CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

High Performance Execution of Service Compositions: a Multicore-aware Engine Design

Achille Peternier, Cesare Pautasso, Walter Binder, Daniele Bonetta

Faculty of Informatics, University of Lugano, Via G. Buffi 13, 6900 Lugano, Switzerland,
email: firstname.lastname@usi.ch

SUMMARY

While modern computer hardware offers an increasing number of processing elements organized in Non-Uniform Memory Access (NUMA) architectures, prevailing middleware engines for executing business processes, workflows, and Web service compositions have not been optimized for properly exploiting the abundant processing resources of such machines. Amongst others, factors limiting performance are inefficient thread scheduling by the operating system, which can result in suboptimal use of system memory and CPU caches, and sequential code sections that cannot take advantage of multiple available cores.
In this article, we study the performance of the JOpera process execution engine on recent multicore machines. We first evaluate its performance without any dedicated optimization for multicore hardware, showing that additional cores do not significantly improve performance, although the engine has a multi-threaded design. Therefore, we apply optimizations based on replication together with an improved, hardware-aware usage of the underlying resources such as NUMA nodes and CPU caches. Thanks to our optimizations, we achieve speedups from a factor of 2 up to a factor of 20 (depending on the target machine) when compared to a baseline execution "as is". Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: performance optimization; multicore; non-uniform memory access architecture; service composition and execution

1. INTRODUCTION

Recent technology trends in micro-processor design have shifted the focus from higher clock frequencies to packing multiple cores on the same chip. Modern hardware features a series of processors and buses aggregated into structures with different hierarchies and latencies, ranging from CPU caches to memory nodes. The consequence of this evolution is that recent machines are fundamentally different from previous micro-processor architectures [1], resembling more a distributed system than a centralized one, and inheriting similar features and drawbacks [2].

In the context of Web service provisioning, systems requiring high-scalability such as Process Execution Engines (PEEs) need particular care in their design to efficiently take advantage of such hardware. PEEs have become crucial components within modern Service-Oriented Architectures (SOAs) [3, 4]. The purpose of such engines is to orchestrate a set of distributed services by executing a business process describing how Web services should interact to achieve a certain goal [5]. Business processes are typically themselves delivered as services to clients [6], requiring PEEs to scale to handle a potentially high and unpredictable number of concurrent client requests [7].

Since business processes are accessible as Web services, PEEs may have to handle a large number of concurrent service requests. Assuming that the composed services are designed to scale (they may be hosted in a Cloud environment), PEEs can easily become performance bottlenecks when the complexity of the executed business processes is high or the number of process execution requests

from clients increases. For example, some scientific workflows require running many thousand process instances for a single experiment [8].

Several existing PEEs, such as OSIRIS [9], SpiderNet [10], or CEKK [11], rely on distribution and replication techniques to ensure scalability in peer to peer environments or in clusters of computers. Modern multicore machines offer a promising alternative to clusters, allowing building a powerful infrastructure with highly parallel shared memory machines. The abundant hardware resources of such machines are often under-utilized unless the application has been tuned to exploit the specific hardware architecture. This tuning phase often requires deep understanding of complex hardware mechanisms, and is becoming crucial for companies providing services.

In this article we show how a significant performance increase can be gained by explicitly considering the characteristics of modern multicore architectures in the design of a PEE. In order to take full advantage of a high number of cores, it is often not sufficient to simply configure an application to use a larger pool of execution threads. In some cases such a naïve approach can even produce adverse results. Instead, specific care has to be taken to deploy a software solution that automatically improves its performance by matching the characteristics of the underlying hardware. Existing work in the area of performance optimization clearly shows that hardware- and OS-related issues, such as thread migrations, data locality, or cache sharing, have an impact on performance [12, 13, 14].

