if the transactions access different fields. Most hardware transactional memory (HTM) systems detect conflicts at word or block (an adjacent fixed size group of words) granularity.

Object granularity has comprehension and implementation advantages [11]. However, word or block granularity permits finer grain sharing than objects do, potentially allowing higher concurrency.

A TM system may use direct update or deferred update. The idea is similar to the immediate and deferred modification in the database log recovery system discussed in Section 2.3.4. A transaction typically modifies a data item. If the TM uses direct update, the transaction directly modifies the data item and the system must record the original values of the modified data item to be able to restore them later. If the TM uses deferred update, the transaction updates the data item in a location private to the transaction. The transaction ensures that it reads the updated values from its private copy. When the transaction commits, it copies the modified values from the private data to the original data item. If the transaction aborts, the system discards its the private copy.

Deferred update use of private copies has the advantage of being more efficient than direct update when aborting a transaction, since direct update must undo the effects done to a data item when aborting the transaction. On the other hand, direct update reduces the amount of work needed to commit a transaction, since it does not need to copy from a private copy. Also, since direct update only logs the original values of a data item instead of doing a full copy, it may need less memory if the data items are only partially modified. Deferred update needs exclusive access to the data items for less time. With direct update one must "lock" the access until the transaction commits or has to deal with unrepeatable reads and similar problems. Deferred update only needs exclusive access to data when copying from the private copy to the final destination.

TM systems require concurrency control schemes. Most HTM and STM systems use optimistic concurrency control (see section 2.3.2). In TM, the concurrency control schemes may have some forward guarantees: the guarantee that a transaction seeking to access an object will eventually complete its computation.

In a TM system with blocking synchronization the threads may deadlock or be delayed for an arbitrary period as a thread waits for a lock that is held by another thread. A TM system may also have non blocking synchronization. This mechanism guarantees that a stalled thread cannot cause all threads to stall indefinitely. Some of the guarantees of non blocking synchronization follow [11]:

- *Wait freedom*. All threads contending for a common set of objects make forward progress in a finite number of their individual time steps;
- *Lock freedom*. At least one thread from a set of threads contending for a common set of objects makes forward progress in a finite number of time steps of the concurrent threads;
- *Obstruction freedom*. A thread will make forward progress in a finite number of its own time steps in the absence of contention over shared objects.

A conflict between two transactions can be resolved by aborting either one. TM systems usually have a contention manager, which implements at least one contention resolution pol-

icy that decides which conflicting transaction to abort. The choice of which transaction to abort can affect the performance of the system and semantic guarantees [11]. For example, some contention resolution protocols may result in starvation — a transaction never gets to be executed, since when it conflicts with other transactions the last is always be favored to prevail.

2.5 Speculative Execution

Speculative execution is the execution of code in which the result may not be needed. Generally it is used as a performance optimization. For example, a system can use speculative execution to predict the result of some costly computation. The result should be highly predictable, in order to minimize the chances of computing an unneeded result, and there should be a potential gain by speculating, such as reducing the time to achieve the necessary result.

Speculative execution requires memory and processing power. Additionally, if a speculation proves to be incorrect, there will be a waste of resource usage during its execution since its results will never be used. Modern computers have multi-core processors and high memory capacity and, as most of the time these resources are not fully utilized, they can safely be used for speculative execution. Moreover, it is very likely that the number of cores will increase as well as the memory capacity, reducing even further the cost of speculative execution.

2.5.1 Kernel Level Speculative Execution

One approach to provide speculative execution is by extending support at the operating system level. Speculator [12] provides Linux kernel support for speculative execution. Distributed file systems (DFS) were the first candidates of Speculator because they can strongly benefit from speculative execution. However, many other applications could benefit from it.

Distributed file systems often perform much worse than local file systems because they perform remote synchronous I/O operations for verifying cache coherence and providing data safety. To overcome this problem, many DFS weaken consistency and safety properties. Local file systems typically guarantee that a process that reads data from a file will see all modifications previously completed by other processes. However, for example, a DFS that provides close-to-open consistency, guarantees only that a client that opens a file will see the modifications made by other clients that have previously closed the file. The advantage of such semantics is that it reduces the number of network messages to validate the cache, thus improving performance.

Instead of weakening consistency and data safety, the approach taken by Speculator to improve performance is to speculate the result of the issued operations and continue execution, rather than block while waiting for the result. Figure 2.4 shows how speculative execution is used in a DFS. Two clients collaborate on a shared project that consists of three files: A, B and C. At start, both clients have up-to-date copies of all files in cache. Client 1 modifies files A and B and then client 2 opens files C and B.

Figure 2.4a illustrates the case where there is no speculation. To modify a file, Client 1 issues

a sequence of write operations and a final commit operation. Each operation only returns when it receives the result it executes from the server. To open a file, Client 2 issues a `getattr` operation and waits for the reply from the server to check whether the cached file is fresh or not.

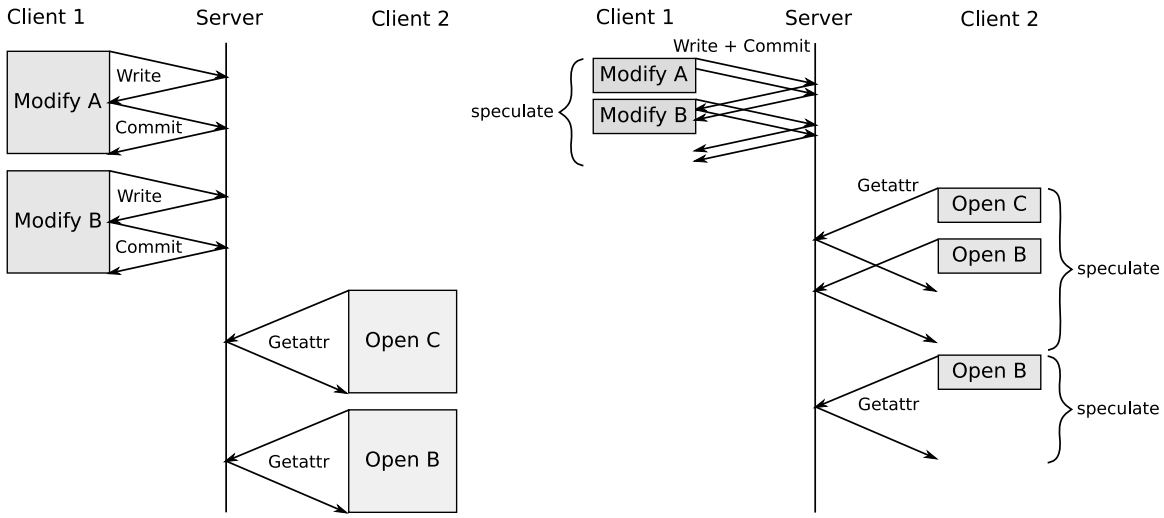


Figure 2.4: Example of speculative execution for a DFS

Figure 2.4b shows the case where there is speculation. When modifying a file, Client 1 does not wait for any reply from the server, simply speculating that all modifications will succeed at the server. Client 2 speculates that its cached copy of C and B are up-to-date. If the cached file is not fresh (e.g., B has been modified by Client 1), when the server replies, the system must ensure that it gets to a state where nobody used the false speculation and then retry the open operation. When speculation is used, the DFS performance increases since latency is hidden: multiple file system operations can execute concurrently, and computation can be overlapped with I/O.

To implement this idea, Speculator creates a checkpoint of the currently running process when a speculation starts. If the speculation fails, the checkpoint is used to restore the process state. The execution then proceeds with the correct value, giving the illusion that speculative execution never happened. If the speculation was correct, the checkpoint is discarded.

Speculator allows interprocess communication (IPC). Speculator tracks interdependencies between speculative processes, so that when a speculation fails, all dependent processes on that speculation rollback to a consistent state. Modifications to file systems and systems calls were necessary to support speculative state (the state of a process when it is executing based in a speculation) and its correct execution.

Speculator guarantees correctness of speculative execution by preventing speculative processes from externalizing output and by preventing a process from viewing speculative state, unless it is already dependent upon that state. If a non-speculative process tries to view speculative state, Speculator either blocks the process until the state becomes non-speculative, or it

makes the process speculative.

In [21] the System V IPC shared memory API is extended to support distributed shared memory (DSM) by using speculative execution. The developed DSM system is part of the Mojave system, which consists of a compiler together with operating system extensions designed to simplify distributed system programming through the use of high-level language primitives. The Mojave system provides primitives to create, commit and abort a speculation. Like in Speculator, it uses checkpointing, so that when a speculation fails, the calling process returns to the captured state where the speculation (and subsequent speculations) never happened.

To ensure consistency of data stored in the shared memory space, the DSM system uses total-order communication among the participants. Total-order protocols are expensive due to the amount of communication required among participants to achieve consensus on message order. Also, the latency of messages in the system is high because a message cannot be delivered until all other participants have acknowledged receipt. To optimize the protocol performance, if a message waits long enough, the system speculates that it can be delivered.

The DSM system also uses speculations on read and write operations to reduce latency and introduce fault tolerance. In a total order protocol, a process issuing a read operation for a page would normally have to wait until all writes issued before the read have been processed. In the DSM system, instead of waiting, the reader process speculates that all pending writes for the page have been processed. If later a write for the page with an earlier logical timestamp is delivered, the speculation aborts. Otherwise, if there are no pending writes for the page with an earlier logical timestamp, the speculation commits.

Additionally, to support fault-tolerance in event of a machine failure, the system speculates that the write calls will succeed. If a participant does not receive the message due to a failure, the write speculation is aborted and the sender may retry the write once it identifies the new set of participants.

Finally, the DSM system uses speculations to improve distributed lock mechanisms. A process that requests a lock speculates it to be granted and, if it is, the speculation commits, otherwise it aborts.

Clearly, both systems have very different goals in mind. While the goal of Speculator was to introduce speculative execution to the Linux kernel, the DSM system could have been developed as a user level application. The reason for developing it at kernel level was in consequence of testing and concluding that having a page management mechanism inside the kernel would reduce the overhead of the system.

Providing support for speculative execution at kernel level does have some advantages, such as transparency. However, support is only available for the system on which it was developed. In other words, such systems lack portability.

2.5.2 Database Speculative Execution

Some systems try to improve the performance of database systems by using speculative execution. Ganymed [16] is a database replication middleware intended to provide high scalability

and strong consistency. Figure 2.5 illustrates the Ganymed architecture. The Ganymed architecture is composed of clients, one or more schedulers, one manager, one master (or primary) replica and a set of slave (or secondary) replicas.

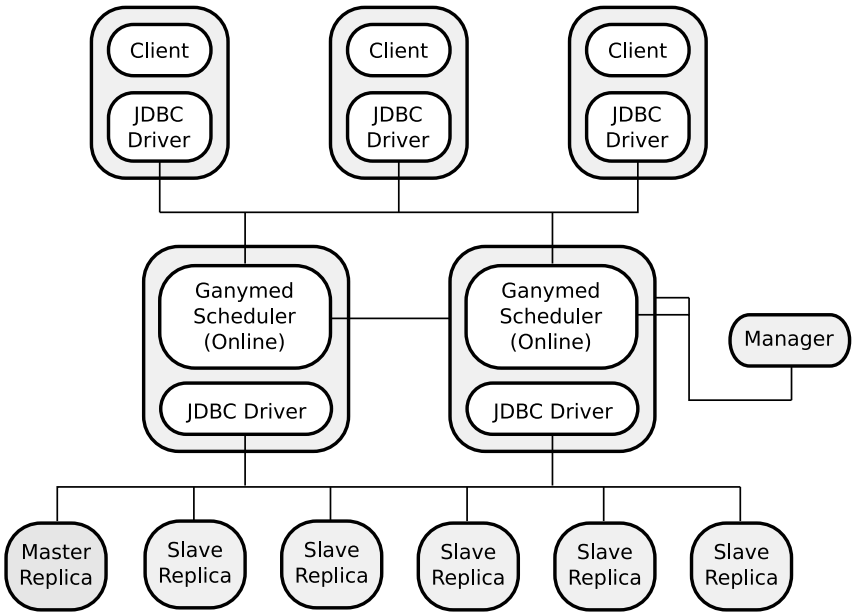


Figure 2.5: Ganymed architecture

The scheduler, called Replicated Snapshot Isolation with Primary Copy (RSI-PC), is the main component of Ganymed. It can be used to schedule transactions over a set of Snapshot Isolation (SI) based replicas. The RSI-PC works by separating the read-only and update transactions and sending them to different replicas. This distinction between transactions has to be marked by the client application in advance.

Update transactions are directly forwarded to the master replica. After a successful commit of an update transaction, the scheduler sends the corresponding write set to all slaves and makes sure every slave applies the write set in the same order as the master replica. Read-only transactions can be processed by any replica. The scheduler freely decides on which replica to execute such transaction.

The Ganymed architecture and the way RSI-PC differentiates and dispatches transactions, makes Ganymed highly scalable, and consequently perform better, in systems with a large amount of read-only transactions and a small number of read-write ones, since any slave replica can process a read-only transaction.

The system in [8] is based in Ganymed but it has a simpler architecture and adds speculative execution. Figure 2.6 shows the architecture of this system. The architecture is only composed of clients, a primary replica and a set of secondary replicas (there is no scheduler or manager). Unlike Ganymed, transactions do not need to be distinctly declared as update or read-only. The system provides user-level speculative execution on the client side.

The primary does the Ganymed's master and scheduler job. The first time a client starts a transaction, the primary randomly chooses a secondary replica to execute it and informs to

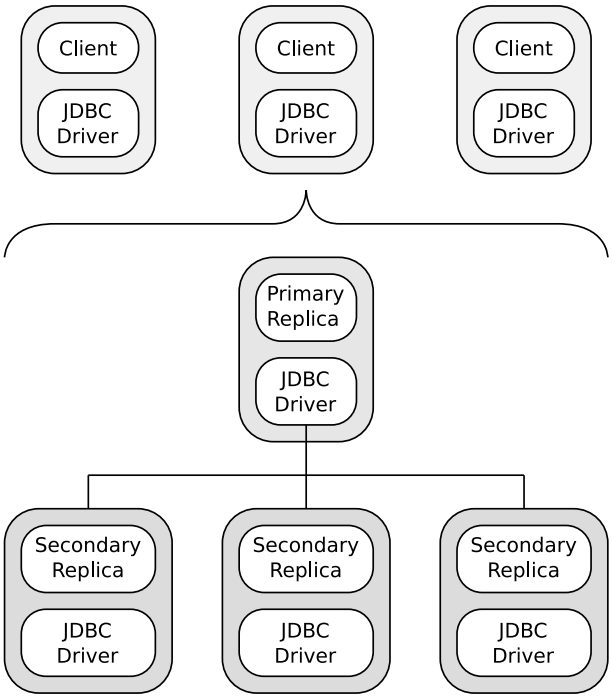


Figure 2.6: [8] system architecture

the client which one it chose. During the transaction execution, if a write operation is found, the transaction continues at the primary instead. After completion of the first transaction, later transactions always begin at the chosen replica. The primary does an equivalent job to Ganymed's scheduler by ensuring that all replicas evolve to the same state and updates are not lost in case of a master replica failure.

The client JDBC driver may use speculative execution when justified. When the client driver issues operations to the server, instead of blocking waiting for each reply, the client can speculate the most probable result allowing the execution to continue. If later the reply is different from the speculation, the transaction needs to abort. When the client issues the next operation, the information about the abort is reported to the application. Only operations on which it is possible to guess the result can be speculated, e.g., creation of prepare statement, connection closure, execution of update statements and execution of prepare statement.

2.5.3 BFT Database Speculative Execution

Zyzzyva [10] is a protocol that uses speculation to reduce the cost and simplify the design of Byzantine fault tolerant (BFT) state machine replication. By using speculative execution, Zyzzyva succeeded in combining in a single protocol, minimal replication, throughput and latency costs.

Zyzzyva is composed of a limited number of clients and $3f + 1$ replicas. Liveness and safety are guaranteed if up to f replicas experience byzantine fault behavior. Zyzzyva is based on three protocols: agreement, view change and checkpoint. The agreement protocol orders requests for execution by the replicas. The view change protocol coordinates the election of a new

primary when the current primary is faulty or the system is running slowly. The checkpoint protocol limits the state that must be stored by replicas and reduces the cost of performing view changes.

Execution is organized into a sequence of views and within a view a designated primary replica is responsible for leading the agreement protocol. The agreement protocol is the one which makes Zyzzyva perform better than previous protocols (e.g., [1,4,6]) through the use of speculation. The agreement protocol works as follows. A client sends a request to the primary. The primary forwards the request to the other replicas proposing an order. The replicas speculatively execute the request protocol assuming the order is correct and send the reply to the client. Replicas replies carry sufficient history information so that the client can determine if replicas are evolving consistently. If the client determines the replicas are consistent, the results return to the application, as shown in Figure 2.7. Otherwise, a recovery process is executed. Unlike previous proposals, replicas do not run an expensive three phase commit protocol to reach agreement order. The advantages of this approach are that replicas execute quicker, less messages are needed to establish request ordering and work is reduced because the agreement protocol stops working once the client knows the request order.

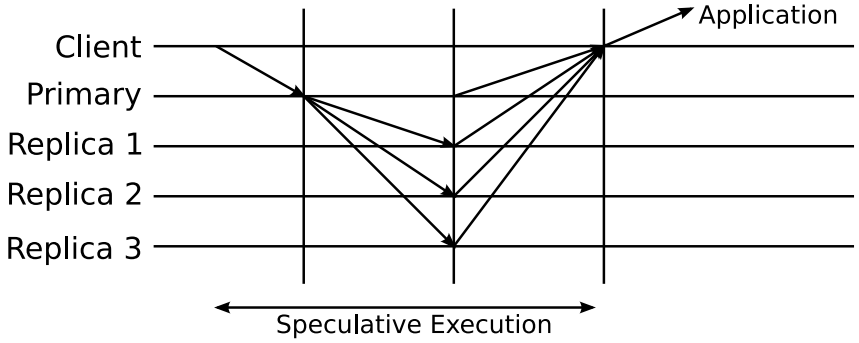


Figure 2.7: Zyzzyva gracious execution

2.5.4 Architecture Speculative Execution

In [15] a different approach towards speculative execution is proposed. Speculative execution is provided through an architecture that supports Thread Level Speculation (TLS). In TLS, a sequential program is divided into threads, which may be dependent upon one another. The hardware executes the threads simultaneously, and monitors the read and write operations of each thread to detect any data dependences that are violated due to parallel execution. If a hazard is detected, the hardware aborts the thread that is supposed to execute later in the original sequential execution (and subsequent threads), eliminating all side effects. By using the TLS machine, [15] proposes a programming paradigm for monitoring program execution and recovery via fine-grain transactions, to enhance the reliability of software.

