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

Dissertação de Mestrado

Mestrado em Engenharia Informática

RepComp - Replicated Software Components for Improved Performance

Paulo Alexandre Lima da Silva Mariano (29619)

Lisboa
(2010)

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

Dissertação de Mestrado

RepComp - Replicated Software Components for Improved Performance

Paulo Alexandre Lima da Silva Mariano (29619)

Orientador: Prof. Nuno Preguiça

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

Lisboa
(2010)

Abstract

The current trend of evolution in CPU architectures favours increasing the number of processing cores in lieu of improving the clock speed of an individual core. While improving clock rates automatically benefits any software executing on that processor, the same is not valid for adding new cores. To take advantage of an increased number of cores, software must include explicit support for parallel execution.

This work explores a solution based on diverse replication which allows applications to transparently explore parallel processing power: macro-components.

Applications typically make use of components with well-defined interfaces that have a number of possible underlying implementations with different characteristic. A macro-component is a component which encloses several of these implementations while offering the same interface as a regular implementation. Inside the macro-component, the implementations are used as replicas, and used to process any incoming operations. Using the best replica for each incoming operation, the macro-component is able to improve global performance.

This dissertation provides an initial research on the use of these macro-components, detailing the technical challenges faced and proposing a design for the macro-component support system. Additionally, an implementation and subsequent validation of the proposed system are presented. These examples show that macro-components can achieve improved performance versus simple component implementations.

Keywords: diverse replication, multi-core, parallel programming, performance

Resumo

A tendência actual de evolução em arquitecturas de CPU favorece o aumento do número de núcleos de processamento em vez de melhorar a velocidade de relógio de um núcleo individual. O aumento da velocidade de relógio beneficia automaticamente qualquer software em execução neste processador, o mesmo não é válido para a adição de novos núcleos. Para tirar proveito de um aumento do número de núcleos, o software necessita de incluir suporte explícito para execução paralela.

Este trabalho busca explorar uma solução baseada em replicação diversificada que permite a aplicações explorar de forma transparente a capacidade de processamento paralelo do sistema: macro-componentes.

As aplicações geralmente usam componentes com interfaces fixas e múltiplas possíveis implementações, com características diferentes. Um macro-componente é um componente com uma interface igual a estas implementações, que no seu interior usa várias implementações como réplicas. Ao receber uma operação, o macro-componente irá utilizar qualquer destas réplicas para a processar. Utilizando a melhor réplica para cada operação de entrada, é possível de oferecer um desempenho global superior a uma implementação simples.

Esta dissertação apresenta uma investigação sobre o uso de macro-componentes, incluindo os desafios técnicos enfrentados e um desenho arquitectural para o sistema de macro-componentes. Além disso, é apresentada uma implementação do sistema e posterior validação. Nesta validação é verificada uma melhoria de desempenho na utilização de um macro-componente comparado com um componente simples.

Palavras-chave: replicação diversificada, multi-cores, desempenho

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Description 2

1.3 Proposed Solution 4

1.4 Contributions 5

1.5 Outline of the Dissertation 5

2 Related Work **7**

2.1 Replication 7

2.1.1 Eager vs lazy replication 8

2.1.2 Operation distribution algorithms 8

2.1.3 Diverse replication 11

2.2 N-version Programming 13

2.2.1 Object Oriented N-Version Programming 14

2.2.2 Abstraction and opportunistic N-version programming 15

2.3 Parallel Computing Models and Applications 16

2.3.1 Classic Concurrency Control 17

2.3.2 Transactional Memory 18

2.3.3 Speculative Execution 19

3 Model **23**

3.1 Macro-component basis 23

3.1.1 Example: Optimal Complexity Set 24

3.2 Challenges 25

3.2.1 Overhead 25

3.2.2 Performance Differences 28

3.2.3 Scheduling . . . 29
3.2.4 Implementation characteristics . . . 29

4 Design **33**
4.1 Assumptions . . . 33
4.2 Proposed Architecture . . . 34
4.2.1 Operations . . . 35
4.2.2 Macro-Component . . . 37
4.2.3 Global Scheduler . . . 38
4.2.4 Supporting tools . . . 39
4.3 Scheduling . . . 40
4.3.1 Internal Scheduling . . . 40
4.3.2 Global Scheduling . . . 44
4.3.3 Speedup . . . 45
4.4 Addressing Low-Granularity Operations . . . 47
4.4.1 Goals . . . 47
4.4.2 Design Changes . . . 48

5 Implementation **49**
5.1 Runtime System . . . 49
5.1.1 Macro-component template . . . 50
5.1.2 Internal Scheduler . . . 52
5.1.3 Global Scheduler and Executors . . . 53
5.2 In-memory Database Macro-Component . . . 56
5.2.1 Why databases? . . . 56
5.2.2 JDBC . . . 57
5.2.3 Macro-component-based JDBC Driver Solution . . . 58
5.3 Macro-Component Programming Support Tools . . . 59
5.3.1 JavaCC and JJTree . . . 60
5.3.2 Code Generation . . . 60
5.4 Optimized Runtime System . . . 61
5.4.1 Maintaining Consistency . . . 62
5.4.2 Minimizing Response Time . . . 64

6 Evaluation **67**
6.1 Runtime System Evaluation . . . 67
6.1.1 Benchmark . . . 67
6.1.2 Experimental setup . . . 70

6.1.3 Results 71
6.1.4 Analysis 73
6.2 Optimized Runtime Evaluation 74
6.2.1 Benchmark 74
6.2.2 Experimental Setup 77
6.2.3 Results 77
6.2.4 Analysis 80

7 Conclusions and Future Work **83**
7.1 Conclusions 83
7.2 Contributions 85
7.3 Future Work 85
7.3.1 Benchmarking 85
7.3.2 Extending the existing system 86
7.3.3 New directions 87

Introduction

1.1 Motivation

Until recently, CPU evolution was tied to improvements on their instruction sets, usually by adding support for more complex instructions, cache size or levels increase and finally a steady rise in clock speed.

Typically, this last characteristic was the most pursued by both CPU manufacturers and end users as any application could directly benefit from increased CPU processing speed. However, in the last few years, the previously steady increase in processor clock speed has slowed to a crawl as hardware manufacturers find it increasingly harder to further improve clock rate on new processors. The power required for high rate clocks and the heat generated by such power consumption are the main limiting factors for this growth [OH05] and, unless a major technological breakthrough is achieved, it is likely that this trend will continue.

Due to the difficulty of further increasing clock rates, hardware manufacturers are shifting their resources from improving individual processing cores to increasing the number of cores on a processor [Gee05], or even the number of processors in a system. This gave origin to the now ubiquitous multi-core processors. These processors currently can have anywhere from two to dozens of cores and some system architectures are up to nearly a thousand. This number continues to increase, as the amount of transistors that can fit in a CPU die keeps rising. The processing cores are largely independent and thus capable of executing programs in true parallel fashion unlike

single-core processors. While it is hard to compare single and multi-core processors directly (saying a dual core processor at the same speed as a single core is twice as fast is a very naive interpretation), individual applications can use multiple cores simultaneously for increased performance.

Unfortunately, while an increase in clock speed in a CPU directly benefits the performance of all programs running on it, the same does not hold for an increase in the number of cores. Unless software is written specifically with support for parallel computing, increasing the number of available cores will have little to no impact on its performance, as it is not likely it will take advantage of the extra available cores. For this reason, research in parallel computing techniques and associated programming language support has gained a renewed importance as the number of cores in the average CPU rises [HS08].

Creating software which takes advantage of high degrees of parallelism is not a trivial task. The next section describes some of the problems faced when building new parallel applications or even adding parallel computing support to existing programs.

1.2 Problem Description

The idea of adding parallel computing support to a sequential program is simple in theory. The first approach is to logically divide the program in independent segments or tasks. These tasks can then be assigned to different execution threads for processing. The various threads will then process the tasks concurrently and cooperate to achieve some common goal. This type of program structuring is usually called parallel programming and it has been used for many years to achieve high performance in the presence of multi-processor architectures, usually in highly specialized domains. In practice, this process can become quite complex and demanding even for an experienced programmer.

The performance gains from parallel programming are highly dependent on the type of software being written. For some programs, the so called embarrassingly parallel problems, the effort required to divide the problem into a number of parallel tasks is minimal. In this case, there is little or no synchronization needed among threads and purely sequential sections are small or non-existent. This is the ideal situation for parallel programming, often allowing for close to linear (and in some rare situations even above linear) speedups. Only a very small subset of the programs fit this category. In general, those conditions rarely hold and the maximum speedup possible is limited. With knowledge of the program's structure and sections that can be executed in parallel it is even possible to use Amdahl's Law [HM08] to infer the maximum speedup

achievable using parallel programming. This law states that the maximum speedup is limited by the time spent in purely sequential sections of the program. For example, imagine we have a program that takes 5 minutes to run in a single processor, but 1 minute of that time is data initialization which must be done sequentially. No matter how many processors we use, it is impossible to run this program in less time than the 1 minute initialization time, as that phase is purely sequential. Therefore, and assuming the remainder of the program can be fully parallelized, the maximum speedup achievable for this case is 5, regardless of the number of processors used.

The challenges in the use of parallel programming can be divided in two aspects: expressing the parallel program and handling the issues introduced by the parallelism itself.

One of the main reasons which contributes to the first challenge is that it is not always easy to identify which program sections can execute concurrently. This is an inherently difficult task due to the possible interactions between different program sections, shared data and so on. Additionally, the average programmer is simply not used to thinking about applications in a parallel fashion. Most programmers view a program as sequence of instructions, executed one after the other. Thus, the adoption of the parallel programming paradigm requires a shift in programmer reasoning. With several execution flows, it is hard or impossible to know exactly what is going to happen at any moment. Adding explicit parallel support to an already existing sequential program can be an even greater challenge, as it requires restructuring the entire program and might even lead to rewriting most of it.

Even after the parallel program is structured, with the use of concurrent flows of execution a whole new set of potential bugs is introduced which are unlikely to happen in sequential versions of a program – e.g. race conditions [NM92]. Multiple execution flows accessing the same resource (for example, the same memory region, file or devices) tend to lead to unexpected behaviours. Furthermore, a concurrent program is much more difficult to debug, for similar reasons as it is more challenging to build one in the first place. For example, errors caused by race conditions are notoriously hard to analyse, as these often depend on a specific interleaving of execution flows that is hard to reproduce in debug situations.

Several mechanisms have been proposed for concurrency control - e.g. locks, semaphores [Dij0 or monitors [Hoa74]. These mechanisms are usually not transparent to the programmer and quite complex to use. In fact, many common software bugs result from the misuse of these mechanisms such as the infamous deadlocks [CES71].

These problems make parallel programming notoriously complex and difficult to use in general programs. To address this problem, there is intense research to improve

currently used or find better abstractions for expressing parallel computations, both in structuring and concurrency control. One such abstraction is transactional memory [HCU+07]. Transactional memory aims to simplify the concurrency control problem brought by parallel execution. The programmer can easily guarantee correct access to shared memory sections by defining code blocks that access those memory regions as transactions. These transactions act similarly to the well known database transactions and guarantee isolation, atomicity and consistency. Transactional memory remains a work in progress and a great deal of research is being done in this field.

As for abstractions that help with program structuring, an example is the use of the Future construct. This abstraction allows a programmer to define a computation to be executed asynchronously. When the program requires the result of the operation this can be obtained with implicit (or explicit, depending on the language support) synchronization operations, which await for the computation to finish if it is not yet known. A different approach to exploit parallelism is the usage of replication [SS05]. This is used extensively in distributed systems for improving both performance and fault tolerance. This work aims to bring the usage of replication to multi-core systems, by allowing distinct cores to run different replicas of the same software component.

1.3 Proposed Solution

This work is executed in the context of the RepComp project. This project aims to study the use of replication techniques, often used in distributed systems with great success, to explore parallelism in multicore architectures. Specifically, the project focuses on a novel approach: the use of diverse replicated software components in multi-core systems. These replicated components, referred in the rest of this document as macro-components, can be seen as containers which hold a number of *functionally equivalent* replicas of a component specification. These replicas have distinct implementations and, therefore, different characteristics. The macro-component attempts to use the differences in these implementations to achieve its goal.

Two main directions for macro-components are identified at the moment: increasing performance and providing fault tolerance. The former is done by using *optimistic* replication and focusing on the performance differences between implementations of this component. Each operation will use the replica with the fastest implementation and, as such, obtain the best performance for every provided operation. The latter is done by using *pessimistic* replication and focusing on the differences in bugs between implementations. Different implementation likely have different bugs, by returning

the result of the majority, the macro-component can provide some degree of fault tolerance.

This work explores the viability of this novel approach while focusing mainly on using macro-components to transparently increase performance for both parallel and single-threaded applications.

A base runtime system was built to support general implementations of macro-components. This system is based on a central scheduler which coordinates the execution of operations by a pool of executor threads. Using this system, a prototype JDBC macro-component was built with positive results. Furthermore, an optimized version of the base runtime system was implemented, which improves the base system in the handling of fine-granularity operations.

1.4 Contributions

This dissertation presents several contributions to the RepComp project. First, the analysis of challenges connected to the basic execution model of the macro-components. Second, the design and implementation of the base macro-component runtime system and support tools. Third, the proposal of different approaches for the scheduling of operations in the implemented runtime system. Fourth, the implementation and evaluation of a complex in-memory database macro-component in both high and low stress situations. Last, the design, implementation and evaluation of an optimized design for low-granularity component support.

1.5 Outline of the Dissertation

The remainder of this document is organized as follows.

Chapter 2 discusses previous work related to our approach. Chapter 3 provides an insight on the basic conceptual model of a macro-component based execution system, as well as the challenges such a system must face. Following that, chapter 4 introduces the design for the runtime system which supports the macro components and a supporting source code generation tool, which greatly simplifies the creation of new macro-components. The implementation of the runtime system, prototype macro-components and supporting tools is described in chapter 5. Chapter 6 pretends an evaluation of the system. Finally, chapter 7 concludes with some final remarks and a discussion on possible directions for future work.

Related Work

This section deal with previous work related to the macro-components introduced in this dissertation. These works can be divided in three main categories, which cover different aspects of our solution: replication, n-version programming and parallel computing. Section 2.1 introduces several concepts related to replication in distributed systems, with emphasis on distributed database environments. Following that, section 2.2 discusses works in N-version programming, technique based on heterogeneous replication used to obtain a high degree of fault tolerance in critical programs. Finally, section 2.3 concludes this chapter with some general purpose parallel programming systems which present mechanisms or technical issues relevant to our work.

2.1 Replication

Replication is an extremely important topic in distributed systems [RS03, HHBB96, WPS+00], especially distributed databases [EZP05, EDP06a, EDZ07], and can be used in its various forms for a variety of purposes such as achieving greater reliability, fault tolerance or improving system performance. There are a multitude of systems that use replication in one way or another, but the initial approaches for each purpose are easy to grasp. The way the replicated resources (data or processes) are accessed depends on the purpose.

For reliability, resources are usually seen as backups. If the main replica is unreachable or damaged for any reason, the system is still available as new requests can be

processed by the remaining replicas. In the case of fault-tolerance, an the operations themselves can be replicated and executed by multiple replicas. The results obtained from the various replicas are then aggregated and compared to detect any incoherent results. Performance increases can achieved through replication through multiple approaches. The most common approach is to use replication to explore a large amount of available hardware resources.

2.1.1 Eager vs lazy replication

A complex task when designing any replicated system is maintaining *consistency* between the replicas. Depending on their consistency strategy, solutions can be classified as either eager (pessimistic) [BHG87] or lazy (optimistic) [SS05] replication.

In eager replication, the system must ensure that all replicas are consistent with each other and progress in the same way during the entire execution period. Eager consistency strategies usually rely on conflict *prevention* mechanisms to completely avoid inconsistent states during execution rather than resolve them.

An alternative to this strategy is the use of lazy consistency. Unlike eager consistency, replicas under lazy consistency are allowed to diverge during execution but are expected to converge at some point. Rather than conflict prevention, these systems employ conflict *resolution* to fix eventual conflicts between replicas, automatically or manually.

While eager consistency offers a stronger consistency model, it has severe drawbacks in terms of performance and scalability in relation to lazy models. Consequently, the consistency model chosen for an application must take into account the specific requirements of the application.

2.1.2 Operation distribution algorithms

Usually, when replication is used as a means to improve system performance, this is directly related to system throughput. In distributed systems, the typical technique is to fully replicate the system in several machines and distribute the system load between these replicas. This distribution of work can have several effects in different parameters of the system's performance. Consequently, the operation distribution or scheduling algorithm is a major cornerstone of any replicated distributed system, having a vital importance in its performance.

A number of techniques have been proposed for this distribution, some with very different goals, others with different ways to achieve the same goal. Below we introduce two systems which use replication and a load distribution algorithm to achieve

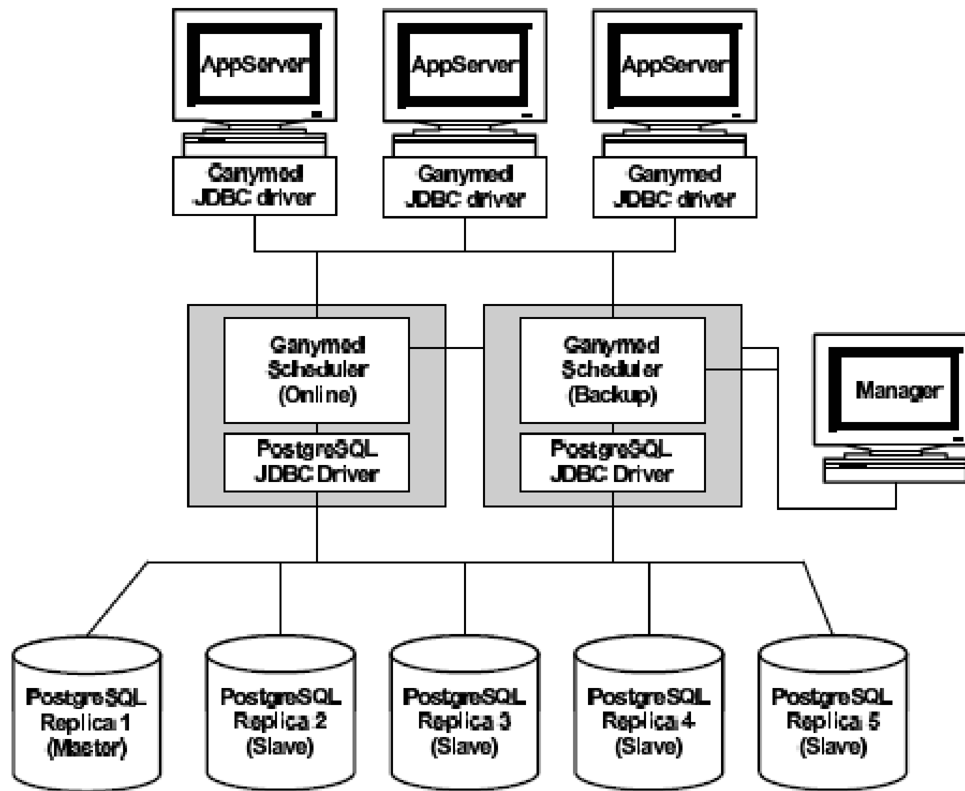


Figure 2.1: Ganymed system architecture. Figure taken from [PA04].

greater system throughput while exploring different resources.

2.1.2.1 Ganymed and the RSI-PC algorithm

Ganymed [PA04] is a database replication middleware-level platform that provides scalability without sacrificing consistency. Ganymed is based on primary-backup architecture and an operation distribution algorithm, in this case a transaction scheduler, named RSI-PC (Replicated Snapshot Isolation with Primary Copy. The Ganymed architecture can be seen in figure 2.1.

In Ganymed, replicas are divided in two categories: a single master replica and a number of slaves. Transactions are distributed among these replicas by the Ganymed Scheduler, using the RSI-PC algorithm. This RSI-PC scheduling algorithm is based on the separation between update and read-only transactions. Update transactions are always directed to the system's current master replica. Read-only transactions, on the other hand, are directed, whenever possible, to any of the remaining replicas, the slave replicas.

