Towards Scalable Service Composition on Multicores

Daniele Bonetta, Achille Peternier, Cesare Pautasso, and Walter Binder

Faculty of Informatics, University of Lugano (USI)
via Buffi 13, 6900 Lugano, Switzerland
`first.lastname@usi.ch`

Abstract. The advent of modern multicore machines, comprising several chip multi-processors each offering multiple cores and often featuring a large shared cache, offers the opportunity to redesign the architecture of service composition engines in order to take full advantage of the underlying hardware resources. In this paper we introduce an innovative service composition engine architecture, which takes into account specific features of multicore machines while not being constrained to run on any particular processor architecture. Our preliminary performance evaluation results show that the system can scale to run thousands of concurrent business process instances per second.

1 Introduction

Service-oriented architectures promote the creation of new applications by orchestrating existing Web services by means of service composition languages [2]. Since compositions are themselves made accessible as Web services, composition runtime engines may have to handle a large number of concurrent service requests. Assuming that the composed services are designed to scale (e.g., they are hosted in a cloud environment), composition runtime engines can easily become performance bottlenecks. Existing engines rely on distribution and replication techniques in order to ensure scalability in peer to peer environments (e.g., OSIRIS [8]) or over clusters of computers (e.g., JOpera [6]). Other approaches (like Lu et al. [5]) propose an optimized architecture for service compositions based on event-driven patterns and message passing interactions.

Modern multicore machines offer a promising alternative to clusters or server farms, respectively allow to build a sufficiently powerful infrastructure with less machines. However, modern multicore architectures are fundamentally different from previous micro-processor architectures [4]. Since it has become difficult to further increase the clock rate of processors, nowadays chip manufacturers are delivering more processing power by increasing the number of cores per CPU. Recent chip multi-processors combine several cores with a hierarchy of caches on a single processor. Typically, each core has its own small L1 and L2 caches, while several or all cores on a chip share a larger L3 cache. Examples include Intel Nehalem, AMD Opteron, and IBM Power7 processors.

In order to take full advantage of the hardware resources on modern multicore machines, it becomes important to explicitly consider the characteristics of the multicore architectures in the design of the process execution engine.

In this paper we introduce the SOSOA process execution engine, an innovative service composition middleware based on a multicore-aware design. While we take into account the specifics of multiprocessor architectures in the design of process execution engines, we do not resort to any low-level implementation and optimization techniques. The resulting engine is thus platform-independent, but capable of adapting according to the actual hardware configuration.

The main contributions of this paper are to take emerging multicore architectures of modern processors into account for the design of process execution engines, and also to demonstrate the clear impact of multicore-awareness on their performance with some preliminary results.

The paper is organized as follows. Section 2 describes the main requirements and architectural characteristics of the process execution engine and how it has been designed to target multicore machines. Section 3 describes the evaluation testbed and presents the results of our first measurements. Section 4 concludes the paper and presents future research directions.

2 Architecture

This section gives an overview of the architecture of the SOSOA service composition engine and how it can adapt to run across different configurations of the underlying hardware resources.

2.1 Components

The logical architecture of the engine is designed following a multi-stage pipeline, comprising three components: the Request Handler, the Kernel, and the Invoker (Fig. 1). The Request Handler makes the composite Web services available to clients. The Kernel performs the actual execution of the processes and manages the state of multiple active instances of the running compositions. The Invoker takes care of interacting with the composed services.

The execution of a composition begins with a request from a client to instantiate a new instance (1). This request is forwarded by the Request Handler to a queue (2) which is read by the Kernel. The Kernel is in charge of retrieving pending requests from the queue (3) and then instantiating and executing the corresponding compositions, while keeping their state up-to-date. In order to interact with the composed Web services, the Kernel delegates the actual service invocations to the Invoker via a second queue (4).

The three components in the SOSOA process execution engine are decoupled using shared queues in order not to slow down the execution of compositions, due to the natural delay involved in the invocation of remote Web services. Once the Web service invocation completes (5), its results are enqueued by the Invoker into the queue shared with the Kernel, so that they can be used to