Programs usually need monitoring functions, for example, to check for unexpected inputs or conditions, and then may need to recover without terminating the program. If the program is correct the monitoring functions represent pure overhead.

To enhance the performance of these checks, the software should convey to the TLS architecture whose functions are monitoring functions. The programmer can write code following the semantics that the monitoring functions and normal computation are executing sequentially. On each call of a monitoring function, the TLS machine may split the sequential code into two threads: the original thread executes the monitoring function and the new thread executes the code after the call speculatively in parallel. The TLS machine ensures that the sequential semantics are preserved.

Typically a monitoring function is independent from the main computation and, since the program is expected to be correct, its result is highly predictable. Consequently, the speculative thread is usually correct and should not need to roll back. When the program is correct the monitoring functions run faster because they are overlapped with the main computation.

The proposed transactional programming model is focused on error recovery. Figure 2.8 shows the syntax of the transactions. Although the `try ... catch` syntax resembles the traditional exception handling constructs, the semantics is different. In conventional exception handling, when an exception is signaled, the side effects made to the data structures are not removed. In the new semantics, the hardware buffers all memory writes and discards them when an `abort` is issued, restoring memory to the state before the `try` statement.

```
1 try {
2   ... original code ...
3   if (error-detected)
4     abort;
5 } catch {
6   return error code;
7 }
8 return ok;
```

Figure 2.8: Example of a transactional programming construct

To support this operation, the original TLS machine was altered to support three new instructions: `TRY`, `ABORT` and `COMMIT`. The `TRY` statement indicates start of a transaction. On the TLS terminology, the thread becomes speculative and all side effects are buffered. The `ABORT` statement indicates that the transaction is to be aborted. The hardware discards the speculative state, restoring memory state before the `TRY` statement. Finally, the `COMMIT` statement indicates that the transaction is to be committed. The machine commits speculative state, and the execution continues as usual.

Although having an architecture to support speculative execution may improve performance of the system, it is even less portable than kernel level speculation, since there is the need of an architecture that supports speculations efficiently. Note that usual multi-core processors are not sufficient to run the described programming paradigms efficiently, since unlike a TLS machine, usual multi-core processors do not have automatic recovery mechanisms in case of a speculation failure.

2.6 Byzantium

This dissertation is part of the Byzantium project [17]. In the long run, the goal of this dissertation is to provide speculative execution by using transactional memory to the Byzantium system. A brief description of the latest version of the Byzantium system follows.

2.6.1 Motivation

As databases systems are the main component behind many of today's applications, it is important that they provide correct and continuous service despite unpredictable circumstances. Many systems attempt to provide continuous service through replication. Most of database replication proposals assume a crash failure model [3]. Although assuming replicas only fail by crashing simplifies replication algorithms, it does not allow the replicated system to tolerate other faults caused by software bugs or malicious attacks.

A service that uses Byzantine-fault-tolerant (BFT) replication can tolerate arbitrary failures from a subset of replicas. Unlike systems that assume crash failure model, BFT not only can resist to crash failures, but also to malicious nodes, hardware errors and software bugs. Previous proposals for BFT database replication only provide strong semantics, limiting performance, or have centralized components, which is problematic when replication is being used to tolerate attacks.

Byzantium [17] is a Byzantine-fault tolerant database replication system that provides snapshot isolation (SI) semantics and does not depend on any centralized components. Providing a weaker semantics than previous proposals allows increased concurrency and, consequently, better performance.

2.6.2 Overview

Byzantium uses the PBFT byzantine fault tolerant state machine replication algorithm [4] as one of its components. System model and assumptions are inherited from this system:

- Assumes a Byzantine failure model where faulty clients or servers may behave arbitrarily;
- The adversary can coordinate faulty nodes but cannot break cryptographic techniques used;
- At most f nodes are faulty out of $n = 3f + 1$ replicas.

Safety is guaranteed in any asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, corrupt them, delay them arbitrarily, or deliver them out of order. Liveness is guaranteed during periods where the delay to deliver a message does not grow indefinitely.

The system architecture is composed by a finite number of clients and $n = 3f + 1$ servers. Figure 2.9 illustrates the Byzantium architecture. Each server is composed by a Byzantium Replica Proxy (BRP) linked to a database system, plus the PBFT replica library. Each replica

contains a full copy of the database and their database systems may differ to ensure a lower degree of fault correlation specially if faults are caused by software bugs. The BRP controls the execution of operations in the database system and receives messages from the PBFT replication library and directly from Byzantium clients.

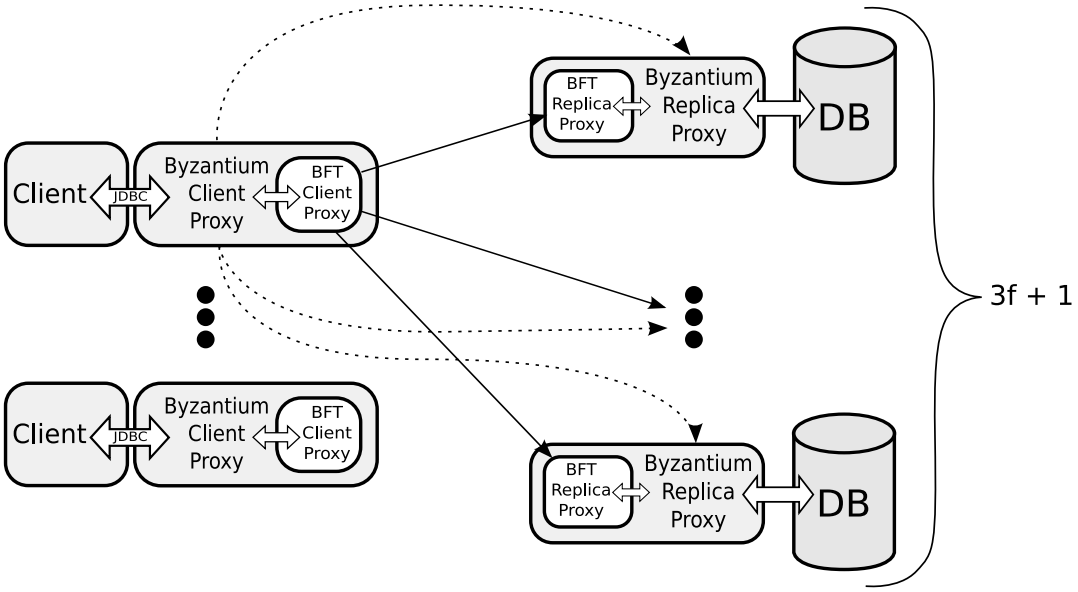


Figure 2.9: Byzantium architecture

Users applications access the system using a custom JDBC interface. This way, applications already using JDBC do not need to be modified to use Byzantium. The JDBC driver implements the Byzantium client proxy (BCP). The client proxy is linked to the PBFT protocol because some parts of the client side need to invoke operations from the PBFT replication protocol.

To improve performance, besides using snapshot isolation, Byzantium minimizes the number of operations that need to execute the three-phase agreement protocol that BFT replication uses to totally order requests, and allows concurrent transactions to execute speculatively in different replicas. The normal transaction execution proceeds as follows.

The client starts by selecting a replica responsible for speculatively executing the transaction: the coordinator replica. At the begin operation, the client issues the corresponding BFT operation from the PBFT library, so that each replica starts a transaction in the same database state. After that, the client executes a sequence of read and write operations. Each of these operations executes only in the coordinator of the transaction. While receiving responses from the coordinator the client proxy returns the results and stores them in a list of operations and corresponding results. On a commit operation, the client issues a BFT operation that includes the list of operations and results of the transaction, so that each replica can verify if the transaction execution is valid before committing it.

The validation proceeds in all replicas. Each replica, besides the coordinator, validates the transaction by executing all the transaction operations and verifying that the results match the results previously obtained by the coordinator. If the coordinator was faulty, the results will not match. In this case, the correct replicas will abort the transaction and the client is informed.

Also, all replicas verify the SI properties.

As explained before, some operations run an expensive BFT protocol, namely the begin and commit operations. The goal of this work is to provide a mechanism that allows the client to speculate on the result of these operations (and others). We expect this to improve the performance of the overall Byzantium system.

3

Abstract Speculative Execution

3.1 Introduction

Section 2.5 provided an overview of a variety of systems using speculative execution. Each of these systems used speculative execution at specific operations to serve a particular purpose. Some of the systems used a mechanism capable of adding speculative execution to a set of operations. One of the main objectives of this dissertation is to create similar mechanisms to enable the user to add speculative execution to the source code of a program.

This chapter starts by describing the program model we use along this dissertation. Afterwards, we present the design of a process [1] that adds speculative execution to a program by transforming its source code. We call this process the *transformation process*. The single objective of the *transformation process* is to improve the performance of a program. Consequently, the *transformation process* must preserve the semantics of the original program. Next, we study the correctness of the *transformation process*. Finally, we present a theoretical analysis of the performance improvement a program can obtain, after being subjected to the *transformation process*.

3.2 Program Model

In this section we describe the program model used along this dissertation. We model a program as a set of computations. In our model we consider a computation as a sequence of instructions that can be given to a computer to perform a task (e.g., statement, method, thread). For practical reasons, we assume computations can only return a single result. However, note that a result may be a set of multiple values.

[1] In this chapter we refer to a process as a series of actions to achieve a result. This does not imply the use of computer programs.

To better understand a program we have also created a visual model compliant with our program model. Figure 3.1 illustrates our visual representation of a computation. The time of execution is included in our visual model. The time line axis is vertical and increases from top to bottom. If the time line is not required, it may be omitted. The beginning of a computation is represented with a squared box. The box label summarizes the objective of the computation. The remaining time of execution of the computation is represented with a vertical line that ends with a dot. The dot represents the end of execution of the computation.

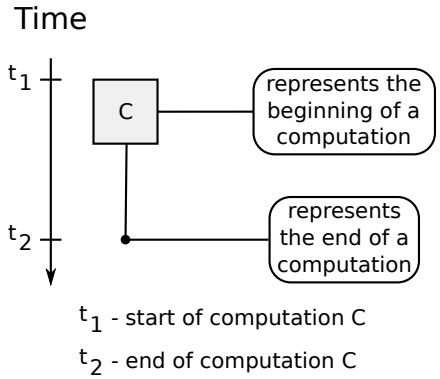


Figure 3.1: Representation of a computation in the program model of this dissertation

In our program model, computations may execute concurrently with each other. The beginning and end of the concurrent execution of a sequence of computations, is visually represented with a horizontal line that ends with an empty diamond. Figure 3.2 illustrates an example of two computations concurrently executing with each other. The first computation, labeled C1, is the main computation. The second computation, labeled C2, results from the execution of a statement in computation C1. The execution of computation C2 ends before computation C1 finishes executing.

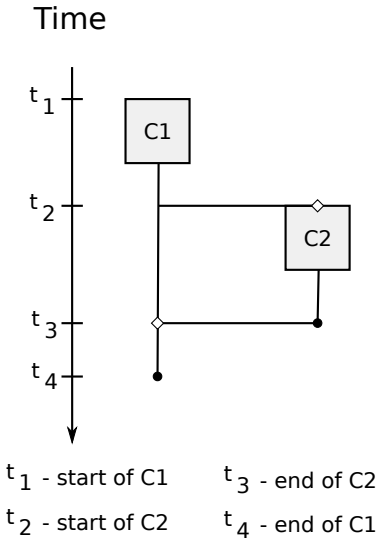


Figure 3.2: Representation of the concurrent execution of two computations in the program model of this dissertation

In our visual model, two computations that execute sequentially are linked by the earlier computation dot and later computation squared box. Figure 3.3 shows an example of a program with sequential computations, compliant with our visual model. The main sequence, labeled S1, is composed of two computations, labeled C1 and C4. From computation C1 results the concurrent execution of another sequence of computations, labeled S2. The sequence S2 is composed of two computations, labeled C2 and C3. The sequence S2 ends before the computation C4 finishes executing.

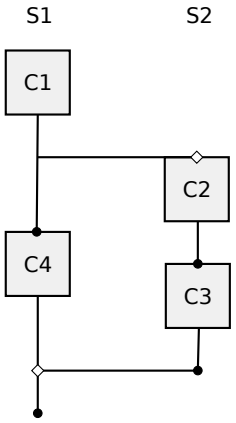


Figure 3.3: Example of a program in the program model of this dissertation

Our program model and respective visual model will be used in the remainder of this dissertation for the following tasks:

- To describe the required transformations to add speculative execution to the source code of a program.
- To describe the possible events that can occur during the execution of a program.

3.3 Design

In this section we present the *transformation process*. The *transformation process* requires in first place the following data:

- The source code of the program where speculative execution should be added. We call this program the target program.
- A set of computations in the target program whose results should be speculated. We call these computations the target computations.
- The most probable result for each of the target computations. We call the most probable result of a target computation, the speculative result. We call the result obtained from executing a target computation, the final result.

The *transformation process* performs the following transformations to add speculative execution to the target program. First, each target computation is transformed to execute concurrently with its following computations. This transformation allows the target computations to execute simultaneously with their subsequent computations, if there are enough free available processors.

The target program may have computations that require the result from a target computation. However, since the target computations were transformed to execute concurrently, the result from a target computation may be required before the target computation is terminated. The result from a target computation is only known, when it finishes executing. For this reason, if a computation requires a result from a target computation, it is transformed to behave in the following way. If the result from the target computation is already known, the computation uses its final result and the program continues executing as it normally would. Otherwise, if the result from the target computation is not yet known, the computation uses its speculative result instead and the computation proceeds as a speculation.

If a speculative result is used, an execution based in a speculation begins. Later, the speculation must be confirmed as correct. A speculation is correct if the speculative result is equal to the correspondent final result. If the speculation was correct, the program may continue as usual. Otherwise, if the speculation was incorrect, the speculative execution must abort. In other words, the execution must return to the state where the speculative result was assumed, leaving no side effects. Afterwards, the aborted computations must be re-executed, now using the final result instead.

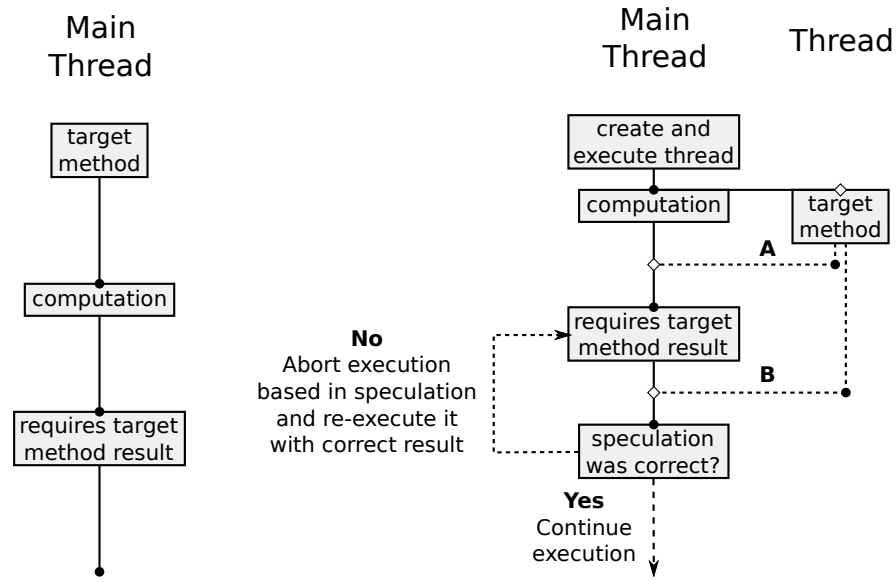
In the next section we present an example of a concrete program that is subject to the *transformation process*.

3.3.1 Transformation Process Example

Figure 3.4 illustrates an example of a program that is subject to the *transformation process*. Figure 3.4a illustrates the program before being subject to the *transformation process*. The program starts with the execution of a method. The method itself is a target computation. If a target computation is a method we call it a target method. Next, the program executes some computation and finishes with the execution of another computation that requires the target method result.

Figure 3.4b illustrates the program after being subject to the *transformation process*. In the original program the target method must complete its execution before proceeding with the next computation. After the *transformation process*, the execution of the target method is replaced by the creation and execution of a thread. The thread executes the target method. This allows the concurrent execution of the target method and its subsequent computation.

When the program executes the computation that requires the target method result, two situations may occur. In the first situation, labeled with the letter A, the target method has already finished executing. In this case the final result of the target method is used and the program proceeds as it normally would. In the second situation, labeled with the letter B,



(a) Program before being subject to the *transformation process*

(b) Program after being subject to the *transformation process*

Figure 3.4: Example of a program subject to the *transformation process*

the target method did not yet finish executing. In this situation the final result from the target method is not yet known. Instead of waiting for the final result to be known, the *transformation process* makes the computation use the speculative result.

If the speculative result is used, an execution based in a speculation begins. Later, the speculation must be checked if it was correct. The speculation is correct if the speculative result is equal to its correspondent final result. If the speculation is correct, the program may continue its execution. Otherwise, if the speculation is incorrect, the computation based in a speculation must abort. The aborted computation is then re-executed using the final result instead.

3.4 Correctness

In this section we discuss the correctness of the *transformation process*. The process we have designed is correct if when applied to a program the resulting transformations preserve the original program semantics.

The correctness of the *transformation process* will be analyzed in three steps. First, we discuss the correctness of transforming a target computation to execute concurrently with its subsequent computations. Second, we discuss the correctness of returning the speculative result instead of the final result. Finally, we consider the correctness of the *transformation process* when multiple target computations exist.

3.4.1 Target Computation Concurrent Execution

A transformation to speculate the result of a target computation is only valid if the concurrent execution of the target computation and its subsequent computations do not break the semantics of the original program. To this end, the subsequent computations must not depend on the results or side-effects of the target computation, including the direct access to modified memory or any dependent I/O mechanism.

For example, consider that a target computation required to modify the value of a global variable. Usually, the *transformation process* would transform the target computation to execute concurrently with its subsequent computations. If so, a subsequent computation can use the global value, before being changed by the target computation. In such case, the semantics of the original program would be broken.