Updates are propagated by the master replica to the several slaves in the form of

ordered write-sets. RSI-PC requires the database replicas to support the serializable transaction isolation level for read-only transactions, guaranteeing they see the same database snapshot for the duration of the transaction. Update transactions can run in either serializable or read-committed mode.

The scheduler maintains a global version number record, which is incremented every time an update transaction commits on the master replica. This version number is used for multiple purposes. First, each write-set is tagged with the number of the transaction that produced it, effectively ordering the write-sets propagated to the replicas. When a slave replica received a write set it will first validate its global version number to assure it hasn't arrived out of order. With this, the slave replicas can guarantee all write-sets are applied by the same order as the master replica, and ensure data consistency.

Furthermore, transactions are also tagged with the global version number upon their arrival. When a read-only transaction is directed to a slave replica where the latest global version number is not available yet, it can wait until the pending write-sets are processed. If a client is unable to support a delay in the transaction, there are two choices: the queries from that client can be sent to the master replica, or a staleness threshold can be set. Sending read-only queries to the master replica is undesirable as it reduces the system's capacity for update transactions. A staleness threshold is a condition expressing an acceptable limit for how outdated the data is, expressed for example in seconds or versions.

2.1.2.2 Tashkent+ and the MALB algorithm

Tashkent+ [EDP06b] is another replicated database system with an interesting operation distribution approach. The Tashkent+ system introduces the Memory Aware Load Balancing load balance algorithm. This algorithm's main goal is to improve replicated database performance by minimizing I/O caused by transactions. This accomplished by using each transaction's working set information (memory usage) to dynamically group transactions types. These transaction types are then assigned them to a set of replicas in such a way that each replica can fit most, or ideally all, of the data required by its assigned transaction types in main memory. In this situation, transactions can run directly in main memory, avoiding costly disk I/O to swap accessed tables in and out of memory.

The Memory Aware Load Balancing (MALB) has several variants, depending on the depth of the information used to group transactions into types. The simplest variant, MALB-S uses only information about the working set's size (the amount of memory used). Transactions types are assigned by their memory usage to a replica using

the Best Fit Decreasing (BFD) bin packing algorithm, taking into account the maximum memory of each replica and the transaction types already assigned to it. As it does not account for intersection on each transaction type's working sets it is probable it will overestimate the size of working sets when multiple transactions types are assigned to the same replica.

MALB-SC is another variant which extends MALB-S by using both working set size and content. With this information, unlike MALB-S, MALB-SC can detect overlap between the working set of different transaction types and, thus, avoid overestimating working sets for multiple transaction types. Lastly, MALB-SCAP uses working set size, content and access pattern. Using the access pattern of a transaction type, this variant will assign transaction types to replicas by taking into only the most heavily used tables, which are likely to have to remain in memory. However, this method runs the risk of underestimating the working set size of a transaction type.

Additionally, update filtering is proposed as an optimization to reduce the overhead of system-wide updates. Using MALB, replicas will usually only deal with the working sets of their assigned transaction types. For this reason, replicas do not generally need to receive information about updates that have no impact on their working set. Nonetheless, some constraints exist to the filtering of updates, to guarantee that the system maintains a level of availability of replicas for each transaction type.

2.1.3 Diverse replication

In most replicated systems, replicas tend to be, as much as possible, homogeneous. Nonetheless, interesting proposals have been presented on the use of diverse (or heterogeneous) replication. In diverse replication, the replicas have small, but important, differences. Using those unique characteristics, the global system can achieve greater performance or a higher degree of fault tolerance than systems which use homogeneous replicas.

In [GPSS03] the author argues in favour of diverse replication applied to database server systems as a mean to improve both fault tolerance and system performance. While replication is already widely used in database systems, it is usually done using homogeneous replicas. To ensure an acceptable degree of fault-tolerance, this setup would only be enough on the assumption that the replicas have a fail-stop behaviour, ensuring easy detection of faults in a given replica. The results obtained from fault studies on several well known database systems show that this assumption is not valid. Also, benchmarks show a significant variance in performances between transaction types in different database systems. These findings support the idea of heterogeneous replication not only to tolerate non-self-evident fault, but also to increase performance,

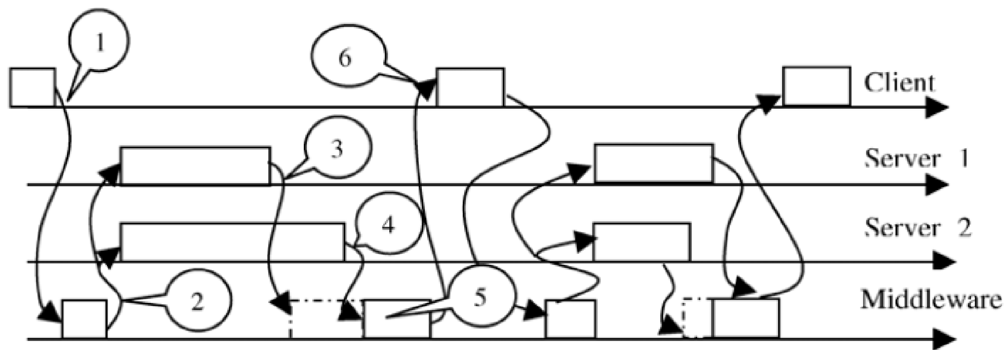


Figure 2.2: Pessimistic FT-node timing diagram. Figure taken from [GPSS03]

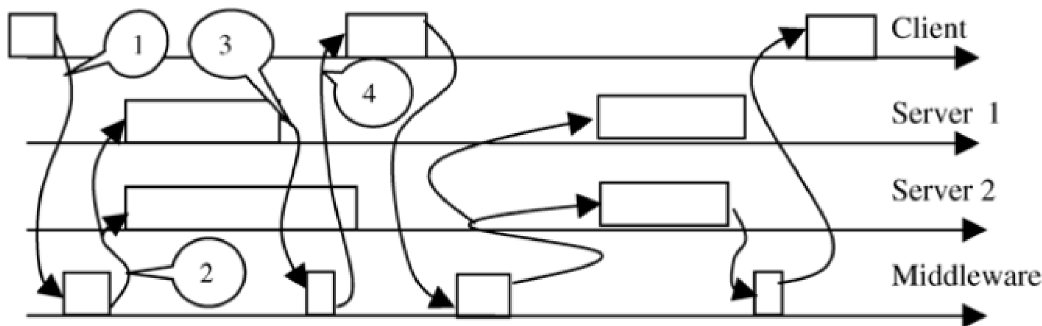


Figure 2.3: Optimistic FT-node timing diagram. Figure taken from [GPSS03]

by always using a replica with the best performance for a given operation.

The problem with using non-diverse replication for fault-tolerance in database systems lies in non-self-evident failures. Self-evident faults (crashes, exceptions or performance failures) are easily detectable without the need of design diversity. On the other hand, non self-evident faults consist of incorrect results without server exceptions within an acceptable time delay. These are usually caused by implementation bugs leading to deviations from the expected behaviour. Since using homogeneous replication, all replicas will generally behave in the same way, these errors can undetectable without diversity as they might be connected to the implementation itself. On the contrary diverse replication would, in most cases, detect these faults. It is not likely that different database implementations will suffer from the same fault. In the fault study done it is shown that more than half of the faults tested are non-self-evident and undetectable by the same-database system server pairs, while diverse pairs detect at least 94% of the faults investigated.

An architecture is proposed to create fault-tolerant server nodes (FT-nodes). These

are composed by two or more diverse servers connected to a middleware layer. Communication between the node and the servers is intermediated by the middleware layer which serves as a proxy between the clients and the replicas. These nodes can be used for both improving fault tolerance and performance, depending on the behaviour of the middleware layer. For fault tolerance, the behaviour would be pessimistic. In this case, the middleware must wait for the results from all the servers and check the consistency of the replies before returning the result to the client the clients. In Figure 2.2 we can see a timing diagram for this behaviour. In message 1 the client sends the request to the middleware, which forwards it to both servers with message 2. The servers process the request (with different processing times) and respond to the middleware. The middleware waits until both responses are received before adjudicating - in 5 - them and sending the result to the client in message 6, or if no consensus exists initiate recovery or signal a failure.

For performance, the behaviour would be optimistic. In this case the middleware immediately returns to the client the result returned by the fastest replica. In Figure 2.3 we show an example timing diagram of an FT-node running in optimistic mode. The middleware no longer waits for the response of server 2 and responds immediately to the client with with the result received in message 3.

2.2 N-version Programming

N-version programming is a particular case of the use of diverse replication to achieve fault-tolerance. This concept was introduced in [Avi85] as a fault tolerance approach to improve software reliability. Up to that point, the usual method of improving reliability was focused on fault avoidance rather than tolerance, although these techniques are not mutually exclusive [Avi75]. Software defects were eliminated prior to deployment with extensive testing. However, software components are too complex for that approach to be viable. Replication techniques such as the ones used on hardware were also unsuitable as, unlike hardware faults, software faults are often deterministic. As such, using replication with the same underlying implementation will only result on a failure in both replicas, failing to increase reliability. N-version program consists in the independent generation of N functionally equivalent programs, or versions, from a detailed common specification. These versions are generated by different teams working on the same objective with no contact between each other, using different programming languages, different algorithms, etc. The main idea behind this approach is that with a high degree of independence, the versions are highly unlikely to suffer from the same faults, although this must be used with care as it is still possible for multiple

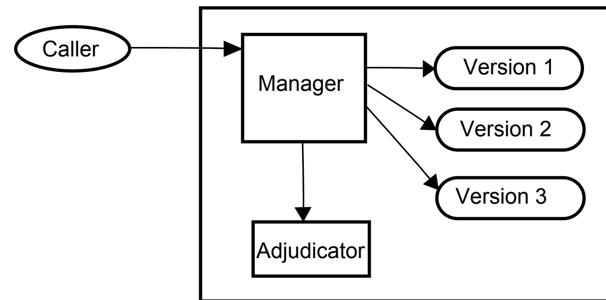


Figure 2.4: Structure of a diversely implemented object as proposed by [Rom00].

versions to present the same fault [KL86].

Together, multiple versions form an N-version unit, similar to the FT-nodes described in section 2.1.3. In this unit, the several replicas can run each operation separately and cross-check information, including internal state or operation results, to detect faulty executions. With N greater than 3, faulty executions can even be masked by using a correct execution selection algorithm based on a majority or threshold.

However, building an N-version unit can have heavy design and implementation requirements. The initial specification must be strong enough to define completely and unambiguously all the functional requirements of the program while leaving implementation choices as open as possible. Also specified should be the special features required for the N-version programming fault tolerance scheme such as cross-check points, format of the information to cross check and the selection algorithm.

2.2.1 Object Oriented N-Version Programming

In [Rom00], the authors address some of the difficulties in creating a structured faulty version recovery scheme for object oriented N-version programming based fault-tolerant systems. Unlike the previous work described, the focus is not on replicating an entire program, but a single object. For this, Diversely-implemented objects are introduced. These objects have a very similar functioning to the macro-components proposed in this dissertation, albeit with a different objective. As the scope of this replication scheme is much smaller than basic N-version programming, it is much simpler to implement and manage.

In the architectural scheme proposed, seen in figure 2.4, a diversely implemented (DI) object is separated into three parts: a manager, an adjudicator and the several class versions. When a method is called on a DI object, the manager component calls the method in the different class versions in parallel and waits for the all the results. Versions are assumed to be deterministic, and as such the method call results should

be equal unless one or more of the versions are faulty. These results are then passed on to the adjudicator which will compare them and return the correct output by majority or, if no majority exists, a failure exception.

Additionally, the authors address a number of principles considered vital for the development of a well-structured diverse replication scheme. First, structure. Applying software diversity should be coupled with the structured techniques used in system development. Second, recursion. If the diversity scheme can be used recursively, diversity can be applied to any system subcomponent no matter at what level. Third, diversity encapsulation. Diversity schemes in component implementation should be hidden from external components. Diversity control, and other details must be internal to the component which offers conventional application interface only. Fourth, version independence. Ideally, the scheme should support complete independence of version development. Meaning, it should avoid forcing versions to conform to some shared common factor outside of the component specification. And last, diversity control and support should be separated from the application code of the component itself.

2.2.2 Abstraction and opportunistic N-version programming

In [RCL01], a replication library named BASE (BFT with Abstract Specification Encapsulation) is presented. This technique extends on BFT [CL99], a replication library that provides Byzantine fault tolerance using homogeneous replication. The BASE library has one main improvement over BFT: the replicas are no longer required to be homogeneous or even deterministic. The methodology used is based on the concepts of abstract specification and abstraction functions.

Take a set of N implementations for a service, the number of implementations is, ideally, equal to the number of replicas but not necessarily so. Each implementation may have the same functionality but follows different specifications. Determinism and homogenization is achieved by defining a common abstract specification. This specification defines the abstract state, an initial state value and the behaviour of the operations on the replicas. The abstract state is built using the information accessible to the client, avoiding the need for knowledge of internal state details on the implementations.

The abstract specification is enforced not by the replicas themselves, but by conformance wrappers that must be developed on a per-replica basis. These conformance wrapper offer the abstract specification operations and map them into replica invocations, effectively acting as a "translation" layer. Conformance wraps also manage any additional data necessary to ensure determinism or correct specification semantics.

The main advantage of this methodology is that it allows for the reuse of existing

code. No changes are needed at the replica-level to use it in this replication scheme with previously existing implementation. There is only need to implement a conformance wrapper and abstract state conversion functions which should not require any special support from the replica implementation.

Consequently, it eliminates one of the greatest drawbacks of the N-version programming technique by allowing opportunistic N-version programming. Instead of having to manage and develop a number of different software versions, at a great cost, it is possible to simply use a number of off-the-shelf implementations. This is especially useful for areas where market competition brought a sizeable number of distinct commercial implementations with similar functionality.

The techniques used for the BASE system are theoretically applicable to every replicated service. However situations where the underlying replica implementations have very different or undocumented behaviours or overly narrow interfaces can be very burdening for the definition of the common abstract specification, abstract states and the implementation of the conformance wrapper.

2.3 Parallel Computing Models and Applications

Given the tendency for modern CPUs to support increasingly high degrees of parallelism, support for parallel execution is vital. The results of this research can usually be classified as either abstractions that help structure a parallel program or new concurrency control mechanisms. This section deals with previous work in this area.

Ideally, as discussed in section 1.1, a programmer should be able to write a parallel program or add parallel support to an already existing program with few or no changes to the code used for a sequential version. This goal is the reason why one of the main goals for these mechanisms, as important as the actual performance, is their simplicity and, whenever possible, transparency.

There are a number of *classic* concurrency control mechanisms which are well known to most programmers who have attempted to structure a parallel program. The common of these include locks, semaphores [Dij02] and monitors [Hoa74]. While these can be used to achieve excellent performance, this task is exceedingly complex and prone to error. This happens even when the model of the mechanism used is very simple, such as the lock model. In fact, many common software bugs are directly linked to the misuse of these mechanisms. As such, there is much research for alternative, more user-friendly, concurrency control mechanisms such as transactional memory.

2.3.1 Classic Concurrency Control

Typically, concurrency control is achieved by forcing process synchronization. If two or more processes attempt to access a shared resource that doesn't support concurrent access, one process is allowed to proceed while others must wait their turn. Code regions where processes these resources are commonly called critical sections. Faced with critical sections, most applications achieve mutual exclusion with concurrency protocols based on the use locks, monitors or semaphores [GR93].

In its primitive form, a lock represents something like an access token to a shared resource. When a process wants to access with resource it obtains the lock using an atomic operation[1]. Until that process releases the lock, no other process can obtain it (and generally blocks until it is released), therefore no other process can access the shared data while it is in use. Other types of locks exist with less restrictive policies however.

Semaphores [Dij02] can be seen as a generalization of locks. While a lock can only be held by a single process, semaphores can be seen as a counter. Processes decrease the value of the semaphore when they want to access the resource, and increase it when the resource is no longer in use. If a process were to bring the value of the semaphore's counter below 0, it will instead be blocked until another process increments it. Monitors [Hoa74] are used to obtain mutual exclusion at the method call level. Only one thread at a time can occupy a monitor, therefore methods executed through a monitor are guaranteed run in mutual exclusion.

2.3.1.1 Futures

The future (or promise) concept [LS88, BJH77] is a useful abstraction to build a parallel program while keeping a low impact on the overall program structure. The future is in it's essence a synchronization object which represents the value of a computation which is not yet known. This computation is usually being run in asynchronously to the main application thread, until the result from the computation is needed. When the main application thread requires the value of the computation, this can be accessed from the future object and is known as resolving or binding the future. If the computation has already finished, the future will simply return the result and the application can proceed as normal. Otherwise, the action taken depends on the semantics of the future, blocking or non-blocking. Futures provide an alternative method to add concurrent execution support, with the advantage over threads of simplifying synchronization for encapsulated operations.

[1] If the locking and release operations were not atomic, the lock itself would need concurrency control.

2.3.2 Transactional Memory

Transactional memory [HM93, ST95, HCU+07] is a programming abstraction which simplifies access to shared data by concurrent threads. This abstraction is based on the concept of memory transactions which provide semantics similar to database transactional systems[2]. The main advantage of transactional memory lies in its transparency to the programmer. All that is needed to ensure the correct concurrent execution of a code that accesses shared data is to encapsulate it in a transaction. This is usually done by explicitly signalling the zones which should be run under transactional memory support, by marking the start and end of the transaction with a special token. Other systems exist which provide implicit transactions. Within the transaction region, he transactional memory engine will guarantee that concurrent executing threads will observe a correct transaction semantic. To this end, several implementation techniques can be used and a large number of systems have been developed - see [HS08] for a survey.

2.3.2.1 Side effects actions

Transactional memory is a very active topic of research currently. An aspect that is being studied it how to handle side-effect actions. The problem is that these actions cannot be transparently undone by the transactional system. This poses several problems to the transactional memory support system when the transactions need to abort. Although several techniques exist to handle these situations none has proved far superior to the others. In [BZ07], the authors address this problem by studying the applicability of several compensation mechanisms. To this end, a study was conducted over the critical sections of two well known multi-threaded applications: Firefox and MySQL.

Side-effect actions can be divided in 3 types: protected, unprotected and real. Protected actions affect only CPU state and memory and can be completely compensated by TM systems. Unprotected actions cannot be compensated directly by the TM system, but the programmer can provide explicit support to compensate them. File system operations are an example of unprotected actions. Finally, real actions are those where there is no adequate compensation support - e.g. printing a document, network output, launching a missile.

Several options exist to address real or unprotected actions within transaction blocks. The simplest approach is to restrict them completely. However, this is not acceptable as it poses a severe limitation on the use of transactional memory for general programs.

[2] Database transactions usually provide the full set of ACID (atomicity, consistency, isolation and durability) properties. Transactional memory provides only atomicity, consistency and isolation.

A less restrictive solution is deferring the execution of the problematic actions until the transaction is sure to commit. This solution has two problems. First, it cannot be used when the transaction code has dependencies on the result of the real or unprotected actions. Second, changes the flow of execution of the program. The programmer expects the transaction actions to run in a sequential order and delaying problematic actions to commit time can cause unexpected effects. Another option is going non-speculative. With this, we must guarantee the transaction that contains real/unprotected actions will not abort. This can be achieved by implicitly using a global lock when such an action is detected, but it limits concurrency between transactions and therefore has an adverse effect on system performance. Furthermore, if a transaction uses this mechanism, it cannot use explicit transaction abort operations as it has no way to revert the effects of the real/unprotected actions. The last method is compensation. The programmer can associate a compensation block to the unprotected code. This block is executed if the transaction aborts and must deal with the effects of the unprotected code. However, not all actions can be rolled back using a compensation code. The authors group actions in 4 categories considering the reaction the transactional memory system must take to compensate their effects.

