A Multicore-aware Runtime Architecture for Scalable Service Composition

Daniele Bonetta, Achille Peternier, Cesare Pautasso, and Walter Binder
Faculty of Informatics
University of Lugano (USI)
via G. Buffi 13, 6900 Lugano, Switzerland
Email: firstname.lastname@usi.ch

***Abstract*—Middleware for web service orchestration, such as runtime engines for executing business processes, workflows, or web service compositions, can easily become performance bottlenecks when the number of concurrent service requests increases. Many existing process execution engines have been designed to address scalability with distribution and replication techniques. However, the advent of modern multicore machines, comprising several chip multiprocessors each offering multiple cores and often featuring a large shared cache, offers the opportunity to redesign the architecture of process execution engines in order to take full advantage of the underlying hardware resources. In this paper we present an innovative process execution engine architecture. Its design takes into account the specific constraints of multicore machines and scales well on different processor architectures, as shown by our extensive performance evaluation. A key feature of the design is self-configuration at startup according to the type and number of available CPUs. We show that our design makes efficient use of the available resources and can scale to run thousands of concurrent business process instances per second, highlighting the potential and the benefits for multicore-awareness in the design of scalable process execution engines.**

***Keywords*-web service orchestration; process execution engine; web service composition; multicores; performance and scalability evaluation; testbed**

I. Introduction

Service-oriented architectures promote the creation of new applications by composing and orchestrating existing Web services using business process models [1]. The resulting compositions are executed by middleware for Web service orchestration, such as business process or workflow execution engines [2]. In this paper, we refer to compositions as "processes" and to Web service orchestration middleware as "process execution engines".

Since processes are accessible as Web services, process execution engines may have to handle a large number of concurrent service requests. Assuming that the composed services are designed to scale (i. e. they are hosted in a cloud environment), process execution engines can easily become performance bottlenecks when the complexity of their processes is high or the number of process execution requests from their clients increase. For example, some scientific computing applications require to run many thousands of workflow instances for a single experiment [3].

Some existing engines, such as OSIRIS [4], JOpera [5], SpiderNet [6], or CEKK [7], rely on distribution and replication techniques so as to ensure scalability in peer to peer environments or in clusters of computers. 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 [8]. 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's Nehalem and AMD's Opteron.

In order to take full advantage of the hardware resources on modern multicore machines, which often comprise many chip multi-processors, it is not sufficient (and in some cases can produce adverse results) to simply configure an engine to use a larger pool of execution threads. Instead, as we are going to discuss, a performance increase up to about 30% can be gained by explicitly considering the characteristics of the multicore architectures in the design of the process execution engine.

In this paper we extend the SOSOA process execution engine, an innovative service composition middleware based on a multicore-aware design, which is not limited to a specific kind of processor architectures [9]. A key feature of our design is its ability to perform self-configuration at startup. This lets the architecture adapt according to the type and number of available chip multi-processors. 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 self-tuning upon startup according to the actual hardware configuration.

The main contributions of this paper are the following:

1) We propose to take emerging multicore architectures of modern processors into account for the design of process execution engines and demonstrate the clear impact of multicore-awareness on their performance.
2) We introduce self-configuration upon startup for process execution engines in order to optimize the use of the available hardware without sacrificing the portability of the engine across different processor micro-architectures.

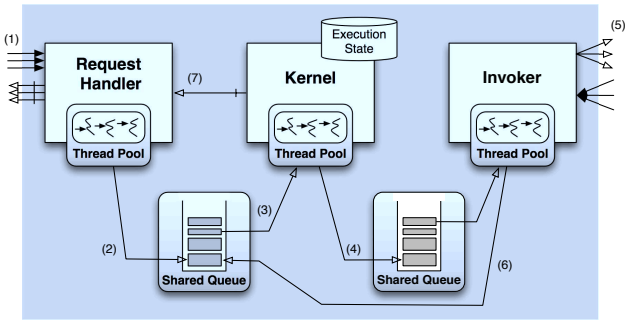

Figure 1. Multi-stage architecture of the SOSOA process execution engine for Web service composition

3) We thoroughly evaluate performance and scalability of the SOSOA engine running on three different processor micro-architectures, highlighting that replication used in conjunction with CPU affinity binding techniques help increase performance.