As a case study, we target JOpera [15], a Java-based PEE, and show how to improve its performance with specific multicore optimizations. JOpera can be deployed as a standalone application on server machines. JOpera executes service compositions defined in a visual modeling language which are then compiled to Java code for efficient execution. The kernel of JOpera features a scalable architecture which was originally designed to run on parallel execution environments such as on clusters [16].

The present work completes and extends our previous contributions on multicore-aware PEEs published in [17, 18, 19]. We improved JOpera's hardware awareness of CPU/core mapping and processor shared caches by also taking into account the way memory is partitioned into NUMA nodes. This article provides a new, thorough evaluation of JOpera's optimizations on several different server-class machines with distinct micro-architectures and hardware resources. While previous work was mostly based on the way cores are aggregated into CPUs sharing a common level of cache among available cores, we now also take into account new factors such as local versus remote memory bindings, and motivate our experiments with a raw evaluation of the hardware capabilities of the machines in our testing environment.

The contributions of this article can be summarized as follows:

1. To highlight the possible performance gains by improving Operating System (OS) scheduling and memory management, we perform a series of detailed tests to benchmark performance differences between default and optimally tuned settings on a series of modern multiprocessor machines with different hardware architectures.
2. We show that JOpera executed "as is", without explicit architecture-aware optimizations, does not exploit all the resources provided by the modern multicore machines of our testing environment. Simply increasing the number of threads of JOpera's internal components is not enough to take advantage of the computational power of the underlying hardware.
3. We improve our software with multiprocessor-specific optimizations that enable it to automatically configure itself on the various computers used for our evaluation. We adopt a replication mechanism coupled with memory and CPU bindings to implement an efficient server consolidation approach. The replication approach mimics the distributed nature of the underlying hardware at the software level, executing each replica on a partition of the hardware to improve data locality.
4. We show that thanks to our strategy, we achieve speedup factors of up to 2, 4, and more than 20 (depending to the underlying machine) when compared to a baseline executed without any multicore optimization. The results shed some light on why some modern computers perform worse than their predecessors, unless software components are properly replicated.
5. We make the tools developed for this article publicly available.

DOI: 10.1002/cpe

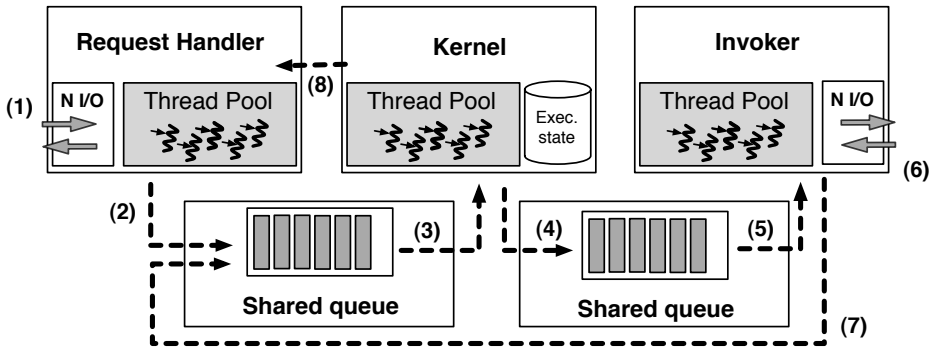


Figure 1. Multi-stage architecture of the JOpera PEE for Web service composition.

This article is structured as follows: Section 2 contains a description of the architecture of the PEE we use as the case study for our optimizations. In Section 3 we motivate our approach. Section 4 presents the testing environment for our measurements. Section 5 contains a first series of synthetic experiments to measure the correlation between selected software configurations and performance. In Section 6 we measure performance of JOpera without any multicore optimization. In Section 7, we describe the performance gains obtained through our improvements. Related work is summarized in Section 8, while Section 9 concludes this article.

2. ARCHITECTURE OF JOPERA

In this section we present the architecture of the latest version (2.6.0) of the JOpera* PEE. JOpera represents a notable case study for a complex multi-threaded system, and we will show how to improve its performance by adopting several multicore optimizations.