Null compensation, for actions which do not require any compensation code. Memory-fixup, where only the kernel state is altered and is easily fixed with a compensation block. Full compensation, where unprotected actions require a more complex compensation block, or the transaction to go non-speculative. And finally, real actions. These last cannot be compensated at the syscall level and require external mechanisms such as I/O buffers or compensation at a higher abstraction level (library or application level). An important detail is that except for real actions, which make up a very small percentage of the syscalls present in the critical regions observed, all syscall categories can be handled at syscall level. As such, if the compensation code is implemented by the system libraries, it is possible to handle the bulk of syscalls in transactions in a completely transparent way for the application developer.

2.3.3 Speculative Execution

In concurrent or distributed environments, many times processes need to perform operations that require synchronous requests. Generally, the caller blocks until the result of the operation is received. If the response takes long time, either due to communication latency or for heavy processing operation, there is a great loss of performance due to the time spent blocked. Speculative execution addresses this issue. Instead of blocking, the process predicts the result of the request and continues execution in speculative mode. When the response arrives, the system must verify if the speculation made

was correct. If so, the process can continue execution in normal mode. Otherwise, the process must rollback to the point where the request was made (or even earlier) and continue execution from there with the result received. Using speculative execution, programs which must block for a large amount of time awaiting results which can be predicted with a fair amount of certainty can drastically increase their performance.

Speculative execution brings two points of interest to the research on macro-components. First, just like transactional memory, it faces similar technical issues at the level of problematic operations during execution – e.g. I/O. Second, speculative execution can become a linking bridge between the two directions for macro components, performance and fault tolerance. An application could begin speculative execution upon receiving the first result from a macro-component replica and resolve the speculation when enough results are received to ensure fault tolerance. However, as a first approach to macro-components oriented at increasing performance, this falls out of the scope of this dissertation.

Below are presented some previous works in speculative execution system which shed some light on technical challenges faced and mechanisms that may be applied to macro-components (further discussion in section 3.2).

2.3.3.1 Speculator

In [NCF05] the authors use speculation to improve efficiency in a distributed file system while maintaining strict consistency. To make speculative execution possible, Speculator was created. Speculator adds support for speculative execution at the operating system level. When a process initiates speculative execution, Speculator creates a checkpoint for that process. The checkpoint comprises a copy-on- write fork of the process, which is not placed in the execution queue, along with all other external states the process execution depends, on such as pending signals and file descriptors. If all of the speculations the process depends on are succeed this information is simply discarded. If one of the speculations failed, the running process is terminated and the checkpoint child process will take its identity.

There are two invariants that must hold for the speculative execution to be considered correct: speculative states should never be visible to the user or an external device, and no process should view speculative state unless it is already speculatively dependent upon that state. The first invariant means that speculative processes cannot output to the screen, network, write on files, etc. The initial approach is to just block those processes upon a syscall that does any of these. However, that limits the amount of work that can be done speculatively, as those operations are quite common. Instead,

Speculator uses kernel level output buffers. The kernel intercepts output of speculative processes and buffers it until the speculations are resolved. For the second one, the process that attempts to view speculative state can, again, be blocked or, alternatively, become speculative itself and depend on the same speculation that created the speculative state.

2.3.3.2 Speck

Security checks are an important tool to secure a system against attacks or intrusions but the impact of powerful security checks in the system's performance could be drastic. This impact can be reduced using lighter security checks but at the cost of system safety. In [NPCF08] the authors propose a system which takes advantage speculative execution in multi-core systems in order to achieve better performance for security checks without compromising safety. Speck (Speculative Parallel Check) decouples security checks from an application and allows it to continue execution using operating system support for speculative execution. As the application continues its execution, later checks can be identified and done in parallel with earlier ones.

Speck uses the Speculator system, described in the previous section, to support speculative execution at the operating system level. When a security check is called by a process, Speck can fork the caller and allows the initial process to continue its execution speculatively, while security checks are done on the forked process which completely replays the execution of the original. To minimize the overhead caused by forking a process for each security check, these are instead aggregated, in epochs, allowing more than one security check executed per fork.

As the clone process must replay the parent's execution exactly, sources of non-determinism are a problem. For example in the case of I/O, by the time the copied process reads the same file as its parent, the file could have already been changed. Similar sources of non-determinism may disrupt proper macro-component functionality.

In the environment used by the developers (single-thread processes and Linux operating system) there are two supported sources of non-determinism to handle: system calls and signal delivery. Processes that execute non-deterministic functions that bypass the operating system are not currently supported.

Both system calls and signal delivery are handled using a transparent kernel-level replay system. For system calls, the replay system logs the result of any system call executed by the original process. When a clone executes that system call, the replay system intercepts the call and delivers the result logged for the parent. As for signal delivering, the replay system guaranteed copies of the parent's received signals are

replayed to the clone process at the same execution point. This is done by keeping a signal replay queue and delivering the queued signals at the proper moment of the clone's execution.

3

Model

This chapter introduces the basic ideas behind our solution. Section 3.1 explains the reasoning which led to the macro-component approach and some preliminary thoughts and examples. Section 3.2 ends this chapter by identifying some of the challenges to implement the macro-component model.

3.1 Macro-component basis

Applications usually use a set of components (e.g. data structures, algorithms, etc.), each one implementing a specific functionality. It is common that a single component specification can have multiple implementations. While their external interface is homogeneous these implementations likely have several internal differences that lead to differences in performance. Typically, there is no single best implementation: different implementations perform best in different conditions.

In this work, we introduce the concept of macro-component, a software component that combines several different implementations of a given component specification. When under the scope of a macro-component, these implementations are called replicas. The macro-component itself implements the same specification as the replicas and can be used in an application transparently in the same way as any other implementation. This basic macro-component concept can be used for two different purposes.

First, it can be used for providing improved reliability, by masking software bugs (in a way similar to N-version programming [Avi85], but for each macro-component

Figure 3.1: Macro-component operation with two underlying replicas example.

individually). For example, to mask a fault in one implementation which caused it to output an incorrect result, three replicas could be used. In this case, the faulty replica would return an unexpected result, but the two others would return the correct result, and this result would be taken as correct by majority.

Second, a macro-component can be used for providing improved performance. In this case, an operation can be executed in all replicas in parallel. The first result obtained from any replica is the result immediately returned by the macro-component - e.g. figure 3.1 illustrates this approach by presenting the execution timeline on a macro-component with two underlying replicas, where the macro-component immediately returns the result of the first replica. Thus, the macro-component could have the best performance for every operation. In this work, we focus on using macro-components for achieving improved performance, although the design proposed could serve as the basis for implementing fault-tolerant macro-components. Next, we present an example of a performance oriented data-structure macro-component and discuss the challenges identified to implement macro-components efficiently.

3.1.1 Example: Optimal Complexity Set

The use of data collections is pervasive in software applications. Collections are used to group multiple data elements, providing methods to store, access and manipulate these data elements. Modern languages provide collections frameworks that include generic interfaces (e.g. set, list) and concrete implementations of these interfaces (e.g. HashSet, TreeSet in Java collections framework).

Each concrete implementation uses a different data structure, for which it is known the complexity of the operations. When a programmer wants to use a collection that implements a given interface, e.g. a set, she must select the implementation that better suits the application's access pattern. For example, if an application only inserts, removes and checks if elements are in the set, an hash table based implementation provides optimal performance. However, if the elements in the set are related according to some total order and the application often needs to access the element in some given rank (or iterate over all elements), a tree based implementation may be preferable.

Thus, the best choice will depend on the access pattern of the application. If the access pattern is unknown or it changes over the running time of the application, no optimal choice is immediately discernible.

By creating a macro-component that includes both implementations as its replicas, it is possible to create an optimal set implementation. Such implementation provides the best performance for all operations, by relying on the result returned by the fastest replica in each operation.

3.2 Challenges

The use of replicated software components has already been proposed in n-version programming [Avi85], as a way to deal with software bugs. At the time, with single-core systems, it would be hard to attempt to use such replicated designs to obtain improved performance, due to a number of factors. First, not being capable of parallel execution, these systems would have to predict and run operations on the best replica only. But accurately choosing the replica where an operation will run fastest might not always be possible. Furthermore, and most importantly, operations that change the internal state of a replica, would necessarily have to be executed in every replica or the internal state of the replicas would diverge. Generally, the overhead imposed by these repeated operations would cancel any gains from the optimized operations unless the mix of operations was very favourable.

Currently, with systems that include several processing cores and large amounts of memory, the use of replicated software components for improving application performance can be pursued. However, the design of a system to support the existence of multiple macro-components inside an application poses several technical challenges that are far from trivial. Next, some of these challenges are discussed, even if only some of them have been address in the proposed design.

3.2.1 Overhead

3.2.1.1 Time

One of the most problematic issues is the execution time overhead introduced by the code of the macro-component itself. Figures 3.2 and 3.3 present, respectively, the execution time for different producer-consumer synchronization options available in Java and insert operations in different data structures implementations. Producer-consumer synchronization options are especially relevant as these are a basic building

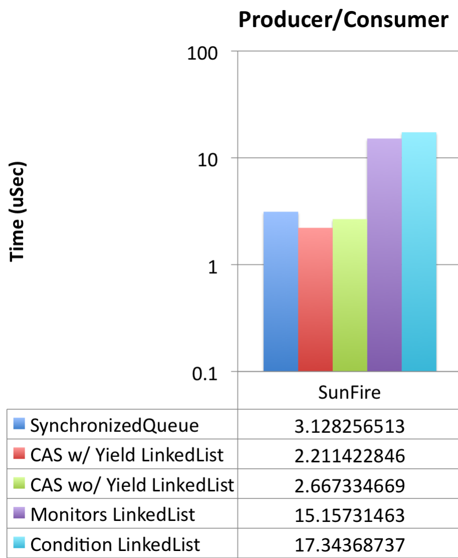


Figure 3.2: Producer-consumer model overhead estimation.

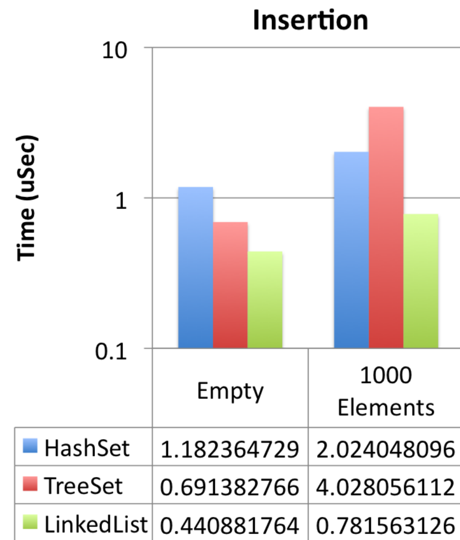


Figure 3.3: Values for an insertion operation in various Collection implementations.

block for coordinating the execution of an operation in multiple replicas by multiple threads.

The results show that, some operations, such as insertions in LinkedLists have an extremely fast running time when compared to the minimum overhead for synchronization among multiple threads. Therefore, if the macro-component design includes, for example, thread synchronization in the form of a producer-consumer scheme, the performance for this type of operations will suffer considerably. On the other hand, the overhead introduced is largely negligible when considering coarser granularity operations.

Also, this counts only the minimum required overhead for synchronized execution. Adding other mechanisms to prevent or resolve some of the challenges found is likely to introduce new sources of overhead to the system. As such, it is important that all mechanisms introduced cause the least overhead possible.

While the base runtime system design provides no special measures to handle fine-grained operations, an optimized version of the base system was implemented to address these operations. Sections 4.4 and 5.4 describe these these optimizations.

3.2.1.2 Memory

A different type of overhead to consider is the memory overhead of a macro-component. Implementations will generally be completely independent, and thus share no data.

This means a macro-component can increase the use of physical memory over a simple component by a factor that is linear to the number of implementations it contains.

While the system can maintain all applications in main memory, the consequences are not dramatic. However, if the memory used by the macro-component grows large enough that it cannot be contained in main memory, the operating system must swap it from main memory to disk and vice-versa on demand. The time for this swap will only exacerbate the execution time overhead already introduced by the macro-component. However, modern machines have large amounts of available physical main memory, and this growth shows no sign of stopping. As such, in most situations, we expect macro-components to fit completely in main memory, and the memory will not be an issue.

3.2.1.3 Dynamic Replica Allocation

Dynamic replica allocation is a technique envisioned to help minimize the overhead caused by macro-components. Instead of a macro-component starting with every available implementation, a single implementation could be used in the starting point when the existence of multiple replicas would not be beneficial. For example, with a very small number of elements, any implementation of a *Set* data structure takes a similar time to execute any operation. Later, new implementations could be added when deemed necessary - e.g. when the number of elements in the data structure increases. This could reduce the memory overhead of macro-components and even help with reducing the execution overhead.

Nevertheless, this approach poses some technical challenges in itself. First, there is the issue of knowing exactly when a new replica should be created. This decision should be based on the workload and state of the macro-component. Considering the example of the optimal complexity set presented in section 3.1.1, the macro-component could start with only a tree set implementation while the workload is favourable and the number of elements is small. When the workload changes or the number of elements grows enough to justify it, the macro component could create an hash set based replica to support this workload. Furthermore, in some situations, it can be advantageous to create a new replica with an implementation already in use on the macro-component. This second replica could allow a greater throughput by enabling the parallel execution of read operations even if the implementation for that replica do not allow for such parallelism.

When a new replica is created, it must be brought up to date with the macro-component. For replicas based on an implementation that is already present and provide a clone method, the process of allocating a new replica is simplified. The macro-component can simply clone one of its replicas and use the clone. However, even in this case the problem remains that during the cloning process this replica would be unavailable. Moreover, if the cloning process takes a long time, it is likely that when it has finished, both the existing replica and the new one will already be outdated due to operations on the macro-component during the cloning process.

One possible solution is to use a solution similar to the *state transfer* [Rom00, CL02] mechanism used in replicated systems.

3.2.2 Performance Differences

Although the macro-component attempts to benefit from the performance differences between implementations, this difference can also lead to problems. When all operations are executed in every replica, extreme performance differences will pose a challenges to the implementation of macro-components. For example, when the mix of operations on a macro-component leans heavily towards a particular operation that is significantly faster in one of the implementations, and all available replicas are required to process every operation, it is possible for the slower replicas to have trouble catching up with the fastest.

The implementation must take this into consideration when deciding where to execute new operations, so that an operation is not scheduled to a replica that will take too long to be up-to-date.

Consider the example of the optimal complexity set described earlier: an ordered listing operation will run much faster on the tree set operation, while an inclusion check will have a better performance on a hash map implementation. If an application requests an ordered listing followed immediately by an inclusion check, the hash map based replica can be kept busy by the first operation and unable to process the second immediately (assuming this replica has no concurrent execution capabilities). In this case, the macro-component will fail to reach the best performance for all operations. Instead, the inclusion check will present the minimum running time between the tree set implementation and the hash map with the added delay of finishing the ordered listing. Neither of those are optimal.

Several approaches can be used to deal with this problem, such as skipping operations which are not required on late replicas. In extreme cases, the abstract state transfer mechanism mentioned in the previous section could be used to update a replica's state without applying every pending operation. This problem and possible solutions

are further discussed in section 4.3.

3.2.3 Scheduling

A macro-component can be composed of any number of different replicas. When an application calls for an operation on the macro-component, this can be executed in any of these replicas. A possible approach is to simply execute the operation in every replica, returning the result of the first replica to finish. However, this is not the only possibility. For example, some operations could be executed in a single replica instead. This decision is tied to a scheduling strategy which decides which replicas will process a given operation or operation type. This problem is related to the scheduling of operations in distributed systems, as discussed in section 2.1

Besides discussing where each operation must execute, there is the problem of synchronizing the different operation executions to maintain replica consistency. Section 4.3 in the next chapter will directly address these scheduling problems.

3.2.4 Implementation characteristics

While the final goal is to allow any implementation to serve as a macro-component replica, some implementation properties bring new challenges to the system. At least three of these characteristics were identified: non determinism, non self-containment and general software faults. This section presents an overview of these characteristics and their effects on the macro-component environment.

3.2.4.1 Non-Determinism

The macro-component model assumes the replicas to be functionally equivalent. It is generally assumed that, while the performance of differently implemented replicas may be different, the results obtained from executing an operation on any replica will be the same. When implementations include non-deterministic operations, such as pseudo-random number generation or timestamps, there is no guarantee that an operation's result will be the same on every replica or even that the replicas will evolve to the same states. As an example, we could define an operation to insert a randomly generated number on the optimal complexity set presented in section 3.1.1. If this number was generated inside the implementation itself ,as each implementation would generate a number independently, these would likely differ. Consequently, the state of each replica would start to diverge, and the macro-component's execution would be compromised.

This problem was already addressed by N-version programming solutions and other parallel execution systems [NPCF08, Avi85]. If non-determinism cannot be avoided, some mechanisms must be implemented to support these operations. Simple non-deterministic operations, for example, can be supported with the use of a replay mechanism similar to the one used in [NPCF08]. In this case, the result of these operations on the first replica is replayed on every subsequent replica. On the example used above, this replay system would result in the same random number being obtained for both replicas, avoiding the divergent state problem. In addition, there are replicated systems which, instead of enforcing replica determinism directly at the replica level, use replica wrapping mechanisms to support some levels of non-determinism in their components [RCL01]. These wrappers would form a layer between the macro-component and its various replicas which attempts to force deterministic behaviour on otherwise non-deterministic operations. In the previous example, the non-deterministic operation to insert a number would generate the number in the wrapper and execute an add with this value.

3.2.4.2 Interactions

Non self-contained replicas can interact with the outside environment through other means than operation results. Examples of this are implementations which do input or output - e.g. to the network, screen or files. This problem is similar to the one faced by speculative execution [NCF05, NPCF08] and transactional memory systems [BZ07]. As in these systems, macro-components could either severely limit interactions with the outside in their replicas or include mechanisms to support these operations. One of such mechanism is an output buffer [NCF05, NPCF08] which captures all output attempts on a per-operation basis and, upon the operation's end, let through only the output of the fastest replica, discarding the rest.

3.2.4.3 Faults

Software implementations are typically far from perfect. Due to programming errors, design flaws, or other situations, software faults are a common occurrence in any system. These faults often lead to failure situations: a component can deliver an unexpected result, fail to respond or even corrupt itself or other parts of the system. The consequences of replica failures in macro-component replicas are similar to those of non-deterministic replicas. They can lead to incorrect results for operations or cause the replicas' states to diverge.

There is intense research in fault detection, tolerance and isolation [KAD+07, AEMGG+05,

CL99, RCL01, VBLM] for both centralized and distributed systems. Some of the proposed techniques, could be applied to the macro-component system. An interesting detail is that the macro-component itself can be used as a fault detection and recovery mechanism by applying the same concepts of N-version programming [Avi85] . As a macro-component contains several replicas, Byzantine fault tolerance techniques can be used to provide fault tolerance – the result from the majority of replicas is taken as the correct result, and the faulty replica can, for example, be flagged as corrupt and be recovered.

4

Design

In this chapter we describe the architecture and runtime used to support the macro-component concept. This design intends to allow further study of viability, performance and the identification of challenges that are faced for building the macro-component solution. The remainder of this section is structured as follows. Section 4.1 discusses the assumptions made in our design. Section 4.2 presents the proposed architecture and details each component. Section 4.3 concludes this chapter with some considerations about scheduling strategies and rules.

4.1 Assumptions

This work is part of a larger project and intends to serve as an initial step into the study of the technical challenges and applicability of the macro-component model. Thus, it was important to create a full system that allowed to understand the aspects involved in the support of the macro-component model. For this reason, the proposed solution tries to be as simple as possible, and makes a number of assumptions to keep the design simple. In particular, the following assumptions are made.

Replicas are deterministic: Considering the issue of determinism, discussed previously in section 3.2.4.1, all implementations of a specification are assumed to be functionally equivalent and deterministic. Thus, for any sequence of operations executed in a replica, the results obtained by executing the same sequence of operations in another replica from the same initial state will be the same. This assumption guarantees

the following properties. First, a macro-component comprising replicas with different implementations will obtain the same result when executing an operation on any of its underlying replicas. Furthermore, the internal states of these implementations will not diverge when executing the same sequence of operations.