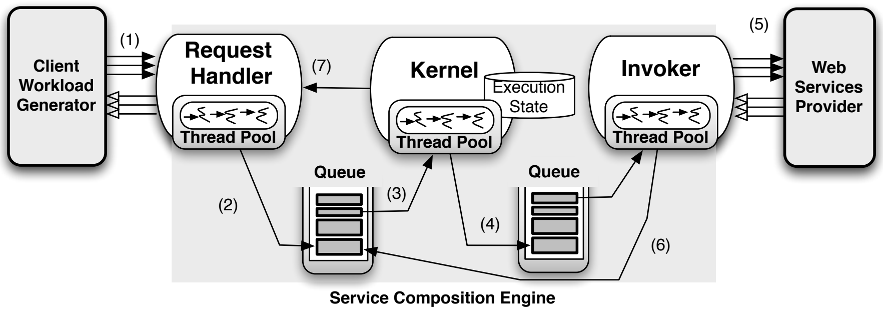

Fig. 1. Architecture of the SOSOA engine for Web service composition

continue the execution of the corresponding instance (6). Once the execution of an entire instance completes, the Kernel component notifies the Request Handler component which sends results to the client (7).

At this level of abstraction, the architecture does not yet define how its three execution stages are mapped to the available execution resources (i.e., OS threads). The goal is to define a scalable system architecture, where a limited number of operating system threads can be leveraged to execute a much larger number of composition instances. Thanks to the separation of the execution stage from the Web service publishing and invocation stages, this architecture makes it possible to use only three execution threads to run any number of process instances that may involve the parallel invocation of any number of Web services. Clearly, allocating at least one thread per component is necessary to make sure the system can operate and run its workload, but is not sufficient to provide an acceptable level of performance. If we need to implement parallel constructs commonly found in most service composition languages, we need to assign a larger number of threads to the Invoker component. Likewise, the Request Handler component needs a pool of worker threads to serve incoming concurrent requests from many clients. The same concerns also apply to the Kernel: as it acts as a bridge between two thread pools, it may become a performance bottleneck unless it can also rely on multiple threads to execute the composition instances.

This design thus adopts thread pools to assign more than one thread to each component. Thread pools not only have the potential to increase overall system throughput, but also provide a straightforward mechanism to leverage the underlying hardware parallelism [3]. In addition, thread pools provide a number of useful performance tuning knobs as well. For example, increasing the size of a thread pool may increase the system throughput for some workloads. In this way, the architecture can efficiently distribute work on the available cores to increase overall throughput.

2.2 Deployment on Multicore

The three-stage architecture characterizing the SOSOA engine should be flexible enough to be deployed on different hardware configurations, ranging from single-core single CPU machines all the way to multi-processor multicore environments. The main constraints driving the deployment decisions of the architecture concern data locality, cache sharing, and the minimization of thread migrations, as these have measurable effects on a system performance [7, 9, 10]. Another important design aspect is portability. Given the wide variety of multicore architectures that are appearing on the market, it is important to avoid making too many assumptions about specific characteristics of the hardware.

As previously discussed, a thread pool is assigned to each component in order to let the threads of each pool execute common code paths. Thread pools communicate through queues, reducing contention, as the number of shared data structures is reduced and can be specifically optimized for concurrent access. For example, only the threads of the Kernel can access the state of the running composition instances. Also, only a subset of the threads of the engine performs I/O operations (in the Invoker and the Request Handler components). This means that when some thread gets blocked (e.g., waiting for a remote Web service to reply), the rest of the engine continues the execution of other composition instances.

Concerning the mapping of threads to cores, we do not assume that each thread pool should simply run on a separate core. Instead, our deployment is based on a replication of the entire engine, where each replica runs on a different set of cores. Given an incoming request, the Request Handler locates an available replica of the engine. The replica manages then the execution of the newly created composition instance on its cores.

Based on this strategy, the engine scans the hardware configuration to determine the structure of the system memory, the total number of CPUs, and the number of available cores. In more detail, the engine identifies sizes and levels of processor caches, finding out (when available) cores under a common cache (usually a L2 or L3). In this way, the engine creates affinity groups composed by core IDs accessing the same last-level cache. According to the information collected, the engine replicates itself while forcing the OS scheduler to constrain the execution of all threads of a replica within a specific affinity group. The replication phase keeps the total number of threads and memory usage constant: the more replicas are instantiated, the less amount of threads and memory is assigned to each of them. Then, after all replicas have started, client requests are forwarded to each replica using a round-robin policy.

This adaptive deployment procedure, based on the ability to "pin" the threads of each replica to the corresponding cores, allows the SOSOA architecture to adapt to different hardware configurations, from a single-CPU setup to a multi-processor multi-core deployment.