JOpera's service compositions are modeled using processes describing in which order a set of tasks should be executed, and specifying the data flow exchanges between tasks. This means that JOpera tasks may involve local computations (e.g., defined using Java code snippets or Java method calls) as well as remote service invocations (e.g., through RESTful HTTP calls). JOpera also enables running atypical business processes, composed for example of WS-* and RESTful Web services [20].

The software architecture of the JOpera engine is designed following a multi-stage pipeline, comprising three components: the RequestHandler, the Kernel, and the Invoker (Figure 1). The RequestHandler 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 RequestHandler 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), which is processed as soon as possible (5). RequestHandler, Kernel, and Invoker 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 (6), its results are enqueued by the Invoker into the queue shared with the Kernel,

*http://www.jopera.org/

DOI: 10.1002/cpe

so that they can be used to continue the execution of the corresponding process instance (7). Once the execution of the entire process instance completes, the Kernel component notifies the RequestHandler component which sends results to the client (8).

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 parallel constructs commonly found in most service composition languages are to be properly supported, we need to assign a larger number of threads to each component. For this reason, the RequestHandler 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 components, 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 distribute computations on the available cores.

However, optimally sizing thread pools is not a trivial task, as an excessive total number of threads in the system can lead to a waste of resources, while undersized thread pools may not exploit all the computational resources available. 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 6), we observe a waste of memory without performance gain by running 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 these resources can be organized in a more efficient way to fully exploit modern multicores.

3. DESIGN DECISIONS FOR MULTICORES

The three-stage architecture characterizing the JOpera PEE can be deployed as a set of replicated OS processes, each one independent from the others. In this configuration, the workload is split and each replica handles different requests alone. Within each replica, we assign a thread pool to each engine component in order to let several threads execute the same code paths. The engine design reduces contention on shared data structures, as they are accessed by only a subset of the threads running in the engine. 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 RequestHandler components). This means that when some thread gets blocked, the rest of the engine continues the execution of other process instances.

In this scenario, the main issues are to determine the optimal number of replicas to execute, and how to allocate their hardware resources efficiently. To deal with these configuration issues, it is important to consider the way multicore hardware resources are grouped into several distributed processing entities, each comprising memory and CPUs. These architectures are briefly summarized hereafter. To clarify our terminology, we refer to a micro-processor as CPU, which may aggregate several cores. For instance, an Intel Core2Quad is a single CPU with 4 cores. We use the term Processing Unit (PU) to define Simultaneous Multithreading (SMT) units. For example, an Intel Nehalem CPU with 4 cores with 2-level SMT per core has a total of 8 PUs. As we will discuss later, SMT has been disabled in our experiments, thus, the number of cores will match the number of PUs.

DOI: 10.1002/cpe

3.1. SMP vs. NUMA

Hardware based on Symmetric Multi-Processing (SMP) architectures is composed of a single memory bus shared by all the CPUs available [21]. While this approach works relatively well for a small number of CPUs, the shared memory bus rapidly becomes a performance bottleneck on systems with many cores competing for concurrent access to the memory.

A Non-Uniform Memory Access (NUMA) architecture reduces bus contention by using several memory buses and limiting the number of CPUs that can use them [22]. A combination of memory plus one or more CPUs is called *NUMA node*. NUMA nodes do not communicate directly through the memory buses but through a dedicated connectivity called the high-speed interconnect. For this reason, two types of memory allocation need to be considered: *local* and *remote*. If a process is executed on a CPU and the accessed data is allocated in the memory belonging to the same NUMA node where the CPU is, accesses are referred to as local. If the data is allocated in another NUMA node's memory, it is considered remote. If a CPU located on the NUMA node 1 requires data from memory within the NUMA node 2, the request passes through the interconnect, reads memory on node 2, returns the data through the interconnect, and stores it in the CPU cache.