3.4.2 Target Computation Result Visibility

A computation can only make the speculative result from a target computation visible to an external source (e.g., output the result to screen), if it is reversible (can be undone).

For example, consider that a computation would request the result from a target computation, but the result was still unknown. Usually the *transformation process* would make the computation use the speculative result instead. Assume that the speculative result was incorrect. If the computation required to output the result to screen, it would display an incorrect result. Later the effect could not be undone.

3.4.3 Multiple Target Computations

The final aspect to be discussed is if the program has multiple target computations. In such case, the *transformation process* transforms each of the target computations to execute concurrently with their subsequent computations. This transformation may cause the target computations to execute in the same scope (e.g., execute the target computations in the same thread) or in separate scopes (e.g., execute each target computations in its own thread).

The target computations can only be transformed to execute concurrently in separate scopes, if they do not depend on each other. If multiple target computations execute in separate scopes, it is possible for their executions to overlap. However, the program may require the target computations to execute sequentially for its semantics to be preserved.

For example, consider a program where the target computations are database operations. Assume the *transformation process* would then transform each target computation to execute concurrently in its own thread. This transformation opens the possibility for the execution of the target computations to overlap. However, to preserve the semantics of the program, the target computations must execute in the same order as the original program. This is because database operations must execute in an order defined by their respective transactions. If the execution of the target computations can overlap, their original order of execution cannot be guaranteed.

3.5 Performance Analysis

In this section we present a theoretical analysis of the performance improvement a program can obtain after being subject to the *transformation process*. In this analysis we assume concurrent computations always execute simultaneously with each other. This is possible if the number of computations concurrently executing is always less or equal to the number of free available processors.

The performance analysis is organized in three sections. We start by analyzing the global performance improvement a program can have when subject to the *transformation process*. Next, we analyze the best time to validate a speculation. Finally, we discuss the ideal properties of a target computation to improve the performance of a program.

3.5.1 Performance Efficiency

In this section we analyze the performance efficiency of a program after being subject to the *transformation process*. In the analysis we use the program of section 3.3.1. The simplicity of the program keeps the analysis simple. Despite its simplicity, our analysis should be sufficient to deduce the performance improvement of any other program, even if the program has multiple target computations.

Figure 3.5 illustrates the program represented in Figure 3.4a after being subject to the *transformation process*. First, we analyze the situation labeled with the letter A. The target method finishes executing in $t_A \in [t_2, t_3]$. Consequently, the speculative result is never used. Since the target method executes simultaneously with its subsequent computation, it overlaps completely the subsequent computation and, thus, corresponds to an absolute performance gain of its execution time.

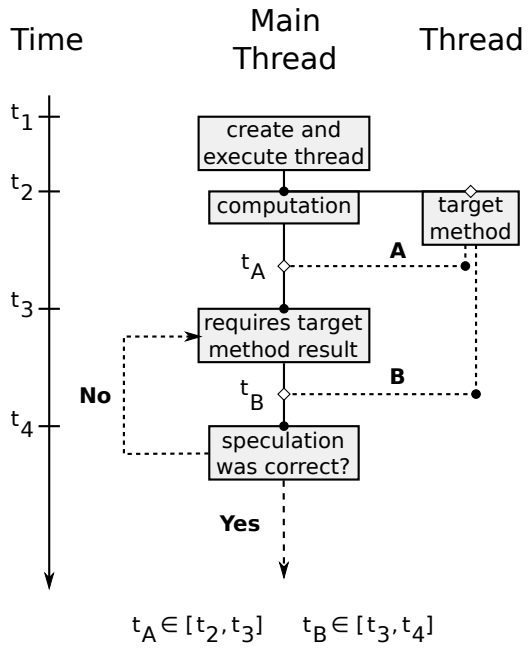


Figure 3.5: Performance gain when using speculative execution

Next, we analyze the situation labeled with the letter B. The target method finishes executing in $t_B \in [t_3, t_4]$. The speculative result is used and an execution based in a speculation begins. Later, the speculation must be checked if it was correct. Like in situation A, if the speculation is correct, the whole time of execution of the target method is gained. However, since the time of execution of the target method is bigger, so is the performance gain. If the speculation is incorrect, the time required for aborting the speculative execution is lost. Moreover, the time executing based in an incorrect speculation (time $[t_B, t_4]$), after the target method finishes executing, is also lost. However, note that while the target method is executing simultaneously with its subsequent computation (time $[t_2, t_B]$), no time is lost.

Based in this analysis, we present the best time to validate a speculation and the ideal properties of a target computation.

3.5.2 Speculation Validation

Consider a program with a single target computation. The maximum time that can be gained is the time of execution of the target computation. When the target computation finishes executing, no more time is gained. Moreover, after the target computation finishes executing, time is lost if the program continues executing based in an incorrect speculation. Therefore, the time to validate a speculation should not surpass the time when the respective target computation finishes computing.

While a target computation is executing simultaneously with another computation, time is being gained. To increase the performance of a program to the maximum, the target computations should execute the maximum possible time simultaneously with their subsequent computations. Moreover, the speculation validation cannot complete until the respective target computation finishes executing. This is because the final result is necessary but it is only known when the target computation finishes executing. Consequently, to achieve the maximum possible improvement with the *transformation process*, the program should validate a speculation when the respective target computation finishes executing.

In conclusion, the best time for the program to validate a speculation, is at the exact moment the respective target computation finishes executing.

3.5.3 Target Computation Ideal Properties

The target computation should have a highly predictable result. If the result from the target computation is highly predictable, the respective speculative result has a high probability of being correct. In theory, if the speculative result is correct, time is gained. On the other hand, if the target computation is not highly predictable, the speculative result has a high probability of being incorrect. If the speculative result is incorrect, time is lost.

The target computation should take a significant amount of time to complete. The maximum amount of time that can be gained with the *transformation process*, is the minimum between the execution time for a target computation and the execution time for all the subsequent computations until the execution of the speculation validation. If the time of execution

of the target computations is not significant, neither will be the performance improvement of the respective program.

4

User-Controlled Speculative Execution

4.1 Introduction

Speculative execution can be implemented in many different ways. In some systems [8, 10], speculative execution is implemented for specific operations to serve a particular purpose. Other systems [12, 15, 21] have a mechanism that enables the user to add speculative execution to some operations of his/her choice. We call such mechanism, a speculative execution mechanism. Clearly the last solution is the best if one wants to add speculative execution to multiple operations. However, the implementation of the speculative execution mechanisms found in the studied systems lacked portability in favor of performance.

The studied speculative execution mechanisms were implemented at two levels: kernel and hardware level. Any implementation at kernel level requires the use of a specific version of the operating system. Any kind of hardware implementation requires the use of the respective specialized hardware. In this dissertation we study the integration of speculative execution to the source code of programs, independently of the operating system and hardware that is used. In this chapter we present our own speculative execution mechanism.

The main objective of our speculative execution mechanism was to enable the programmer to easily add speculative execution to a set of operations in the source code of a program. To accomplish our objective, our design favors easy integration of speculative execution over performance improvement of the program.

4.2 Design

The design of our speculative execution mechanism should enable programmers to apply a variant of the *transformation process* (see Chapter 3), which we call *nospec transformation process*.

The *nospec transformation process* is designed to enable easy integration of speculative execution to the source code of programs. In the next section we describe this process.

4.2.1 NoSpec Transformation Process

In this section we describe the *nospec transformation process*. The *nospec transformation process* requires the following data:

- The source code of the program where speculative execution should be added. We call this program the target program.
- A set of computations from the target program to be transformed. We call these computations the target computations.

Note that unlike the *transformation process*, the speculative result from each target computation is not required. The *nospec transformation process* then performs the following transformations to the target program. First, each target computation is transformed to execute concurrently with its subsequent computations. This transformation makes possible for a computation to request the result from a target computation, that did not yet finish executing. At the time of request, the result from the target computation is still unknown. Consequently, the computations that require a result from a target computation must be transformed.

The computations that require a result from a target computation are transformed to behave in the following way. If the result from the target computation is already known, the computation uses it and the program continues as it normally would. Otherwise, if the result from the target computation is not yet known, the computation waits until the target computation finishes executing. This is the main difference between the *transformation process* and the *nospec transformation process*. With the *transformation process*, if the result from the target computation was not yet known, the speculative result would be used instead.

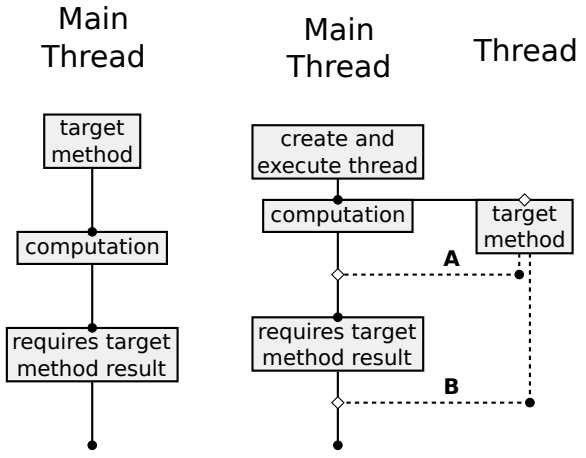
Since speculative results are never used, there is never an execution based in a speculation. Therefore, the speculation validation becomes unnecessary. This makes the implementation of the *nospec transformation process* much easier than the *transformation process*. On the other hand, the *nospec transformation process* may not improve the performance of the target program as much as the *transformation process* would.

With the *transformation process*, computations never wait for the result from a target computation. However, if the program uses an incorrect speculative result, time is lost executing useless computations. The transformed program may even perform worst than the original program, if it uses a sufficient number of incorrect speculative results. With the *nospec transformation process*, if a computation requires the result from a target computation, it must wait for it to be known. Nevertheless, this also guarantees that the transformations do not add the possibility of executing useless computations.

In the next section we show an example of a program that is subject to the *nospec transformation process*.

4.2.2 NoSpec Transformation Process Example

We will now illustrate the impact of the *nospec transformation process* in a program. Figure 4.1a illustrates the program before being subject to the *nospec transformation process*. The program starts with the execution of a target method (a target computation that is a method). Next, the program executes some computation and ends with another computation that requires the target method result.



(a) Program before being subject to the *nospec transformation process*

(b) Program after being subject to the *nospec transformation process*

Figure 4.1: Example of a program subject to the *transformation process*

Figure 4.1b illustrates the program after being subject to the *nospec transformation process*. The execution of the target method is replaced by the construction and execution of a thread which executes the target method. This results in the concurrent execution of the target method and its subsequent computations.

When the program executes the computation that requires the target method result, two situations may occur. In the first situation, labeled with the letter A, the target method already finished computing. In such case, the computation uses its result and the program continues as it normally would. In the second situation, labeled with the letter B, the target method did not yet finish executing. In such case, the computation waits until the target method finishes executing and then proceeds as usual.

Our speculative execution mechanism enables the programmer to apply the *nospec transformation process* to the source code of programs. Our mechanism is based in the use of futures, which are described in the following section.

4.2.3 Futures

A future is an object that acts as a proxy for the result of an associated computation. The result the future holds is initially unknown, usually because its value was not yet computed. Later, when the result is computed, it can be retrieved from the future object itself.

Futures may be implicit (e.g., Alice [1] programming language [19]), where any use of the future automatically obtains its result, or explicit (e.g., Java 5 programming language), where the user must call a method to obtain its result. The computation associated with an implicit future may either start eagerly, when the future is created, or lazily, when its result is first needed. Explicit futures may start its associated computation by calling a specific method. In both types of futures the associated computation may execute as a thread, allowing concurrent execution.

Figure 4.2 shows an example of a possible explicit future implementation, based in the Java language. The class that implements explicit futures is named `FutureTask<V>`. The constructor of the `FutureTask<V>` class requires a single `Callable<V>` object. The `Callable<V>` interface has a single method `call()`, which executes a user defined computation and returns a value of type `V`.

The `FutureTask<V>` class has three methods: `run()`, `get()` and `done()`. The `run()` method executes the `call()` method of the `Callable<V>` object and stores its result. Since the `run()` method is inherited from the `Runnable` interface, it can execute as a thread. The `get()` method waits if necessary for the `call()` method to complete, and then retrieves its result. The `done()` method returns true if the `call()` method already finished computing. Note that the `call()` method represents the associated computation of the future.

4.2.4 Speculative Execution Mechanism

In this section we present the design of our speculative execution mechanism and how it can be used to perform the *nospec transformation process*. The transformations involved in the *nospec transformation process* are the following:

- Transform each target computation to execute concurrently with its subsequent computations.
- Transform each computation that requires the result from a target computation to behave as follows. If the result from the target computation is known, use it and proceed as usual. Otherwise, if the target computation result is unknown, wait until the target computation finishes executing and then proceed as usual.

Our mechanism uses futures for both transformations. Each target computation is replaced by the construction of a future, with the target computation as the associated computation. The futures are then executed concurrently with their subsequent computations. If a computation requires the result from a target computation, it is transformed to obtain the result from the respective future instead. When a computation attempts to get the result from a future, it should wait until the result is known.

Figure 4.3 illustrates an example of a program that is subject to the *nospec transformation process*, by using our speculative execution mechanism. The target program is the one represented in Figure 4.1a. The target method is replaced with the construction and execution of a

[1] `http://www.ps.uni-sb.de/alice/`

```
1   public interface Callable<V> {
2     /* Computes a result */
3     public V call();
4   }
5
6   public class FutureTask<V> implements Runnable {
7     private Callable<V> callable;
8     private V result;
9     private boolean done;
10
11    public FutureTask(Callable<V> callable) {
12      this.callable = callable;
13    }
14
15    /* Waits for the associate computation to complete, and then
16    retrieves its result */
17    public synchronized V get() {
18      while (!done)
19        wait();
20      return result;
21    }
22
23    /* Executes the associated computation */
24    public synchronized void run() {
25      result = callable.call();
26      done = true;
27      notifyAll();
28    }
29
30    /* Returns true if the result of the associated computation has
31     * already finished executing */
32     public synchronized boolean done() {
33       return done;
34     }
35  }
```

Figure 4.2: Example of an explicit future implementation

future. The future is constructed with the target method as its associated computation. The future execution opens a thread that executes the target method.

The computation that requires the target method is transformed to obtain the result from the future. When attempting to obtain the result from a future, the computation must wait until the result is known. In our previous future implementation this behavior corresponds to calling the `get()` method. After the result is obtained, the program proceeds as usual.

In Section 3.4 we have discussed that some programs may require multiple target computations to execute sequentially, to preserve the semantics of the original program. Next, we present the design of a mechanism that solves this problem.

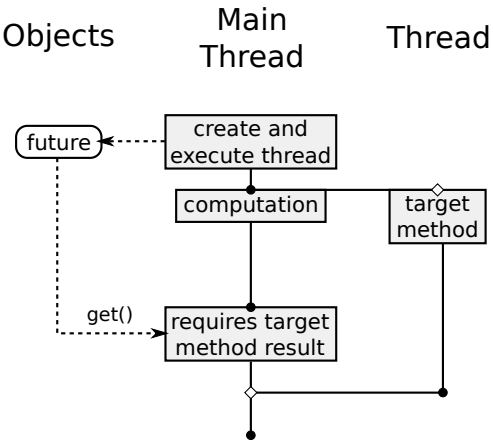


Figure 4.3: Program after being subject to the *transformation process* using our speculative execution mechanism

4.2.5 Multiple Target Computation Coordinator

In this section we present a solution for programs that require multiple target computations to execute sequentially. In our solution, instead of executing the target computations in separate scopes (e.g., execute each target computation in its own thread), we give them to a coordinator. The coordinator decides when and in which scope should the target computations execute.

The objective of our coordinator is to make all accepted target computations to execute sequentially in the same scope. We call our coordinator, the *multiple target computation coordinator*. For the coordinator to execute the target computations sequentially, we use a blocking queue. A blocking queue is a queue with a particular characteristic: the dequeue operation waits until an element is available. When the coordinator accepts a target computation, it inserts it into the blocking queue. The order of insertion in the queue is the order in which the target computations will execute.

When the coordinator starts, a new scope is created. The scope is continuously trying to dequeue and execute a target computation from the blocking queue. If there are no target computations available, the scope waits until there is one. With this, the coordinator can execute all target computations sequentially in the same scope.

Figure 4.4 shows an example of a Java implementation of the *multiple target computation coordinator*. The class that implements the *multiple target computation coordinator* is called `SerialQueueExecutor`. The constructor of the `SerialQueueExecutor` creates a blocking queue and executes itself as a thread. The blocking queue is composed of `Runnable` objects. Consequently, the target computations must implement the `Runnable` interface. The thread is set to daemon, so that when the program ends, it does not wait for the `SerialQueueExecutor` to finish its execution.

The `SerialQueueExecutor` as two methods: `execute(Runnable task)` and `run()`. The `execute(Runnable task)` method, inherited from the `Executor` interface, adds the `task` argument to the blocking queue. The `run` method, inherited from the `Runnable` interface, is continuously dequeuing and executing a `Runnable` object from the blocking queue.

```
1   public class SerialQueueExecutor implements Executor, Runnable {
2     private BlockingQueue<Runnable> queue;
3
4     public SerialQueueExecutor() {
5       queue = new LinkedBlockingQueue<Runnable>();
6       Thread thread = new Thread(this);
7       thread.setDaemon(true);
8       thread.start();
9     }
10
11    /* Adds the given task to the blocking queue */
12    public void execute(Runnable task) {
13      queue.add(task);
14    }
15
16    /* Continuously tries to take and execute a task
17     * from the blocking queue */
18    public void run() {
19      while (true)
20        queue.take().run();
21    }
22  }
```

Figure 4.4: Example of an implementation of the *multiple target computation coordinator*

Next, we show our implementation of the speculative execution mechanism and the *multiple target computation coordinator*.

4.3 Implementation

In this section we present our implementation of the speculative execution mechanism and the *multiple target computation coordinator*. Both implementations are written in Java programming language and used the Sun JDK 6 library.

4.3.1 Speculative Execution Mechanism

The design of speculative execution mechanism only requires futures. Consequently, our implementation of the speculative execution mechanism simply uses the Java explicit futures implementation.

All the classes we used are located at the `java.util.concurrent` package. The future implementation we used was the `FutureTask<V>` class. Although the implementation has two constructors, we only used one which requires a single `Callable<V>` object. The `Callable<V>` interface has a single method `call()` that returns a value of type `V`. The the `call()` method represents the future associated computation.

