Software Component Replication for Improved Fault-tolerance: Can Multicore Processors Make It Work?[⋆]

João Soares, João Lourenço and Nuno Preguiça

CITI/DI-FCT-Univ. Nova de Lisboa

Abstract. Programs increasingly rely on the use of complex component libraries, such as in-memory databases. As any other software, these libraries have bugs that may lead to the application failure. In this work we revisit the idea of software component replication for masking software bugs in the context of multi-core systems. We propose a new abstraction: a *Macro-Component*. A *Macro-Component* is a software component that includes several internal replicas with diverse implementations to detect and mask bugs. By relying on modern multicores processing capacity it is possible to execute the same operation in multiple replicas concurrently, thus incurring in minimal overhead. Also, by exploring the multiple existent implementations of well-known interfaces, it is possible to use the idea without incurring in additional development cost.

1 Introduction

Despite the large number of techniques developed for detecting and correcting software bugs during development and testing phases, software bugs remain a major problem in production releases [13,4]. Software updates or patches, designed to correct existing bugs often end up introducing new bugs - studies show that up to 70% of patches are buggy [22].

Multicore processors have made a push for increased concurrency in applications, leading to an increase of concurrency related bugs. This problem is being actively investigated, with a large number of works addressing the subject in the last few years [19,14,10,16,4,23].

The increasing complexity of software has led programmers to build their applications relying on the (re)use of third party off-the-shelf libraries and components, such as in-memory databases and XML parsers. If some of these components undergo systematic quality control procedures, others are provided by communities that cannot afford such procedures. If in the former case components already include bugs [8], we can expect the situation to be far worse in the latter case. Thus, these components become an important source of bugs for applications. The situation is magnified by the fact that application programmers have little or no control over these components.

[⋆] This work was partially support by FCT/MCT projects PEst-OE/EEI/UI0527/2011 and PTDC/EIA-EIA/108963/2008. João Soares was partially supported by FCT/MCTES research grant # SFRH/ BD/ 62306/ 2009.

For dealing with faults caused by software bugs, several fault tolerance techniques have been proposed [17]. Some of these techniques improve software quality by relying on replication and redundancy techniques, normally combined with design diversity. The drawbacks of these approaches are an increase in development time and costs, since diverse solutions need to be designed, implemented and tested, and a compromise in application performance, due to result and state validation.

In this work we revisit the idea of software component replication for detecting and masking bugs, by proposing the *Macro-Component* abstraction. A *Macro-Component* is a software component that includes internally several diverse component replicas that implement the same interface. Assuming that different component replicas exhibit different bugs [3,6,7], by executing each operation in all replicas and comparing the obtained results, it is possible for a *Macro-Component* to detect and mask software bugs.

By exploring the power of multicore processors, the same operation can be concurrently executed in multiple replicas with minimal overhead. By exploring the multiple available implementations for the same standard interfaces, it is possible to create *Macro-Components* without incurring in additional development time or cost. This allows to put in practice the old idea of N-Version Programming [3] at the component level.

Although the idea seems simple, putting it to work involves a number of technical challenges that we explore in the remaining of this paper. In particular, we show how to minimize computational overhead by executing operations in as few replicas as possible and by minimizing the required number of coordination points among replicas. Our preliminary results suggest that this approach is promising, exhibiting acceptable performance. The results also show an important result for the practicality of the solution: the amount of memory used is not directly proportional to the number of replicas. The reason for this is that a large number of objects can be shared among the replicas - e.g. strings in database fields.

The remainder of this paper is organized as follows. The next section discusses related work. Section 3 introduces the *Macro-Component* model. Section 4 presents our current prototype, the implementation of Macro-Components for in-memory databases and presents some preliminary evaluation. Section 5 concludes the paper with some final remarks.

2 Related Work

Macro-Components share an identical model with n-Version Programming (NVP) [3,1]. Contrarily to the original NVP, *Macro-Components* work at the component granularity [18], taking advantage of third party components to minimize the impact in development time and costs. Additionally, unlike previous works, our design addresses modern multicore processors, which seem an suitable architecture to make software component replication work with minimum performance impact.

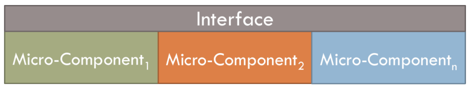


Fig. 1: Macro-Component

Replication has been a highly researched topic in distributed systems [9], with most of the proposed techniques addressing only fail stop faults. Byzantine fault tolerance techniques [12,20,2] have been proposed for dealing with other fault models. Some works, e.g. Eve [11], have been addressing replication in distributed settings with multicore machines. In our design, we re-use some of the ideas proposed in these works.