Local accesses are faster than remote ones since they do not require communication through the interconnect. According to the kind of hardware and implementation details, remote accesses may take more than twice the time when compared to local accesses.

3.2. Multicore Awareness

The multicore-aware optimizations we propose combine the flexibility offered by JOpera's multi-stage architecture with its capability of replicating different instances of the engine and controlling how threads are allocated to a specific NUMA node for 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, because the homogeneous nature of this configuration does not match the heterogeneity of memory access patterns that could be present on multi-CPU and NUMA machines. Instead of a single application instance scaling in the number of worker threads as the number of available PUs increases, multicore hardware has to be addressed with specific approaches to exploit memory locality and to balance the load among all available cores. To tackle these issues, we implemented a multicore-aware design based on replication.

With this approach, we do not change only the overall number of threads running on the machine but we also modify the way they are mapped to the underlying hardware. The working capacity of the engine is partitioned among multiple replicas depending on the number and the type of CPUs and NUMA nodes. The size of each partition is defined in terms of the number of threads and the memory assigned to each replica, and the partitioning strategy changes according to the hardware architecture. In this way, we mimic a distributed system made by several stand-alone instances of the main application, each executed on a single NUMA node to improve CPU cache usage and local memory accesses.

3.3. Self-Configuration on Startup

To adapt itself to the underlying hardware configuration, JOpera relies on a self-configuration startup strategy. First, a single instance of the engine is executed, which scans the hardware configuration to determine the system architecture. More precisely, JOpera identifies the presence of NUMA nodes, the number of CPUs, the way cores belong to a CPU, and the way a CPU belongs to a NUMA node. The engine also gathers information on levels and sizes of CPU caches to identify, when available, cores sharing a common cache (usually a L2 or L3 cache). In this way, the engine creates a list of

DOI: 10.1002/cpe

Computer	Total number of CPUs	cores	SMT units	nodes	PUs	Freq.	RAM	Year
Xeon-16	4	16	N/A	N/A	16	2.4 GHz	16 GB	2008
Opteron-12	2	12	N/A	2	12	2.6 GHz	64 GB	2009
Nehalem-24	4	24	2	4	48	2.0 GHz	128 GB	2010
P7-32	4	32	4	4	128	3.3 GHz	128 GB	2010

Table I. Hardware used in our experiments.

groups composed of IDs of cores running on the same NUMA node or under the same shared CPU cache.

According to the information collected, the engine decides whether and how many times to clone itself by starting new replicas and forcing the OS scheduler to constrain the execution of each replica within a specific combination of memory and cores belonging to the same NUMA node (also referred to as its "affinity group"). Typically, the replication phase keeps the total number of threads and memory usage constant: if more replicas are instantiated, a smaller number of threads and less memory are assigned to each of them. Each replica receives also a different IP address and/or port number, such that the RequestHandler of each instance can receive requests independently. In such a scenario, requests are forwarded to one of the replicas by using a load-balancer. Further client-server communication can bypass this step by keeping track of the proper replica to use (in case of session-oriented data exchange), e.g., by using HTTP redirection.

Since in this article we mainly deal with hardware performance related to memory and CPU caches on a single machine, we are not using any kind of load-balancer or DNS sprayer on the front-end. In our experiments, requests are distributed among the available instances: clients always equally communicate with the same replica they have been assigned to for testing, in order to generate a similar amount of traffic. Front-end load-balancing is out of the scope of this article.

This adaptive startup procedure, coupled with the ability to "pin" replicas to selected cores and NUMA nodes, allows JOpera to adapt to different hardware configurations, from a single-CPU setup to a multi-CPU deployment, in a transparent and autonomous way. As our experiments will show, a good configuration consists in the creation and allocation of at least one replica for each instance of shared hardware (i.e., one for each NUMA node or shared CPU cache available).

4. TESTING ENVIRONMENT

In this section we describe the testing environment used for our experiments. Detailed specifications are given about the hardware infrastructure, the used tools, and the software settings.