The paper is organized as follows. Section II describes the main requirements and architectural characteristics of the process execution engine. Section III refines the architecture describing how it has been designed to target multicore machines with the self-configuration mechanism triggered at startup. Section IV describes the evaluation testbed and presents the results of our measurements. Section V gives an overview of related work, while Section VI concludes the paper and presents future research directions.

II. Architecture

The logical architecture of the SOSOA process execution engine is designed following a multi-stage pipeline, comprising three components: the Request Handler, the Kernel, and the Invoker (Fig. 1). The Request Handler publishes processes as Web services. The Kernel performs the actual execution of the processes and manages the state of multiple process instances. The Invoker takes care of interacting with the composed services.

The execution of a process begins with a request from a client to instantiate a new process 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 processes, while keeping their state up-to-date. Processes, modeling how to compose Web services, require interactions with the composed Web services. To this end, 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 processes, 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 continue the execution of the corresponding process instance (6). Once the execution of the entire process 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. The goal is to define a scalable system architecture, where a limited number of execution threads can be leveraged to execute a much larger number of process instances. Thanks to the separation of the process 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 one thread per component is necessary to make sure the system can function 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 multiple concurrent 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 its process instances. The architecture adopts thread pools to leverage the underlying hardware parallelism. By assigning more than one thread to each component, we can efficiently distribute computations on the available cores.

However, optimal pool sizing is not a trivial task, as the total amount of threads in the system could easily become a crucial issue for performance. In fact, it is not always guaranteed that increasing the number of threads automatically translates into better performance. On the contrary (as shown in Section IV), we observed a significant performance degradation by (mis)configuring the engine with an oversized number of threads. This issue motivates the need for a different approach to scale engine performance: instead of increasing the number of worker threads of each pool, the engine should make better usage of a bounded number of them. In the next section we discuss how such limited number of execution resources can be organized in an optimal way to fully exploit multicore hardware capabilities.

III. Design Choices for Multicores

Existing work in the area of performance optimization clearly shows that hardware-related issues, such as thread migrations, data locality, or cache sharing, have relevant effects on execution times and performance [10]–[12].

The three-stage architecture characterizing the SOSOA process execution engine is flexible enough to be deployed as a set of replicated OS processes, each one independent from the others but altogether sharing the workload by handling different requests cooperatively. This is possible because of the pipelined design, which is abstract enough

to allow its deployment to be adapted without requiring the semantic of the execution to change.

Within each replica, we assign a thread pool to each component in order to let several threads run the same code paths. Thread pools communicate through queues, minimizing synchronization issues and 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 process 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, the rest of the engine continues the execution of other process instances.

A. Multicore Awareness

Our multicore-aware design combines the flexibility offered by the multi-stage architecture with its capability of replicating different instances of the engines and controlling how many threads have been allocated to the thread pools found in each replica.

The basic principle guiding our approach is that different hardware architectures have to be considered as potential sources of performance degradation, unless the system is deployed taking into account their specific characteristics. Any system that claims to be portable should deal with this aspect, and the wide adoption of machines with multiple CPUs and cores makes this aspect a relevant issue.

For example, targeting a multi-CPU machine with a single instance of our engine and scaling only in the number of threads does not result in optimal performance, mainly because the homogeneous nature of this configuration is in contrast with the heterogeneity of memory access patterns that could be present on multicore machines. Instead of a single application instance scaling in the number of worker threads as the number of cores increases, multi-processor hardware has to be addressed with specific approaches to reduce contention, exploit locality, and balance the load among all of the available cores. To tackle these issues, we implemented a multicore-aware design strategy based on replication.