From the many methods the `FutureTask<V>` class implements, the most interesting are the `run()` and the `get()` methods. The `run()` method executes the `call()` method from the associated `Callable<V>` object. Additionally, the `FutureTask<V>` object implements

the `Runnable` interface. This enables the `run()` method to be executed as a thread. If necessary, the `get()` methods waits for the `call()` method to finish executing, and then retrieves its result.

4.3.2 Multiple Target Computation Coordinator

In this section we present our implementation of the *multiple target computation coordinator*. Our implementation is equivalent to the one shown in Figure 4.4. All Java classes we used are located at the `java.util.concurrent` package. Our *multiple target computation coordinator* is a Java `Executor` implementation. The `Executor` interface has a single method `execute(Runnable task)`. In our implementation we also use the Java blocking queue implementation. The Java blocking queue implementation we used was the `LinkedBlockingQueue<E>`.

Our implementation of the *multiple target computation coordinator* is a straight forward implementation of the design shown in Section 4.2.5. Our implementation of the `Executor` interface is called `SerialQueueExecutor`. The `SerialQueueExecutor` constructor creates an empty blocking queue of `Runnable` objects and a thread. The thread is set to daemon and immediately starts executing. The thread is continuously trying to take and execute an element from the blocking queue. The implementation of the `execute(Runnable task)` simply adds the `task` argument to the queue.

5

Automatic Speculative Execution

5.1 Introduction

In the previous chapter we have designed a mechanism to enable the programmer to add speculative execution to the source code of a program. Despite our attempt to make the speculative execution mechanism as simple as possible, inserting speculative execution to the source code of a program has two main problems: it consumes time and it is error prone. An alternative solution is to develop a mechanism to automatically add speculative execution to the source code of a program. We call such mechanism, an automatic speculative execution mechanism.

Our automatic speculative execution mechanism should meet the following requirements. The mechanism should take the least possible time and effort from the user to add speculative execution to a program. The mechanism must preserve the semantics of the program after adding speculative execution to it. Both of this requirements cover the inconveniences of the speculative execution mechanism, reported in the previous chapter.

The automatic speculative execution mechanism may have some disadvantages compared to programming speculative execution directly. An automatic speculative execution mechanism is harder to implement, since additionally we have to define a set of rules to add speculative execution. Moreover, the set of rules may not be sufficient to work in all programs or to fully optimize a range of systems. Even so, having a mechanism that can reduce significantly time and effort to add speculative execution to a program, makes it worthy of being developed.

5.2 Design

In this chapter we present the design of our automatic speculative execution mechanism. The automatic speculative execution mechanism is based in the *transformation process* described in

Chapter 3. Before proceeding we are first going to review the referred *transformation process*.

The *transformation process* is an abstract representation of a series of transformations to the source code of a program to add speculative execution. The *transformation process* first requires the following data:

- The source code of the program to add speculative execution. We call this program the target program.
- A set of computations in the target program whose results should be speculated. We call these computations the target computations.
- The most probable result for each of the target computations. We call the most probable result of a target computation, the speculative result. We call the result obtained from executing a target computation, the final result.

The *transformation process* then performs the following transformations to add speculative execution to the target program. First, each target computation is transformed to execute concurrently with its following computations. Because of this transformation, the following situation may occur: a computation requests the result from a target computation, that did not yet finish executing. At that time the result is unknown. Consequently, if a computation requires the result from a target computation, it is transformed to behave in the following way. If the result from the target computation is already known, the computation uses it and continues as usual. Otherwise, if the result is not yet known, the computation uses the speculative result instead.

If a speculative result is used, an execution based in a speculation begins. Later, the speculation must be checked if it was correct. A speculation is correct if the speculative result is equal to the correspondent final result. If the speculation is correct the program may continue as it normally would. Otherwise, if the speculation is incorrect, the execution based in the speculation must abort and re-execute using the final result instead.

The operation of the automatic speculative execution mechanism is straightforward: applying the *transformation process* to the whole source code of a program. To be able to do so, the mechanism uses three components: speculative futures, transactional memory and a source-to-source compiler. Speculative futures are an extension to explicit futures which adds support to speculative execution. Speculative futures can be used to transform a target computation to execute concurrently and to return its final or speculative result when necessary. Transactional memory enables to abort an execution based in a speculation and restart it. The source-to-source compiler allows to transform the source code of a program as required. The following sections describe each of these components and how are they used to accomplish the *transformation process*.

5.2.1 Speculative Futures

Speculative futures are an extension to explicit futures (see Section 4.2.3). As the name suggests, the extension adds support to speculative execution. Our automatic speculative execu-

tion mechanism uses speculative futures to satisfy the following requirements:

- Concurrently execute a computation.
- Return the final result of the computation, waiting for the computation to complete if necessary.
- Return the speculative result of the computation.

Explicit futures already solve the first two requirements. Explicit futures are associated with a user defined computation. The associated computation can be executed concurrently with other computations. Explicit futures can also return the final result of its associated computation, waiting for the computation to complete if necessary. However, explicit futures cannot return the speculative result of its associated computation.

Speculative futures address the last requirement. All features from explicit futures are inherited. Additionally, the following features are also supported:

- Return the speculative result from the associated computation when required.
- Return the final result, if the associated computation finished executing, or the speculative result otherwise.

Next, we show a possible implementation of speculative futures.

Speculative Futures Implementation

Figure 5.1 illustrates a possible implementation of speculative futures. This speculative future implementation is an extension of the explicit futures implementation of Section 4.2.3.

Speculative futures are implemented by the `SpecFuture<V>` class. The constructor of the `SpecFuture<V>` class requires a `Callable<V>` object and a value of type `V`, which represents the speculative result. The `Callable<V>` interface has single method `call()` which executes a user defined computation and returns a value of type `V`. The `call()` method represents the associated computation of the speculative future.

The methods inherited by the `SpecFuture<V>` class are the following: `run()`, `get()` and `done()`. The `run()` method executes the `call()` method of the `Callable<V>` object and stores its result. The `get()` method waits for the execution of the `call()` method to complete, and then retrieves its result. The `done()` method returns true, if the `call()` method has already finished its computation.

The new methods of the `SpecFuture<V>` class are the following: `spec()` and `getSpec()`. The `spec()` method simply returns the speculative result. The `getSpec()` method returns the result from computing the `call()` method, if the result is already known, or the speculative result otherwise. Note that unlike the `get()` method, the `getSpec()` method never waits to return a result.

Next, we show how we use speculative futures in the *transformation process*.

```
1  public class SpecFuture<V> extends FutureTask<V> {
2    private V spec;
3
4    public SpecFuture(Callable<V> callable, V spec) {
5      super(callable);
6      this.spec = spec;
7    }
8
9    /* If done, return the final result. Otherwise, return the
10   speculative result */
11   public V getSpec() {
12     if (done())
13       return get();
14     else
15       return spec;
16   }
17
18   /* Returns the speculative result */
19   public V spec() {
20     return spec;
21   }
22 }
```

Figure 5.1: Example of Speculative Future implementation

Speculative Futures in the Transformation Process

Speculative futures are used in the *transformation process* for the following transformations:

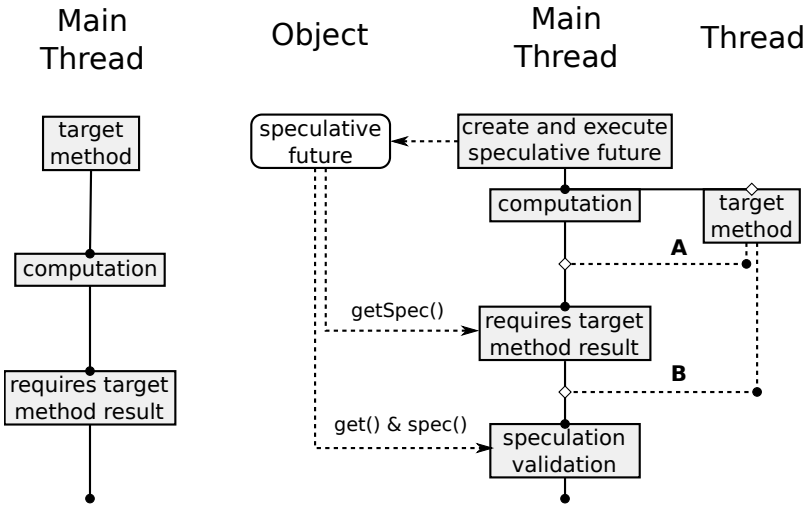
- Transform each target computation to execute concurrently with its following computations.
- Transform each computation that requires the result from a target computation to behave as follows. If the result from the target computation is already known, use it and proceed as usual. Otherwise, if the result from the target computation is not yet known, use the speculative result instead.
- If the speculative result is used, check later if it was correct. The speculative result is correct, if it is equal to the final result.

Each target computation is replaced by the construction of a speculative future. Each speculative future is constructed with the respective target computation as its associated computation and a predefined speculative result. Each speculative future is then executed concurrently with its following computations.

If a computation requires the result from a target computation, it is transformed to obtain the result from the speculative future instead. The result should be returned in the following manner: return the final result of the associated computation, if it is already known, or the speculative result otherwise.

If a speculative result is used, later it must be checked if it was correct. A new computation is added to do so. We call this computation, the speculation validation. The speculation validation gets the final and speculative result from the respective speculative future and compares the value of each other. If the speculative result is equal to the final result, the speculation is correct.

Figure 5.2 shows an example of a program that is subject to the *transformation process*, by using speculative futures. Figure 5.2a shows the program before being subject to the *transformation process*. The program starts with the execution of a target method (a target computation that is a method). Next, the program executes some computation and ends with another computation that requires the target method result.



(a) Program before being subject to the *transformation process*

(b) Program after being subject to the *transformation process* by using speculative futures

Figure 5.2: Example of a program subject to the *transformation process* by using speculative futures

Figure 5.2b shows the program after being subject to the *transformation process*. The target method is replaced by the construction and execution of a speculative future. The speculative future is constructed with the target method as its associated computation and a predefined speculative result. The execution of the speculative future opens a thread that executes the target method.

The computation that requires the result from the target method is transformed to obtain the result from the speculative future. The result should be returned in the following way: return the final result of the target method, if it is already known, or the speculative result otherwise. In our previous speculative future implementation, this behavior corresponds to calling the `getSpec()` method. Consequently, two situations may occur. In the first situation, labeled with the letter A, the target method did not yet finish executing. In such case, the final result is returned. The speculation validation considers no speculation was used and the program proceeds as usual. In the second situation, labeled with the letter B, the target method

did not yet finish executing. In such case, the speculative result is returned, which starts an execution based in a speculation.

If the speculative result is used, later it must be checked if it was correct. A speculation is correct if the speculative result is equal to its correspondent final result. Thus, the speculation validation compares the speculative result with the final result. In our previous speculative future implementation, getting the speculative result and final result corresponds to calling the `spec()` and `get()` methods, respectively.

If a speculation is incorrect, the execution based in the speculation must abort and restart using the final result instead. However, using only speculative futures we cannot easily do so. For that we use transactional memory, as described in the next section.

5.2.2 Transactional Memory

If a speculation is found to be incorrect, the execution based in the speculation must abort. That is, the execution must return to the state where no speculation was ever used and guarantee that there are no side effects. To be able to do that, our design uses transactional memory. With transactional memory, code can be encapsulated into an atomic block (see section 2.4.2). The execution of the atomic block acts as a memory transaction. As a result, the code inside an atomic block can commit or abort and return if necessary.

Next, we show how we use transactional memory in the *transformation process*. First, we show the case where there is a single target computation, and then the case where there are multiple target computations.

Transactional Memory in the Transformation Process

Consider a program with a single target computation, that is subject to the *transformation process*. If a computation requests the result from the target computation, it should behave in the following way. If the result from the target computation is already known, the computation uses it and the program proceeds as it normally would. Otherwise, if the result from the target computation is not yet known, the computation uses the speculative result instead.

If the speculative result is used, an execution based in a speculation begins. The speculation must later be checked if it was correct. The speculation is correct if the speculative result is equal to the correspondent final result. If the speculation is found to be correct, the program may continue as usual. Otherwise, if the speculation is incorrect, the execution based in the speculation must abort.

In our design, every execution based in a speculation should be done inside a memory transaction. Also, the speculation validation should be executed at the end of each memory transaction. If the speculation is found to be correct, the memory transaction commits. Consequently, all modifications made inside the memory transaction become definitive and the program may continue its execution. Otherwise, if the speculation is incorrect, the memory transaction aborts. Thus, all modifications made inside the memory transaction are discarded, while guaranteeing no side-effects. The memory transaction is then re-executed using the final

result instead.

To execute code as a memory transaction, we place it inside an atomic block. In our design, all computations that have the possibility of executing based in a speculation, must be transformed to execute inside an atomic block. Therefore, in a program with a single target computation, at least the first computation that requires the result from the target computation, must be transformed to execute inside an atomic block. This atomic block may also encompass subsequent computations, including other computations that also require the result from the target computation.

At the end of the atomic block the speculation validation is executed. The speculation validation if necessary waits for the target computation to finish executing. Therefore, it is guaranteed that after executing the atomic block, the target computation has already finished executing. Consequently, in a program with a single target computation, it is not possible for further computations to execute based in a speculation. Furthermore, it is also guaranteed that if the atomic block must re-execute, the final result will be used and that the atomic block re-executes at most once.

Our design does not specify when should the atomic block end. Ideally, the atomic block should end at the exact moment the target computation finishes executing (see Section 3.5.2). However, only at run-time would be possible to know when would that happen. Thus, the end of a memory transaction would have to be decided dynamically. Since our design is based in static transformations, we leave this decision to the implementation.

A computation can only make use of a speculative result if it is reversible (can be undone) (see Section 3.4.2). One limitation of our design is that it does not solve this particular problem. For example, consider that a computation inside an atomic block would request the result from a target computation, but the result was still unknown. In this case, the speculative result would be returned. Assume the returned speculative result was incorrect and the computation would attempt to output this result to screen. Depending in the transactional memory implementation, the incorrect speculative result could be immediately output to screen before the commit operation of the atomic block.

Figure 5.3 continues the example shown in Figure 5.2. It shows the program after being fully subject to the *transformation process*, by using speculative futures and transactional memory. The computation that requires result from the target method, is transformed to execute inside an atomic block. The speculation validation is placed at the end of this atomic block.

When the program executes the computation that requires the target method result, two situations may occur. In the first situation, labeled with the letter A, the target computation already finished executing. In this case, the computation uses the final result of the target method. In the second situation, labeled with the letter B, the target computation did not yet finish executing. In this situation, the result from the target computation is not yet known. Thus, the speculative result is used instead.

Next, the speculation validation is executed. If the speculative result was used and it was found to be correct or if it wasn't used at all, the atomic block commits. In such case, all modifications made inside the atomic block are applied to memory and the program may continue

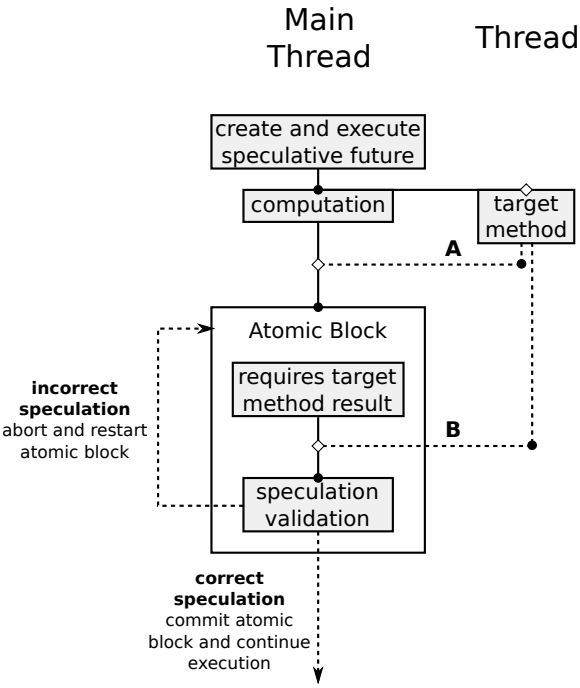


Figure 5.3: Program illustrated in Figure 5.2a after being fully subject to the *transformation process*, by using speculative futures and transactional memory

its execution. Otherwise, if the speculative result was used and it was incorrect, the atomic block aborts. That is, the execution state is reverted to the point where the atomic block began, while guaranteeing no side-effects. The atomic block re-executes afterwards. At the end of the atomic block, it is guaranteed that the target computation has already finished executing. Therefore, in the re-execution of the atomic block, only the final result will be used. In any case, the semantics of the program is preserved.

So far we have only discussed how we use transactional memory in programs with a single target computation. However, if multiple target computations exist we may have to perform additional transformations. In the next section we show how we use transactional memory in the *transformation process* when multiple target computations exist.

Multiple Target Computations

Consider a program with multiple target computations. In our design, if a computation requires the result from one of the target computations, it is placed inside an atomic block. This atomic block may also encompass subsequent computations, including other computations that also require the result from one of the target computations. At the end of the atomic block, all speculative results that have been used, must be checked if they were correct. A speculative result is correct if it is equal to its correspondent final result. If all speculative results are correct, the atomic block commits and the program may continue its execution. Otherwise, if one of the speculative results is incorrect, the atomic block aborts and it is re-executed afterwards.

The computations inside each atomic block may require additional transformations. Assume a target computation is inside an atomic block. If a computation requires the result from

this target computation and it is inside the same atomic block, it is transformed to execute inside a closed nested memory transaction (see Section 2.4.2). Closed nested memory transactions have one important property which allows them to be used in the *transformation process*: if a closed nested memory transaction aborts, only that transaction is terminated and the parent is not. In this way, even if a speculation is incorrect inside a closed nested memory transaction, we can abort and restart it, leaving no side effects. To execute code as a closed nested memory transaction, we place it inside a closed nested atomic block. All transformations that were applied to the atomic blocks are also applied to closed nested atomic blocks.

If this last transformation inside each atomic block was not applied, the execution could enter an infinite loop. For example, consider that a target computation would execute inside an atomic block. Assume that some subsequent computation would use the speculative result of the target computation and that the speculative result was incorrect. Later, the atomic block would have to abort and re-execute. However, there would be no difference in the re-execution. That is, the computation would use again the incorrect speculative result of the target computation and the atomic block would have to abort and re-execute again. This loop would never end. However, with the last modification it is guaranteed that at the end of the closed nested atomic block, the target computation already finished executing. Consequently, at the re-execution of the closed nested atomic block, only the final result would be used.