4.1. Hardware Configuration

We use four server computers, each with a different micro-architecture. Since our aim is the creation of software automatically tuning to get the best out of each underlying architecture, we give hereafter a detailed description of the machines used. To improve mnemonics, computers are named after their processor family and number of cores. The hardware characteristics of our testing environment are summarized in Table I.

The first computer (henceforth referred to as Xeon-16) is a Sun Fire X4450 with four Intel Xeon E7340 2.4 GHz CPUs. Each CPU has four cores with 32 kB of L1 cache, and two 4096 kB blocks of L2 shared by two cores (for a total of 8192 kB of L2 per CPU). The total number of PUs for this machine is 16. Xeon-16 is equipped with 16 GB of 667 MHz DDR2 RAM and does not run in a NUMA configuration (that is, all the four CPUs access the same memory banks at the same speed).

The second computer (Opteron-12) is a Dell PowerEdge M605 with two AMD Opteron 2435 2.6 GHz CPUs. Each CPU has six cores, each core with 128 kB of L1 cache and 512 kB of L2 cache. An additional L3 cache of 6 MB is accessible by all cores within the same CPU. The total

DOI: 10.1002/cpe

number of PUs is 12. The machine is equipped with 64 GB of 667 MHz DDR2 RAM. The memory is partitioned into two NUMA nodes of 32 GB. Each CPU is directly connected to one NUMA node, thus giving optimal memory speed results when instructions executed on a CPU are accessing data allocated on the same NUMA node.

The third computer (Nehalem-24) is a Dell PowerEdge M910 with four Intel Xeon E7540 2.0 GHz CPUs. Each CPU has six cores, each core with 32 kB of L1 cache and 256 kB of L2 cache. An additional fully inclusive L3 cache of 18 MB is accessible by all the cores within a CPU. Each core also supports 2 SMT units, for a total of 48 PUs. Nehalem-24 is equipped with 128 GB of 1066 MHz DDR3 RAM. The memory is distributed into four NUMA nodes of 32 GB each, one per CPU. Nodes communicate through the Intel QuickPath Inter-Connect (QPI).

The fourth computer (P7-32) is an IBM P755 with four IBM Power7 3.3 GHz CPUs. Each CPU has eight cores with 32 kB of L1, 256 kB of L2, and 4 MB of L3 cache each. L2 and L3 caches can be accessed by all the cores within one CPU. Each core also supports up to 4 SMT units, for a total of 128 PUs. P7-32 is equipped with 128 GB of 1066 MHz DDR3 RAM. The memory is distributed into four NUMA nodes of 32 GB each, one per CPU. P7-32 uses slot-based modules instead of CPU sockets, where each module is a pluggable daughterboad aggregating a CPU and its RAM.

In order to obtain measurements for a baseline that represents a well-configured system, we apply tuning suggestions directly given by the hardware producers and we run benchmarks to find configurations that yield good performance. We turned SMT off on Nehalem-24 and P7-32 since it was not giving any benefit in our measurements (as confirmed in [23]). We also turned Turbo Mode off on Nehalem-24 to remove unpredictable and non-reproducible speedups given by this technology [24]. For the same reason, all our experiments are performed with speed-step and power-saving functionalities deactivated. Finally, experiments are mainly consuming computational resources and memory access: disk I/O occurs mostly during startup and is not affecting measurements, since JOpera keeps all relevant data in memory. To avoid memory swapping to the hard-disk or spreading across several NUMA nodes, we chose the maximum heap size to match the amount of RAM available on the target machines. NUMA node interleaving (where available) has been turned off.

Computers are connected through a private 1 Gbit LAN, with an average message round-trip time of 0.5 ms. The network is used to synchronize the concurrent execution of experiments among replicas and to aggregate results after each test.

