Replicated Software Components for Improved Performance[⋆]

Paulo Mariano, João Soares and Nuno Preguiça

CITI / Dep. de Informática - Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa,
Quinta da Torre, 2829 -516 Caparica, Portugal

Abstract. In recent years, CPU evolution has shifted from the continuous increase of speed to an increase in the number of processing cores. In this paper, we propose to take advantage of the new multicore systems, by diverse replication of software components. This approach, complementary to other directions that are being tackled, attempts to obtain a better performance for a *macro-component* consisting of several diverse implementations of the same specification, by returning, for each operation, the result from the replica that is faster for that operation. We present an early design of a system that implements this approach.

Keywords: replication, parallel programming, multicore systems

1 Introduction

Until recently, CPUs evolution was a mix of improved functionality and a steadily increase of clock speed. However, this path of evolution have reached its end, with an increasing difficulty on further increasing clock speed [4]. Currently, hardware manufacturers are deploying CPUs with an increasing number of processing cores. With multicore CPUs, programs must include multiple concurrent threads of activity to take benefit from the multiple cores available.

Previous experience in the field of concurrent/parallel programming shows that creating such applications is a highly demanding task even for experienced programmers. This problem has led to an intense research activity for finding good abstractions for expressing parallel computations and to build suitable runtime support [2,5,3] that simplifies this task in multicore environments.

In this paper, we propose a complementary approach that can be used by both applications that include multiple threads and by applications that include a single thread. The main insight for our approach is that applications almost always resort on a set of components with standard interfaces - e.g. data structures, algorithms, etc. For these components, several implementations are available, which have different performance for different inputs or for different

[⋆] This work was partially supported by FCT/MCTES, project #PTDC/EIA/74325/2006 and #PTDC/EIA-EIA/108963/2008 and CITI.

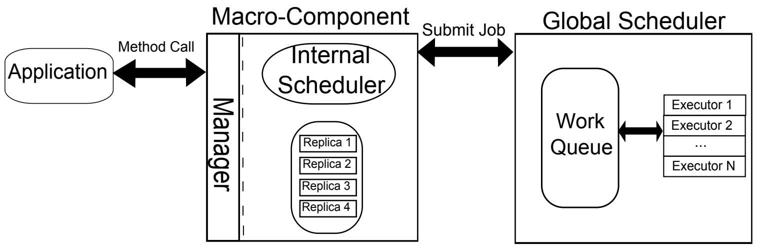

Fig. 1. Macro-component system model.

operations. Thus, we propose to locally replicate these components using different underlying implementations, building a *macro-component*.

This basic macro-component concept can be used for two different purposes. Reliability, by masking software bugs, similar to n-version programming [1] but on a smaller scale. Or improved performance, by returning the result obtained by the fastest replica. In this case, an operation would be executed in the underlying replicas concurrently and the first result returned by these would be the result returned by the macro-component.

Although the idea seems simple, making it work in practice has a number of technical challenges that must be addressed, including performance issues, coordination among macro-components in an application, etc. In this paper we present our initial design for building and supporting *macro-components*.

The remaining of this paper is organized as follows: Section 2 describes our initial design and Section 3 concludes the paper with some final remarks.

2 Design

In this section we present the initial design for a macro-component runtime system. To simplify this initial design a number of assumptions are made about the macro-component underlying micro-components (or replicas). First, it is assumed that the replicas do not fail arbitrarily. Second, they must be self contained. Implementations are not allowed to interact with their exterior through any means other than operation results. Third, operations must be deterministic.

In figure 1, we present the 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 of a manager, which provides the interface of the implemented specification to the exterior, a set of underlying replicas and an internal scheduler. It is not necessary to have any knowledge on the implementation details of the underlying replicas as long as they all implement the same specification. The global scheduler keeps track of the several execution jobs in a work queue and schedules them for execution by a pool of *executor* threads in a producer-consumer scheme. More details on the purpose and functioning of both the internal schedulers and the global scheduler can be found in section 2.1.

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.

2.1 Scheduling

The scheduling of operations in a macro-component environment takes two forms in the proposed model, *internal* scheduling and *global* scheduling. Internal scheduling is done internally in the macro-component and consists on the creation of *jobs* to be submitted to the global scheduler. These jobs consist on the execution of an operation on a given replica. As such, the internal scheduler is responsible to decide which replicas will run an operation. Global scheduling, on the other hand, handles the choice of which *job* to execute at any given moment.

Internal Scheduling As replica states must be maintained coherent, operations which update this state must be executed everywhere. However, for read operations this is not required. In our prototype we have implemented the following three internal scheduling strategies based on this property.

Read-all Reads are executed in all replicas and the result returned will be that of the replica that finishes processing first. If the macro-component experiences a light load, this strategy ensures the best performance for all operations. However, stress tests reveal a problem for this strategy as replicas can be held back by unnecessary slow operations instead of useful work.

Read-one With read-one, the macro-component directs a read operation to a single replica. This replica is, hopefully, the one with the best performance for this operation type. The issue with this strategy is how can we predict which replica best fits an operation.

Read-multiple A compromise between read-all and read-one modes. An operation is assigned to a *subset* of the replicas.

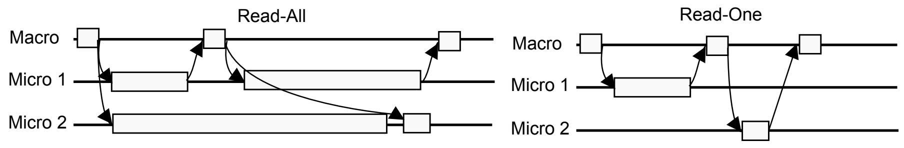

Fig. 2. Read-all vs read-one example timeline.

Global Scheduling This scheduler works as a distributor of jobs for the executors. Various scheduling strategies can be implemented, but some concerns must be addressed in order to maintain proper functionality. First, the relative order of update operations must be preserved. This means that update operations must be executed in the same order on all replicas. If these operations are executed out of order, macro-component replicas' internal state may diverge, leading to incorrect or unexpected results. Also, while read operations can be reordered they must be executed in a state that reflect all previous updates.

For simple independent macro-components, these ordering rules can be enforced for each macro-component independently, as there is no relationship between the replicas. In more complex cases, this independence may not hold due to relationships between macro-components. For example, in a macro-component based JDBC driver, the Connection macro-component cannot be allowed to commit before all previous operations on its Statements are finished.

3 Final Remarks

In this paper, we introduce the concept of macro-component, a software component that combines several different implementations of a given component specification. We propose the use of macro-components as a mechanism to improve the performance of applications in multicore systems and present an initial design of a runtime system to support this concept.

The initial evaluation with an early prototype of our system shows that the runtime system imposes non-negligible overhead. However, even with a non-optimized implementation it was possible to obtain a better overall performance for a *macro-component* that implements an in-memory SQL database.

References

1. A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11(12):1491–1501, 1985.
2. J. Larus and C. Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, 2008.
3. E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In S. J. Eggers and J. R. Larus, editors, *ASPLOS*, pages 308–318. ACM, 2008.
4. K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.
5. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.