Replicas do not fail: determinism is not enough to fully guarantee these properties in face of arbitrary failures as discussed in section 3.2.4.3. These failures can cause the macro-component to obtain conflicting results from different replicas, or even cause these replicas to diverge by corrupting their internal state. Consequently, for the purposes of this work, it is assumed the replicas cannot fail.

Replicas are self-contained: The problem of non self-contained implementations, discussed in section 3.2.4.2, is also not addressed. Instead, implementations are assumed to be self-contained. As a result, we avoid including any mechanisms to handle the execution of I/O operations, calls to other components, etc. Nonetheless, some degree of non self-containment can be supported by the system. If relationships exist among macro-components, the global scheduler can support these by enforcing an order between operations from these components. An example of this situation is found in our prototype and is discussed further in section 4.3.2.

Methods are not thread safe: and, as such, it is not possible for a single replica to process multiple operations concurrently. This has a direct impact on scheduling strategies but it is necessary to ensure simplicity in the design and in the implementations used as replicas.

Memory resources: it is assumed there is enough memory available in the system to maintain all replicas in main memory, avoiding the extra overhead introduced by paging. This assumption is realistic in most systems considering that modern computers tend to have vast amounts of physical memory.

4.2 Proposed Architecture

In figure 4.1, we present the proposed macro-component runtime architecture. The architecture can be divided in two main components: the global scheduler and the individual macro-components.

The macro-components are composed by a manager, which provides the interface of the component; a set of replicas, that implement the functionality of the macro-component; and an internal scheduler, responsible for deciding in which replica each operation must execute. This design only requires the replicas to implement the same specification, making them functionally equivalent and guaranteeing their interface is homogeneous.

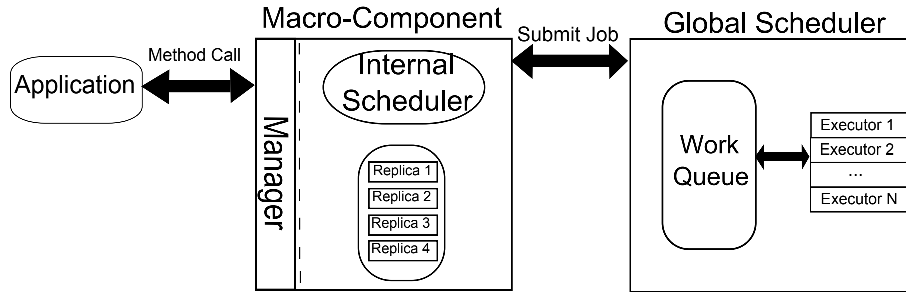


Figure 4.1: Macro-component system model.

The global scheduler is responsible for managing the execution of operations for all macro-components in the application. To this end, the global scheduler keeps track of the several execution jobs, representing the execution of one operation in a specific replica, in a work queue and schedules them for execution by a pool of *executor* threads in a producer-consumer scheme. More details on the internal global schedulers can be found in section 4.3.

When the application calls an operation on the macro-component, the manager forwards it to the internal scheduler. The internal scheduler creates one or more *jobs* for the operation which are then submitted to the global scheduler. If the operation can be executed asynchronously (i.e., it returns no result and can never fail), the application thread can immediately continue execution, otherwise it must block until a result is available. Operations submitted to the global scheduler are kept in a work queue until they are handed for execution to one of the executor threads. The executors will process the operation, set the result and signal the global scheduler the current job is finished. If the application thread is blocked, waiting for the result, it will be unblocked by the global scheduler.

4.2.1 Operations

Before detailing the proposed macro-component runtime system, this section discusses the alternative semantics of macro-component operations, and its impact in the execution model of our system. Operations can be generally classified based on: granularity, synchronicity and their effect on the replica state (read-only vs update).

4.2.1.1 Granularity

For our purpose, the granularity of an operation is related to the ratio between execution time of an operation and the time overhead introduced by the macro-component runtime. Coarse granularity operations are those with a long running time compared

to the overhead. For these operations, the overhead introduced will be negligible, and the macro-component performance will not be affected. Fine grained operations are those with a small execution time. For these operations, the overhead introduced by the system can have a deep impact on the performance of the macro-component when compared to the performance of a simple implementation. Thus, it is important to minimize the overhead for these operations.

Is it important to note that it is difficult to accurately estimate an operation's granularity. In fact, many operations have a variable execution time, and therefore variable granularity, depending on the parameters or size of the internal state of the component - e.g. search operations in data structures or database queries.

4.2.1.2 Read vs Update

The distinction between a read and an update operations is important for the macro-component system. Update operations are simply defined as operations that can affect the result of subsequent operations on the replica they are executed on, such as inserting or removing an element in a data structure. Read operations do not affect the result of following operations. Inclusion checks on a set data structure, for example, are read operations under this rule.

Note that it is possible for a read operation to change a replica's internal state without changing the result of following operations. For instance, in a search tree data structure with promotions on read, reading an element will bring this element to the top of the tree. This will make subsequent reads of that particular element more efficient in this data structure, but the result of operations will be unaffected. In our model, these operations are still classified as read operations.

Our system introduces support for handling read and update operations in a different way, by allowing the internal scheduler to decide in which replica an operation needs to be executed based on this distinction.

4.2.1.3 Synchronicity

Operations can be classified as synchronous or asynchronous. In a synchronous operation, when the application thread invokes the operation on the macro-component, the thread will be blocked until the execution completes (and usually, a result can be obtained). Therefore, the thread will remain blocked up to the moment the operation is executed in at least one replica. In an asynchronous operation the application does not need to block until the end of its execution. While it is possible to use asynchronous execution for operations with a result by fetching the result at a later time, it can be more commonly used for operations with no result.

The support for asynchronous operations can allow the macro-components to execute in a similar fashion to the futures model [ZKN07]. The application submits an operation to the macro-component, which will process this operation in background. This allows the application to continue as normal and fetch the result, if any, implicitly or explicitly, only when required.

In our system, we only support asynchronous operations that return no result.

4.2.2 Macro-Component

In this subsection we detail the internal components of a macro-component.

4.2.2.1 Replicas

Replicas are the implementations of the specification of the macro-component. When an operation is invoked on the macro-component, it must be finally executed in the replicas. The execution model can vary for different macro components. The macro-components can execute a given operation in one, some or all replicas. For improving performance, it is expected that the performance of different replicas differ and that no single replica is the best for all operations.

4.2.2.2 Manager

The manager provides the interface of the macro-component. This interface should be the same as the one provided by the macro-component's underlying replicas. When an application calls an operation on the manager, it will first submit the operation to the internal scheduler and obtain a number of *execution jobs* for that operation. These execution jobs can then be submitted to the global scheduler. The manager can submit the jobs to the global scheduler either synchronously, which will block the application thread until a result is obtained, or asynchronously, which allows the application thread to continue executing, depending on the semantics of the operation.

4.2.2.3 Internal Scheduler

The internal scheduler decides *which* replicas an operation will be executed on. For each operation, the internal scheduler creates one or more *execution jobs*, which are then submitted to the global scheduler. The jobs created by the internal scheduler can contain different information but, at a bare minimum, they must contain an operation and the replica it will be executed on. Different scheduling approaches can be used for internal scheduling. Section 4.3.1 details these strategies.

4.2.3 Global Scheduler

The global scheduler decides the order of execution of pending jobs. The global scheduler is composed of two components: the executors and the work queue for jobs. The global scheduler can be seen as structured according to a simple producer/consumer scheme. The macro-components submit jobs, which are stored in the work queue. The executors continuously process these jobs. Results of these jobs are then returned to the macro-components by the global scheduler.

Several scheduling strategies can be implemented as discussed in section 4.3.2.

4.2.3.1 Executors

The global scheduler's executors are simple worker threads. These threads execute a simple loop: request a job from the global scheduler, process the job and set the result and signal the global scheduler. When no pending jobs are available these threads block on the job request operation until a new job is submitted to the scheduler.

These executors are created with the global scheduler, implementing a thread-pool pattern. While it was possible to create and destroy executors during runtime and therefore adjust their number, we decided to maintain a constant number of worker threads. This model avoids the additional overhead of creating new threads during runtime. In the future, we plan to explore maintaining a variable number of executors depending on the current system load.

4.2.3.2 Work Queue

In the presence of asynchronous operations or multiple consecutive calls to macro-components, it is probable that the working queue grows to store several jobs awaiting execution. The global scheduler's work queue stores the jobs until there is an executor free to execute them.

Choosing an adequate data structure for this queue can also simplify the implementation of the scheduling strategy for global scheduling. For example, the queue can be implemented as a linked list if the global scheduler follows a simple FIFO scheduling strategy.

In the case of our runtime system implementation, the work queue is divided into several FIFO queues representing different logical replica groups. This grouping of execution jobs allows the scheduler to maintain consistency in the face of operations with dependencies between different macro-components. This is further explained in chapter 5.

4.2.4 Supporting tools

The design presented in the previous section allows for any programmer with knowledge of the global scheduler's interface to implement their own custom-made macro-components. However, for simplifying the use of macro-components, the system should include suppoer tools to help the programmer in creating the code of the macro-components, including all the internal components – e.g. the manager and the internal scheduler.

We have implemented a simple version of this tool. This version is a pre-processor that, given the interface of a macro-component, creates the template of the macro-component including the classes for the manager, the internal scheduler and other auxiliary classes such as those to encode jobs. The programmer can then complete the code by selecting the implementations used for the replicas. Additionally, the programmer can tweak the code of the internal scheduler to implement an optimized solution. The implementation details of this code generation tool are presented in section 5.3

The current implementation of the code generation tool only supports the creation of macro-components which use the read-all internal scheduling strategy. However, an extension to this tool could use annotated interfaces to give the code generator several hints about operation properties which cannot be expressed through normal Java interfaces. An example of a possible solution is presented in figure 4.2

```
public interface TestClass {

    @update (
        granularity = update.Granularity.COARSE
    )
    public void methodA(String arg0, boolean arg1);

    @read (
        bestReplica = "example.TestClassImpl1",
        granularity = read.Granularity.COARSE
    )
    public int methodB(int arg0) throws Exception;

}
```

```
public @interface read {

  public enum Granularity { FINE, COARSE }

   String bestReplica();
   Granularity granularity();
   ...
}

public @interface update {

    public enum Granularity { FINE, COARSE }

   Granularity granularity();
   ...
}
```

Figure 4.2: Annotated interface with code-generation hints.

In this example, additional information about the methodA and methodB operations is encoded using the @update and @read annotations. Furthermore, these can also hold additional information such as the best replica for read operations or operation granularity in both cases.

4.3 Scheduling

The scheduling of operations in the macro-component runtime can be divided in two parts in the proposed design: internal scheduling, done by the internal scheduler of each individual macro-component; and global scheduling at the global scheduler level. These are different processes with different objectives. In this section the scheduling models for both internal and global scheduling are described, as well as some considerations on limitations and rules the scheduling strategies must follow.

4.3.1 Internal Scheduling

Internal scheduling consists on deciding in which replica an operation must execute. As a result, a number of *execution jobs* is created. Each macro component contains an individual scheduler which creates these execution jobs based on the semantics of the component type and the operation to be executed. An execution job, in its basic form, can be as simple as the reference to a replica, and the operation to be executed in that replica. However, extra information can be added to a job, such as operation characteristics (synchronous or asynchronous).

This information can then be used to enable complex global scheduling strategies.

The main concern in this version of the system is improved performance. With the parallel execution mentioned before, the macro-components take advantage of the difference in performance between replicas. However, it is possible to add another technique for increased performance, which is widely used in distributed systems [SKSS02] or distributed databases [PA04, EDP06b]: distribute the read operations between the existing macro-component replicas.

The basis for this approach is that read operations, unlike updates, do not modify the results of subsequent operations or the state of the replica itself. As such, it is not required that the macro-component execute these operations in every replica. Also, it is not necessary to execute these operations in any particular order, as long as they are executed in a state that reflects ecectly all previous update operations and no more.

Considering the distribution of read operations for the replicas, three internal scheduling strategies for macro-components are envisioned: read-all, read-one and read-many. The rest of this section will present the three strategies in detail and discuss each strategy's advantages and disadvantages.

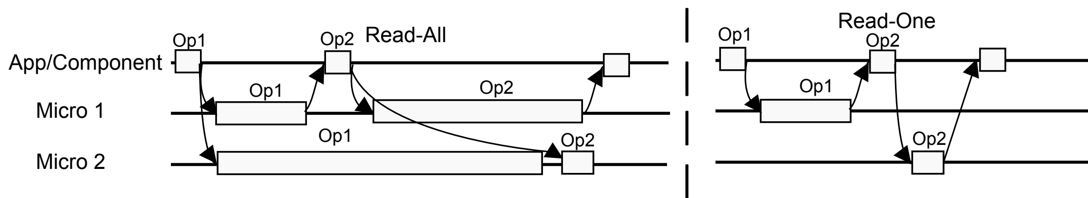


Figure 4.3: Read-all and read-one internal scheduling examples.

4.3.1.1 Read-All

Read-all is the default and most simple mode of internal scheduling: in this model, reads are executed in every available replica and the result returned will be that of the replica that finishes processing first. If the execution starts as soon as the jobs for that operation are submitted to the global scheduler, the operation will execute in parallel in all replicas. This way, it is guaranteed that the operation on the macro-component will have the best performance possible among its replicas. When the macro-component system experiences a light load, this strategy ensures the best performance for all operations.

However, that is not always the case and stress tests reveal a problem for this strategy. Since every replica must process every operation, it is possible for replicas to be held back by processing unnecessary slow operations that completed in other replicas while pending operations could be executed instead. Figure 4.3 illustrates this problem for a single macro-component with 2 replicas. In the example, while Rep 2 has much better performance for the second operation, it must also process the first operation. As the execution time for the first operation is quite long in Rep 2, the replica is kept busy even after there is already a result available for the first operation. In fact, the execution time is so long, that the macro-component will execute the second operation in replica Rep 1 before finishing its execution in replica Rep 2. Consequently, the macro-component will not achieve the best performance of its replicas for that particular operation.

This situation assumes that a replica cannot process two operations concurrently, otherwise Rep 2 could be used processing the second operation without finishing the first. Even in a situation where concurrency within the replicas themselves is possible, a similar situation could occur as we have a limited number of processing units (be it limited by executor threads or processing cores) and, therefore, a limited degree of concurrency.

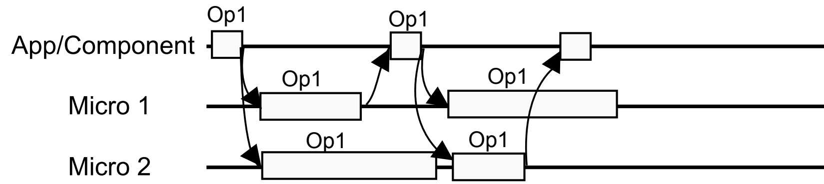


Figure 4.4: Read-all improving overall performance.

Nonetheless, there are two advantages in the read-all internal scheduling mode. First, its simplicity as it requires no special knowledge about the operations or about the macro-component replica implementations. Identifying reads and updates can be useful, and allows for some interesting optimizations, but it is not required. As such it can be applied to any macro-component automatically. The second advantage lies in optimal handling of operations with *unpredictable* execution times. When there is no best replica for a given operation, the macro-component can have a better *overall* performance than either of its replicas. Figure 4.4 illustrates this situation. The first time the macro-component executed *op1*, replica *Rep1* has the best performance for this operation. The second time this operation is executed, *Rep2* outperforms *Rep1*. If either replica were used exclusively for the processing of this operation, the execution time would not be optimal unless it was possible to predict which replica had the best performance for the operation at any moment.

4.3.1.2 Read-One

With read-one, the macro-component's internal scheduler attempts to direct each read operation to a single replica. This replica should be the one with the best performance for the operation. Theoretically, this should minimize "wasted" processing by the macro components. In read-all mode we can look at the macro-component replicas as *competitive*, each replica competes with the remaining and the macro-component returns the result of the winner. In read-one replicas become specialized in executing some operations. Each replica will process only the read operations it is better suited for. Figure 4.3 shows a comparison between the read-all and read-one internal scheduling modes. As we can see, read-one improves on the "busy replica" situation that read-all suffers from. However, it is still possible to suffer from a similar effect in situations with either asynchronous read operations or multiple application threads sharing the same macro-component.

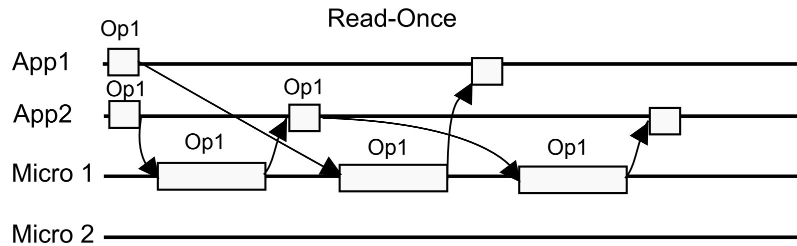


Figure 4.5: Read-one replica overburden example.

If no concurrency is possible between operations in the same replica and the assignment of operations to replicas is completely static, read-one runs the risk of *overburdening* a single replica while not taking advantage of the remaining. In figure 4.5 this situation is shown for a macro-component shared between two application threads. While Rep 1 has the best performance for the operation types submitted by the application thread, in this example the macro-component is taking no advantage of the parallel execution capabilities of the host system. If some of those operations were directed to Rep 2, the macro-component could process them in parallel, and possibly improve its throughput.

An important aspect for this strategy is how to predict which implementation will have the best performance for a given operation. When the operations have a stable or predictable running time (obtained, for example, through complexity analysis), it is possible to use a static approach to the internal scheduler. In this case, the internal scheduler will contain a table of operation-to-replica assignments and will simply check this table for deciding which replica the current operation should be executed on. However, it is not always possible to apply this approach. For example, for some components it is difficult to correctly estimate the running time of an operation for every case. Also, this analysis likely requires a high level of expertise on the domain, and deep knowledge of the implementations of the macro-component replicas. For these reasons, a dynamic approach where the macro-component could self-configure to adapt to the performance of its replicas during runtime might be preferable.

For example, a new macro-component could start execution in the simple read-all internal scheduling mode and gather information, such as the average operation runtime on each replica for any given operation. At some point, this could be used to decide which replica should execute each operation and the macro-component could switch to read-one internal scheduling mode. Nevertheless, such an approach will have to be carefully examined as it is likely it will have an impact on the overhead of macro-component usage. In our prototype, we have only experimented with the static assignment.

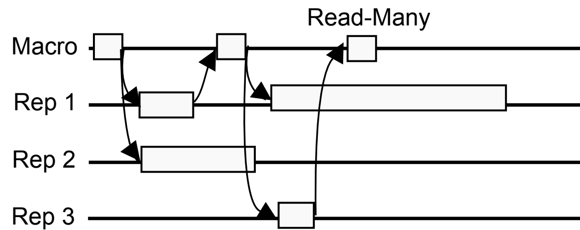


Figure 4.6: Read-many internal scheduling example.

4.3.1.3 Read-Many

Read-many mode is a compromise between read-all and read-one modes. In this mode, an operation is assigned to a subset of the replicas. This scheduling mode avoids part of the problems identified on the read-one and read-all strategies. Nevertheless, new drawbacks can be identified.

First, this strategy requires a larger number of replicas, and therefore implementations of the desired interface, to really be distinguishable from read-one and read-all. If the macro-component only contains two replicas, a subset of those replicas can only be composed of one replica or both replicas. The first case, where the subset holds a single replica, will be identical to read-one. The second case, with both replicas, is identical to read-all. Consequently, a macro-component needs at least 3 replicas for the read-many strategy to be applicable.

Second, just like read-all, read-many can lead to needless wasting processing resources when replicas are competing between themselves and processing the same operations in parallel. Figure 4.6 presents an example of a macro-component running in read-many internal scheduling mode with 3 underlying replicas. In this example, op1 is executed in replicas 1 and 2, while op2 is executed in replicas 1 and 3.