4.2. Software Configuration

Our machines, apart from P7-32, are running under Ubuntu Linux Server 64 Bit 10.10, with a 2.6.35-25-server kernel version. P7-32 runs under Red Hat Enterprise Linux (RHEL) Server for PowerPC 64 Bit version 6.0, with a 2.6.32-71.14.1.el6.ppc64 kernel.

The C code used in Section 5 is natively compiled on each hardware architecture using GCC version 4.4.5 with full optimization enabled (`-O3` flag). On P7-32, C code is compiled by using Advanced Toolchain 4.0 GCC version 4.5.2.

Java is executed in the Oracle Hotspot Server VM 64 Bit version 1.6.0_23. On P7-32, Java is executed in the IBM J9 VM 64 Bit PowerPC version SR9. In both cases, the Java Virtual Machine (JVM) is always started with the `-server`, `-Xms`, and `-Xmx` flags set. Minimum and maximum heap sizes are set to the same value, which is always slightly less than the total amount of memory available on the whole machine or on a single NUMA node, accordingly to the computer used.

4.3. Benchmark Workflow

Benchmarking PEEs is a challenging task [25]. Whereas in our previous work [17, 18] we used different synthetic workflows inspired by common workflow patterns (e.g., Sequence, Parallelism, Iteration), in this article we use a single but more realistic and complex workflow as benchmark.

The workflow has been designed to combine a number of representative control flow and interaction patterns in a single process. As shown in Figure 2 using the Business Process Modeling

DOI: 10.1002/cpe

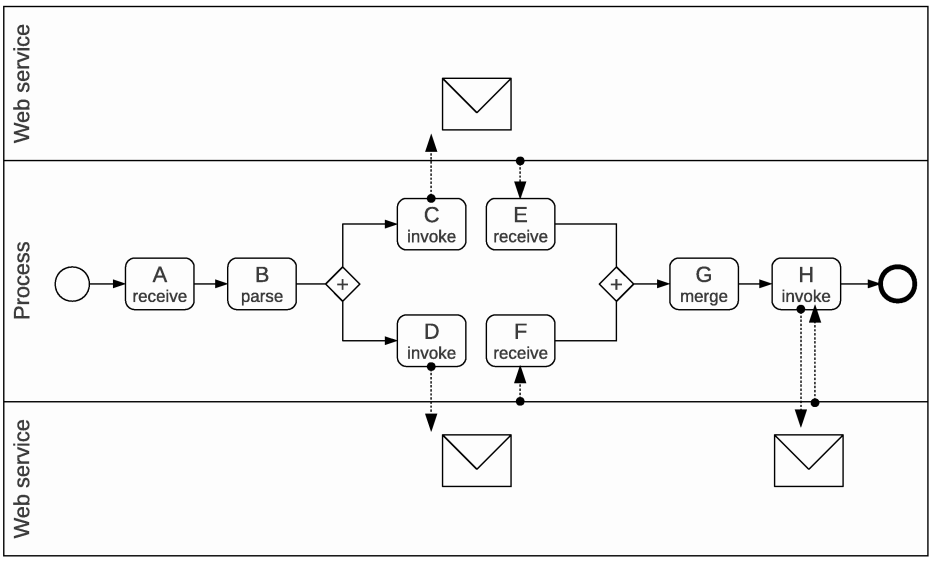


Figure 2. Benchmark workflow represented in BPMN [26].

Notation (BPMN)[†], the process begins with a receive activity A which accepts a message consisting of two string fields. The input message is parsed (activity B) and its content is passed to two parallel activities (C, D) which invoke two asynchronous Web services. The asynchronous responses are received by the following receive activities (E, F) which block the process until both replies are received by the process. The process continues by merging both replies into a single message that is passed to another Web service through a synchronous invoke activity H.

This workflow makes use of synchronous and asynchronous message exchange patterns in combination with sequential and parallel execution control flow patterns, thus stressing most critical paths within the JOpera engine architecture. In all our experiments, processes exchange messages of limited size (less than 1 kB) with the corresponding Web services.