Most of the research on concurrency bugs has focused on techniques for finding and avoiding bugs [10,13,14,4,23]. Our work share some of the goals with these works, but differs from most of these approaches by relying on diverse replication.

Gashi et al. [6,7] have also focused on using third party components for improving fault tolerance in SQL Database Servers and Anti-Virus engines. Our proposal differs from these works, as it provides a generic framework and runtime support for producing fault tolerant components, based on diverse implementations of the same common interface.

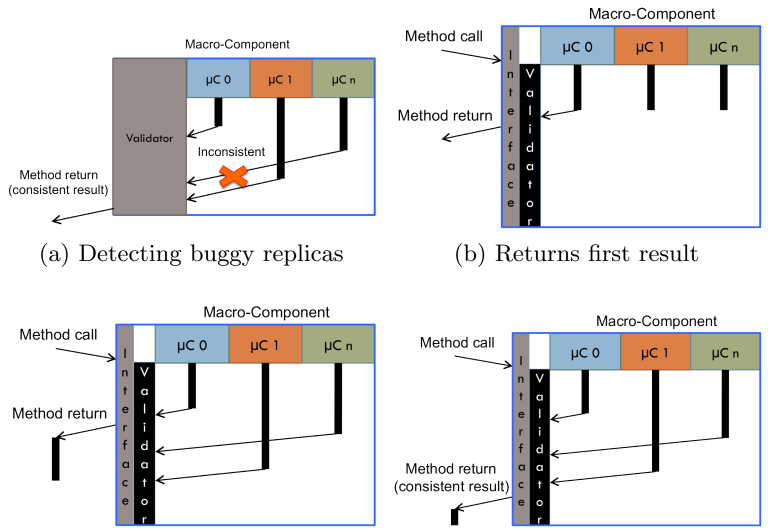
3 *Macro-Components*

A *Macro-Component* is a software component implemented using a set of diverse components (called replicas) that implement the same interface, as presented in figure 1. Diversity allows for each replica to have its own implementation while offering the same functionality and maintaining the same abstract state as all other replicas. This provides the means for *Macro-Components* to detect buggy behaviour of replicas, by identifying state or result divergences amongst replicas, thus preventing these bugs from being exposed and affecting the reliability of applications.

With this approach, an application can use a *Macro-Component* as it would use any other component. The only difference is that a *Macro-Component* has improved reliability. Thus, a single application may include a large number of *Macro-Components*.

Next, we detail how *Macro-Components* can be used to address several goals.

Detecting and Masking Bugs: Macro-Components, as NVP, follows the assumptions that different implementations incur in different bugs, and that a divergent result from the majority occurs due to the presence of bugs. Thus, to detect buggy behaviour, diversity in component replicas is crucial, and detection is achieved by comparing the results from the several replicas. Whenever a method is invoked on a *Macro-Component*, the following steps (illustrated in figure 2a)



(a) Detecting buggy replicas (b) Returns first result

(c) Background validation (d) Resuming execution

Fig. 2: *Macro-Component* Method Calls

occur: *i*) the corresponding method is invoked on all component replicas; *ii*) replicas execute the same method concurrently; *iii*) wait for $f+1$, i.e., the majority, equal results from component replicas; and *iv*) return the result from the majority of replicas.

For keeping the overhead low, unlike solutions that validate both results and object state [24,21], *Macro-Components* detect buggy behaviour primarily by validating results, while object state is compared periodically in background. Whenever inconsistent results are detected, the corresponding faulty replicas are marked for recovery, and temporarily removed from the set of active replicas. Also, if some replica is unable to produce a result within a certain time limit, the replica is considered faulty and threads executing in the replica are aborted. The time limit is defined by the time taken by the majority of the replicas to reply plus an additional tolerance.

Detecting Concurrency Bugs: Macro-Components are not restricted, in any way, to the use of diverse replicas. When using homogenous replicas, Macro-Components can still be used for detecting and masking concurrency bugs.

To this end, the following approaches are possible. First, the imposed overhead due to the *Macro-Component* runtime may result in different inter-leavings of concurrent operations in different replicas. Second, it is possible to impose random delays on the method execution in different replicas, thus leading to different inter-leavings. Third, it is possible to execute method invocations sequentially in some of the replicas (as in [5]).

For minimizing the overhead, the latter two solutions can be used only when some problem is detected by running operations in the default mode. Additionally, sampling can be used when the latter two approaches are being used.

4 Implementation and Runtime

We are currently building a system for supporting applications that use Macro-Components, in Java. In this section, we present our current prototype.

A Macro-Component is composed by three main components: the *manager*, responsible for coordinating method execution on the replicas, the *validator*, responsible for validating the results returned by the replicas, and the *replicas*, the components responsible for maintaining the state. Applications remain oblivious to the replicated nature of Macro-Components since it offers a single copy view of the underlying state. To this end, each replica maintains an associated *version*, that registers the number of updates performed on the replica. The *manager* guarantees that operations execute on replicas in the same state, i.e., with the same version. This version is kept in shared memory, as an atomic *counter*.