With this approach, we fix the overall amount of threads running on the machine but we modify the way they are mapped to the underlying hardware. This way, the working capacity of the engine is partitioned among multiple replicas depending on the number and the type of processors and cores. The size of each partition is defined in terms of number of threads and memory assigned to each replica, and the partitioning strategy changes according to the hardware architectures.

B. Self-Configuration on Startup

To adapt itself to the actual hardware configuration, the engine relies on a self-configuration on startup strategy. First, a single instance of the engine is executed, which scans the hardware configuration to determine the structure of the system memory, the total number of CPUs, and the number of available cores. More precisely, 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 decides if and how many times to clone itself by starting new replicas and forcing the OS scheduler to constrain their execution 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 the startup phase has completed, the main instance receives and forwards requests to replicas using a round-robin policy.

This adaptive startup procedure, coupled with the ability to "pin" replicas to cores, allows the runtime to adapt to different hardware configurations, from a single-CPU setup to a multi-processor deployment in a transparent and autonomous way.

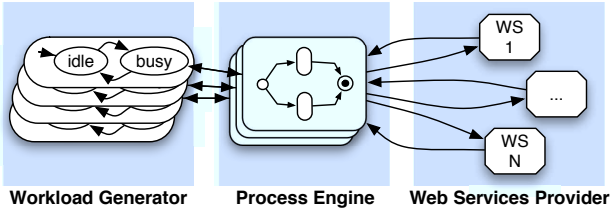

Figure 2. Testbed setup

IV. Evaluation

In order to validate the SOSOA execution engine adaptability, we have extended the approach presented in [13] by developing a new workload generator targeting our needs. The testing platform used to evaluate SOSOA performance has been tuned to execute exhaustive tests with several kinds of workload. Each workload represents different load factors, allowing to analyze the average throughput under different stress conditions.

A. Testbed Setup

The overall testing environment to evaluate SOSOA's performance is depicted in Fig. 2. The testbed environment has been configured to stress the process execution 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.

1) 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

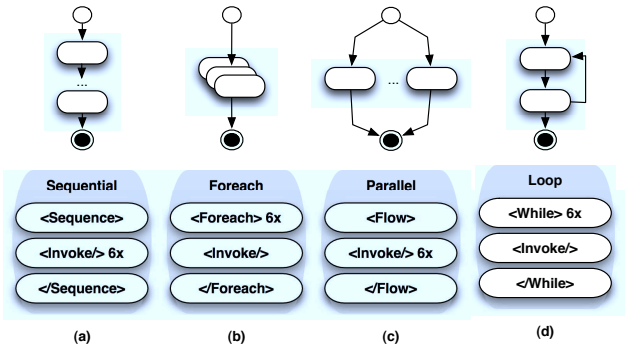

Figure 3. Benchmark workflows executed by the middleware for performance evaluation

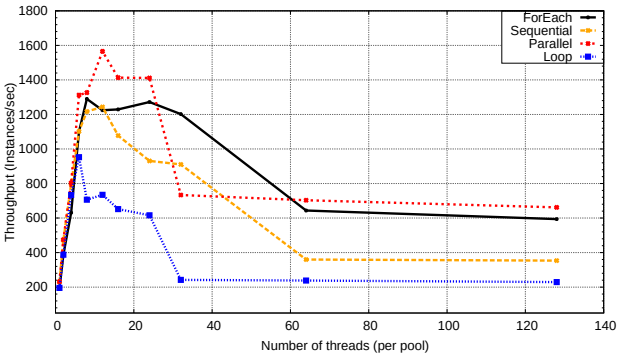

Figure 4. Scalability test using a growing number of threads assigned to the Kernel and Invoker's pools

$\mu = 1.0$ 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 workflow instance. The client then waits for the response message. Once the process 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.

2) Workflow Engine Benchmark: the second component of the testbed is the SOSOA process execution engine. The engine has been tested with four different composite Web services, chosen because each represents a common workflow pattern used also in other benchmarking contexts (such as [13], [14]).