Since our aim is to improve the way JOpera exploits CPU caches and memory locality, external Web services are simulated through a loopback interface directly embedded in the JOpera engine. Compositions are executed by using all the modules provided by JOpera but network traffic is not generated, to avoid bias introduced by I/O operations over the network. In this way, we can make sure that the system is not I/O-bound by traffic generated on the back-end, since no traffic at all is actually being generated.

4.4. Workload Generator

Like other Web services, JOpera receives process requests via its Web service interface. To this end, JOpera uses a RESTful interface based on the HTTP protocol [27]. The way requests can be sent to the front-end is heterogeneous: in our case, we send a unique request asking the engine to execute a specified number of workflow instances. In this way, we avoid the network overhead occurring when a single HTTP request is necessary for each workflow invocation.

Thanks to this approach, we can make sure that the system is not limited by network communication I/O bottlenecks (which are out of the scope of this article) and focus only on what is happening at the memory level and computational resources. For example, with a single HTTP command, we can ask JOpera to execute one thousand processes of a given workflow, with a given list of parameters. We also use the engine Web interface to dynamically and remotely modify

[†]`http://www.bpmn.org/`

DOI: 10.1002/cpe

internal engine settings, such as the kind of performance counters being tracked or the thread pool sizes of the various components.

To synchronize the execution of tests spread over different replicas, we rely on shell scripts using `curl`[‡] to generate HTTP requests that are simultaneously sent to all the JOpera replicas available. In this way, a single HTTP request per replica ignites the whole workload, which is locally generated as a queued list of X requests (where X is a parameter passed through the HTTP request itself). Once all the workload activated by a request has been processed, JOpera returns a list of measurements that we use for the evaluation. These measurements contain metrics such as wall clock times of the process executions, hardware performance counters, garbage collection events, and CPU usage.

4.5. Hardware-Level Tools

In this section we describe the tools we used to perform hardware-related operations such as topology surveying, performance counter retrieval, and manipulating thread and NUMA node bindings.

4.5.1. Numactl and Taskset. A very simple but efficient tool for manipulating the OS policies about NUMA-aware memory allocation is `numactl` [28]. `Numactl` is a command line application that allows users to specify the NUMA node's memory and CPU set to use for the execution of the OS process passed as one of the arguments. In this way, all the memory allocations and threads created by the OS process (including child OS processes as well) will happen only within the boundary specified through the `numactl` parameters.

While `numactl` allows specifying explicitly on which NUMA node's memory and CPU an OS process is executed, `taskset` enables a finer control through the customization of the per-core affinity mask. Affinity masks are used to tell the kernel on which PUs a thread can be scheduled. `Taskset` allows restriction of the execution of an OS process to a series of selected cores.

By using both `numactl` and `taskset`, it is possible to specify an application's memory and computational hardware resources in a detailed way. The main limitations of this approach, typically used for synthetic benchmarks or testing, are the requirement of command line interaction (e.g., a terminal), and that policies cannot be dynamically changed during the execution of the OS process, once the OS process has started.

4.5.2. Overseer API. Our optimization strategy requires many low-level functionalities to identify, monitor, and interact with the underlying hardware. Since these features are not directly available in high-level languages such as Java, we implemented an interface over native methods to support our needs. Our API is called Overseer and is made of several components, including a JVMTI[§] agent and a Java API to be used in application code. Each component provides abstractions for a set of hardware-related functionalities, as summarized below.

Hardware Topology. A detailed report about the underlying machine where the application is running is important for automatic optimization strategies based on hardware features, such as the ones proposed in this article. Overseer analyzes the computer architecture and returns detailed feedback on the hardware topology of the machine. This report includes information on the number of CPUs available, the number of cores per CPU, the number and organization of caches, presence of NUMA nodes and relative packaging, etc.