4.3.2 Global Scheduling

The global scheduler is responsible to schedule the execution jobs from multiple macro-components. Although different strategies can be implemented at this level several concerns must be addressed to maintain proper macro-component functionality.

First, the relative order of update operations in each replica must be preserved. Take for example two operations, one that inserts an element in a data structure and another which removes the same element. If these operations are executed out of order, macro-component replicas' internal state would diverge. Namely, replica 1 would first apply the insertion operation and then the removal. Replica 2 would apply the removal operation first, and then the insertion. As a result, replica 1 would not contain

the element, but replica 2 would. Subsequent operations on that macro-component would have different results in either replica, leading to incorrect or unexpected results. Together with the internal scheduling rule which states that update operations are executed in all replicas, this means that update operations must all be executed in the same order on all replicas.

Second, while read operations can be reordered they must execute in a state that reflects all previous updates and before any later updates. In other words, each update operation pair creates a view, where read operations between these two updates can be reordered at will. Furthermore, depending on the replica's characteristics, it might be possible to execute read operations in the same replica in parallel without sacrificing correctness.

Third, for simple independent macro-components, these ordering rules can be enforced for each macro-component independently, as there is no relationship between the components. In more complex cases, where the macro-component replicas do not follow the self-containment limitation, this independence may not hold - an example, is the JDBC interface used in our evaluation, where a Statement has an associated Connection. In this case, the ordering of operation execution must be preserved across the set of related macro-components. To implement this functionality, in our prototype, the global scheduler maintains several work queues. Each independent macro-component has its own queue. A group of related macro-components share the same queue. This approach guarantees that the system preserves the expected ordering among operations.

Finally, simple optimizations can be done at the global scheduling level to improve global macro-component performance. One of these optimizations, which is applied on our prototype is the discarding of read execution jobs for which a result was already obtained. This happens frequently when macro-components use read-all internal scheduling modes. When an executor requests a new job, the scheduler can verify if there is already a result for the next job and, if that is the case, discard it and verify the next pending job until a suitable one is found.

4.3.3 Speedup

In this section, we try to estimate the expected maximum performance increase attainable by a macro-component in relation to a given implementation. This estimation intends only to give a rouch idea of the possible performance improvements of macro-components.

The estimation is always connected to a given *setting*. The definition of what exactly is part of the component's setting can vary between different components and must

include at least 2 aspects: the relative weight of different operations on the operation mix and the state of the macro component (including which implementations are used as its replicas).

The second aspect is required to identify the average running time of the different operations for each replica, which is used in the speedup computations. Different states will lead to different performances for the operations. Exactly what comprises this state must be adjusted to the component being analysed. For example, the state of a data structure can be defined as the elements it contains, while the state of a database is slightly more complex, including the current database schema, information in those tables, serialization mode, cache size, etc.

Assuming we are considering a low stress situation, where the several replicas have time to synchronize between consecutive operations, the speedup estimation can be obtained with a simple formula. This computation disregards the overhead imposed by the macro-component system and assumes that operations will not overlap. This assumption is likely to hold for read-one macro-components and read-all in a low stress situation.

Given these assumptions, we can estimate the speedup of a macro-component with I possible implementations (numbered 1..I) which provide O operations (numbered 1..O) with the following formula:

$$Sp(impl) = \sum_{op=1..n} F(op) \times \frac{T(op, impl)}{Tmin(op)} \tag{4.1}$$

Where:

- Sp(x) – The maximum speedup achievable by the macro-component when compared to implementation x.
- T(o,x) – The average execution time of operation o on implementation x.
- Tmin(o) – The execution time of operation o on the best possible implementation.
- F(o) – The relative frequency of operation o on the operation mix for the macro-component's setting.

For example, take a component which supports 2 operations, *op1* and *op2*, and has 2 possible implementations *Impl1* and *Impl2*. With the average execution times and relative frequencies listed in table 4.1 the maximum speedup obtainable by the macro-component in relation to *Impl1* is computed as:

Table 4.1: Speedup estimation example

Impl	op1 (50%)	op2 (50%)
1	15 ms	10 ms
2	10 ms	15 ms

$$Sp(Impl1) = 0.5 \times \frac{15}{10} + 0.5 \times \frac{10}{10} = 0.5 \times 15 + 0.5 = 1.25 \quad (4.2)$$

High stress situations are inherently more complex. In these cases, the assumption of no overlapping operations in the macro-component is very unlikely to hold. While the maximum speedup cannot be determined in the same way as low-stress settings, it is possible to estimate this using simulation. This should require no more information about the macro-component setting than what is required for the formula described earlier.

4.4 Addressing Low-Granularity Operations

The base design presented previously in this section is meant to be simple and general. While this design provides a good initial starting point for research in macro-components, it does not handle fine-granularity operations well. Some changes were made to this base design which led to an alternative implementation capable of supporting finer granularity operations.

4.4.1 Goals

The main goal of the implementation of an optimized runtime system is to study the viability of macro-component usage in the case of components which provide mainly fine-granularity operations or that include a large number of these operations. As described earlier in section 3.2.1, the main obstacle to fine-granularity operations is the execution time overhead imposed by the runtime system, especially due to thread-coordination.

The optimized runtime system attempts to minimize this overhead, if necessary at the cost of a loss of other properties – e.g. the current implementation shows better performance but demands considerably more processor resources for the reasons detailed in section 5.4.2. By obtaining an implementation with minimum time overhead, we aim to evaluate the impact of this overhead by using a data structure based macro-component.

4.4.2 Design Changes

The main design changes introduced to the base runtime system are in the way operations execute. The largest sources of overhead of the base system is the coordination between the application thread and the executors. This happens at two points: the coordination in the access to the shared work queue, and the blocking/waking up of threads during synchronous operations. In this design, we attempt to minimize this overhead based on the following insight:

Part of the update operations can be done by the application thread. In the base system, each update operation generates a job task for each replica in the macro-component, which are then submitted to the global scheduler. With this optimization, one of the replicas is prioritized, we call this the primary replica.

While the non-primary replicas remain updated by the Executors from the submitted job tasks, this primary replica is updated directly by the application thread. Using this approach, the overhead of the update being handed to the Executor is avoided for the primary replica and it should always be ready to process new operations.

Read operations can execute in the application thread. With the read-one internal scheduling strategy, each read operation is executed in a single replica. As such, there is no need for the operation to be submitted to execution by the Executor pool, as it will not benefit from concurrent execution. Instead, it can execute in the selected replica in the application thread.

However, some mechanism to ensure consistency must be used to guarantee some coordination between the application thread reads and the Executor threads updating the replicas. If the application thread simply executes a read operation on a non-primary replica it is possible that replica has pending updates which were not yet processed by the Executors, which may lead do inconsistent results for that read. In the same way, if either multiple threads are sharing a macro-component, or several Executors are attempting to update the same replica with no coordination, updates could be done out of order.

The consequences of these design changes in the runtime system implementation, and the coordination method used are presented in section 5.4.

5

Implementation

This chapter describes the implementation of the prototype of the macro-component runtime system. This is divided in three main parts. Section 5.1 details the implementation of the core runtime system. Section 5.2 describes the prototype macro-component implemented: a replicated in-memory database. Section 5.3 presents the tools for automatically generating macro-components. Finally, section 5.4 conclused the chapter by presenting the implementation details for the optimized design with support for fine-granularity operations.

5.1 Runtime System

The runtime system was implemented in Java 6.0 and comprises both an initial implementation for the global scheduler as well as the definition of the interfaces and subsequent code style for the macro-component Java classes.

This system implements the design presented in section 4, trying to address two main concerns. First, the development of the individual macro-components and the global scheduler should be as independent as possible. This means there is no need for macro-component programmers to have any knowledge about implementation details of the global scheduler. The only interaction between the two is during job submission and result retrieval. Second, simplicity and generality. These characteristics are not only helpful to simplify the programming of new macro-components, but are also essential to allow the creation of a framework for automatic macro-component code

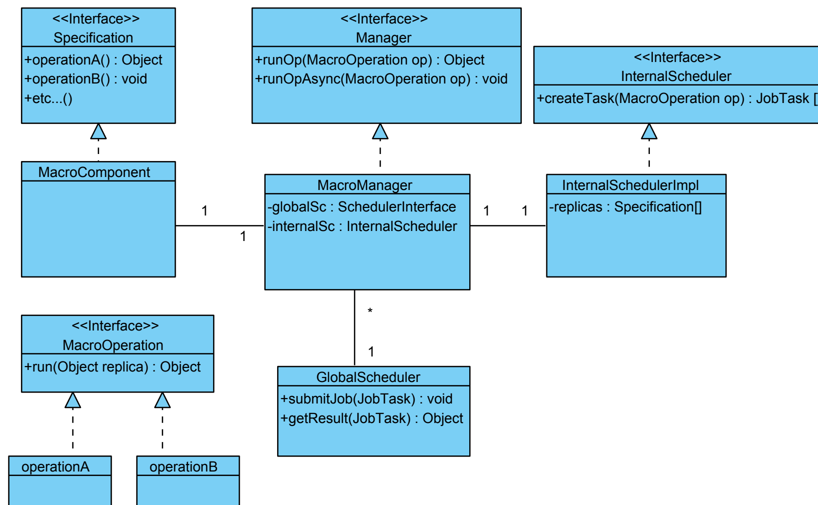


Figure 5.1: Generic class diagram for proposed macro-component structure.

generation, as detailed in section 5.3.

5.1.1 Macro-component template

Figure 5.1 presents the basic class structure for a generic macro-component. The mapping from the design to Java classes is almost direct. The MacroComponent class of a macro-component that implements interface *X* will also implement interface *X*, making the macro-component transparent to the application: the macro-component can be used as any other simple implementation of the interface.

The macro-component class includes two constructors: the first, that hides from applications the replicas that will be used, which are predefined. The second, which the application can use to specify the replicas which will be used.

This MacroComponent class itself is just a front-end. Every method on this front-end will generate a request to the macro-component's manager (MacroManager) to run the corresponding operation, either synchronously (through the *runOp()* method) or asynchronously (*runOpAsync()* method). The manager performs as described in chapter 4, calling the internal scheduler on each method call and submitting the resulting execution jobs to the global scheduler.

5.1.1.1 Execution Jobs and Operations

Execution jobs encode calls to a method on a specific replica. These jobs must be generic enough to allow uniform handling by the global scheduler. Also, the jobs must be able to hold all necessary information to support the chosen global scheduling strategy. Currently, the only information supported is whether it is a read-only or update operation. However, this can be easily extended to support additional information. Finally, these jobs are also used as containers for the result of the operation, as this makes for easy access to this result by both the macro-component, scheduler and executors. The interface of the job is presented in figure 5.2.

```
public interface JobTask {

     public boolean isUpdate();
     public Object getResult();
     public void setResult(Object result);
     public Object run();

}
```

Figure 5.2: Basic JobTask interface.

The *run* method allows the executor to transparently execute operations from any macro-component in a generic way. When the Executor receives the job from the global scheduler it simply executes the run method on the job and sets the job result with the return of the run method.

To avoid having to keep all the parameters of a method in all jobs created for that method call, we have created the MacroOperation interface (figure 5.3). Implementations of this interface store the parameters of the method call and include a method to execute the operation in a given replica, passed as a parameter.

```
public interface MacroOperation {

    Object run(Object rep);

}
```

Figure 5.3: MacroOperation interface.

For example, if our MacroTest component implements the interface Test, which contains an *add()* method with an integer argument, the resulting MacroOperation class

```
public class Add implements MacroOperation {

    public Add(Integer arg0) {
        this.arg0 = arg0;
    }

    public Object run(Object rep) {
        return ((Test) rep).add(arg0);
    }

}
```

Figure 5.4: Add MacroOperation implementation.

code for this method is the one found in figure 5.4.

This allows for a generic JobTask implementation that receives in the constructor both a MacroOperation and a reference to the replica the operation should be executed on. The run method for the JobTask implementation simply calls the run method of the MacroOperation on the given replica as shown in figure 5.5. As such, only the MacroOperation classes for any new macro-component have to be generated.

```
public class JobTaskImpl implements JobTask {

    public JobTaskImpl(MacroOperation op, Object replica ...) {
        (...)
        public Object run() {
            return op.run(replica);
        }
    }
}
```

Figure 5.5: Partial JobTask implementation.

5.1.2 Internal Scheduler

Two scheduling modes were implemented in this runtime system: read-all and read-one. Implementing the read-all strategy is straightforward. Any operation submitted to the internal scheduler of a macro-component will cause the scheduler to create an execution job for each available replica. An example of the implementation for this strategy is found in figure 5.6

Read-one, however, requires the internal scheduler to assign operations to different replicas. This assignment is currently done statically. The internal scheduler has the

```
public JobTask[] createTask(MacroOperation op) {
        JobTask[] result;

        result = new JobTask[replicas.length];

        for (int i = 0; i < reps.length; i++)
            result[i] = new JobTaskImpl(op,replicas[i],wrap ...);
    }
    return result;
}
```

Figure 5.6: Read-all internal scheduling implementation.

information about which operation goes to which replica hard-coded in its createTask() method. Figure 5.7 presents an example of an internal scheduler that executes in read-one mode based on static assignment of operation types to specific replicas.

5.1.3 Global Scheduler and Executors

5.1.3.1 Scheduling Strategy

The global scheduler strategy is based on a group FIFO algorithm designed to be able to handle relationships between operations from different macro-components. The basis for this scheduling strategy is the separation of operations in groups of dependent components. For operations on a specific group, the global scheduler is guaranteed to maintain a FIFO ordering in their execution. This means that all previous operations from that group are required to have *finished* their execution before the next operation is handed for processing.

This, in turn, limits the parallel execution capabilities inside a group. It is possible to relax this rule and allow for parallel execution of read-only operations within a group. In this case, the scheduling rule is divided in two parts. First, read operations require that all previous *update* operations from its group have finished their execution. Update operations, on the other hand, require that all previous operations from its group, both read and updates, have finished before they can be applied. However, the applicability of this strategy depends on replica semantics and, as such, has not yet been implemented in the current runtime system.

5.1.3.2 Work Queue

This scheduling strategy had a direct impact in the scheduler's work queue implementation. Instead of a single queue, the work queue contains several separate FIFO

```
public JobTask[] createTask(MacroOperation op) {
     (...)
     JobTask[] result;
     if (isUpdate(op)) {
          //Updates execute in all replicas
          result = new JobTask[replicas.length];
          for (int i = 0; i < replicas.length; i++)
               result[i] = new JobTaskImpl(op,replicas[i],wrap);
          return result;
     }
     if (op instanceof MethodA)  {

          //Method A executes only on replica 0
          result = new JobTask[1];
          result[0] = new JobTaskImpl(op,replicas[0],wrap);
          return result;
    }
    else {

         //Method B executes only on replica 1
         if (op instanceof MethodB) {
              result = new JobTask[1];
              result[0] = new JobTaskImpl(op,replicas[1],wrap);
              return result;
          }

   (...)
  }
}
```

Figure 5.7: Read-one internal scheduling implementation.

queues, one for each scheduling group. Operations from each group are placed in their respective queue. As each executor requests a new job, the global scheduler will simply remove the job at the head of one queue and hand it to the requesting executor. This is enough to guarantee operations in a group are executed in the correct order. However, it is not enough to guarantee these will not overlap and be processed in parallel.

As there is no parallelism between operations in the group, a trivial solution is to assign a single executor to each group. As only this executor thread will process jobs in that group, it is guaranteed these will not overlap, and correct group semantics will be maintained. Other distributions can be used, such as sharing executors between groups, but as it is assumed operations in the same group cannot be executed in parallel (so that dependencies can be maintained) a number of executors greater than the number of groups will lead to idle executors.

To allow this, and other possible extensions to this model, executors were extended from simple "worker bee" threads to hold an executor identifier and a queue identifier,

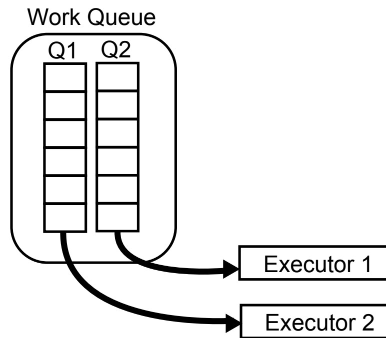


Figure 5.8: Global scheduling with a per-group work queue.

as can be seen in figure 5.9. This queue identifier is sent with every request for a new job, allowing the global scheduler to pass the executor a job from the correct queue. Figure 5.8 offers a rough sketch of this scheduler model.

```
public class Executor implements Runnable {

    SchedulerInterface scheduler;
    int executorID;
    int queueID;

    public Executor(int executorID,SchedulerInterface scheduler,int queueID) {
        this.scheduler = scheduler;
        this.executorID = executorID;
        this.queueID = queueID;
    }

    (...)

    public void run() {
        while(true) {
            JobTask task = scheduler.getJob(queueID);
            Object result = task.run();
            task.setResult(result);
            scheduler.signalFinish(task);
        }
    }
}
```

Figure 5.9: Executor implementation example.

For controlling access to the job queues, the BlockingQueue interface of Java was used. In Blocking queues the methods used to add and remove elements are atomic, ensuring thread-safety. Furthermore, these methods can block the caller threads until

a condition is verified, normally that the queue is not empty/not full. This is can be used to block Executor threads when there are no pending jobs to execute.

From the several available BlockingQueue implementations, the one chosen for our scheduler was the unbounded BlockingLinkedQueue. This allows the producer (scheduler/macro-components) to insert new jobs on a queue without blocking, as it is not bounded by capacity. Consequently, application threads submitting operations to the macro-component are never required to block due to a full job queue. This means asynchronous operations will never block when submitting operations to the system, and synchronous operations only block waiting for their result and not in intermediate steps.

5.2 In-memory Database Macro-Component

To evaluate the runtime system, we have developed a macro-component for in-memory databases. A custom JDBC driver was created which uses this macro-component and allows any program to transparently access a replicated in-memory database as it would use a single database.

The database engines used for the underlying database replicas were H2[1] and HSQLDB[2], which both feature an in-memory mode database configuration. In this section, we start by discussing the reasons for the choice of this example. Next, after a small introduction to the JDBC API, we detail the custom JDBC driver macro-component's design and implementation.

5.2.1 Why databases?

There are several characteristics of the in-memory database environment that led to its choice as a test case. First, due to heavy interest and market competition, there is a large number of well known and established database engines both proprietary and open-source. Furthermore, a database engine's external interface (in this case, the JDBC API) is uniform and, in the absence of inherent bugs or dialect changes, it is functionally equivalent to other engines. This makes it easier to select a number of possible implementations for a macro-component.

Second, it is well known and documented that there are significant performance differences between different database engines [GPSS03]. This is caused by several reasons, such as different indexing methods, storage methods, table join algorithms, etc.

[1]http://www.h2database.com
[2]http://hsqldb.org/

Third, we expected the running time of the queries to be largely constant in a given implementation, although that was not always observed. However, it is possible to predict which database has the best performance for a set of selected operations. Thus, it is possible to encode in the internal scheduler this information and implement a read-one policy.

Fourth, queries in mid to large-size databases have a relatively long running time, in the order of tens to hundreds of miliseconds depending on the operation and the machine it is run on. Therefore, most operations in this macro-component have coarse granularity. As mentioned before, for this type of operations the overhead imposed by the macro-component runtime system becomes negligible.

Finally, the JDBC interface is fairly complex, with inter-related objects. This allows to experiment with macro-components in a non-trivial environment.

5.2.2 JDBC

The Java Database Connectivity (JDBC) API provides a simple, transparent and uniform API for programmers to access databases when building Java applications. In this section we present a small description of the basic objects used when programming using JDBC along with some code examples.