All processes executed in the experiments contain the same number ($N = 6$) of service invocations and have the following control flow structures (see Fig. 3):

(a) *Sequential* — Each service invocation depends on the previous one, thus the workflow 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 2.0, 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 dependencies 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 sequential iteration. It corresponds to a BPEL `<while>` block.

3) 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. 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. This way, we can ensure that the measured throughput is not limited by the WSP component.

4) Multicore Hardware and Software Environment: To avoid interferences, components WG and WSP have been deployed to separate machines for all experiments. We analyzed the behavior of the SOSOA engine component with different workloads and configurations by deploying it on three multicore machines with different hardware characteristics.

The first machine adopted for testing (M1) is equipped with two 32GB RAM slots 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: each CPU is optimally connected to a dedicated RAM slot and can efficiently access its data with minimal latency. When a CPU requires to access data stored on another memory slot, request times increase significantly.

The second machine (M2) is equipped with 16GB RAM and four Intel Xeon 2.4GHz processors with four cores each, for a total of 16 cores. Processors on this second machine miss L3 caches and have a total of 8MB L2 cache per CPU and 32KB L1 cache per core. L2 caches are grouped into two 4096KB blocks per processor, each shared by two cores (4096KB L2 for core 1 and core 2, 4096KB L2 for core 3 and core 4).

The third machine (M3) is a 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. Like in M2, L2 caches are grouped into two 6144KB blocks per CPU, shared by cores 1-2 and cores 3-4, respectively.

The WG and WSP components are deployed on two additional dedicated machines with the same specifications of M3. 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

Table I
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.

Machine Name	CPUs (cores)	Number of Replicas	Threads used by Kernel	Threads used by Invoker	Total Threads
M1	2 (12)	1	12	12	24
		2	6	6	24
		6	2	2	24
		12	1	1	24
M2	4 (16)	1	32	32	64
		4	8	8	64
		8	4	4	64
		16	2	2	64
M3	1 (4)	1	8	8	16
		2	4	4	16
		4	2	2	16

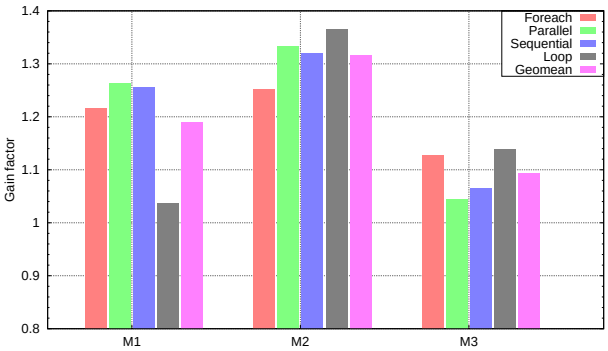

Figure 5. Relative speedup factor comparison for the best configurations running on different machines: M1 (2 replicas), M2 (8 replicas) and M3 (2 replicas)

distribution. We also used the standard Oracle Hotspot JVM 64bit Server version 1.6.20, since the SOSOA engine prototype is written in Java.

B. Deployment Configurations: Resource Allocation and Replication

The preliminary experiment shown in Fig. 4 explores the effect of simply increasing the number of threads allocated to the thread pools of a *single* replica of the engine. The performance of the SOSOA engine is measured in terms of the throughput of an increasingly large number of threads. Results (obtained on the M1 machine with a constant workload of 2000 requests per second) clearly show that the performance does not grow proportionally with the execution resources that are allocated to the engine. Depending on the underlying hardware configuration, and for each workload type, there is an optimal, limited number of threads that should be used for sizing the thread pools.

Using the same approach, we identified the optimal number of threads on each machine: the values adopted for experiments are summarized in Table I.

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 (e.g. 4096MB for 1 JVM, 2048 MB for 2 JVMs, 1024MB for 4 JVMs, etc.).

C. Results

The results of our extensive experiments show the average throughput scalability and workflow performance comparisons for M1, M2, and M3 using the different engine configurations summarized in Table I. In order to minimize the noise introduced by the Java runtime, we repeated all test runs 10 times.