3 Preliminary Evaluation

In order to compare the performance of the different configurations of the SOSOA execution engine, we follow the approach presented in [1]. In this preliminary evaluation we focus on a limited set of workload types (based on 4 composition

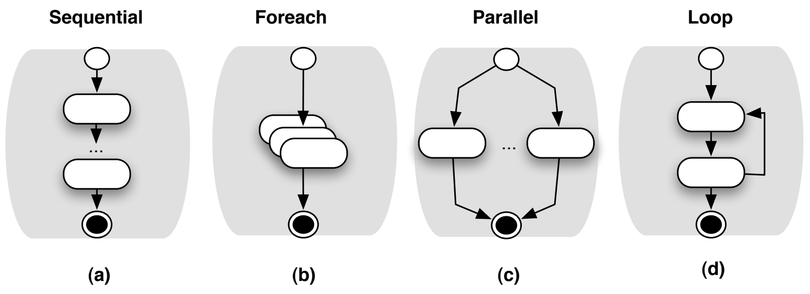

Fig. 2. Benchmark patterns executed by the service composition engine

patterns) and use the overall system throughput as the main evaluation and comparison metric.

3.1 Testbed Setup

The testbed environment has been configured to stress the service composition engine while minimizing the effect of the composed Web services running on the back-end. In this way, the components Workload Generator (WG) and Web Service Provider (WSP) never become performance bottlenecks and measurements are mostly influenced by SOSOA's behavior.

Workload Generator. Each test begins with the activation of the WG client component. This component drives the test generating a pseudo-random stream of service requests to the SOSOA engine. The component internally executes a specified number of clients, each one performing concurrent service requests according to a simple finite state machine composed of two states, "idle" and "busy". When "idle", a client sleeps for a random amount of seconds determined by a Gaussian distribution (fixed at $\mu = 1.0$ and $\sigma^2 = 0.5$). When the sleep time elapses, the client wakes up, moving to state "busy". In this state, the client makes a service request to the SOSOA engine, starting a new composition instance. The client then waits for the response message. Once the execution completion acknowledgement is received (or a timeout fixed at 30 seconds occurs), the client moves back to state "idle". This procedure is executed concurrently for the desired number of clients and repeated for a given number of iterations, in order to effectively measure the system throughput under reproducible conditions and to reduce the observed variance.

Benchmark Patterns. The second component of the testbed is the SOSOA service composition engine. The engine has been tested with four different composite Web services, chosen because each represents a common pattern used also in other benchmarking contexts (such as [1,5]). All compositions executed in the experiments contain the same number ($N = 6$) of service invocations and have the following control flow structures (see Fig. 2):

(a) *Sequential* — Each service invocation depends on the previous one, thus the engine invokes services sequentially. This is equivalent to a BPEL `<sequence>` block.

(b) *ForEach* — The composite service performs a parallel invocation of a variable number of services. This pattern occurs when data needs to be scattered to a number of independent services for parallel processing. Then, results are gathered by the composite service which aggregates them and continues the processing until all services have replied. In terms of BPEL, this is equivalent to a `<foreach>` block.

(c) *Parallel* — In this case, each service invocation is fully independent from the others and therefore they can be invoked in parallel. This is equivalent to a BPEL `<flow>` block without any control flow links between its child elements.

(d) *Loop* — The control flow of this composite service executes a loop for a fixed number of times (6 iterations), invoking a service at each iteration. It corresponds to a BPEL `<while>` block.

Web Service Provider. The third component of the testbed, WSP, is a common Web Server hosting the Web services invoked by the SOSOA engine. The component is deployed to an independent machine hosting $N = 6$ services. In order not to influence the overall execution time with delays caused by the Web Server, each service responds to any request with the same message after a controlled time interval. The size of each request and response message is negligible. In this way, we can ensure that the measured throughput is not limited by the WSP component.

Multicore Hardware and Software Environment. We measured the behavior of the SOSOA engine with different workloads and configurations by deploying it on a multicore machine equipped with 64GB RAM and two 2.6GHz Six-Core AMD Opteron processors, for a total of 12 cores. Each CPU comes with a high-capacity last-level cache (6MB L3 cache) shared by all cores. Each core also features 512KB L2 and 64KB L1 caches. This machine exploits a cache-coherent non-uniform memory access (cc-NUMA) architecture. To avoid interferences, the WG and WSP components are deployed on two additional dedicated machines with the following specifications: single-CPU Intel Core2-Quad desktop machine with a 3.0GHz processor (12MB L2 cache, 32KB L1 cache per core) and a total of 4GB RAM.