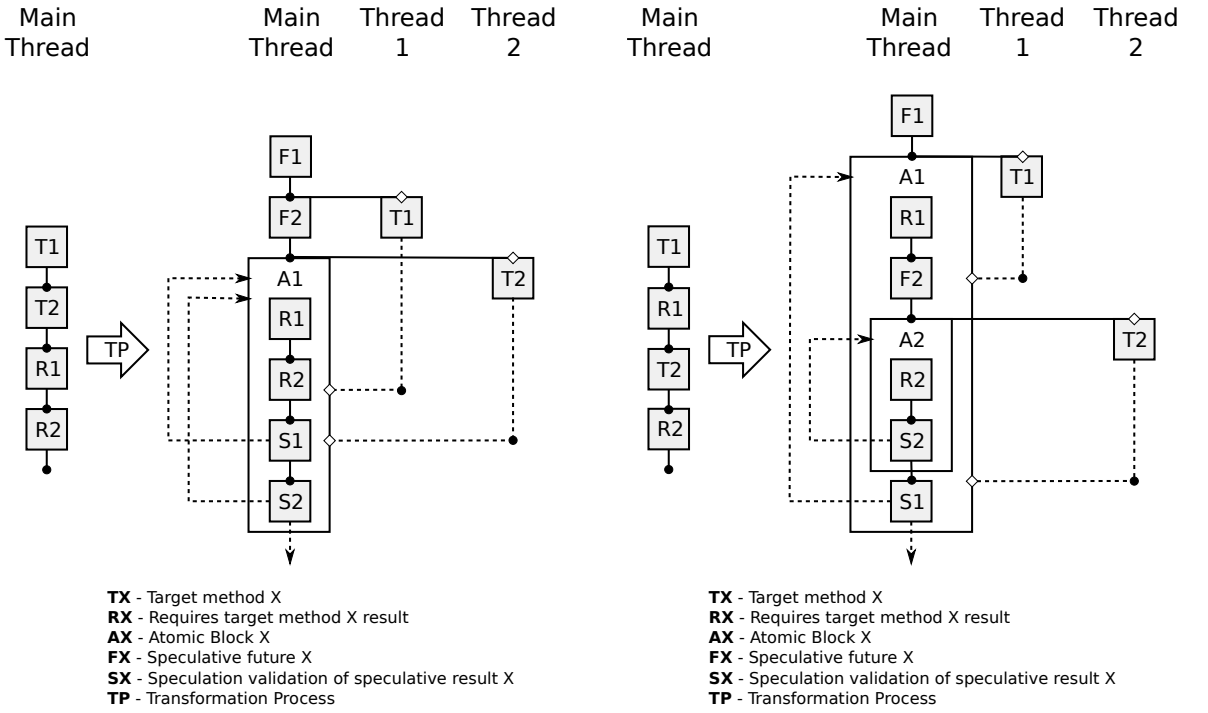
Figure 5.4 illustrates two examples of programs that are subject to the *transformation process*, by using speculative futures and transactional memory. Figure 5.4a shows an example of a program that starts with two target methods, one followed by another, labeled T1 and T2. The program continues with two other computations, labeled R1 and R2. R1 requires the result from T1 and R2 the result from T2. Figure 5.4b shows the same program, but with the order of execution of T2 and R1 switched.

In both examples, each target computation is replaced by the construction and execution of a speculative future. Thus, T1 is replaced with F1 and T2 with F2. Each speculative future is constructed with the respective target computation as the associated computation and a predefined speculative result. The execution of a speculative future opens a thread that executes the target computation.

The computations R1 and R2 must be transformed to execute inside an atomic block. In the example of Figure 5.4a, both R1 and R2 are placed in the same atomic block. This atomic block is labeled A1. At the end of A1, all speculative results that were used must be checked if they were correct. The computations S1 and S2 are used for this task. If both speculative results are correct, A1 commits. Otherwise, if one of the speculative results is incorrect, A1 aborts and it is re-executed afterwards.

In the example of Figure 5.4b, R1 and all its subsequent computations are placed inside the same atomic block. This atomic block is labeled A1. Therefore, F2 and R2 are also encompassed into A1. Since F2 and R2 are in the same atomic block, R2 must be placed inside a closed nested atomic block. The closed nested atomic block is labeled A2.

Inside each atomic block, all speculative results that were used must be checked if they were correct. Thus, A1 must check if the speculative result correspondent to F1 was correct and



(a) Program with in line multiple target methods

(b) Program with scattered multiple target methods

Figure 5.4: Example of programs with multiple threads subject to the *transformation process* using transactional memory

A2 the speculative result correspondent to F2. If all speculative results that were used inside an atomic block are correct, the atomic block commits. Otherwise, if one of the speculative results that was used inside an atomic block is incorrect, the atomic block aborts and re-executes afterwards. Note that if A1 aborts, A2 also aborts. However, if A2 aborts, A1 does not.

The combination of speculative futures and transactional memory allows us to meet all the requirements to perform the *transformation process*. However, even having met all requirements for the *transformation process*, the transformations must still be automatically applied to the source code of a program. To do this we use a source-to-source compiler.

5.2.3 Source-to-source Compiler

A source-to-source compiler is a type of compiler that takes the source code of a program specified in a high level programming language and generates another program in another high level language. Note that the input and output high level programming languages can be the same.

Our automatic speculative execution mechanism uses a source-to-source compiler to apply the *transformation process* to the source code of a program. The source-to-source compiler does the necessary transformations to add speculative execution to a given program. To apply the transformations, the source-to-source compiler resorts to speculative futures and transactional memory.

5.3 Implementation

In this section we present our implementation of speculative futures and the source-to-source compiler. All the components were implemented in Java programming language and used the Sun Java 6 library.

5.3.1 Speculative Futures

In this section we describe our implementation of speculative futures. Our speculative future implementation is equivalent to the one shown in Figure 5.1. All classes we have used can belong to the `java.util.concurrent` package. The speculative future implementation is an extension of the Java `FutureTask<V>` class. It is a straightforward implementation of the design shown in Section 5.2.1.

The extension we created is called `SpecFuture<V>`. Our implementation has a single constructor that requires a `Callable<V>` object and a `V` object. The `Callable<V>` object is handled by inheritance through the `super` call. The `Callable<V>` interface has a single `call()` method which represents the associated computation of the speculative future. The `V` object represents the speculative result.

The most interesting methods inherited from the `FutureTask<V>` class, are the following: `run()` and `get()`. The `run()` method executes the `call()` method from the associated `Callable<V>` object. Additionally, the `FutureTask<V>` class implements the `Runnable` interface, allowing to execute the `run()` method as a thread. The `get()` method if necessary waits for the `call()` method to finish computing, and then retrieves its result.

The new methods from the `SpecFuture<V>` class are the following: `spec()` and `getSpec()`. The `spec()` method simply returns the speculative result. The `getSpec()` method returns the result from executing the `call()` method, if the result is already known, or the speculative result otherwise.

5.3.2 Source-to-Source Compiler

In this section we describe our implementation of the source-to-source compiler. Our implementation uses Polyglot5 [1] as a source-to-source compiler. Polyglot5 is an extension of Polyglot [13] that adds support to the Java 5 programming language. From our experience Polyglot5 is easily understandable if one first learns about Polyglot. We present Polyglot in the next section.

Polyglot

Polyglot is an extensible compiler framework for the Java programming language. By default the Polyglot framework is simply a semantic checker for Java 1.4. However, a programmer may create a compiler for a new language by extending the Polyglot framework. The framework

[1] `http://www.cs.ucla.edu/~milanst/projects/polyglot5/`

is implemented using design patterns to promote extensibility. Using Polyglot, compilers for new languages can be implemented without duplicating code from the framework itself.

In Polyglot the default supported language is called the base language. By default the base language is Java 1.4. The new language is called language extension, even if it is not backwards compatible with the base language. A Polyglot extension is a source-to-source compiler that accepts the source code of a program written in a language extension and generates a new program written in Java source code.

The Polyglot compilation process offers several opportunities for the programmer to change its behavior. Figure 5.5 illustrates the steps in the Polyglot compilation process. The name `Ext` stands for language extension. The first step in compilation is parsing the input source code to produce an abstract syntax tree (AST). Polyglot includes an extensible parser generator, that allows the implementer to define the syntax of the language extension as a set of changes to the base grammar. The produced AST may contain new kinds of nodes to represent the syntax added to the base language.

Figure 5.5: Polyglot compilation process.

The next step in the compilation process is a series of compilation passes applied to the AST. Both semantic analysis and translation to Java may comprise several such passes. A pass scheduler selects passes to run over the AST of a single source file, in an order defined by the extension. Each compilation pass, if successful, rewrites the AST producing a new AST that is input to the next pass. After all compilation passes complete, the usual result is a Java AST.

Polyglot5 is an extension of the original Polyglot. By default, the Polyglot5 is a semantic checker for Java 5 programming language. Polyglot extensions are by themselves also extensible. Our implementation is an extension to the Polyglot5 framework. The result from the extension is a source-to-source compiler that adds speculative execution to the Java source code of a target program.

Target and Stop Methods

In our implementation the user must first assign a set of target methods and stop methods. The target methods are the set of methods where their results should be speculated. Sections 3.4 and 3.5 describe which properties are suitable for a method to be a target method.

The stop methods are the set of methods that define the limit that any active target method can execute concurrently due to speculative execution. For example, suppose one of the stop methods is a method that outputs to screen the result of a target method. If the target method was still executing and the next computation in the main thread was the stop method, the speculative result could not be used. If the speculative result was used and it was incorrect, the program would be outputting an incorrect result to screen. Thus, the transparency of our implementation would be broken. Instead, if a stop method must execute, it must first wait for

all active target methods to finish their execution. After replacing a target method with a speculative future, it is possible to wait for the target method to finish executing by getting its final result. In our speculative future implementation getting the final result corresponds to a call of the `get()` method. There is no advantage in including a stop method or its subsequent computations inside an atomic block, since any target methods must finish their execution before the execution the stop method. At that point the final result of any target method is already known. For that reason, a stop method also defines the limit which an atomic block can contain subsequent computations.

Both set of methods are defined in respective files. The target method description includes the method signature, return type and speculative result. The stop method description only includes the method signature. These methods are then used for the passes we have added to Polyglot5. We describe all passes added to Polyglot5 in the next sections.

Call Graph Pass

The first pass we have added to Polyglot5 constructs a graph of all method calls in the program. The produced graph is then used in the next pass. The second pass creates two reduced graphs. To create a reduced graphs we first input a set of methods. The reduced graph contains only the routes that end in one of the methods that was input. The reduced graphs that are created are a subgraph of target methods and of stop methods.

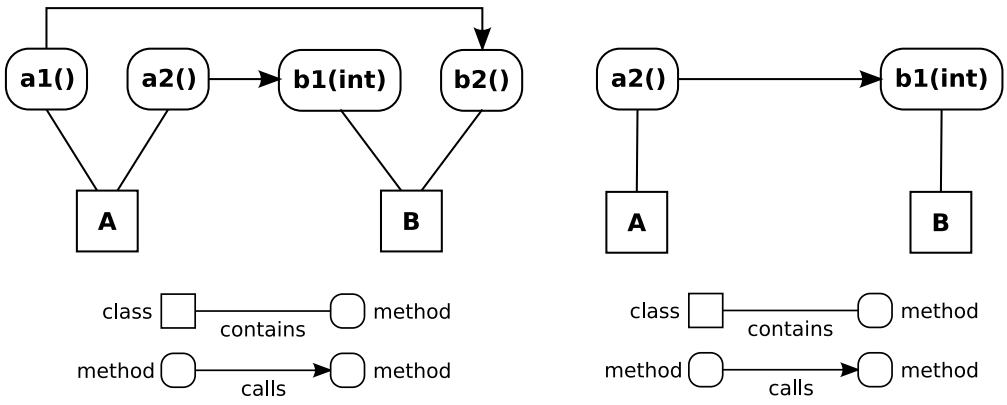
The reduced graph of target methods is used in later steps to know the methods (or the class(es) where these methods are contained) that call at least one of the target methods. Knowing the methods that call a target method makes the implementation of further steps easier. If a stop method cannot execute based in a speculation, neither can a method that calls a stop method. For that reason, any method that is inside the reduced graph of stop methods is also considered a stop method.

Figure 5.6a represents an example of one possible graph of method calls. The program has two classes: `A` and `B`. Class `A` contains the methods `a1()` and `a2()`. Class `B` contains the methods `b1(int)` and `b2()`. The methods are represented by their signatures. The arrows represent the method calls that exist inside the code of a method. The method `a1()` calls the method `b2()` and the method `a2()` calls the method `b1(int)`. One limitation of our implementation is that anonymous classes are not supported. If an anonymous class is encountered it is simply ignored.

Figure 5.6b represents a possible reduced graph of the graph illustrated in Figure 5.6a. The only input method was `b1(int)`. The only method that calls the method `b1(int)` is `a2()`. All other methods do not exist in the reduced graph.

Executor Pass

In the third pass we add an `Executor` to the fields and constructors of the classes where a call of a target method exists. The classes where a call of a target method exists are obtained from the reduced graph of target methods. The default `Executor` is the *multiple target compu-*



(a) Graph of method calls of a program

(b) Reduced graph of method calls

Figure 5.6: Example of a graph of method calls

tation coordinator (see Sections 4.2.5 and 4.3.2). The *multiple target computation coordinator* forces each `Runnable` object to execute sequentially within a thread.

The reason to add an `Executor` is to facilitate the execution of speculative futures. We use speculative futures in the next pass. The reason why the default `Executor` is the *multiple target computation coordinator*, is because there may be programs that require the target methods to execute sequentially (see Section 3.4.3). We may sacrifice performance but a larger number of programs can use our automatic speculative execution mechanism correctly.

Transformation Process Pass

In the fourth pass the *transformation process* is applied to the source code of the program. This pass is based in the design shown in Section 5.2.1 and 5.2.2. Each time a target method is found, it is replaced by the construction and execution of a speculative future. The speculative future is constructed with the target method as its associated computation and a predefined speculative result. If a variable was used to store the result of the target method, it is placed next to the construction of the speculative future with no value. This variable is used later. Afterwards, the speculative future concurrently executes with its subsequent computations, by resorting to the `Executor` that was added in the previous pass.

In the same pass, any computations that requires the result from one of the target methods is searched. If such computation is found it is encompassed into an atomic block. This atomic block encompasses all the subsequent computations until one of the stop methods is found or the block where it is contained ends. If variable that used to store the result from a target method is used by a computation inside the atomic block, a new statement is placed at start of the atomic block, for this variable to store the result from the `getSpec()` method of the respective speculative future. This way, any method that uses such variable will use the final result of the target method, if it is already known, or the speculative result otherwise.

The speculation validation is inserted at the end of the atomic block. The value of each variable that used to store the result from a target method, variable which was used inside the atomic block, is compared with the final result from the respective speculative future. The final

result from a speculative future can be obtained by calling the `get()` method. If the value from one the variables is not equal to its correspondent final result, the atomic block aborts. The next pass makes the atomic block re-execute if it is aborted.

In our implementation, no transformations are done to target methods that are found inside an atomic block or to computations that require the result from one of these target methods. This is because the transactional memory framework used in our current implementation does not support closed nested memory transactions.

Another limitation in our implementation is that there is no verification if a method can be a target method. A method can only be a target method if its subsequent computations do not depend on the results or side-effects from the execution of the target method (see Section 3.4.1).

Atomic Block Passes

In the final step we transform the atomic block to use a transactional memory framework. The framework and this pass were programmed by Ricardo Dias. The pass converts the atomic block to a while loop that only ends when the correspondent memory transaction does not abort, either due to a explicit abort call or due to a conflict with another memory transaction. The memory transaction starts at the beginning of the while loop and commits in the end.

In Section A.1 we present concrete example of a program that is subject by our implementation of the automatic speculative execution mechanism.

6

Speculative Execution in Database Clients

6.1 Introduction

In the two previous chapters we have presented two mechanisms to add speculative execution to the source code of a program. In Chapter 4 we presented a speculative execution mechanism for the programmer to add speculative execution to the source code of a program. In Chapter 5 we presented a mechanism to automatically add speculative execution to the source code of a program. The objective of this chapter is to measure the performance improvement of a database client after integrating speculative execution to its source code.

The methods of the database client we are going to add speculative execution are database operations in a database system. When a database client sends an operation to a database, usually the database client must wait for the database response. The time the database client must wait to receive the response from the database can be quite large. Instead of waiting, the database client can speculate the result from the database operation and continue the execution based in the speculation.

Adding speculative execution to database operations poses two problems. The first problem is that the database operations in a transaction must execute sequentially with each other. Otherwise, the order in which the database operations appear in their respective transactions may be different than in the original database client. The other problem is when the database operations are sent to the database, based in an incorrect speculation. In such case, the database operations must abort.

In this chapter we start by presenting the design of a system that allows us to measure the performance improvement of database clients when using speculative execution. Afterwards,

we present our implementation of this design. Finally, we present an experimental study aimed at evaluating the performance improvement that a database client can have when using speculative execution.

6.2 Design

In this section we present the design of a system that allows to measure the performance improvement of a database client when using speculative execution. The model of the system is client-server. This section describes both the database client and the database server. Additionally, this section describes the requirements for the automatic speculative execution mechanism to be able to add speculative execution to a database client.

6.2.1 Database Client

The database client is the component we are attempting to improve the performance by adding speculative execution. A database client is a computer program that accesses a database system in a client-server model. In our design, it is mandatory for the database client to be able to measure its own performance. With such feature it is possible to measure the performance improvement of the database client when using speculative execution. The performance improvement can be measured by comparing the performance of the database client using speculative execution with its performance without speculative execution.

Our design uses a transaction processing benchmark as a database client. A transaction processing benchmark usually has two components: a database constructor and a client. The database constructor creates a specific database into a target database management system (DBMS) (e.g., PostgreSQL, Oracle). The client component sends transactions to the DBMS, to access the previously created database, and generates performance statistics based in the timings for the responses from the DBMS (e.g., the number of transactions processed by time unit).