To access a database using JDBC, the first thing a programmer needs to do is to establish a connection with the intended database. This is done by requesting a Connection from the DriverManager after the driver for the database is loaded. This Connection object represents an open session with the database. The code snippet below shows the creation of a connection to a custom database. In the getConnection method, the first argument is a URL that identifies both the database engine, testEngine, and the database name, schoolDatabase.

```
Class.forName("test.jdbc.driver.TestDriver");
Connection con = DriverManager.getConnection("jdbc:testEngine:schoolDatabase", "admin","admin_pass");
```

With the Connection object, it is possible to start querying the database. This can be done through Statement objects associated with the open connection. Three different types of statements can be created from a connection: Statements, PreparedStatements and CallableStatements. Their uses are slightly different, but in abstract they are all objects which allow the programmer to send SQL statements to the database. The code example bellow shows the execution of a simple *select* operation, using a Statement object. The result of the operation is printed on the standard output.

```
Statement stat = con.createStatement();
ResultSet rs = stat.executeQuery("select * from table1");
while (rs.next()) {
```

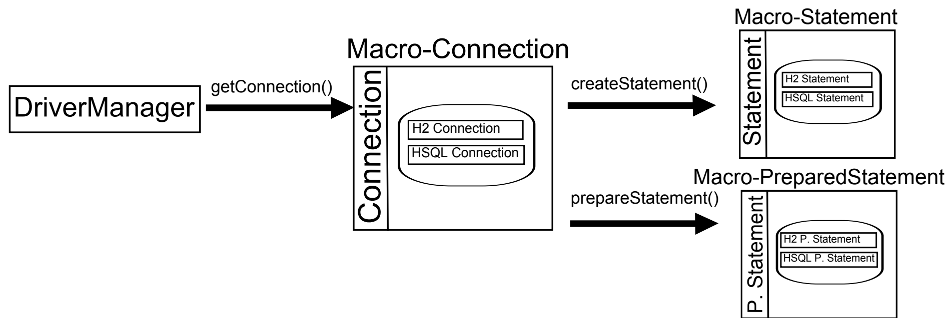


Figure 5.10: JDBC macro-component hierarchy.

```
        System.out.println(rs.getString(1);)
    }
    rs.close();
    stat.close();
```

Following this simple introduction to the JDBC API, we will next detail the custom driver using macro-components.

5.2.3 Macro-component-based JDBC Driver Solution

In the macro-component-based custom JDBC driver solution, Connection, Statements and PreparedStatements are macro-components, with the actual H2 and HSQLDB equivalents being the underlying replicas. When a new connection to the macro-component database is created, the Connection object is a macro-component named MacroConnection. This MacroConnection uses as replicas Connection objects to the underlying databases, in this case one Connection for the H2 database and another for the HSQLDB. In the same way, Statement and PreparedStatement objects created from the MacroConnections are also macro-components, with the same underlying objects. This hierarchy is illustrated in figure 5.10

An interesting detail in the creation of the MacroStatements and MacroPreparedStatements components is that it requires a different approach when compared with normal operations. For example, the *createStatement* method executed the *createStatement* in both replicas and creates a new MacroStatement macro-component that uses the returned Statement objects as replicas.

5.2.3.1 Global Scheduling

In this solution, macro-components are not self-contained. The Connection, Statement and PreparedStatement objects interact with the underlying database engines. For

this reason, special care must be taken while handling the operations for these macro-components as it is undesirable to execute them out of order. This means that operations from different macro-components are not independent from each other, and their relative order must be taken into account by the global scheduler. For example, we cannot allow a Connection macro-component to commit without ensuring all previous operations on the Statement and PreparedStatements it has spawned have finished.

The solution for this issue is based on the creation of a group of related macro-component replicas as described earlier in section 5.1. The group assigned to an operation depends on which database engine the operation accesses. This can be trivially verified as different replicas modify different database engines. Operations executed on H2-implemented Connection, Statement or PreparedStatement replicas are considered as belonging to one group, and therefore are sent to a specific logical queue on the global scheduler. Operations on HSQL, are considered as a different group, and are handled in a different logical queue. Operations on the same logical queue are guaranteed to execute in FIFO order, thus the relative order of operations for each database engine is maintained.

5.2.3.2 Internal Scheduling

The read-all internal scheduling mode is implemented in a trivial way. Read-one mode implementation, however, has an interesting detail. While it is established that different database engines have different performances for different SQL queries, all queries are executed using the same method. Thus, the internal scheduler must differentiate between queries by considering the parameters of the operation - i.e. the SQL code of the query is matched against a list of known queries to determine which replica has the best performance. If no replica is known to have the best performance, the query is executed in all replicas.

5.3 Macro-Component Programming Support Tools

This section details the program support tools for creating macro-components. Taking as input the interface of replicas, this tool can automatically generate all classes of a macro-component with read-all internal scheduling.

With some Java annotations to this interface – i.e. indicating for example if an operation is a read or an update, its granularity or the expected best replica – it is possible to create macro-components with read-one internal scheduling. This is not available in the current implementation, but is a planned extension to the tool.

5.3.1 JavaCC and JJTree

Java Compiler Compiler (JavaCC) is a portable java based parser generator. It takes a formal grammar expressed in EBNF notation and from it generates a Java parser for that grammar. In this work, Java CC was used to generate a parser capable of parsing Java interface files. The grammar used to this end is a subset of the full Java 1.5 grammar provided with the default Java CC distribution. We have also used JJTree to build the abstract syntax tree of the parsed interface, which is later used to generate the needed code.

5.3.2 Code Generation

With the approach described in the previous section, we go from having a Java interface file, to the abstract syntax tree holding all the information defined in the interface. From this information, the classes needed to create a macro-component are created using pre-defined templates. Namely, class files are generated for the macro-component manager, an internal scheduler that is ready to schedule operations in read-all mode, and for all operations specified in the interface. For example, the template for an operation can be seen in figure 5.11:

```
public class <operation name> implements MacroOperation {

    <argument declaration>

    public <operation name>(<argument list>) {
        this.<argument> = <argument>;
    }

    public Object run(Object rep) {
        try {
            return ((<interface name>) rep).<operation name>(<arguments>);
        } catch (Exception e) {
          throw new RuntimeException(<message>);
        }
    }

}
```

Figure 5.11: Template for a MacroOperation class file.

Where fields marked with < > are simple markers to be replaced with the information from the parse tree. For example, the object created to encode a call to the methodB

```
public interface TestClass {

    public void methodA(String arg0, boolean arg1);

    public int methodB(int arg0) throws Exception;

}
```

Figure 5.12: Example interface.

```
public class methodB implements MacroOperation {

    int arg0;

    public methodB(int arg0) {
        this.arg0 = arg0;
    }

    public Object run(Object rep) {
        try {
            return ((TestClass) rep).methodB(arg0);
        } catch (Exception e) {
            throw new RuntimeException("Error executing methodB");
        }
    }
}
```

Figure 5.13: Class generated from MacroOperation template.

defined in the interface of figure 5.12, is presented in figure fig:methodb:

The full code-generation process is illustrated in 5.14 step by step.

The templates for the other classes of a macro-component are presented in figures 5.15, 5.16 and 5.17. Note that these templates are used to implement the base of simple macro-components. The result can be later modified to implement more complex components.

5.4 Optimized Runtime System

This section presents the relevant implementation details for our optimized design. This system offers increased support for fine-granularity operations based on the design presented in section 4.4. Section 5.4.1 shows how consistency is maintained in the face of operations which are not submitted to the global scheduler and section 5.4.2

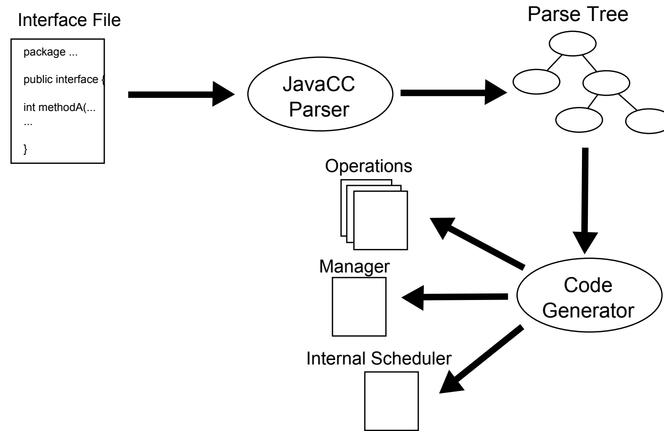


Figure 5.14: Code generation process.

```
public class Macro<interface name> implements <interface name> {

    Manager manager;

    public Macro<interface name>(<interface name>[] replicas, SchedulerInterface schedule
        super();
        this.manager = new <interface name>Manager(scheduler,replicas);
    }

    //for each synchronous operation...
    public void <operation name>(<operation argument list>) {
        manager.runOp(new <operation name>(<operation arguments>);
    }

    //for each asynchronous operation...
    public void <operation name>(<operation argument list>) {
        manager.runOpAsync(new <operation name>(<operation arguments>);
     }
}
```

Figure 5.15: Template for a macro-component class file

explains how the optimized design attempts to minimize the response time of the application and Executor threads.

5.4.1 Maintaining Consistency

Due to the optimizations previously discussed, not all operations are submitted to the global scheduler for execution by the Executor threads. This poses an issue with maintaining consistency in the replicas. For example, if the application thread blindly executes a read operation on a replica, it is possible that replica is still outdated as previous

```
public class <interface name>Manager implements Manager {

    SchedulerInterface globalScheduler;
    InternalScheduler internalScheduler;
    <interface name>[] replicas;

    public MacroConnectionManager(SchedulerInterface scheduler,<interface name>[] re
        this.replicas = replicas;
        this.globalScheduler = scheduler;
        internalScheduler = new <interface name>Scheduler(replicas);
    }

    public Object runOp(MacroOperation op) {
        JobTask[] task = internalScheduler.createTask(op);

        for (int i = 0; i < task.length; i++)
            globalScheduler.submitJob(task[i]);

        return globalScheduler.getResult(task);
    }

    public void runOpAsync(MacroOperation op) {
        JobTask[] task = internalScheduler.createTask(op);

        for (int i = 0; i < task.length; i++)
            globalScheduler.submitJob(task[i]);
    }
}
```

Figure 5.16: Template for manager class file

updates (which are handled asynchronously by the executors) might not have been applied yet.

This led to the introduction of *replica versions*. Each replica holds a version number, initialized when the replica is created. This version number is incremented whenever the replica is updated (after the update has been processed by the Executor or the application thread).

Additionally, macro-components maintain an *expected version number*. This number is incremented by the manager at the start of every update operation, before the update is submitted for background execution. At any point, the expected version number represents the latest state a replica can reach.

Read-operations have no impact on either the replica's version number or the macro-component's expected version number.

These version numbers are used to coordinate the execution of reads by the application thread. When the application thread starts a read operation, it must compare

```
public class <interface name>Scheduler implements InternalScheduler {

     <interface name>[] replicas;

     public <interface name>Scheduler() {
          this.replicas = replicas;
     }

     public JobTask[] createTask(MacroOperation op) {

          JobTask[] result = new JobTask[replicas.length];
          ResultWrapper wrap = new ResultWrapper();

          for (int i = 0; i < stats.length; i++)
               result[i] = new DatabaseJobTask(op,replicas[i],wrap);

         return result;
     }

}
```

Figure 5.17: Template for an internal scheduler class file

the expected version number to the replica's version number. If they are different, the application thread must wait for the remaining updates to be applied before it can continue.

In the same way, if multiple Executors are working on the same replica or set or related replicas, these can use the version numbers along with the expected version number at the moment the update operation was issued to guarantee updates are executed in proper order.

5.4.2 Minimizing Response Time

While the Executors and the application thread can coordinate using the version numbers described in the previous section, it is important this coordination have the lowest overhead possible.

Assuming low-granularity operations, a difference of microseconds can have a large impact on the overall performance. The solution adopted for the coordination between threads was "busy waiting", with shared *volatile* conditional variables.

Busy waiting consists on the process of repeatedly checking a condition which must be valid before the program can continue. This is usually done by implementing a loop with the desired condition and an empty ("do nothing") body. While this is generally wasteful of processor resources, the response time proved to be faster than the more elegant Java sleep/wait model.

Figure 5.18 presents an example of a coordination point using replica versions and busy waiting. The process continuously checks the current version of the replica against the expected (latest) version. When the condition fails – because the replica version is the expected one – the process can continue and execute the read on the replica which is now up to date. Busy waiting must be used carefully and can have different results in different settings due to different models. As multiple processes are accessing the same variable concurrently this can have unexpected effects, especially if the reads and updates to those variables aren't atomic.

```
public boolean operation(int arg0) {

    while (replica.version < expectedVersion)
        //do nothing...
        ;

    replica.operation(int arg0);

}
```

Figure 5.18: Coordination using busy waiting example.

Under the Java memory model, reads and updates on reference fields and most primitive types (which is the case in our implementation, as replica versions are integers) are atomic. Consequently, the concurrent accesses to the version numbers will not corrupt these values and this will not be problematic for the current implementation.

Nonetheless, a different issue can alter the behaviour of busy waiting coordination. As replica versions, and other conditional variables, are accessed by different processors – i.e. the macro-component is waiting on the loop while the Executor finishes an update and increments the version number – it is necessary to guarantee consistency in the values seen by the processors. If the application thread is allowed to view "stale" values for these variables during a busy waiting loop, for example due to processor local caches, it will likely enter an infinite loop. For this reason, all the conditional variables are marked with the Java *volatile* keyword. This keyword limits the caching and reordering of accesses to these variables, guaranteeing that each read will see the value of the last write even if done by a different processor.

6

Evaluation

This chapter presents an evaluation of the macro-component runtime system. Section 6.1 presents the evaluation of the base runtime system with the prototype database macro-component and high granularity operations. Following that, section 6.2 presents the evaluation of the optimized runtime system using the fine-grained data structure macro-component implementation.

6.1 Runtime System Evaluation

This section deals with the evaluation work related to the system with no fine-granularity optimizations and is organized as follows. Section 6.1.1 describes the benchmark used for the evaluation. Section 6.1.2 presents the experimental setup, including the execution environment, host machine and database scheme and, finally, section 6.1.3 presents and discusses the results obtained.

6.1.1 Benchmark

6.1.1.1 Data Model

The in-memory database macro-component prototype was validated using a custom micro-benchmark. This benchmark simulates a school database and features three tables: students, courses and grades. The student tables table information about each student: a student ID as primary key, a name and an address (represented by a single

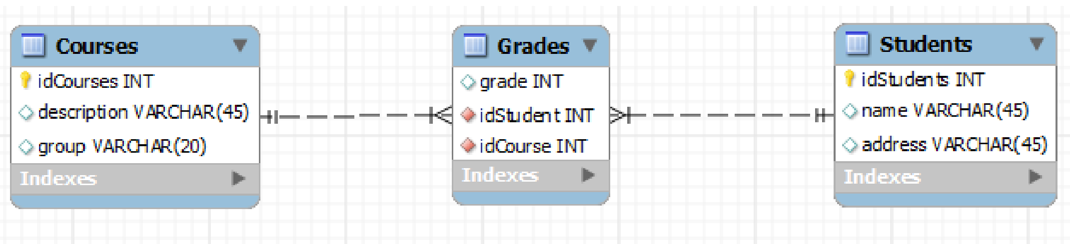


Figure 6.1: Entity-relationship diagram for the test database.

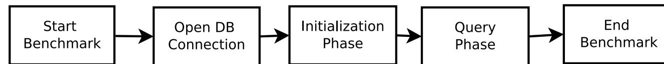


Figure 6.2: Flowchart detailing the complete benchmark execution.

location/city). The courses table holds a course ID as primary key, a small description and a course group (i.e. computer science, biology). Finally, the grades table is a relationship table between the students and courses which holds a student ID, and course ID as foreign keys along with a 0-20 grade. Figure 6.1 presents the Entity-Relationship diagram for this database.

6.1.1.2 Execution Model

The benchmark application, running in a single thread, consists of: (1) an initial startup phase, where the databases tables are created and initialized with data; (2) a query phase, where a pre-determined sequence of queries is executed. Both the initialization phase and the query phase include sequences of JDBC operations.

The benchmark was run in three different modes: *read one*, queries executed by the replica with best performance for that query; *low-stress read-all*, where all operations execute in all replicas but each operation must complete in all replicas before a new operation executes; and *high stress read-all*, where all operations execute in all replicas but a new operation is executed as soon as the previous one completes in a single replica.

6.1.1.3 Intialization Phase

The application thread will initialize the database tables using pseudo-randomly generated data (with a configurable seed). The final size for each table can be configured and easily changed.

Although the results presented in this chapter do not feature the time taken to initialize the databases, an interesting mechanism was used to mitigate much of the

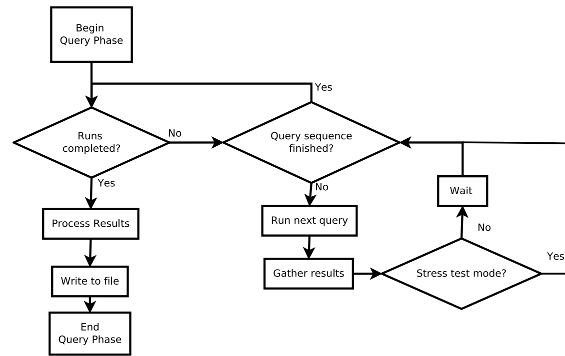


Figure 6.3: Flowchart detailing the query phase execution.

overhead of the macro-component for this phase. The intialization phase comprises a great number of low-granularity update operations, which is a worst-case scenario for macro-components. However, these operations are easy to group as batches, both at runtime-system level, by creating job tasks which batch several operations, and at application level, by using multi-value insert queries. In the benchmark implementation, this was done at the application level.

6.1.1.4 Query Phase

After the initialization phase the benchmark enters the query phase. This phase consists of a number of identical *runs*. In each run, the application thread executes a sequence of mixed queries (which comprises both read and update operations). The exact sequence of queries to execute is loaded from a file in the beginning of the benchmark, thus making it easy to test different workloads.

When the benchmark is running in normal mode, there is an inter-operation waiting time between two consecutive queries where the application thread is halted, giving the macro-component replicas time to synchronize. This behaviour is similar to an application that does not depend heavily on the component. When running in high stress mode, there is no inter-operation waiting time. Consequently, the next query is executed as soon as the previous one finishes, giving the macro-component no time to synchronize its replicas.

The benchmark executes each run N times, computing the average execution time for the results obtained.

After each run is finished, the benchmark will process the results obtained. From these, it obtains the average execution time for each query and run. Finally, a results file is created with the result (execution time and rows found) for every query executed, the total execution times for each run and all runs, and the computed averages.

6.1.2 Experimental setup

All tests were performed on a Sun Fire X4600 M2 x86-64 server machine, with eight dual-core AMD Opteron Model 8220 1.0 GHz processors and 32 GByte of RAM running the Debian 5 (Lenny) operating system. The Java runtime environment used was OpenJDK Runtime Environment (build 1.6.0_0-b11).

The custom JDBC driver macro-component used 2 in-memory database implementations as underlying replicas: H2 (2010-03-21 release) and HSQLDB (2.0.0 RC9).

The database tables are filled with identical pseudo-randomly generated data for all tests. The courses table is the smallest table in this test, containing 20 rows. The students table is medium sized, holding 10.000 rows. The grades relationship table contains 5 grades per student, totalling 50.000 rows.

With their size diversity, these tables are enough to perform a number of distinct queries, from large table joins to range searches in small tables, where different database engine implementations show different performances.

All time values are represented in milliseconds and the results obtained are the average of 4 independent tests, each with 100 runs of the benchmark's query phase.

The results shown in the next section are based on the query sequence presented in tables 6.1 and 6.2. The first column holds the query identifier. Queries are executed in ascending order by this identifier. The second, is a small description of the query's processing in natural language. All queries are expressed in standard SQL in order to guarantee compatibility with the maximum number of database engines possible. Finally, the last column displays the database engine in which the query was performed faster in average.