Fig. 5 summarizes the results showing the relative speedup that can be obtained with the best configuration for each workflow and on each hardware configuration (X axis). Fig. 6 shows the average throughput (Y axis) of the four workflows for an increasingly large number of clients (X axis). The charts help to compare the scalability of the engine for different numbers of replicas on the three hardware configurations. Fig. 7 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.

The two *(a)* charts in Fig. 6 and Fig. 7 show how different replicas increase the throughput on M1 by an average of $\sim 20\%$ when compared to the single instance configuration. Results on M1 can be 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. This helps to reduce latency times due to unoptimized memory access. Among the several configurations tested, M1 shows the best results when the number of replicas is equivalent to the number of physical CPUs available. Since CPUs on M1 share a large L3 cache among all cores, results confirm that two replicas make the most efficient usage of the available computing resources.

Results obtained on M2 are reported in the *(b)* charts, showing speed gains introduced by the adoption of replicas over the single instance configuration. While benefits given by replication show an average speedup of about 30%, they also look almost identical among the three replicated configurations. The explanation for this behavior is related to the hardware of M2, which features more processors than M1, but a less efficient memory architecture (no NUMA). Unlike M1, the M2 machine tends to saturate the front-bus with memory requests coming from its CPUs to access different locations concurrently. This effect is less evident with configurations with four and eight JVMs, for which replicas have been pinned