In our design the transaction processing benchmark supports an option to activate speculative execution. When the speculative execution option is activated, the transaction processing benchmark uses the speculative execution mechanism we have designed at specific points of its execution. That way, we are able to compare the performance of an instance of the transaction processing benchmark using speculative execution with the performance of another instance of the transaction processing benchmark without speculative execution (e.g., the instance that processes more transactions per time unit performs better).

6.2.2 Database Server

The database server is the component that processes the transactions sent by the database client. A database server is a computer program that provides database services to another computer program in a client-server model. Database management systems usually provide a

database server functionality. In our design the database server can be any one, as long as the database client supports it.

As mentioned previously, the database client will use speculative execution in methods that send database operations to a database. The time such methods take to complete is usually equal to sum of the following times: the time the database operation takes to reach the database, the time the database server takes to process the database operation and the time the response from the database takes to arrive to the database client. The maximum performance that can be gained from using speculative execution is the minimum between the execution time for the methods using speculative execution and its subsequent computations (see Section 3.5.1). Consequently, the choice of the database server and the way it is accessed has an important role in the performance improvement of the database client using speculative execution.

6.2.3 Automatic Speculative Execution Mechanism in Database Clients

Our design of the automatic speculative execution mechanism must meet a set of requirements to be able to automatically add speculative execution to the source code of a database client. The operations we are going to add speculative execution are database operations. However, all database operations must execute sequentially with each other. Otherwise, the order the database operations appear in their respective transactions could be different from the original execution order in the database client. In our design all database operations using speculative execution must be managed by the *multiple target computation coordinator* (see Section 4.2.5). The *multiple target computation coordinator* forces the sequential execution of all speculative operations within a single thread.

In the design of the automatic speculative execution mechanism, all the computations that can possibly execute based in a speculation are encapsulated inside an atomic block. If a computation inside the atomic block executes based in an incorrect speculation, later the atomic block aborts and is re-executed. This assures that the program goes back to the state where the speculation never occurred. However, aborting the atomic block does not guarantee that changes made to external systems during the execution of the atomic block are rolled back. To solve this problem, our design ends the atomic block when the next database operation needs to be executed. In this way, database operations are never sent based in speculated values.

6.3 Implementation

In this section we present the implementation of the system we have designed in the previous section. We also present how we have used our implementation of the automatic speculative execution mechanism to add speculative execution to the database client.

6.3.1 Database client

Our first attempt for a database client was to use an existing transaction processing benchmark and extend it to support speculative execution. We have attempted to add speculative exe-

cution to both: an open-source TPC-W implementation [1] and RUBiS [2]. However, our attempt failed due to problems with the compilation and documentation of both applications.

The open-source TPC-W implementation was developed using Java 1.4 and required a servlet capable server, a database system and MATLAB. Although its authors claim to have successfully run the application in various conditions, the test conditions used old software and some of it was proprietary. We adapted the TPC-W source code to support the newer software and we were able to execute it. Despite the successful execution, the generated performance results were incomprehensible mainly due to the lack of documentation. As for RUBiS, as its authors admit, it proved hard to compile properly, manly due to the lack of proper documentation.

Having failed to use an existing transaction processing benchmark, we have decided to develop our own transaction processing benchmark. The next section describes the implementation of our transaction processing benchmark.

Benchmark Application

Our benchmark application was programmed in Java 6, designed to be simple and to support a wide range of options. JDBC is used to contact a target database system. Most of the SQL code used in the benchmark conforms to the SQL-92 standard (except that `CREATE SEQUENCE` and `FETCH ONLY` are used, conforming respectively to SQL-2003 and SQL-2008), enabling support with most of the DBMS available.

The transaction processing benchmark has two components: the database constructor, which creates a specific database into a target DBMS with JDBC support, and the benchmark executor, which operates over the database and returns performance statistics. Each of the components will be described in the following sections.

Database Constructor

In this section we describe the database constructor of our transaction processing benchmark. The objective of the database constructor is to construct a specific database into a target database system.

The database that is constructed is a video games database. Figure 6.1 represents the ER (Entity-Relationship) model of the database. The database is composed of genres, games, platforms and categories. Each game must have a single genre and be available at one or more platforms. Each platform must belong to a single category. There cannot be repeated genres, game titles, platforms or categories in their respective tables. After creating the tables the database constructor populates the database with a 1990 games, 7 genres, 24 platforms, 2183 game to platform associations and 2 categories.

[1] http://www.ece.wisc.edu/ pharm/tpcw.shtml
[2] http://rubis.ow2.org/

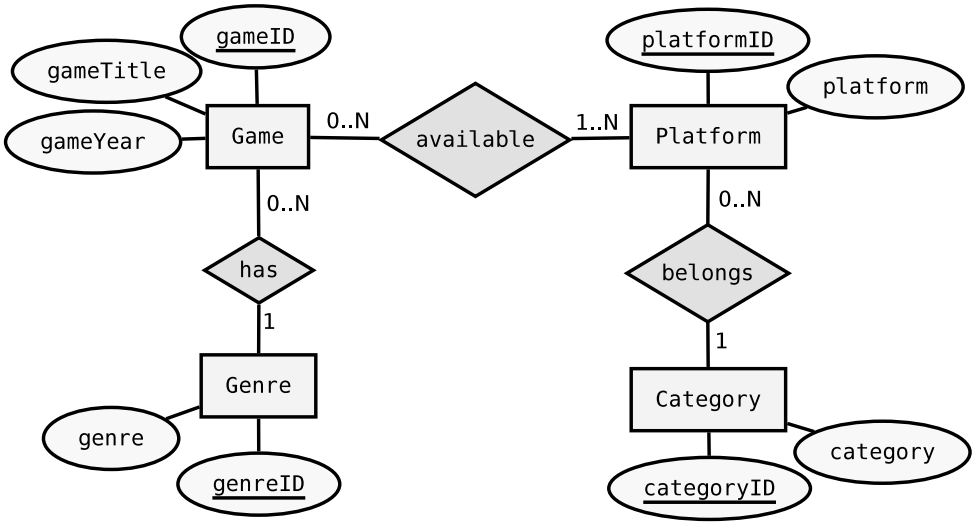


Figure 6.1: Games Database ER Diagram

Benchmark Executor

In this section we describe the benchmark executor of our transaction processing benchmark. The objective of the benchmark executor is to operate over a target video games database and generate performance statistics based in the response of the database server.

Figure 6.2 shows the model of execution of the benchmark executor. The main process, called master, starts by creating a set of threads, called clients, to operate over a video games database. The master then waits for the response of the clients. Each client sends transactions to the database and generates performance statistics based in the responses from the database. The clients repeat these operations until a time specified by the user elapses. When the time ends, each client sends its performance statistics to the master and terminates. The master gathers all the client performance statistics, joins them and stores the final results in the disk.

Figure 6.3 represents the model of execution of a client. The client execution is a continuous loop that only ends when a time specified by the user elapses. Each iteration of the loop starts with the simulation of the think time of the user. The think time of a user is implemented with the thread sleep method. The time the sleep takes is a random value from a time interval specified by the user. The client then sends a transaction to search for a set of random game titles in the video games database. The returned game titles are used afterwards to send a read-only or a read-write transaction.

If the client chooses to send a read-only transaction, it retrieves the details of the previously searched set of random games (game title, year, genre, platforms and respective categories) by sending the look up transaction. Otherwise, if a read-write transaction is chosen, the client removes or inserts a set of games. If it is the first time the client chooses to send a read-write transaction or the last sent read-write transaction was an insert transaction, the client removes the previously searched set of games by sending the remove games transaction. On the other hand, if the last sent read-write transaction was a remove transaction, the client inserts the removed games. This behavior keeps the number of games in the database constant. Since it is

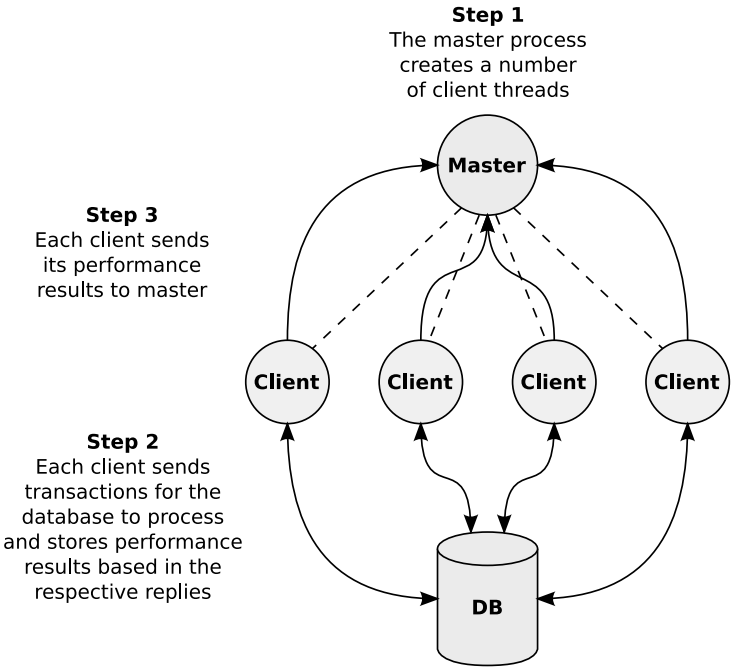


Figure 6.2: Benchmark executor model of execution

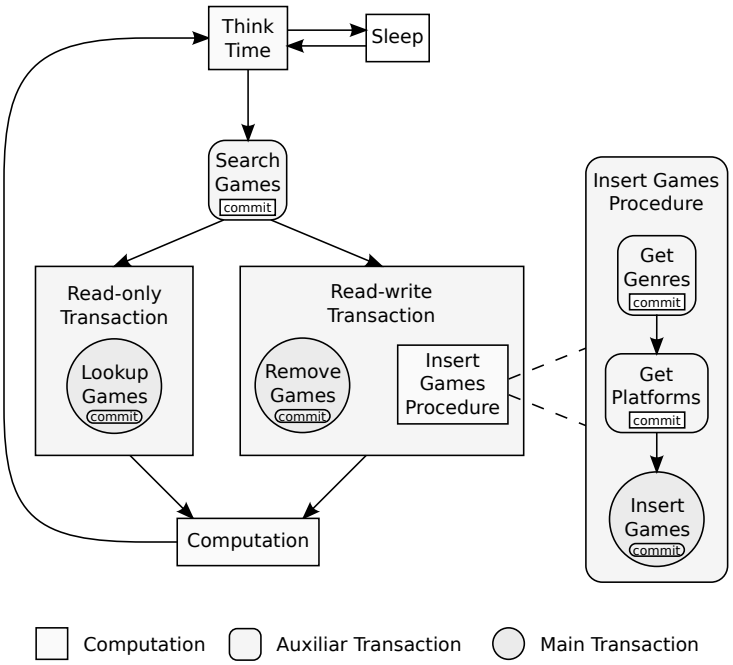


Figure 6.3: Client model of execution

only possible to insert complete games (game title, genre and platforms) and we only have the titles of the removed games, the insert procedure generates a random genre and platforms for each game title. The insert procedure first sends the get genres and get platforms transactions to get the random genre and platforms. Only then, the insert games transaction is sent. We refer to the look up, remove and insert transactions as the main transactions, since we consider the other transactions as auxiliary transactions. However, note that all transactions are independent from each other and commit in the end. Finally, after the commit of the main transactions, the client does a calculation. The calculation simulates the post processing of the data after each operation of the database. In our case we create an image of a bar chart with the number of committed transactions.

The next section describes the available options in the benchmark executor.

Benchmark Executor Options

As previously mentioned the benchmark application was designed to support a large number of options. In this section we describe the most important options available in the benchmark executor.

One of the options of the benchmark executor is to activate speculative execution in the commit operation of the main transactions (the look up, remove and insert transactions). Since in JDBC the commit operation does not return any value, the commit operation can execute concurrently until another database operation needs to execute. Consequently, the commit operation can only execute concurrently with the creation of the image of the bar chart.

We also added an option to activate speculative execution the main transactions. All the database operations of a main transaction are implemented at a respective method, except for the commit operation. We call these methods the main transaction methods. The commit operation of the main transactions is located outside the main transaction methods. We forced the sequential execution of the main transaction methods and the commit operation by using the *multiple target computation coordinator* (see Sections 4.2.5 and 4.3.2). All main transaction methods return a result. The main transaction methods are executed concurrently with its subsequent computations, but when the result from one of the main transaction methods is required, the benchmark executor waits until the respective main transaction method ends its execution. The result of the main transaction methods can be obtained afterwards. This option allows the main transactions to execute concurrently with the creation of the image of the bar chart.

The benchmark executor allows to set the percentage of read-only transactions each client will automatically choose. The importance of this option is that it allows the benchmark have various workloads. The performance improvement by using speculative execution may be different depending in the workload that is selected.

Another option is to set the size of the bar chart image. The larger the bar chart image, the longer it takes to create. After creation of the bar chart image, the benchmark executor waits for the end of execution of all methods executing concurrently due to speculative execution.

Thus, the time the bar chart image takes to complete, is one of the limits we can gain from using speculative execution.

6.3.2 Automatic Speculative Execution Mechanism in the Benchmark Application

In this section we present the target and stop methods we have selected, to automatically add speculative execution to our benchmark application. Our implementation of the automatic speculative execution mechanism can be found in Section 5.3.

Target Methods

The target methods are the set of methods where speculative execution should be added. These methods execute concurrently with their subsequent computations. If a computations requests the result from a target method, and the result is not yet known, the computation uses most probable result of the target method instead. We call the most probable result of a target method, the speculative result.

The methods we have selected to be target methods are a subset of the JDBC write-only database operations. The reason for the selection of the following methods is that their results are highly predictable. The interfaces of the target methods are located at the `java.sql` package. The interfaces are the following: `Statement`, `PreparedStatement` and `Connection`.

We have selected a single target method from the `Statement` interface. Below we present the target method description, the value it returns and the speculative result we have defined for it:

- **`int`** `executeUpdate(String sql)`
  - **Description**: Executes the given SQL statement, which must be a write-only database operation.
  - **Returns**: Returns the number of updated, inserted or deleted rows.
  - **Speculative Result**: Defined as 1. In our benchmark application all write-only database operations insert or delete a single row. However, note that in general the delete and update results can be different from 1.

The `PreparedStatement` interface is an extension of the `Statement` interface. Consequently, the target method we have selected from the `PreparedStatement` interface is similar to the `Statement` interface. The only difference is that in the `PreparedStatement` the SQL statement is part of the object itself. Finally, we have selected the following target method from `Connection` interface:

- **`void`** `commit()`
  - **Description**: Makes all changes made since the previous commit/rollback permanent and releases any database locks currently held by the `Connection` object.

Stop Methods

The stop methods are the set of methods which define the limit that any active target method can execute concurrently due to speculative execution and the limit that any active atomic block can encompass subsequent computations.

The interfaces of the stop methods we have selected are all from the `java.sql` package. The stop methods belong to the same interfaces as the target methods: `PreparedStatement`, `Statement` and `Connection`. We have included all the previous set of target methods into the set of stop methods, plus the following methods. Below we present the additional stop methods from the `Statement` interface:

- **`boolean`** `execute(String sql)`
  - **Description**: Executes the given SQL statement, which may be any kind of SQL statement.
  - **Returns**: True if the first result is a `ResultSet` object; false if it is an update count or there are no results
- **`ResultSet`** `executeQuery(String sql)`
  - **Description**: Executes the given SQL statement, which returns a single ResultSet object.
  - **Returns**: a ResultSet object that contains the data produced by the given query

The additional stop methods we have selected from the `PreparedStatement` are similar to the ones described above but without the `sql` argument. The reason to have assigned such methods as stop methods is because, according to our design, database operations cannot be sent based in a speculation.

6.4 Evaluation

This section reports the results of an experimental study aimed at evaluating the performance improvements database clients can obtain when using speculative execution. Namely, we use the transaction processing benchmark we have implemented as our case study.

In the next sections, we start by describing the methodology for this evaluation. Afterwards, we describe the experimental settings and the benchmark parameters we have used in all performed tests. Next, we present and analyze the results we have obtained in two different configurations. Finally, we present some final remarks regarding the results we have obtained in both configurations.

6.4.1 Methodology

To achieve our goal we have performed a series of tests with our benchmark application using different parameters and configurations. In all our tests, we compare the performance of

an instance of our benchmark without speculative execution with the performance of another instance of the benchmark using speculative execution.

In each test, we have used one of the following three different methods for our benchmark application to use speculative execution:

- Activate the option to use speculative execution in the commit operation of the main transactions.
- Activate the option to use speculative execution in the main transactions.
- Use our automatic speculative execution mechanism to add speculative execution to the source code of our benchmark.

To improve the reliability of our results, we have executed each instance of the benchmark 10 times. Afterwards, the executions that gave the best and worst results were removed to reduce the involvement of specific fluctuations in performance of the system. The presented results are the average of the remaining.

6.4.2 Experimental Settings

In our experiments the database client and the database server were executed in separate machines. The database client ran in a machine with eight AMD Opteron 8220 processors and 32GB of DDR2-RAM. The AMD Opteron 8220 is a dual core processor with maximum frequency of 2.8GHz. For the database server we used PostgresSQL 8.4 running in an independent machine with an Intel Core 2 Duo P8600 processor and 4GB of DDR2-RAM. The Intel Core 2 Duo P8600 is a dual core processor with maximum frequency of 2.4Ghz. Both machines ran in Debian 5.0 operating system, using a 64-bit Linux 2.6 kernel.

The tests were performed in two different configurations. In the first configuration, the database client and database server were located at the same local area network (LAN). In the second configuration, the database client and database server were located at remote networks. The latency between the database client and the database server was of 0.5 ms in the local area network and 50 ms in the remote network.

6.4.3 Transaction Processing Benchmark Parameters

The main objective for this evaluation was to know if speculative execution is worth using in database clients. Hence, realism was an important factor in this evaluation. In a realistic setting a large amount of users may concurrently access a database system. However, we also required every client to be always capable of executing concurrent threads simultaneously with each other, even though this lessens realism. Only this way it was possible to know the maximum performance that our benchmark could obtain by using speculative execution. Since the machine we used to run the benchmark had eight dual core processors and each client concurrently executes a maximum of two threads, we set the benchmark to run with 8 clients.