The whole testbed is connected through a private 100MBit LAN, with an average message round-trip time of 0.5 milliseconds. All machines in our testing environment run on the Ubuntu Linux Server 10.04 64bit distribution. We also used the standard Oracle Hotspot JVM 64bit Server version 1.6.20, since the SOSOA engine prototype is written in Java.

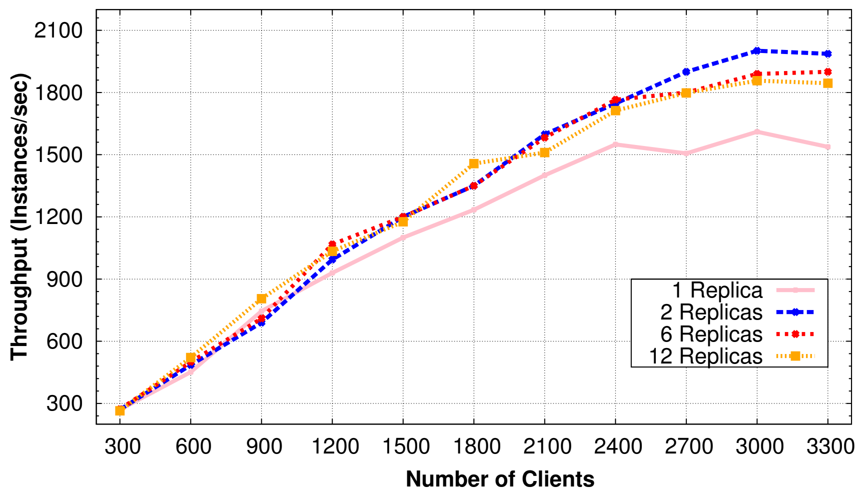

Fig. 3. Average throughput for an increasing number of clients

3.2 Test Configurations

For each machine, we first fix the total number of execution threads that are dedicated to run each replica of the SOSOA engine. Then we allocate the available threads to the pools associated with each engine component. Since the Request Handler uses non-blocking I/O, we observed that it does not require a large number of threads to handle client requests. Thus, we kept their total amount fixed at 32. The remaining number of threads are allocated equally among the other engine components. For replicated configurations where we run multiple instances of the engine, we reduce the number of threads of each replica in order to keep the total amount of threads constant. The same policy has been adopted for memory allocation. Since the total amount of available memory is constant, we reduce each replica's JVM maximum heap size as the number of them increases.

CPUs	Number of	Threads used by		Total
(cores)	Replicas	Kernel	Invoker	Threads
2 (12)	1	12	12	24
	2	6	6	24
	6	2	2	24
	12	1	1	24

Table 1. Deployment configurations: the fixed amount of computational resources per machine (Total Threads) are allocated to a variable number of replicas of the engine's components.

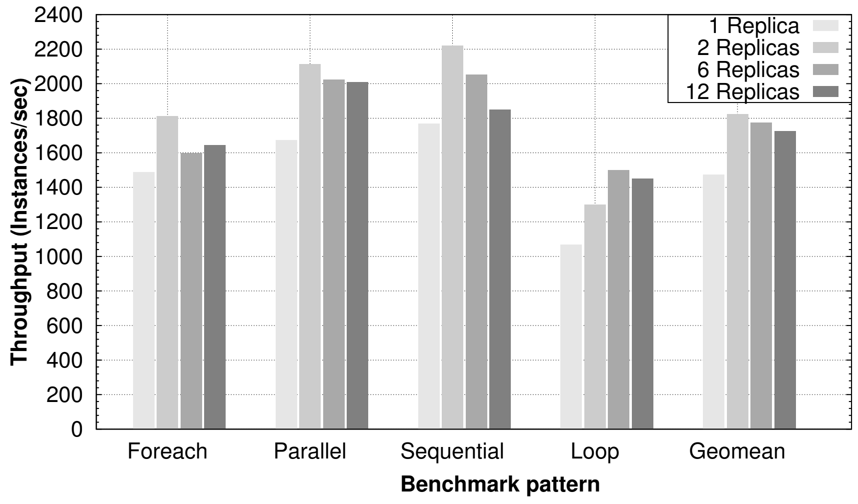

Fig. 4. Throughput for different patterns at the saturation point

3.3 Results

The results of our experiments show the average throughput scalability for the engine configurations summarized in Table 1. In order to minimize the noise introduced by the Java runtime, we repeated all test runs 10 times and show the average.