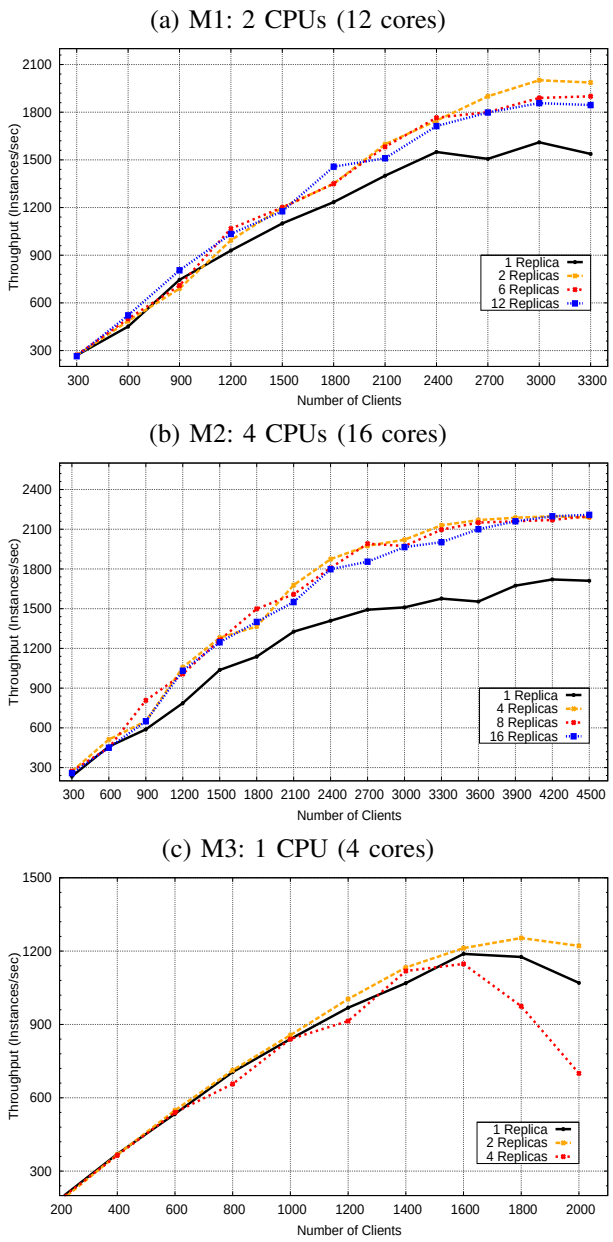

Figure 6. Average throughput for an increasing number of clients

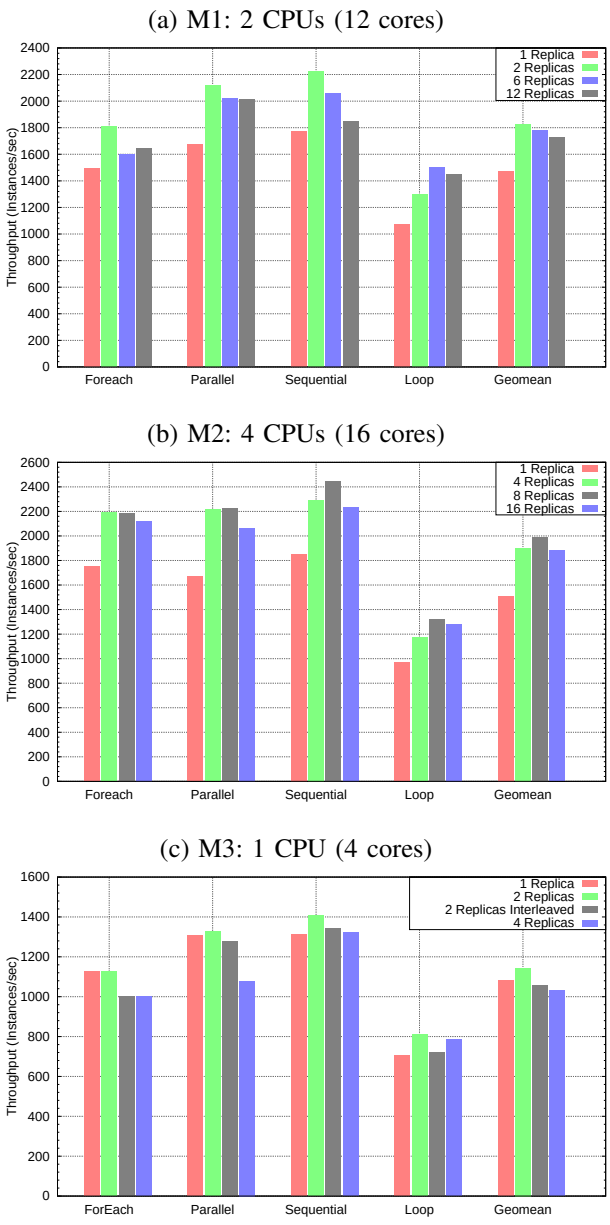

Figure 7. Throughput gain for different workflows

to use the same last-level caches (assigning the threads of each replicas to cores sharing an L2 cache), partially decongesting the stress on the memory bus (although less efficiently than architectures recurring to NUMA and a larger L3). However, results converge towards the same value at the highest throughput peak.

In the two *(c)* charts we show how the speedup on M3 is introduced by a better usage of caches. M3 is a single-CPU quad-core machine sharing a L2 cache between two groups of cores. Running two replicas within these sets of cores leads to a speed gain up to $\sim 15\%$ (*Loop* workflow). For this hardware configuration, the optimal usage of cache memory is confirmed by repeating the test and constraining the two replicas to cores not sharing the same top level cache (in Fig. 7 (c), the "2 Replicas Interleaved" line shows the results obtained with this CPU-affinity setting). The "2 Replicas Interleaved" configuration has been designed to invalidate the benefits introduced by the optimal cache usage of the "2 Replicas" configuration. Results clearly show the negative impact on performance of non-optimal cache utilization. In some cases, and on average (Geomean), the misconfigured engine performs worse than the baseline non replicated configuration. Similar arguments can be brought forward concerning the usage of four replicas on M3: whereas, for some workflows, replication gives a small speedup over default settings (*Sequential* and *Loop*), these results are not as robust as in the "2 Replicas" configuration, and the average values show the highest slowdown in performance.

As summarized in Fig. 5, these results highlight the advantages of our replication based approach, where for some hardware configurations and some workflow type we observed performance gains of more than 30%. Our experiments support the validity of the multicore-aware approach under the following viewpoints.

First, results show that partitioning a set of threads

across multiple replicas always leads to better performance over the baseline approach which does not use replication.