Table 6.1: Queries

ID	SQL code	Best
0	3 table join, *lower than* range search on integer field	HSQL
1	2 table join, *greater than* range search on integer field	H2
2	3 table join, emphbetween x and y range search on integer field	HSQL
3	update on several rows of students table	H2
4	2 table join	H2
5	single table, *greater than* range search on integer field	H2
6	2 table join	H2

Table 6.2: Queries (continued)

ID	SQL code	Best
7	update on several rows of students table	H2
8	update on courses table based on subquery	H2
9	2 table join with computation of average	H2
10	2 table join with computation of average	HSQL
11	3 table join, *equal or greater than* range search, boolean check	HSQL
12	3 table join, *equal or lesser than* range search, boolean check	HSQL

6.1.3 Results

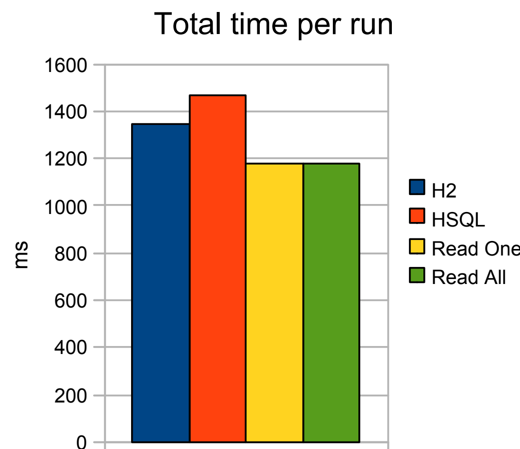


Figure 6.4: Average response time per run in low stress mode.

Figures 6.4 and 6.5 present the average total time taken by the benchmark application for a single run of the query phase for the low stress and high stress modes respectively. The results compare the performance of the H2 and HSQLDB database engines with our custom macro-component in both read-all and read-one internal scheduling modes.

For the low stress setting, H2 presents a better overall performance than HSQL, while read-all and read-one show the best performance with very similar values. The speedup values of the macro-component in this setting are approximately 13% and 24% in relation to H2 and HSQL respectively.

In the high stress situation, HSQL and H2 maintain their relative performances. The

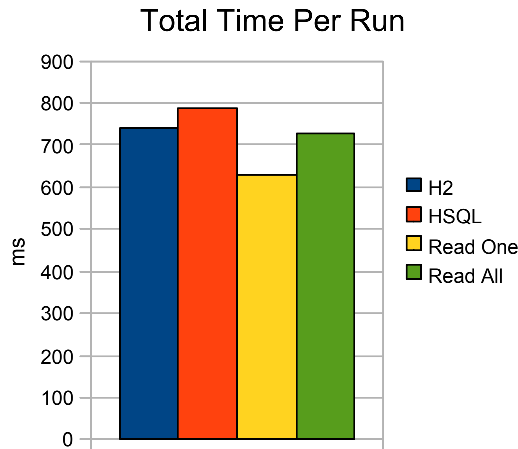


Figure 6.5: Average response time per run in high stress mode.

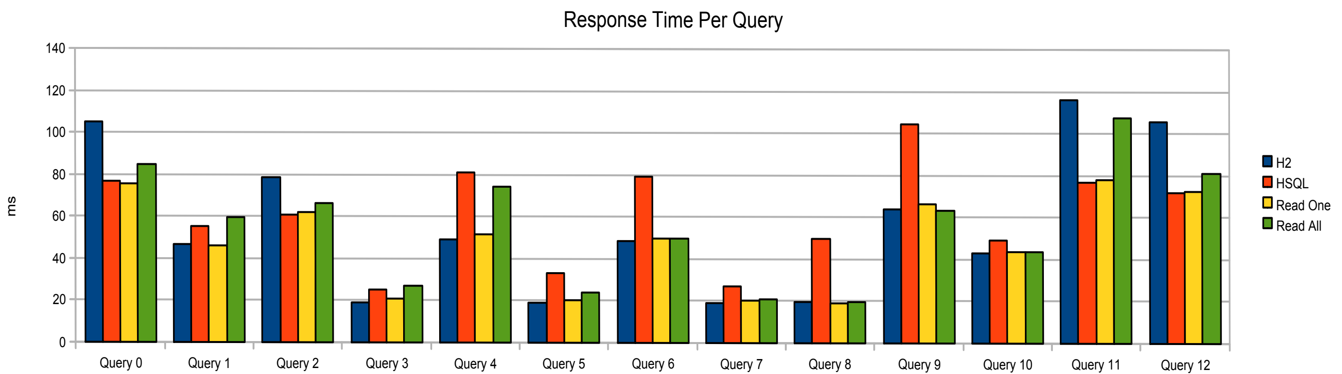


Figure 6.6: Average response time per query in high stress mode.

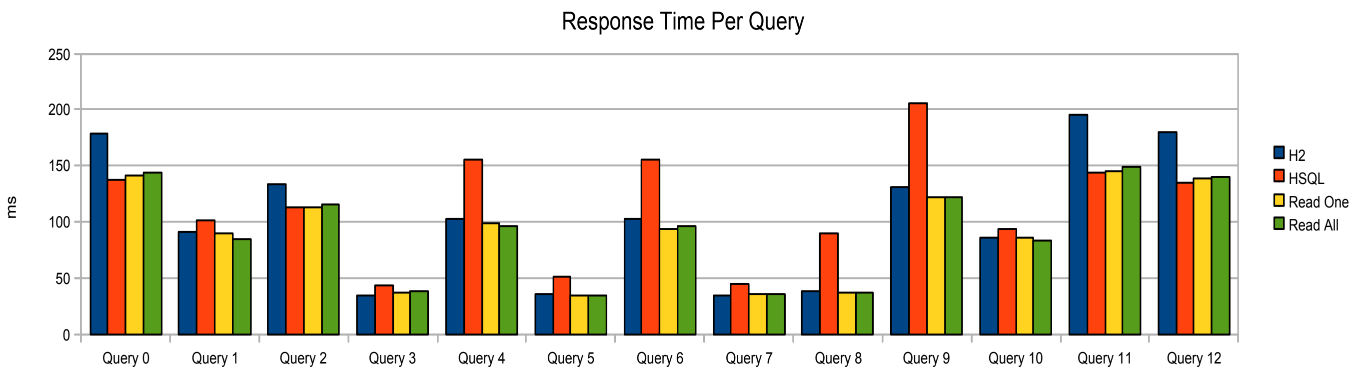


Figure 6.7: Average response time per query in low stress mode.

read-all internal scheduling mode still shows performance improvements over HSQL, with values close to the ones shown by H2. Read-one internal scheduling presents the best performance in this setting, with speedups of 17% over H2 and 26% over HSQL.

Figures 6.7 and 6.6 show the average execution times for individual queries in each

run for both low and high stress settings. In low stress mode, both read-all and read-one have individual performances similar to the best value between H2 and HSQL for all queries. Read-one continues exhibiting the best performance of the two database engines in high stress situations, but the same is not true for read one. In queries 1, 3, 4, and 11 especially, it is noticeable that the performance is closer to the worst case than the best.

6.1.4 Analysis

6.1.4.1 Low Stress

Both read-all and read-one internal scheduling modes show a considerable improvement over the simple implementations in the low stress setting. The performance for the macro-component is very close, with only slight deviations, to the performance of the best implementation on all operations in the sequence and consequently for the total run time. The pure parallel processing strategy of the read-all mode seems to introduce a slightly larger overhead than read-one on some queries. However, given the coarse granularity of the operations this becomes negligible.

Using the expected speedup calculations detailed in section 4.3.3, with the average values for H2 and HSQL tests, the speedup numbers obtained for the macro-component in this setting are approximately 13% in relation to H2, and 25% for HSQL. These numbers were reached by both the read-all and read-one macro-component implementations.

As expected, the read-all and read-one internal scheduling modes present very similar performances. Nevertheless, read-all still uses significantly more processing power to obtain similar performance. For this reason, although slightly more problematic to implement, read-one presents an advantage over read-all even in low-stress environments.

6.1.4.2 High Stress

Unlike the low stress situation, in high stress it is possible for different operations to overlap in the macro-component. This means that the average response time numbers in figure 6.6 cannot be analysed independently. This is due to the possibility of overlapping operations on the macro-component, causing read-all to suffer from the "busy replica" effect explained in section 4.3.1. This effect is the reason for the decline in performance of the read-all internal scheduling mode in this setting.

For example, the performance for the read-all macro-component on query 11 is closer to H2's than HSQL. This is undesirable, as HSQL has a much better performance

than H2 for this particular query. The answer as to why this happens can be found by looking back to queries 7 through 10. The HSQL replica has a significantly worse performance for these queries, but still process them due to the internal scheduling mode used. By the time the macro-component receives query 11 from the application, the HSQL replica is still processing the previous query, and unable to immediately begin execution of the newly received operation. This creates an added delay to the processing time of query 11, leading to the performance drop seen in the graph.

In fact, for this type of high stress situation, read-all internal scheduling will typically present a performance roughly equal to the *best* replica for the operation sequence. Any performance gains over the best implementation will come from the operation skipping optimization, or the possible advantage of unstable operation running times. The total times per run observed in figure 6.5 confirms this. For this test, the performance of read-all is very close to the single H2 implementation, with a slight improvement caused by the operation skip optimization.

6.2 Optimized Runtime Evaluation

This section presents the evaluation of the optimized runtime system using fine-granularity operations. Section 6.2.1.1 introduces the macro-component built for this evaluation. Section 6.2.1.2 explains the custom benchmark used to evaluate the optimized runtime system. Section 6.2.3 presents the results obtained. Section 6.2.4 concludes the chapter by discussing the results obtained.

6.2.1 Benchmark

6.2.1.1 Data Model

The component used for this fine-granularity test is based on the *optimal complexity Set* data structure described in section 3.1.1. This component implements 4 operations: an insert, which inserts a single element into the set if it is not already present; a removal, which removes an element from the set; an inclusion check, which verifies if an element is present in the data structure; and an ordered listing, which returns a partial ordered list of the elements contained in the set (in ascending order). For this implementation, the elements in the set were not generic, but integers. Figure 6.8 presents the interface for this Set.

The implementations of this interface used as replicas are based on the Java HashSet (based on a hash table) and TreeSet (based on a binary search tree, BST) classes. The

```
public interface OptSet {

    void insert(int element);

    void remove(int element);

    boolean contains(int element);

    int[] getOrdered();

}
```

Figure 6.8: Interface for Set component.

HashSet based implementation is generally more efficient for the *contains*, *insert* and *remove* operations, while the TreeSet performs better on *getOrdered*. This information is used to implement a read-one internal scheduling strategy for this component – all getOrdered operations are executed on the TreeSet replica and all contains on the HashSet replica.

6.2.1.2 Execution Model

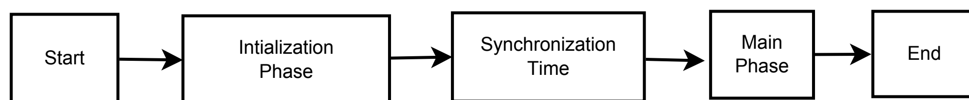


Figure 6.9: Fine granularity benchmark overview.

Figure 6.9 presents an overview of the benchmark process used for this evaluation. The benchmark begins with an initialization phase, where the data structure is initialized with pseudo-randomly generated data up to a parametrisable size (number of elements present in the structure). Following that, there is a small waiting period, used to allow the macro-component replicas to synchronize, so that when the main benchmark phase begins none of the replicas have pending updates to be processed. The main benchmark phase is where the performance evaluation takes place, as detailed in the next section.

6.2.1.3 Main Phase

The main phase of the benchmark simulates a parametrisable workload of mixed read and update operations as is common, for example, in transactional memory micro-benchmarks. The parameters that control this workload are the number of operations to be executed, the update ratio, and the inclusion check fraction.

The update ratio defines the percentage of operations in the workload that are updates (insert/remove). The remaining operations are reads (contains/lists). An update can be either an insert or removal operation, with a 50% chance each. Insertions are done by generating a random integer. As any integer can be generated, it is unlikely for an insert to fail except for very large initial data structure sizes (i.e. approximately 0.002% chance to fail on a data structure with 100.000 elements). Removals are done by keeping track of the values currently in the data structure and removing one of these values, so removals never fail. As inserts are unlikely to fail and removes never do, the data structure will maintain a size close to the initial size during the execution.

The inclusion check fraction controls the number of inclusion checks and ordered listings in the read-operations of the workload. For example, an update ratio of 0.3 with a workload of 100.000 translates into approximately 30.000 updates operations (30%), and 70.000 reads (70%). With an inclusion check fraction of 0.5, these 70.000 reads would generate approximately 35.000 inclusion checks (50%) and 35.000 ordered listings.

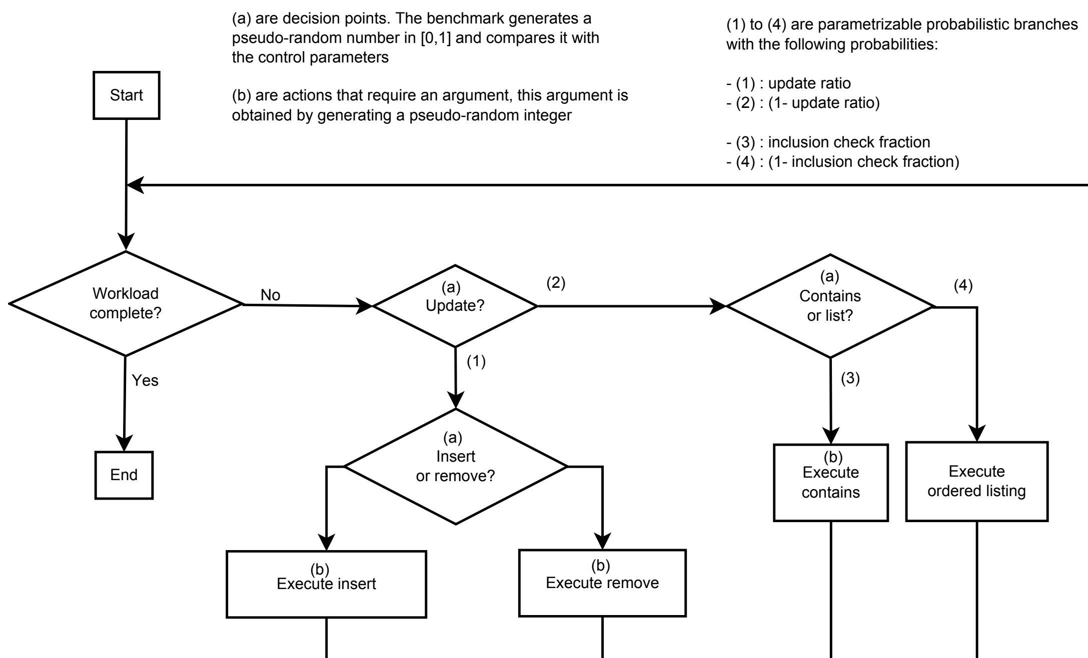


Figure 6.10: Fine granularity benchmark main phase.

Figure 6.10 presents a flowchart detailing the main phase of the benchmark. Notice that there is no time given for replicas to synchronize at any point during the workload, making this similar to the *high stress* situation of the base system benchmark.

6.2.2 Experimental Setup

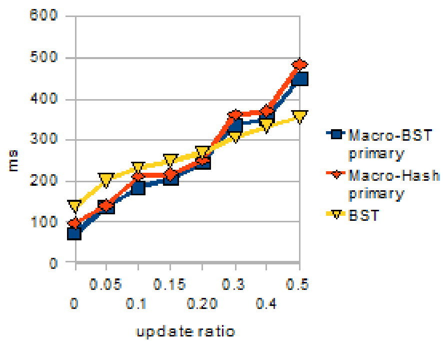
All tests were performed on a Sun Fire X4600 M2 x86-64 server machine, with eight dual-core AMD Opteron Model 8220 1.0 GHz processors and 32 GByte of RAM running Debian 5 (Lenny) operating system with the 2.6.26-2-amd64 kernel. The Java runtime environment used was OpenJDK Runtime Environment (build 1.6.0_0-b11).

In the results, we show different set ups by varying the control parameters of the benchmark. The only constant parameter for these tests is the size of the workload which consists of 100.000 operations. Results present the average time taken by the benchmark's main phase and are the average of at least 3 independent executions.

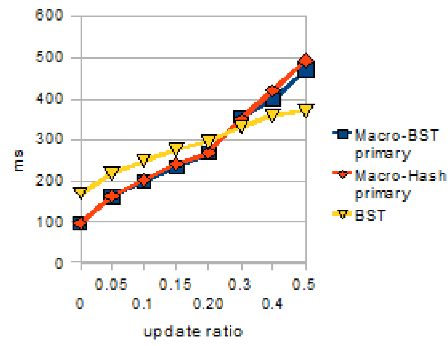
Results present three different data structures: Binary Search Tree (BST), which is the Set implementation using a Java TreeSet class; Macro-BST primary, the macro-component using the BST implementation as the primary replica and the HashSet as secondary; and Macro-Hash primary, where an HashSet based implementation is used as primary with a BST secondary.

6.2.3 Results

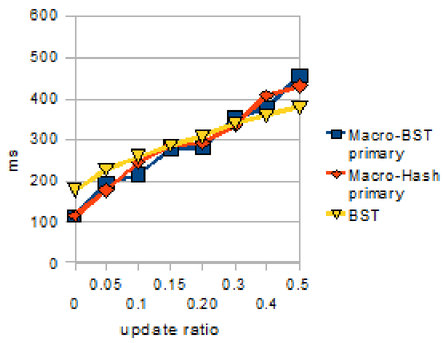
Figure 6.11 presents the results for the benchmark with an initial data structure size of 10.000 elements and with varying update and inclusion check ratios. Figures 6.12 and 6.13 present the results of the same tests for 55.000 and 100.000 elements respectively.