The supporting runtime guarantees that when a method call is performed on a Macro-Component, the equivalent method is concurrently executed in all replicas. To this end, method calls are recorded as tasks and queued for execution. These tasks represent the method to be executed, and the replica in which the method is to be executed on. For each replica, an associated thread is responsible for dequeuing assigned tasks and execute them on that replica. Our current prototype currently supports concurrent execution only for operations that do not modify the state of the Macro-Component - operations that modify the state of the Macro-Component are currently executed serially.

This decouples the execution of the callee from the method, i.e., the thread calling the method can be different from the thread that executes it. This allows Macro-Components to provide independent execution models, allowing methods to execute asynchronously from the application threads.

We currently support two execution model. The first, based on non-transparent speculation of results, provides improved performance. The second, based on a prior verification of execution correctness, allows for transparent replacement of components by their Macro-Components siblings.

In the speculative execution mode, a Macro-Component returns the result from the fastest replica (figure 2b), while validating the result on background (figure 2c). If the result is found to be incorrect, the execution must be cancelled and re-started with the correct value (figure 2d). This approach can even improve the performance over standard components, when there are no faults, by exploring the differences in performances for the different replicas.

We currently do not support automatic transparent speculation. Thus, the Macro-Component notifies an error on a previous call when some method is called or when the application queries the Macro-Component for errors. This requires the application to be modified to include support for such calls. In general, this is not too complex as the verification calls can be added in the

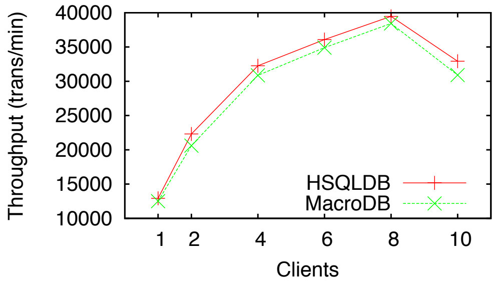


Fig. 3: TPC-C results

	Replicas		
MacroDB	2	3	4
HSQL	1.64×	2.29×	2.46×

Fig. 4: Memory overhead

end of methods that use Macro-Components (or before some externalization of results is done).

In the prior-verification model, the results of a method is only returned when a majority of the replicas has returned the same result.

4.1 Database Macro-Component

We now describe the design and implementation of *MacroDB*, a *Macro-Component* for in-memory database systems. *MacroDB* is composed by a set of database replicas, each potentially supported by a different in-memory database engine. Applications remain oblivious of the replicated nature of the system since it offers them a standard JDBC interface, and standard transaction isolation levels.

Applications do not communicate directly with the database engines, instead they communicate with the *manager*, a JDBC compliant front-end which coordinates client operations in the underlying replicas. The *manager* receives statements from clients and forwards them, without modification, to the replicas, guaranteeing their ordered execution by the runtime support system. For statements inside a transaction, the first result is returned to the application while it is compared in background with the results from other replicas. Additionally, (read-only) queries execute initially only on $f + 1$ replicas (with f the number of replicas that can be faulty) - the queries are only executed in other replicas if returned results differ. When the application wants to commit a transaction, if there has been any error detected on the previously returned results, the commit fails. Otherwise, the commit executes is all replicas and returns to the application after it is confirmed. This approach combines the speculative and prior-verification execution models in a way that is transparent to database applications.

MacroDB is still under active development, missing the code for verifying state divergence and the wrappers to support the small differences in multiple database engines [20]. Even though, our solution is already operational with an homogeneous configuration, with all replicas running the same database engine. To provide an approximate value on the overhead that our runtime incurs for providing fault-tolerance, we ran the TPC-C benchmark on the Macro-Component, and compared the obtained results against the standalone database version. In all cases, the HSQLDB in-memory database was used. The results, presented on figure 3, show a small overhead for *MacroDB* version, averaging a 4% decrease in performance.

Although these are preliminary results, we expect that relying on multiple database engines can improve this overhead, as the result from the fastest replica will be returned in each case. On the other hand, the performance will be penalized by checking for the difference in the database state.

As an additional test, we also measured the memory overhead imposed by replicating the database. Contrarily to what was expected, the memory overhead is not proportional to the number of replicas, as presented in figure 4. This is due to the fact that replicas share immutable Java objects, such as Strings. The obtained results show that, a MacroDB configured with HSQL replicas, uses at most *2.5 times* more memory than the standalone engine, when using a 4 replica configuration. This makes deploying MacroDB practical on single machine multicores, even with large numbers of replicas.