Fig. 3 shows the average throughput (Y axis) of the four patterns for an increasingly large number of clients (X axis). The charts help to compare the scalability of the engine for different numbers of replicas. Fig. 4 shows a more detailed performance comparison, breaking down the average throughput obtained for each workflow and each engine configuration fixing the number of clients at the saturation point.

Different replicas increase the throughput by approximately 20% when compared to the baseline configuration. Results can ben explained considering the architecture of the machine: on NUMA systems, memory is allocated on the optimal RAM slot connected to the CPU where threads are running on. Hence, constraining the execution of each JVM to a specific CPU makes sure that threads do not migrate across different processors. This helps to reduce latency times due to unoptimized memory accesses. Among the several configurations tested, the best results are obtained when the number of replicas is equivalent to the number of physical CPUs available. Since the two CPUs share a large L3 cache among all cores, data confirm that two replicas make the most efficient usage of the available computing resources.

Overall, these preliminary results highlight the potential benefits of our replication based approach, where on the tested hardware configuration and for some types of patterns we observed performance gains of more than 20%. Our experiments support the validity of the multicore-aware approach under the following viewpoints. This improvement is due to the partitioning of a set of threads across multiple replicas, which have been tied to a specific CPU. Using a finer grained partitioning (where replicas are tied to individual cores) also provides better performance compared to the baseline, but does not improve over the configuration with two replicas.

4 Conclusion

Modern multi-processor machines have very heterogeneous and sophisticated architectures, featuring several cores aggregated into various hardware configurations with hierarchic, shared or privileged caches and memory access paths. These newer architectures offer higher computational power through improved parallelism but also require specific software optimizations to maximize performance gains.

In this paper we have presented the SOSOA process execution engine for service composition designed to scale on multi-processor multi-core machines. Our design allows the engine to adapt to the different processor architectures at deployment time. This is performed taking into account the number of available processors and the way cores and caches are physically mapped. This hardware analysis process is performed at startup to let the engine automatically decide if and how many replicas should be started.

Our results show that this approach is significantly faster (up to about 20%) when compared to the baseline design, which uses the same number of execution threads, but keeps all of them within a single JVM. Our experiments also show that overhead introduced by the replication of the engine components is compensated by the speedup gains obtained on multi-processor architectures when replicas correctly exploit the locality of the underlying hardware.

We plan to extend our work to target other complex scenarios. Regarding future work, we plant to take QoS aspects into account. We also plan to perform additional experiments on a broader range of multicore architectures with different cache configurations in order to validate our claims of portability. Another important extension concerns the work sharing policy between the replicas. Adding a more sophisticated dispatching policy (e.g., based on work-stealing) could potentially improve the load-balancing among replicas and further increase performance.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "SOSOA: Self-Organizing Service-Oriented Architectures" (SNF Sinergia Project No. CRSI22_127386/1).

References

1. Bianculli, D., Binder, W., Drago, M.L.: Automated performance assessment for service-oriented middleware: a case study on BPEL engines. In: Proceedings of the 19th International Conference on World Wide Web (WWW 2010), Raleigh, NC, USA. pp. 141–150. ACM (April 2010)
2. Dustdar, S., Schreiner, W.: A survey on web services composition. Int. J. Web and Grid Services 1(1), 1–30 (August 2005)
3. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley (2006)
4. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. IEEE Computer 41(7), 33–38 (2008)
5. Lu, W., Gunarathne, T., Gannon, D.: Developing a concurrent service orchestration engine in ccr. In: Proceedings of the 1st international workshop on Multicore software engineering. pp. 61–68. ACM (2008)
6. Pautasso, C., Alonso, G.: JOpera: a toolkit for efficient visual composition of web services. International Journal of Electronic Commerce (IJEC) 9(2), 107–141 (Winter 2004/2005 2004), `http://www.gvsu.edu/business/ijec/v9n2/`
7. Rajagopalan, M., Lewis, B., Anderson, T.: Thread scheduling for multi-core platforms. In: Proceedings of the 11th USENIX workshop on Hot topics in operating systems. pp. 1–6 (2007)
8. Schuler, C., Weber, R., Schuldt, H., Schek, H.J.: Peer to peer process execution with OSIRIS. In: Proc. of the Service-Oriented Computing Conference (ICSOC 2003). pp. 483–498 (2003)
9. Teng, Q., Sweeney, P.F., Duesterwald, E.: Understanding the cost of thread migration for multi-threaded Java applications running on a multicore platform. In: ISPASS '09: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software. pp. 123–132 (Apr 2009)
10. Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In: PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 203–212. ACM, Bangalore, India (2010)