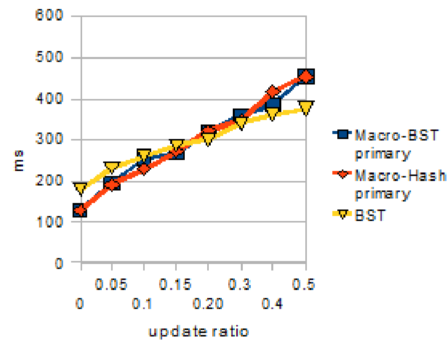
(a) 100% inclusion check fraction



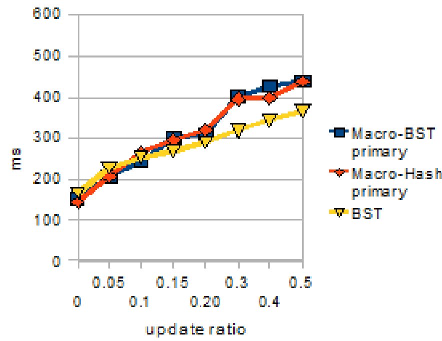
(b) 70% inclusion check fraction



(c) 50% inclusion check fraction

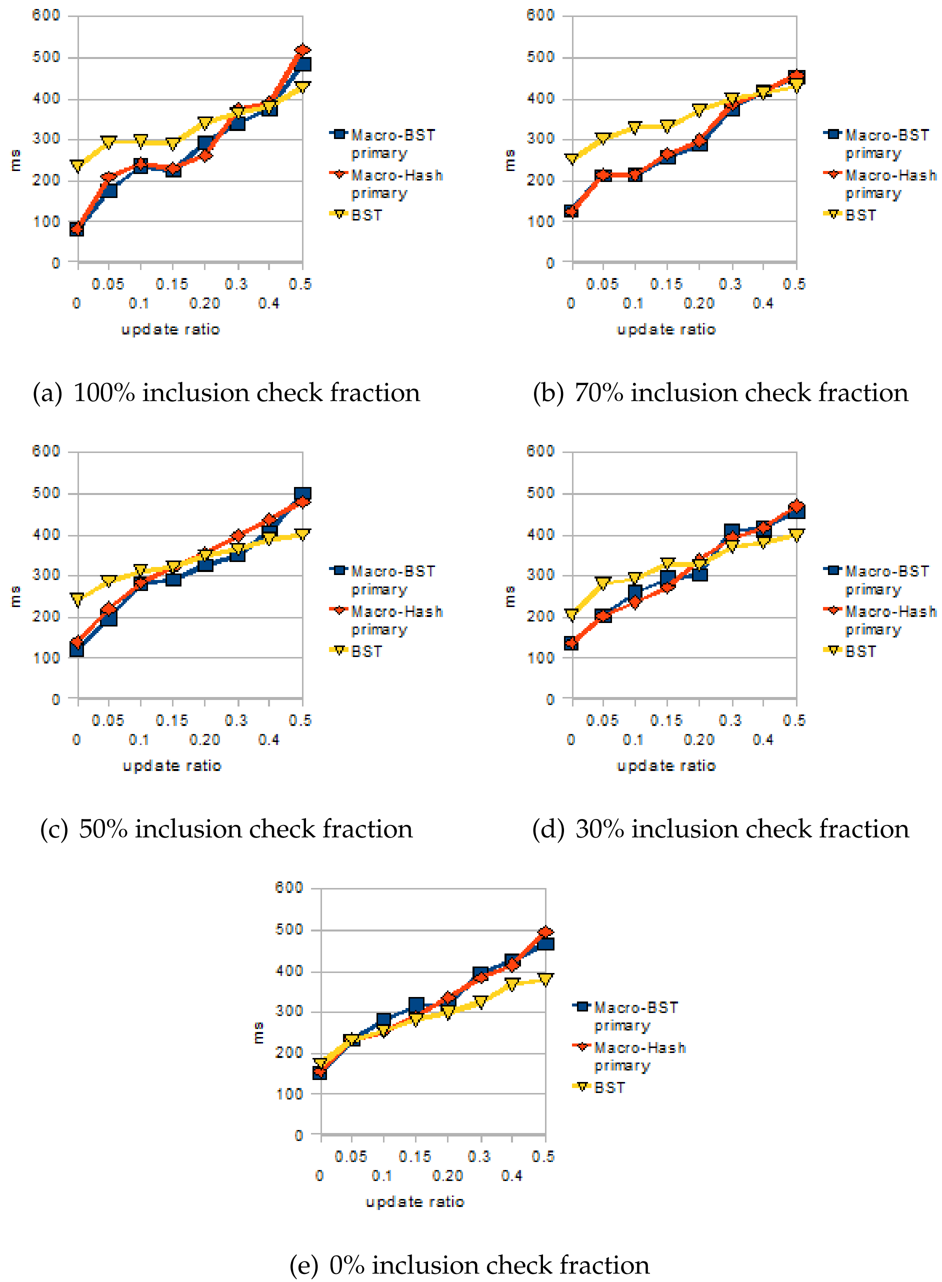


(d) 30% inclusion check fraction



(e) 0% inclusion check fraction

Figure 6.11: Average times for benchmark's main phase with 10.000 elements initial size.



(a) 100% inclusion check fraction

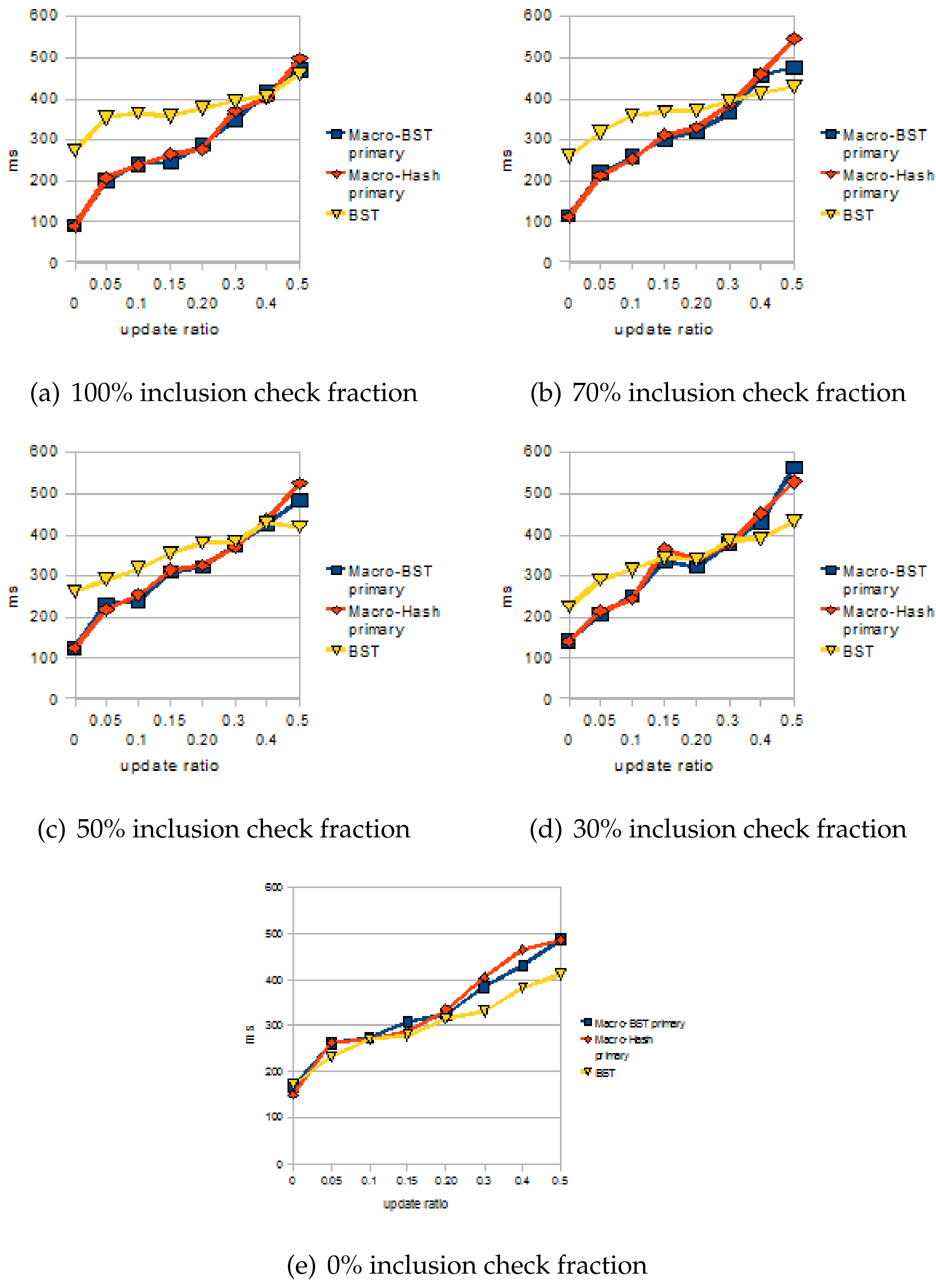
(b) 70% inclusion check fraction

(c) 50% inclusion check fraction

(d) 30% inclusion check fraction

(e) 0% inclusion check fraction

Figure 6.12: Average times for benchmark's main phase with 55.000 elements initial size.



(a) 100% inclusion check fraction

(b) 70% inclusion check fraction

(c) 50% inclusion check fraction

(d) 30% inclusion check fraction

(e) 0% inclusion check fraction

Figure 6.13: Average times for benchmark's main phase with 100.000 elements initial size.

6.2.4 Analysis

As can be seen by nearly all results (figures 6.11, 6.12 and 6.13) the performance difference between the primary replica choices seem negligible for this data structure, with both primaries showing similar performance and scalability up to 50% updates. This can be explained by two details. First, the running time of the insert/remove operations on both replicas is both very small, and not too different. Second, between

each operation there is some processing done by the benchmark in generating pseudo-random numbers and maintaining support data structures which slightly delays the start of the next operation on the macro-component. This slight delay combined with the low execution time of the update operations can mean that by the time the next method is called on the macro-component, both replicas are already in sync, and therefore the main performance gain of using one primary over the other becomes less relevant.

The data structure macro-component always processes inclusion checks in the HashSet based replica and ordered listings in the BST replica. For this reason, the tests done with 100% and 0% inclusion check fraction parameters represent the best and worst case respectively for the use of macro-components. For the best case, the full inclusion check test (figures 6.11(a), 6.12(a) and 6.13(a)), this means the macro-component always uses an implementation that is more efficient for the read operations than the simple BST component. For the worst case, (figures 6.11(e), 6.12(e), and 6.13(e)), the macro-component processes all reads using an implementation that is equal to the simple BST component. Consequently, there are no direct performance gains for read operations but the macro-component will still impose a performance overhead.

Looking at the concrete results for the best case in figures 6.11(a), 6.12(a) and 6.13(a) we can see that, as expected, the macro-component has a large performance advantage over the BST implementation. This is true even when the HashSet replica is the one being updated on background. For these tests, the macro-component shows an improvement in performance for up to 20% updates in the workload for 10.000 elements, and up to 40% updates for 100.000. When over those points in update ratio, the performance gains in read operations does not compensate the overhead imposed by the macro-component in update operations (which must be executed in all replicas).

As for the worst case results in figures 6.11(e), 6.12(e), and 6.13(e) we can see that the macro-component performance is roughly equal to the BST implementation when there are no updates on the data structure (and, therefore, the overhead of the background execution and coordination is avoided), but as the amount of updates increases the overhead begins to be noticeable. Nonetheless, up to the 10% update mark the performance of the macro-component is still comparable to the BST implementation.

The remaining test cases represent different balances of workloads between the best and worst cases and show no large differences from the expected result. For these, the macro-component seems to have hold a better performance for workloads close to the best case (figures 6.11(b), 6.12(b), 6.13(b)) and slowly declining as we move closest to the worst case (figures 6.11(c), 6.12(c), 6.13(c) and 6.11(d), 6.12(d), 6.13(d)).

The number of elements in the data structure also has a deep impact in the performance of the macro-component. The performance improvement being relatively small even in the best cases for small data structures (figures 6.11(a) and 6.11(b)), but much more significant in larger ones (figures 6.13(a) and 6.13(b)). There are two reasons for this effect. First, the coarser granularity of the operations in the data structure with more elements mitigates part of the overhead of the macro-component. Second, in the case of the data structure with less elements, the performance differences between the operations are not significant enough to obtain higher performance increases with the use of the macro-component.

Conclusions and Future Work

7.1 Conclusions

The RepComp project, in which this work is included, aims to bring replication techniques to multi-core environments with the purposes of improving software performance and fault tolerance. This dissertation focused on the design of a runtime system for transparently improving component performance using a new abstraction: the macro-component. These macro-components are based on the replication of heterogeneous software components with well-defined interfaces but different performances for different operations. The performance differences, along with the parallel processing capabilities of modern multi-core architectures allow for transparent improvement of performance by the macro-components over simple implementations.

This dissertation proposes an initial design of a macro-component runtime for improving software performance. In this context, it also presents a study of the major technical challenges posed by the design of the system.

An initial design for the runtime system is proposed and implemented. This design provides general support for the use of macro-components and is divided in two parts: a global scheduler and the individual macro-components.

An important aspect of this design is the scheduling mechanism which can be divided in internal and global. Internal scheduling is related with the decision of which replicas will process an operation. This is done internally by the macro-component

with no interaction by the support system and 3 strategies, tied to the handling of read-only operations, were proposed for this scheduling: *read-one*, when reads are executed by a single replica; *read-all*, when reads are executed by every replica, and *read-many*, when reads are executed by a subset of the available replicas.

For global scheduling, several scheduling strategies can be implemented – FIFO, priority classes, etc. In this work, a global scheduling strategy based on a group FIFO algorithm was implemented. This strategy allowed for the runtime system to coordinate the execution of operations from multiple macro-components with dependencies among them.

In addition to the base design and runtime system implementation, a second design with support for fine-granularity operations is proposed and implemented. This design improves on the base system by minimizing the overhead imposed for operations. This is done by running read operations inline on the application thread and changing the thread coordination points from the Java sleep/wait model to busy waiting.

As much of the macro-component's source code can be automated, a support tool was created to help with the implementation of these new macro-components. This tool can generate all of the code necessary to implement the individual macro-components with minimal programmer intervention, requiring only the interface of the component as input.

A prototype JDBC Driver macro-component was built, implementing a replicated database using two underlying replicas with the H2 and HSQLDB database engines. This is a complex component with multiple classes and dependencies between different objects and provides a range of coarse-granularity operations in the form of SQL queries for our initial tests.

The results of evaluation using this macro-component show that the base runtime achieves significant performance improvements over simple components in either low-stress and high-stress settings. The comparison of scheduling modes shows that for low stress situations both the read-all and read-one internal scheduling strategies showed similar performances. However, read-all still spends considerably more processing resources to achieve that performance, as operations as processed in parallel. On high stress situations, read-all drops in performance, making read-one preferable in these cases.

A prototype of a macro-component implementing a *Set* data structure, with two underlying implementations based on an hash table and a binary search tree, was also built. This macro-component includes only low-granularity operations.

The results of evaluation using this macro-component also show an improvement over a single implementation. With a read-only workload, the macro-component can

reach speedups of over 2. This value decreases with the ratio of updates in the workload – ranging from 30% to 50% improvements for low update ratios – but the macro-component still shows better performances with up to 40% updates in the best cases, while in the worst case it shows comparable performance with up to 10% updates.

In conclusion, the results obtained in this research were promising for further study in macro-components, both in improving performance and other uses such as fault-tolerance. The prototype implementations of both runtime systems and test macro-components were successful and proved to be superior in most cases to simple implementations. Additionally, we believe this research provided valuable insight on the general use of macro-components described in our model, both in advantages and the technical issues a complete system must address.

7.2 Contributions

During this work, we have provided the RepComp project with several important contributions. First, the proposed macro-component execution model and subsequent analysis, leading to the identification of challenges and possible mechanisms for both the performance and fault-tolerance oriented macro-componentes. Second, the design and implementation of two runtime systems, one general and one which attempts to address the challenge of fine-granularity operations, along with some initial scheduling strategies for these systems. Last, the evaluation of these systems with custom benchmarks, which allowed several important conclusions about the viability of macro-components in both fine and coarse-grain environments and the performance of different scheduling strategies.

7.3 Future Work

There are several directions which can be taken to continue this work: further evaluation of the existing system through more sophisticated benchmarks, extending this system or even usage of macro-components for other purposes.

7.3.1 Benchmarking

The development of a small scale benchmark application alongside with the macro-component system was highly useful. This allowed for focused testing of several macro-component characteristics leading to a greater understanding on many runtime details which would have been difficult using a standard database benchmark framework. However, due to its small scale, it is desirable that the prototype, or even a new

macro-component be tested on a real-world test case with well known applications or benchmarks to verify its behaviour in a large scale environment.

As the macro-component can be used as any other JDBC driver, it should require no significant changes on the application used for this test. However, many database engines accept non-standard SQL dialects for their queries. If the application is deeply connected to its underlying database, making use of these dialects, it is possible some changes to the queries are necessary to make them compatible with the database engines used as macro-component replicas for this validation.

7.3.2 Extending the existing system

Another aspect of future work is to improve on the current system, both in terms of design and its implementation. One especially pertinent aspect when it comes to extending the current base design is the handling of fine grained operations.

An approach that remains unexplored is the use of mechanisms to arbitrarily enlarge the granularity of the operations on the macro-component. In the benchmark, this is done at application level by grouping initialization queries into a single database query. However, it is possible to implement macro-component level mechanisms with similar effect, grouping fine grained operations in a single job and executing these in a batch fashion by a single executor. Even when there are no large groups of fine granularity operations, these can be merged into a job task of higher granularity.

If the batch size is large enough, the overhead imposed by the system should be negligible. However, this solution likely requires a constant stream of operations to guarantee batch groups are created efficiently. Otherwise, batches will either be too small, or the macro-component will have to significantly delay job submission to completely finish filling each batch.

On the supporting tools, the code generation tool is still in a very simplistic state, generating only read-all macro-components from Java interfaces. Subsequent implementations could support a wider variety of options, such as read-one macro-components and annotated interfaces. Another useful supporting tool addition is a component profiler which is capable of identifying the best implementations from a set for a given situation. This tool could be used by the automatic code generation tool to create read-one internal scheduling macro-components without explicit programmer intervention. Additionally, this is also a first step in the creation of self-configurating macro-components, which could dynamically adapt their internal scheduling to their current environment.

7.3.3 New directions

As discussed in chapter 3, macro-components can have uses other than performance increase. The most direct one is using the macro-components as fault-tolerant components. There are different ways this can be done, such as the usage of a pessimistic behaviour in regards to the results obtained by the macro-component. Instead of returning the first result obtained, the macro-component awaits for several, or all, of the results and returns the result in majority to provide some degree of fault tolerance. While this work can be used as a basis for these fault tolerant macro-components and some of the challenged encountered will be similar, the goals and technical issues faced may vary significantly.

Bibliography

[AEMGG+05] 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 Press.

[Avi75] A. Avižienis. Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing. In *Proceedings of the international conference on Reliable software*, pages 458–464. ACM, 1975.

[Avi85] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11(12):1491–1501, 1985.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BJH77] H.C. Baker Jr and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59. ACM, 1977.

[BZ07] L. Baugh and C. Zilles. An analysis of i/o and syscalls in critical sections and their implications for transactional memory. In *Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing, Portland, OR*. Citeseer, 2007.

[CES71] EG Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 173–186. USENIX Association, 1999.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 2002.

[Dij02] E.W. Dijkstra. *Cooperating sequential processes*. Springer-Verlag New York, Inc., 2002.

[EDP06a] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. In *EuroSys*, 2006.

[EDP06b] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *Proceedings of the 1st ACM EuroSys European Conference on Computer Systems 2006*, pages 117–130. ACM Press, 2006.

[EDZ07] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases. In *EuroSys*, 2007.

[EZP05] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84. IEEE Computer Society, 2005.

[Gee05] David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[GPSS03] Ilir Gashi, Peter T. Popov, Vladimir Stankovic, and Lorenzo Strigini. On designing dependable services with diverse off-the-shelf sql servers. In *WADS*, pages 191–214, 2003.

[GR93] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.

[HCU+07] T. Harris, A. Cristal, O.S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, pages 8–29, 2007.

[HHBB96] A.A. Helal, A.A. Heddaya, B.B. Bhargava, and B.K. Bhargava. *Replication techniques in distributed systems*. Kluwer Academic Pub, 1996.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[HM08] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan and Kaufmann, Burlington, 2008.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: speculative byzantine fault tolerance. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 45–58. ACM, 2007.

[KL86] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming *. *IEEE Transactions on Software Engineering*, 12:96–109, 1986.

[LS88] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM, 1988.

[NCF05] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 191–205, New York, NY, USA, 2005. ACM Press.

[NM92] R.H.B. Netzer and B.P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[NPCF08] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In Susan J. Eggers and James R. Larus, editors, *ASPLOS*, pages 308–318. ACM, 2008.

[OH05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[PA04] Christian Plattner and Gustavo Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174. Springer-Verlag New York, Inc., 2004.

[RCL01] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: using abstraction to improve fault tolerance. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 15–28. ACM Press, 2001.

[Rom00] A. Romanovsky. Faulty version recovery in object-oriented N-version programming. *IEE Proceedings-Software*, 147(3):81–90, 2000.

[RS03] M. Rabinovich and O. Spatscheck. Web Caching and Replication. *SIGMOD Record*, 32(4):107, 2003.

[SKSS02] Vijay Subramani, Rajkumar Kettimuthu, Srividya Srinivasan, and P. Sadayappan. Distributed job scheduling on computational grids using multiple simultaneous requests. *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, pages 359 – 366, November 2002.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[VBLM] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Database fault-tolerance with heterogeneous replication. Available at http://nms.csail.mit.edu/projects/hrdb/.

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *icdcs*, page 464. Published by the IEEE Computer Society, 2000.

[ZKN07] Lingli Zhang, Chandra Krintz, and Priya Nagpurkar. Language and virtual machine support for efficient fine-grained futures in java. In *PACT*

'07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, pages 130–139, Washington, DC, USA, 2007. IEEE Computer Society.