5 Final remarks

In this paper we revisited software component replication techniques, presenting a new abstraction for improving software fault tolerance, called *Macro-Component*. *Macro-Components* put in practice the old idea of N-Version Programming [3] at the component level. Existing software products can benefit from improved fault tolerance, simply exchanging components by their *Macro-Component* siblings, preventing developers from rewriting code, and preserving development methodologies.

We have presented the design of a system that supports the use of *Macro-Components* in applications. Our design focused on keeping the overhead low, by minimizing the overhead of computational resources by executing operations in as few replicas as possible and by minimizing the required number of coordination points among replicas. Our preliminary results suggest that this approach is promising, exhibiting acceptable performance. The results also show an important result for the practicality of the solution: the amount of memory used is not directly proportional to the number of replicas.

Our current prototype still misses some important features, namely result and state comparison, and improved recovery of replicas. We are currently conducting additional experiments to evaluate our prototype with standard benchmarks.

References

1. A. Avizienis, *The n-version approach to fault-tolerant software*, IEEE Trans. Softw. Eng. **11** (1985), no. 12, 1491–1501.
2. Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa, *Depsky: dependable and secure storage in a cloud-of-clouds*, In EuroSys '11, 2011, pp. 31–46.
3. Liming Chen and Algirdas Avizienis, *N-version programming: A fault-tolerance approach to reliability of software operation*, In Proc. FTCS-8, 1978, pp. 3–9.

4. P. Fonseca, Cheng Li, V. Singhal, and R. Rodrigues, *A study of the internal and external effects of concurrency bugs*, In DSN '10, 2010, pp. 221 –230.
5. Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues, *Finding complex concurrency bugs in large multi-threaded applications*, In EuroSys '11, 2011.
6. Ilir Gashi, Peter T. Popov, Vladimir Stankovic, and Lorenzo Strigini, *On designing dependable services with diverse off-the-shelf sql servers.*, In WADS '03, 2003, pp. 191–214.
7. Ilir Gashi, Vladimir Stankovic, Corrado Leita, and Olivier Thonnard, *An experimental study of diversity with off-the-shelf antivirus engines*, In NCA '09, 2009, pp. 4–11.
8. Sudipto Ghosh and John L. Kelly, *Bytecode fault injection for java software*, Journal of Systems and Software **81** (2008), no. 11, 2034 – 2043.
9. Abdelsalam A. Helal, Bharat K. Bhargava, and Abdelsalam A. Heddaya, *Replication techniques in distributed systems*, Kluwer Academic Publishers, 1996.
10. Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea, *Deadlock immunity: Enabling systems to defend against deadlocks*, In OSDI '08, 2008.
11. Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin, *All about eve: execute-verify replication for multi-core servers*, in OSDI'12, 2012, pp. 237–250.
12. Leslie Lamport, Robert Shostak, and Marshall Pease, *The byzantine generals problem*, ACM Trans. Program. Lang. Syst. **4** (1982), no. 3, 382–401.
13. Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai, *Have things changed now?: an empirical study of bug characteristics in modern open source software*, in Proc. ASID '06, 2006, pp. 25–33.
14. Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou, *Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs*, in SOSP '07, 2007, pp. 103–116.
15. Paulo Mariano, João Soares, and N. Preguiça, *Replicated software components for improved performance*, in InForum 2010, 2010, pp. 95–98.
16. Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu, *Finding and reproducing heisenbugs in concurrent programs*, in OSDI'08, 2008, pp. 267–280.
17. Laura L. Pullum, *Software fault tolerance techniques and implementation*, Artech House, Inc., USA, 2001.
18. James M. Purtilo and Pankaj Jalote, *An environment for developing fault-tolerant software*, IEEE Trans. Softw. Eng. **17** (1991), 153–159.
19. Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou, *Rx: treating bugs as allergies—a safe method to survive software failures*, in SOSP '05, 2005, pp. 235–248.
20. Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov, *Base: using abstraction to improve fault tolerance*, in SOSP '01, 2001, pp. 15–28.
21. Alexander Romanovsky, *Class diversity support in object-oriented languages*, Journal of Systems and Software **48** (1999), no. 1, 43 – 57.
22. Stelios Sidiroglou, Sotiris Ioannidis, and Angelos D. Keromytis, *Band-aid patching*, in HotDep '07, 2007.
23. Kaushik Veeraraghavan, Peter Chen, Jason Flinn, and Satish Narayanasamy, *Detecting and surviving data races using complementary schedules*, in SOSP '11, 2011, pp. 369–384.
24. J. Xu, B. Randell, C. Rubira-Calsavara, and R.J. Stroud, *Toward an object-oriented approach to software fault tolerance*, in Proc. FTPDS '94, 1994, pp. 226 –233.