Option	Value
Number of clients	8
Percentage of Read-Only transactions	95%
Client minimum think time	200 ms
Client maximum think time	500 ms
Benchmark time of execution	5 min
Number of games to search	20
Number of games to look up, insert or remove	1

Table 6.1: Evaluation options

A user accessing our games database most probably only wants to check the details of a set of games. Also, the insert and remove operations should be reserved for users with special attributes (e.g., administrator). Hence, in a realistic setting the percentage of read-write operations should be very low. For those reasons, we ran the benchmark with 95% read-only transactions, leaving only 5% read-write transactions.

A user should also take some time thinking on what operations he/she wants to execute in the games database. Moreover, time should also be spent inputing the necessary data to execute the operations. The total time for a user to think and enter input data to execute any of the available operations for the games database, should be around 10 to 25 seconds. With such think time the benchmark should be running for 4 hours to obtain a reasonable number of committed transactions. From our experiments [3] we concluded that the benchmark behaves similarly if we lower all times proportionally. Hence, we set the think time in the interval of 200 and 500 milliseconds and a total time to run the benchmark of 5 minutes.

During the execution of the benchmark, the client searches a set of random games and looks up, inserts or removes a number of those games from the database. From our experiments [3] we concluded that varying the number of games the client looks up, inserts or removes does not have a significant impact in the performance improvement of the client when using speculative execution. We have set the number of random games to be searched to 20 and the number of games to look up, insert or remove to 1.

Table 6.1 summarizes the options we use in this evaluation. Next we present the results we have obtained in the LAN configuration.

6.4.4 Local Area Network Tests

One important factor in this evaluation is the latency between the database client and the database server. The importance of such factor is that the latency has a high impact in the time the database operations take to complete. In the LAN configuration the latency between the database client and the database server was of 0.5 ms.

Another important factor is the time the bar chart creation takes to complete. The time for computing the bar chart limits the maximum concurrent execution the benchmark can obtain when using speculative execution. The time the bar chart takes to complete depends on the

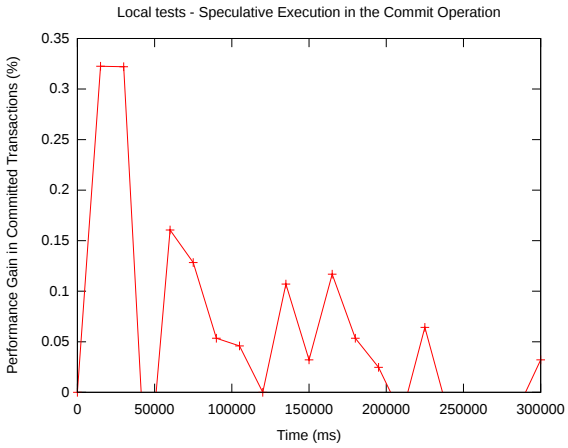
[3] These particular experiments are not shown in this dissertation.

size of the produced image. From our experiments we concluded that a bar chart image of size 100×100 pixels gives us acceptable results.

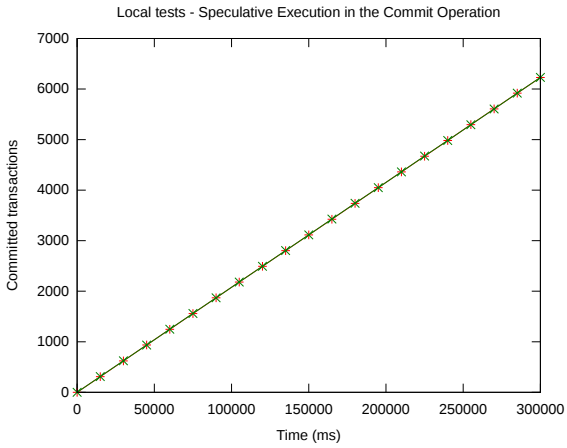
Next, we present the performance results of our benchmark when using three different methods for it to use speculative execution. First, we present the performance results when the option to use speculative execution only in the commit operation of the main transactions was active. Afterwards, we present the performance results when the option to use speculative execution in the main transactions was active. Finally, we present the performance results when we used the automatic speculative execution mechanism to add speculative execution to the source code of the benchmark.

Speculative execution in the commit operation

Figure 6.4 illustrates two charts comparing the performance of a benchmark instance using speculative execution only in the commit operation of the main transactions against another benchmark instance not using speculative execution. Figure 6.4a illustrates the percentage of additional committed main transactions of the benchmark instance using speculative execution when compared with the benchmark instance not using speculative execution. In other words, it represents the performance gain of the benchmark instance using speculative execution. Figure 6.4b illustrates the number of committed main transactions of both instances. Table 6.2 shows the average time each operation took executing sequentially in both instances. That is, in this table the time an operation was concurrently executing with any other operation is not account.



(a) Performance gain of the instance using speculative execution



(b) Number of committed main transactions of both instances

Figure 6.4: Speculative execution in the commit operation - Performance comparison of a benchmark instance using speculative execution against another benchmark instance not using

Figure 6.4a shows that there was very little performance improvement, around 0.1% in average. Also, there are many intervals in time were there is no performance improvement at all. Figure 6.4b shows that the number of committed main transactions is almost equal over time. Table 6.2 shows that all the operations took similar times executing sequentially in

Operation	Avg. time without Spec. Exec.	Avg. time with Spec. Exec.
Think	349.94 ms	350.73 ms
Search Games	3.38 ms	3.41 ms
Get Genres	2.40 ms	2.43 ms
Get Platforms	2.77 ms	2.79 ms
Lookup Games	4.18 ms	4.2 ms
Remove Games	4.18 ms	4.18 ms
Insertion Games	10 ms	10.07 ms
Commit Operation	1.1 ms	0.05 ms
Bar Chart Creation	26.05 ms	26.12 ms

Table 6.2: Speculative execution in the commit operation - Time each operation took executing sequentially

both instances, except for the commit operation. The benchmark instance using speculative execution took in average less 1.05 ms executing sequentially the commit operation than the benchmark instance not using speculative execution.

In each of the presented charts along this dissertation, there can be spikes or variations at start, such as the initial spike shown in Figure 6.4a. These variations can appear for several reasons, such as the database initial warmup or spikes in the connection. Because these variations are not significant, we only discuss the end results where the performance gain is already established.

The performance results can be explained if one knows the time that is gained in each iteration of the client loop. Figure 6.5 represents the time that a client spends sequentially executing the operations of a single iteration of its loop. The benchmark instance without speculative execution always executes these operations sequentially. However, the benchmark instance using speculative execution concurrently executes the commit operation with the bar chart creation. Since the time to execute the commit operation is much lower than the time to create the bar chart, the execution of the commit operation completely overlaps with the creation of the bar chart. Consequently, for each iteration of the client loop, the time executing the commit operation is gained. Since the time executing the commit operation is very small compared with the time executing an iteration of the loop, there is not much concurrent execution, which explains why there is very little performance gain.

$$InsertProcedure = GetGenres + GetPlatforms + InsertGames$$

$$MainTransaction = LookupGames \vee RemoveGames \vee InsertProcedure$$

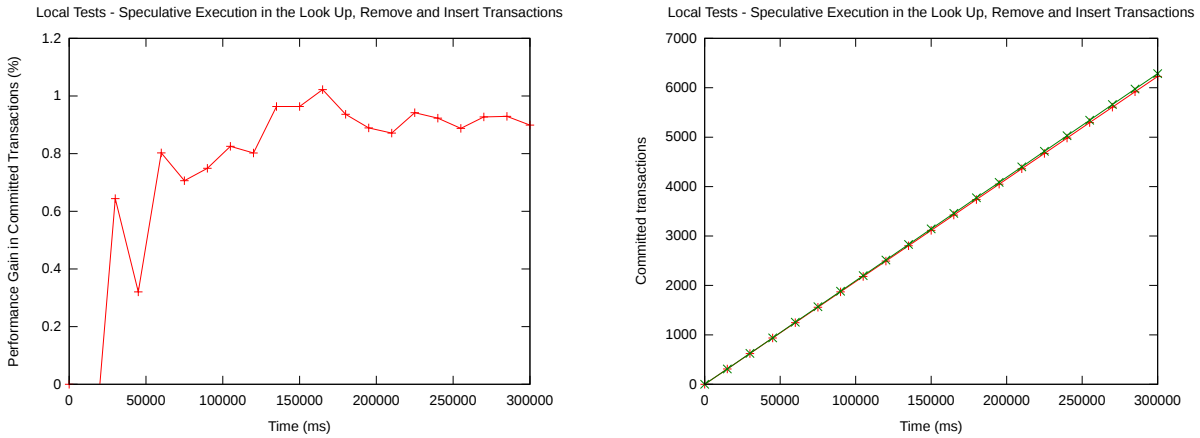
$$ClientIteration = Think + SearchGames + MainTransaction + Commit + BarChart$$

Figure 6.5: Time a client spends executing sequentially the operations of a single iteration of its loop

The performance of our benchmark can be improved if speculative execution is used in whole main transactions. Next, we present the performance results when the option to use speculative execution in the main transactions was active.

Speculative execution in the main transactions

Figure 6.6 presents two charts comparing the performance of a benchmark instance using speculative execution in the main transactions against another benchmark instance not using speculative execution. Figure 6.6a illustrates the performance gain of the benchmark instance using speculative execution. Figure 6.6b illustrates the number of committed main transactions of both instances. Table 6.3 shows the average time each operation took executing in both instances.



(a) Performance gain of the instance using speculative execution

(b) Number of committed main transactions of both instances

Figure 6.6: Speculative execution in the main transactions - Performance comparison of a benchmark instance using speculative execution against another benchmark instance not using

Operation	Avg. time without Spec. Exec.	Avg. time with Spec. Exec.
Think	349.94 ms	351.6 ms
Search Games	3.38 ms	3.39 ms
Get Genres	2.40 ms	2.53 ms
Get Platforms	2.77 ms	2.71 ms
Lookup Games	4.18 ms	0.1 ms
Remove Games	4.18 ms	0.11 ms
Insertion Games	10 ms	0.3 ms
Commit Operation	1.1 ms	0.03 ms
Bar Chart Creation	26.05 ms	26.24 ms

Table 6.3: Speculative execution in the main transactions - Time each operation took executing sequentially

Figure 6.6a shows that at start the performance gain is null, but later it raises to 1% and keeps a similar percentage over time. Figure 6.6b shows that the gap in committed main transactions of both instances is slowly increasing over time. Table 6.3 shows that all operations took the same time executing sequentially in both instances, except for the look up, remove, insert and commit operations. The time the benchmark instance using speculative execution took executing sequentially the look up, remove, insert and commit operations was much lower than with the benchmark not using speculative execution.

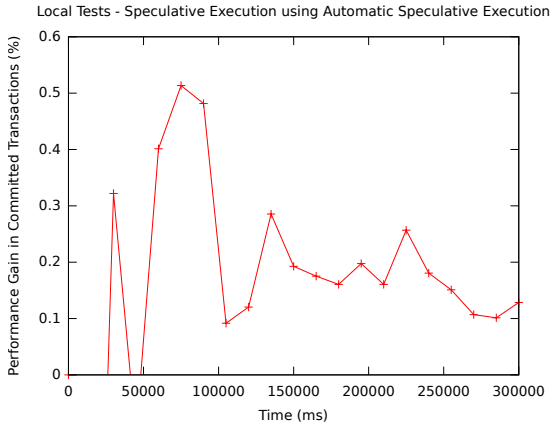
In each iteration of the client loop, the benchmark instance using speculative execution concurrently executes any of the main transactions with the bar chart creation. Since the time to execute any of the main transactions is much lower than the time to create the bar chart, the execution of the main transactions completely overlaps with the creation of the bar chart. For that reason, the time for executing sequentially the look up, remove, insert and commit operations is almost null and can be ignored. Consequently, in each iteration of a client loop, the time executing the look up, remove, insert and commit operations is gained.

The time to concurrently execute the whole operations of the main transactions is much larger than only concurrently executing the commit operation. For that reason, there was a much larger performance gain in this test when compared with the previous one. Also, note that if the main transactions took a larger time to complete the performance gain would be larger, since the transactions would execute concurrently with the bar chart creation for a longer period.

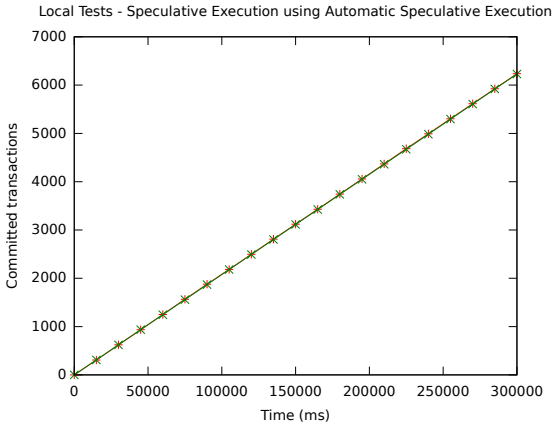
Next, we present the performance results of our benchmark when adding speculative execution to its source code by using our automatic speculative execution mechanism.

Speculative execution using the automatic speculative execution mechanism

Figure 6.7 presents two charts comparing the performance of a benchmark instance using speculative execution, through the use of our automatic speculative execution mechanism, against another benchmark instance not using speculative execution. Figure 6.7a illustrates the performance gain of the benchmark instance using speculative execution. Figure 6.7b illustrates the number of committed main transactions in both instances. Table 6.4 shows the average time each operation took executing sequentially in both instances.



(a) Performance gain of the instance using speculative execution



(b) Number of committed main transactions of both instances

Figure 6.7: Speculative execution by using the automatic speculative execution mechanism - Performance comparison of a benchmark instance using speculative execution against another benchmark instance not using

Figure 6.4a shows the performance gain starts with two spikes with respective percentages of 0.35% and 0.5%, and later keeps an average percentage of 0.15%. Figure 6.4b shows that the

Operation	Avg. time without Spec. Exec.	Avg. time with Spec. Exec.
Think	349.94 ms	350.4 ms
Search Games	3.38 ms	3.44 ms
Get Genres	2.40 ms	2.53 ms
Get Platforms	2.77 ms	2.68 ms
Lookup Games	4.18 ms	4.25 ms
Remove Games	4.18 ms	4.24 ms
Insertion Games	10 ms	10.14 ms
Commit Operation	1.1 ms	0.08 ms
Bar Chart Creation	26.05 ms	26.04 ms

Table 6.4: Speculative execution by using the automatic speculative execution mechanism - Time each operation took executing sequentially

number of committed main transactions is almost equal over time. Table 6.2 shows that most operations took the same time executing sequentially in both instances, except for the commit operation. The time executing the commit operation is almost null in the benchmark using speculative execution.

Despite having performed a little better, the results are very similar to when the option to use speculative execution in the commit operation was active. The reason for this is because, of all the transformed database operations, the only one that concurrently executes for a significant amount of time is the commit operation. Besides the database operations the only computation that takes a significant amount to complete is the bar chart creation, and the commit operation is the only one that can concurrently execute with it. Because of the way the source code of our benchmark is structured the other database operations were not transformed to execute concurrently with the bar chart creation.

Next, we present the performance results we have obtained in the remote network configuration.

6.4.5 Remote Network Tests

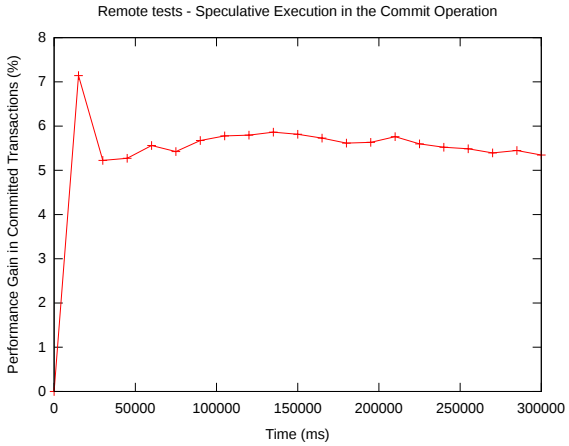
The latency between the database client and database server in the remote network configuration was of 50 ms. Because of the higher latency, the database operations took much more time to complete than in the LAN configuration. For the database operations to concurrently execute for a significant amount of time, we have adjusted the bar chart image size to 300×300 pixels.

Next, we start by presenting the performance results when the option to use speculative execution in the commit operation of the main transactions was active.

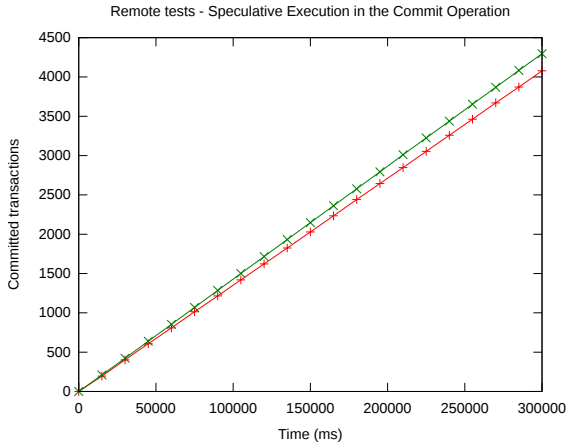
Speculative execution in the commit operation

Figure 6.8 presents two charts comparing the performance of a benchmark instance using speculative execution in the commit operation of the main transactions against another benchmark instance not using speculative execution. Figure 6.8b illustrates the performance gain of the benchmark instance using speculative execution. Figure 6.8a illustrates the number of commit-

ted main transactions of both instances. Table 6.5 shows the average time each operation took executing sequentially in both instances.



(a) Performance gain of the instance using speculative execution



(b) Number of committed main transactions of both instances

Figure 6.8: Speculative execution in the commit operation - Performance comparison of a benchmark instance using speculative execution against another benchmark instance not using

Operation	Avg. time without Spec. Exec.	Avg. time with Spec. Exec.
Think	350.40 ms	350.75 ms
Search Games	50.47 ms	48.62 ms
Get Genres	49.14 ms	48.76 ms
Get Platforms	49.33 ms	50 ms
Lookup Games	52.09 ms	49.63 ms
Remove Games	72.59 ms	73.05 ms
Insertion Games	170.23 ms	167.52 ms
Commit Operation	24.19 ms	0.16 ms
Bar Chart Creation	105.95 ms	102.79 ms

Table 6.5: Speculative execution in the commit operation - Time each operation took executing sequentially

Figure 6.8a shows the performance gain starts with a spike of 7%, and afterwards keeps an almost constant gain around 5.2%. Figure 6.8b shows that gap in committed transactions between both instances is getting larger with time. Table 6.5 shows that most operations took the same time executing sequentially in both instances, except for the commit operation. In the benchmark instance using speculative execution, the time executing sequentially the commit operation is almost null.