Second, results hint at the existence of a direct correlation between number of replicas, their performance, and the underlying multicore platform architecture. By comparing data obtained using the optimal number of replicas with the amount of shared last-level caches available on M1, M2, and M3, we can claim that the optimal amount of replication is related to the number of available last-level caches.

Finally, our measurements provide useful feedback to drive the self-configuration on startup procedure, as the engine always exhibits best performance when the number of replicas corresponds to the number of last-level shared caches present in the system.

V. Related Work

In prior work [13] we have considered the importance of automated middleware performance assessment as a first order issue for modern SOAs, and we have proposed the class of benchmarks adopted in this paper. Other approaches in the field of SOA performance evaluation shift the focus of the experimentation towards the services running in an SOA, dealing with testbed generation for Web services.

A relevant example for this class of testbed generators is the Genesis [15] framework. Genesis is aimed at the generation of complex Web services that provide support for creating and deploying services, for simulating QoS metrics, and steering the service execution by changing runtime and QoS parameters with a plug-in mechanism. Genesis can be seen as a complement to what we have done in our testbed, since it generates a testbed of services that could be composed and executed to benchmark and stress our middleware. Another notable example is Puppet [16]. Puppet is a model-based generator of Web service stubs, which can be used before the actual deployment of a service, in order to test its behavior when interacting with externals services that are unavailable for testing.

Concerning multicore-aware solutions for high performance service composition, Lu et al. propose in [14] an approach not based on thread parallelism but on event-driven patterns and message passing interactions, using the CCR runtime available in the .Net framework.

Outside the realm of service-composition engines, there have been many proposals for multicore-aware middleware solutions. For example, OCCAM [17] is a software platform for developing multicore adaptive applications. The main characteristics of OCCAM are its API-based structure and its design-time platform, consisting of data structures that allow developers to specify the performance contract to be guaranteed. However, the platform seems to target only a specific range of applications.

In the field of thread scheduling, there are many approaches showing the potential of correct thread-to-CPU mapping. Tam et al. [18] propose an OS-level thread scheduler for SMP-CMP-SMT machines able to identify at runtime the best allocation strategy for each executed thread. Unlike our approach, their scheduler is based on constant hardware performance counter feedback, and the allocation strategy is managed by a smart scheduler able to monitor stall breakdowns and consequently to detect so-called *sharing patterns* to obtain a realistic thread clustering aimed at forcing affinity migrations of related threads (i.e. threads influencing the same performance counter by sharing the same memory). In our approach the sharing pattern corresponds to the different threads owned by each replica.

Rajagopalan et al. propose in [11] another approach for affinity-based programming by introducing a thread scheduler which makes use of a constraint called *Related Thread ID*. The programmer is asked to specify a set of related threads that the scheduler should consider to be placed together. For example, a group of threads might have a common RTID when sharing some data or lock. The scheduler adopted is an extension of the one present in McRT [19], implemented through a single task-queue per processor core and a combination of existing work-stealing and work-distribution scheduling policies. When new software threads are initialized, the scheduler first tries to distribute them taking into account the number of available cores, and once all of the physical resources are in use the scheduler submits jobs taking into account their RTID value. This way, the scheduler tries to guarantee locality of threads with a same RTID by associating them to a same core, in order to minimize contention costs.

VI. 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. We have shown that the simple approach following the idea of "just add an higher amount of execution threads to scale a system's performance" is not enough. Instead, it is important to consider *how* threads of the engine components are mapped among processors and cores.

Our design allows the engine to adapt to the different processor architectures at deployment time. This self-configuration on startup 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 bootstrapped. For best results, our experimental validation suggests to create one replica for each last-level shared cache available.

Our results show that this approach is significantly faster when compared to the baseline design, which uses

the same number of execution threads within a single JVM. Our experiments have also shown 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. Conversely, we showed that the simple idea of replication can also be detrimental to performance when incorrectly applied.

Overall, our multicore-aware design performs up to about 30% better than the baseline.

An interesting possible extension of our approach should go beyond self-configuration on startup, but also take into account dynamic QoS aspects at runtime. It should be possible to dynamically allocate and deallocate replicas as the system load conditions change. A slightly modified architecture could also reuse the replication strategy to improve system reliability, by automatically creating a new replica when another fails. Finally, a more sophisticated dispatching policy (e.g. work-stealing) would improve load-balancing among replicas with the potential to give an additional boost to performance.

ACKNOWLEDGMENT

This work is funded by the Swiss National Science Foundation with the SOSOA project (SINERGIA grant nr. CRSI22_127386), and by the European Community under the grant agreement no. EU-FP7-215483-S-Cube.

REFERENCES

[1] S. Dustdar and W. Schreiner, "A survey on web services composition," *Int. J. Web and Grid Services*, vol. 1, no. 1, pp. 1–30, August 2005.

[2] F. Leymann, D. Roller, and M.-T. Schmidt, "Web services and business process management," *IBM Systems Journal*, vol. 41, no. 2, pp. 198–211, 2002.

[3] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[4] C. Schuler, R. Weber, H. Schuldt, and H.-J. Schek, "Peer to peer process execution with OSIRIS," in *Proc. of the Service-Oriented Computing Conference (ICSOC 2003)*, 2003, pp. 483–498.

[5] C. Pautasso and G. Alonso, "JOpera: a toolkit for efficient visual composition of web services," *International Journal of Electronic Commerce (IJEC)*, vol. 9, no. 2, pp. 107–141, Winter 2004/2005 2004. [Online]. Available: http://www.gvsu.edu/business/ijec/v9n2/

[6] X. Gu, K. Nahrstedt, and B. Yu, "Spidernet: an integrated peer-to-peer service composition framework," in *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, 4-6 2004, pp. 110–119.

[7] W. Yu, "Scalable services orchestration with continuation-passing messaging," in *Intensive Applications and Services, 2009. INTENSIVE '09. First International Conference on*, 20-25 2009, pp. 59 –64.

[8] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[9] D. Bonetta, A. Peternier, C. Pautasso, and W. Binder, "Towards scalable service composition on multicores," in *LNCS, proceedings of OTM 2010 Workshops*, Springer-Verlag, Ed., vol. 6428, 2010.

[10] E. Z. Zhang, Y. Jiang, and X. Shen, "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*. Bangalore, India: ACM, 2010, pp. 203–212.

[11] M. Rajagopalan, B. Lewis, and T. Anderson, "Thread scheduling for multi-core platforms," in *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, 2007, pp. 1–6.

[12] Q. Teng, P. F. Sweeney, and E. Duesterwald, "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*, Apr. 2009, pp. 123–132.

[13] D. Bianculli, W. Binder, and M. L. Drago, "Automated performance assessment for service-oriented middleware: a case study on bpel engines," in *WWW '10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 141–150.

[14] W. Lu, T. Gunarathne, and D. Gannon, "Developing a concurrent service orchestration engine in ccr," in *Proceedings of the 1st international workshop on Multicore software engineering*. ACM, 2008, pp. 61–68.

[15] L. Juszczyk, H. Truong, and S. Dustdar, "Genesis-a framework for automatic generation and steering of testbeds of complex web services," in *Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'08)*. Citeseer, pp. 131–140.

[16] A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini, "Model-based generation of testbeds for web services," *Testing of Software and Communicating Systems*, pp. 266–282.

[17] D. Fay, L. Shang, and D. Grunwald, "A platform for developing adaptable multicore applications," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 2009, pp. 157–166.

[18] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*, Lisbon, Portugal, 2007, pp. 47–58.

[19] S. Bratin, A.-T. Ali-Reza, G. Anwar, R. Mohan, H. R. L., P. Leaf, M. Vijay, M. Brian, S. Tatiana, S. Eric, R. Anwar, C. Doug, and F. Jesse, "Enabling scalability and performance in a large scale cmp environment," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*, Lisbon, Portugal, 2007, pp. 73–86.