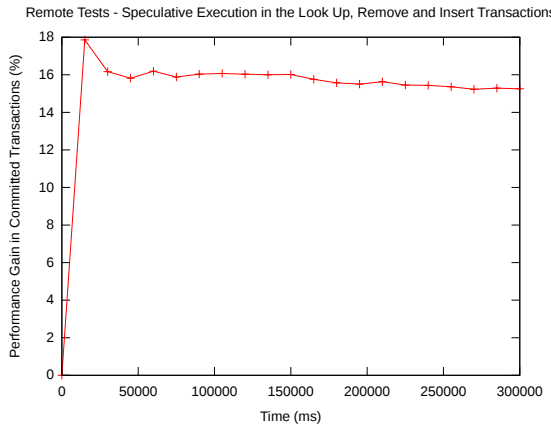
The time the commit operation took to complete is much smaller than the time to create the bar chart. For that reason, in the benchmark instance using speculative execution, the commit operation almost fully executes concurrently with the bar chart creation and its time executing sequentially is almost null. The time of the commit operation compared with the time of an iteration of a client loop is larger in proportion than with our equivalent test in the LAN configuration. Consequently, there was a larger performance gain in this test.

Next, we show the performance results when the option to use speculative execution in the

whole operations of the main transactions is active.

Speculative execution in the main transactions

Figure 6.9 presents two charts comparing the performance of a benchmark instance using speculative execution in the whole operations of the main transactions against another benchmark instance not speculative execution. Figure 6.9a illustrates the performance gain of the benchmark instance using speculative execution. Figure 6.9b illustrates the number of committed main transactions of both instances. Table 6.6 shows the average time each operation took executing sequentially in both instances.



(a) Performance gain of the instance using speculative execution

(b) Number of committed main transactions of both instances

Figure 6.9: Speculative execution in the main transactions - Performance comparison of a benchmark instance using speculative execution against another benchmark instance not using

Operation	Avg. time without Spec. Exec.	Avg. time with Spec. Exec.
Think	350.40 ms	351.02 ms
Search Games	50.47 ms	48.60 ms
Get Genres	49.14 ms	47.94 ms
Get Platforms	49.33 ms	49.52 ms
Lookup Games	52.09 ms	0.22 ms
Remove Games	72.59 ms	1.32 ms
Insertion Games	170.23 ms	64.32 ms
Commit Operation	24.19	1.58 ms
Bar Chart Creation	105.95 ms	104.67 ms

Table 6.6: Speculative execution in the main transactions - Time each operation took executing sequentially

Figure 6.9a shows the performance gain initially spikes to 18%, and afterwards keeps an almost constant gain around 16%. Figure 6.9b shows that the gap in committed transactions between both instances is rapidly getting larger with time. Table 6.6 shows that all operations took the same time executing sequentially, except for the look up, remove, insert and commit operations. The time executing the look up, remove and commit operations is almost null. The time executing the insert operation took less 105 ms to complete.

The time executing the look up and remove transactions is much smaller than the time executing bar chart creation. Because of that, the look up and insert almost fully execute concurrently with the bar chart creation execution. Consequently, the execution time of the remove and insert operations is almost null.

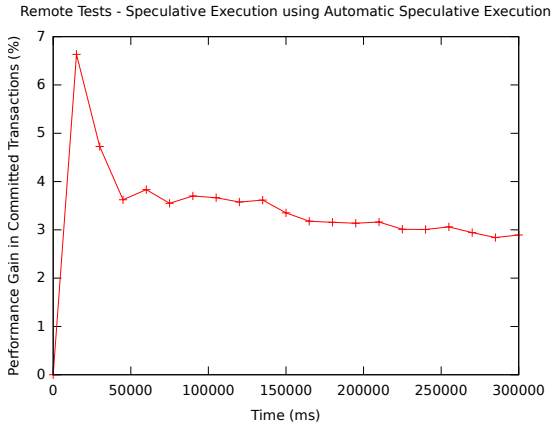
The time executing the insert transaction is larger than the time executing the bar chart creation. Since our benchmark does not allow further concurrent execution, the time concurrently executing the insert transaction is limited by time it takes to create the bar chart. For that reason, the time executing the insert operation took less 105 ms to complete, which is the time it takes to create the bar chart.

Note that when the insert transaction executes, the commit operation executes sequentially, because there cannot be further concurrent execution. However, since the majority of operations that are sent are read-only transactions, in the majority of iterations the commit operation almost fully executes concurrently with the bar chart creation. For that reason, the average time to execute the commit operation is near null, but not as low as in the tests we have done previously.

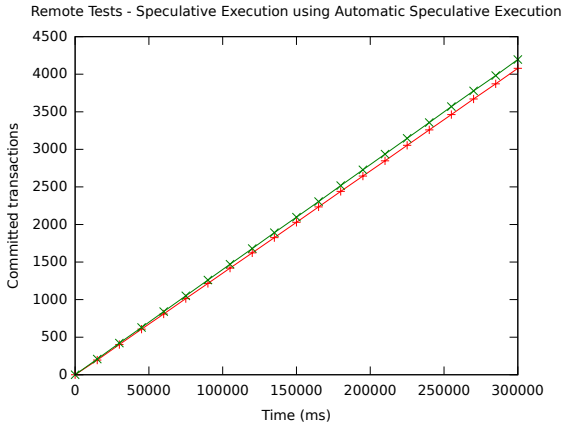
Next, we present the performance results when speculative execution is added to the database client through the use of our automatic speculative execution mechanism.

Speculative execution using the automatic speculative execution mechanism

Figure 6.10 presents two charts comparing the performance of a benchmark instance using speculative execution through the use of our automatic speculative execution mechanism against another benchmark instance not speculative execution. Figure 6.10a illustrates the performance gain of the benchmark instance using speculative execution. Figure 6.10b illustrates the number of committed main transactions of both instances. Table 6.7 shows the average time each operation took executing sequentially in both instances.



(a) Performance gain of the instance using speculative execution



(b) Number of committed main transactions of both instances

Figure 6.10: Speculative execution by using the automatic speculative execution mechanism - Performance comparison of a benchmark instance using speculative execution against another benchmark instance not using

Operation	Avg. time without Spec. Exec.	Avg. time with Spec. Exec.
Think	350.40 ms	350.42 ms
Search Games	50.47 ms	55.12 ms
Get Genres	49.14 ms	52.22 ms
Get Platforms	49.33 ms	53.74 ms
Lookup Games	52.09 ms	55.25 ms
Remove Games	72.59 ms	79.94 ms
Insertion Games	170.23 ms	185 ms
Commit Operation	24.19 ms	0.28 ms
Bar Chart Creation	105.95 ms	103.9 ms

Table 6.7: Speculative execution by using the automatic speculative execution mechanism - Time each operation took executing sequentially

Figure 6.10a shows the performance gain starts with a spike to 6.8%, and afterwards keeps an almost constant gain around 3.5%. Figure 6.10b shows that the gap in committed transactions between both instances is getting larger with time. Table 6.7 shows that most operations took the same time executing sequentially, except for the commit operation. The time executing the commit operation is almost null in the benchmark using speculative execution.

As previously mentioned in our equivalent test using the LAN configuration, the only computation that takes a significant time executing concurrently is the commit operation. For that reason, despite the performance gain being a little lower, the performance results are similar to the results when using speculative execution only in the commit operation.

Next, we present some final remarks regarding the results we have obtained in both configurations.

6.4.6 Final remarks

In our experiments, every benchmark instance using speculative execution performed better than an instance with similar configurations without speculative execution. From our experiments speculative execution can increase the performance of a database client. However, the program only performs significantly better if speculative execution can introduce a large amount of concurrent computation to the original program.

The time that can be gained by using speculative execution is the amount of time the target methods and its subsequent computations execute concurrently. In our experiments in the LAN configuration, the time of execution of the target methods was small. For that reason, although all the target methods fully executed concurrently, the performance gain was low. In our experiments in the remote configuration, the time of execution of the target methods was large. As a results, the performance gain was high. However, note that not all target methods in the remote configuration fully executed concurrently. In our experiment with speculative execution in the main transactions, the insert transaction could not fully execute concurrently, which limited the performance gain.

When our benchmark was subject to the automatic speculative execution mechanism the results obtained were similar to the as original benchmark using speculative execution in the

commit operation. The advantage of the automatic speculative execution mechanism is that a user does not require deep knowledge about speculative execution to add it to his database client.

7

Conclusions

The objective of this dissertation was to study and develop mechanisms to explore speculative execution in programs.

To achieve our goal we first designed an abstract process to add speculative execution to the source code of a program. During the development of the process we have also studied the following aspects. First, we studied the conditions for adding speculative execution to the source code of a program. Second, we have studied the maximum performance gain we can obtain by adding speculative execution to the source code of a program. This process was the base for all the speculative execution mechanisms we have developed afterwards.

Later, we have developed a speculative execution mechanism that enabled a programmer to easily integrate speculative execution to the source code of a program. The mechanism, based in explicit futures, is simple enough for a programmer to easily and quickly add speculative execution to the operations of his/her choice.

Finally, we have developed a mechanism to automatically add speculative execution to the source code of a program. By doing source-to-source transformation, this mechanism allows a user to add speculative execution to the source code of a program, even without any deep knowledge about speculative execution. This mechanism uses an extension of explicit futures, which we have called speculative futures, and transactional memory.

We have tested our developed mechanisms in a custom transaction processing benchmark. From our experiments we have concluded that speculative execution can increase the performance of database clients. It should be able to obtain similar improvements for programs that include operations with long execution times. We have also studied the ideal properties for a program to be able to improve its performance with the use of speculative execution.

7.0.7 Future Work

During the development of our work, several aspects that can be improved in the future were identified. This sections highlights the most important.

Our design of the *multiple target computation coordinator* forced the execution of all target methods to be sequential within the same scope. We did this by resorting a blocking queue that would execute each target method in the order they were inserted. Although this design was sufficient for our experiments, the design can be improved to solve the following two problems. First, in the original program the target methods could already execute in separate scopes. Second, some of the target methods could not depend on any other target method. In both cases, it would not be necessary for all target methods to execute sequentially in the same scope. A better design would be for the *multiple target computation coordinator* to execute target methods in dependency order.

One of the main limitations of the automatic speculative execution mechanism is that to be able to abort a speculation, we encapsulate the code inside an atomic block. Being the atomic block a language constructor, we are limited to maximum amount of code we can encapsulate within it. An alternative solution would be to start and end a memory transaction through method calls. This way, we could have more code within a memory transaction.

Having the memory transactions starting and ending as methods would solve part of the problem. However, we still have to define when should a speculation be validated. The best moment to validate a speculation is when the final result is known. This way we exploit the maximum concurrent execution and reduce the cost of aborting in case the speculation was incorrect. A solution would be to have a subtask controlling the speculations. This subtask could decide assincronously when to validate a speculation and, if necessary, abort and re-execute the respective memory transaction.

Our mechanism already can improve the performance of a program by integrating speculative execution. However, we believe these sugestions would help in an attempt to exploit the maximum possible performance that can be gained through the use of speculative execution.

Appendix

A.1 Automatic Speculative Execution Mechanism Example

This section shows an example of a program that is subject to our implementation of the automatic speculative execution mechanism. Figure A.1 illustrates the program before being subject to the automatic speculative execution mechanism. The program contains five methods: `a()`,`b()`,`c(boolean c)`,`d(boolean d)` and `run()`. The `a()` method executes some costly computation and returns true. The `b()` method executes some computation and does not return any value. The `c(boolean c)` method executes some computation and returns its `c` argument. The `d(boolean d)` outputs its `d` argument to screen. The `run()` method executes each of these methods sequentially in alphabetical order. The `c(boolean c)` and `d(boolean d)` methods use the value that is returned by the `a()` method as their argument.

In our implementation of the automatic speculative execution mechanism, first a set of target methods and stop methods must be input. Figure A.2 shows the set of target methods have been selected. A target method is defined by its package, method signature and speculative result. The only method that was defined as a target method was the `a()` method. The speculative result that was defined for the `a()` method was `true`, since the `a()` method always return `true`. Figure A.3 shows the set of stop methods that have been selected. A stop method is defined by its package and method signature. The only method that was defined as a stop method was the `d(boolean)` method. The reason for the selection of this method is because it requires to output the its argument to screen.

Figure A.4 shows the program after being subject to the automatic speculative execution mechanism. First, our implementation of the *multiple target computation coordinator*, called `SerialQueueExecutor`, is placed in the fields and constructor of the `ASEM_Example` class. The variable that holds the `SerialQueueExecutor` object is called `executor`. Then, the `run()`

```
1  package example;
2
3  public class ASEM_Example {
4
5    public ASEM_Example() {
6    }
7
8    private boolean a() {
9      /* Executes some costly computation */
10     return true;
11   }
12
13   private void b() {
14     /* Executes some computation */
15   }
16
17   private boolean c(boolean c) {
18     /* Executes some computation */
19     return c;
20   }
21
22   private void d(boolean d) {
23     System.out.println(d);
24   }
25
26   public void run() {
27     boolean a = a();
28     b();
29     c(a);
30     d(a);
31   }
32 }
```

Figure A.1: Program before being subject to the automatic speculative execution mechanism

method is transformed to support speculative execution.

The execution of the `a()` method is replaced by the construction of a speculative future. This speculative future is constructed with a `Callable<Booblean>` object, whose `call()` method executes the `a()` method, and a `true` value. The `call()` method represents the associated computation of the speculative future and the `true` value the speculative result of the `a()` method. The speculative future then executes concurrently with its following computations, by resorting to the previously introduced *multiple target computation coordinator*.

Since the `c(boolean c)` method requires the result from a target method, it is transformed to execute inside an atomic block. The code shows the atomic block after being transformed by our last pass of the source-to-source compiler. The atomic block is transformed into a while loop that starts with the creation of a memory transaction and ends with the commit of this memory transaction. All code executed after the creation of the memory transaction is contained inside a `try ... catch` block, that aborts and restarts the memory transaction in case of a `RollbackException`. A `RollbackException` can be triggered either by a conflict

```
1   /*
2    * Target Methods
3    *
4    * <Package> {
5    *   <methodSignature>:<Speculative Result>;
6    * }
7    */
8
9   example.ASEM_Example {
10    a():true;
11  }
```

Figure A.2: Target Methods

```
1   /*
2    * Stop Methods
3    *
4    * <Package> {
5    *   <methodSignature>
6    * }
7    */
8
9   example.ASEM_Example {
10    d(boolean);
11  }
```

Figure A.3: Stop Methods

between concurrent memory transactions or an explicit abort of the memory transaction. Since the `d(boolean d)` method is a stop method, the atomic block must end before its execution.

A new statement is placed at start of the atomic block, to change the value from the `b`, so that it hold the result from the `getSpec()` method from the speculative future. Later the `b` variable is used by the `c(boolean c)` method. At the end of the atomic block, the speculation validation is executed. The speculation validation compares the speculative future final result with the value that the `b` variable holds. If the final result is equal to `b`, the memory transaction commits and the program proceeds. Otherwise, if the final result is different from `b`, the memory transaction aborts and it is re-executed. Before the execution of the `d(boolean d)`, the variable `b` stores the final result from the speculative future, through the call of the speculative future `get()` method, to force the ending of the execution of the target method.

```
1  package example;
2  ...
3  public class ASEM_Example {
4    /* Adds the SerialQueueExecutor */
5    private SerialQueueExecutor executor;
6
7    /* Adds the construction of the SerialQueueExecutor */
8    public ASEM_Example() {
9      this.executor = new SerialQueueExecutor();
10   }
11   ...
12   public void run() throws ExecutionException {
13     /* Replaces the execution of the a() method with the
14      * construction and execution of a speculative future */
15     SpecFuture<Boolean> aFut = new SpecFuture<Boolean>(
16       new Callable<Boolean>() {
17         public Boolean call() {
18           return a();
19         }
20       },true
21     );
22     boolean b;
23
24     /* Executes the speculative future with the SerialQueueExecutor */
25     executor.execute(aFut);
26     b();
27     boolean atomicDone = false;
28     while (!atomicDone) {
29       /* Initiates a memory transaction */
30       TransactionManager.begin();
31       try {
32         /* Get final result, if it is known, or the speculative result
33          * otherwise */
34         b = aFut.getSpec();
35         c(b);
36         /* Speculation Validation */
37         if (!aFut.get().equals(b)) {
38           /* Speculative result was not equal to the final result */
39           TransactionManager.abort();
40         }
41         /* Memory transaction commits and the program proceeds */
42         TransactionManager.commit();
43         atomicDone = true;
44       } catch (RollbackException e) {
45         /* Memory transaction aborts and re-executes */
46         atomicDone = false;
47         TransactionManager.clean();
48       }
49     }
50     b = aFut.get();
51     /* d method gets the result from the future instead */
52     d(b);
53   }
54 }
```

Figure A.4: Program after being subject to the automatic speculative execution mechanism

Bibliography

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 59–74, New York, NY, USA, 2005. ACM.

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM.

[3] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[5] Rance Cleaveland and Scott A. Smolka. Strategic directions in concurrency research. *ACM Comput. Surv.*, 28(4):607–625, 1996.

[6] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: a hybrid quorum protocol for byzantine fault tolerance. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[7] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[8] André Abecasis Gomes Ferreira. Efficient middleware for database replication. Master's thesis, Faculdade de Ciências e Tecnologia / Universidade Nova de Lisboa, 2008.

[9] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[10] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007.

[11] Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[12] 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.

[13] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[14] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[15] Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. *SIGPLAN Not.*, 37(10):184–196, 2002.

[16] Christian Plattner and Gustavo Alonso. Ganymed: scalable replication for transactional web applications. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[17] Nuno M. Preguiça, Rodrigo Rodrigues, Cristóvão Honorato, and João Lourenço. Byzantium: Byzantine-fault-tolerant database replication providing snapshot isolation. In *HotDep*. USENIX Association, 2008.

[18] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[19] Jan Schwinghammer. A concurrent lambda-calculus with promises and futures. Master's thesis, Programming Systems Lab, Universität des Saarlandes, February 2002.

[20] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill Higher Education, 2001.

[21] Cristian Tapus. Kernel level speculative dsm. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 487, Washington, DC, USA, 2003. IEEE Computer Society.