Hardware Performance Counters. HPCs are registers embedded into processors to keep track of hardware-related events such as cache misses, number of CPU cycles, retired instructions, etc. Counters are vendor and processor-architecture specific and require an abstraction layer allowing for dynamic enumeration and binding to make applications that use HPCs portable. The Overseer API allows listing all the counters supported by the system running the application. Selected counters

[‡]`http://curl.haxx.se/`
[§]`http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/`

DOI: 10.1002/cpe

can then be initialized and assigned for monitoring either to a specific core (per core profiling) or to a specific thread (per thread profiling). Since hardware counters are directly implemented at the processor level, their overhead on performance is negligible. Counters are commonly used as efficient profiling instruments [29].

Affinity and Scheduling. While hardware surveys and HPCs are mainly passive tools to acquire knowledge from the machine and an application runtime behavior, Overseer also offers active methods to influence the way threads and memory allocations are managed by the OS kernel. The Overseer API provides a dynamic and per-thread implementation of the functionalities found in `numactl` and `taskset`. That is, at runtime, it is programmatically possible to change the affinity mask of a given thread and the NUMA node where its memory allocations are performed. These features, coupled with the hardware survey, are the foundations of our hardware-aware optimizations.

Thanks to Overseer, we are able to accurately embed hardware-level monitoring in the software, giving a more detailed and fine-grained feedback about the way the machine is actually executing specific portions of code. Unlike other profiling tools, Overseer becomes part of the JOpera engine itself and produces reports about selected hardware-level metrics for a group of threads specialized in a task (Kernel-specific, Invoker-specific, etc.).

By using the native JVMTI agent loaded with the JVM, Overseer intercepts events such as thread creation and termination to dynamically configure core affinity masks and NUMA node bindings, and to track HPCs on a per-thread basis. For example, we can observe the impact of cache misses occurring within the Kernel or the Invoker thread pools, or we can measure the amount of CPU time used by the embedded HTTP server and client pools. Finally, using HPCs provided by the hardware, the overhead incurred by the usage of Overseer is negligible and therefore measurement perturbations are reduced.

More details about the Overseer API are given in [30], and the software is publicly released[¶].

5. EXPLORING HARDWARE PERFORMANCE WITH A SYNTHETIC MICRO-BENCHMARK

In this section we use a synthetic benchmark to measure and discuss the impact on performance of a memory-intensive application running under optimal respectively under intentionally inefficient NUMA and CPU cache configurations.

5.1. Experiment

Our first experiment aims at revealing the relation between performance and the hardware topology of the machines in our testing environment. In this section we use a synthetic benchmark to measure the impact on execution time of a memory-intensive application running under intentionally inefficient settings (allocating all the data on a remote NUMA node) versus an optimal configuration (using local memory only). The goal of this first series of tests is to explore possible differences in performance that can be observed by running a memory-intensive application. Additionally, these first examples give an overview of differences in the computational power among the computers used for our evaluation. Later, we will show that the best multicore machine (according to its specifications) is not necessarily the fastest one in practice when complex software is executed "as is".

We use a simple micro-benchmark written in C which allocates 1 GB of memory using an array of bytes. Each array entry is read, incremented by one, and stored back in memory. The test allows two variants: *linear* and *random*. The linear option accesses the elements of the array one after the other in a sequential way. The random option accesses them in a randomized way. To make sure

[¶] `http://sosoa.inf.usi.ch/`

DOI: 10.1002/cpe

(a) Time summary (seconds)

Computer	*linear*			*random*			
	remote	local	speedup	remote	local	speedup	prefetch eff.
Xeon-16	4.41		N/A	102.95		N/A	N/A
Opteron-12	4.25	3.35	21.1%	70.99	61.10	13.9%	18.2
Nehalem-24	5.97	4.69	21.4%	63.51	52.74	16.96%	11.2
P7-32	1.88	1.32	29.8%	135.70	100.99	25.6%	76.5

(b) HPC summary for the linear test

Computer	hardware performance counter	*linear*	
		remote	local
Xeon-16	RESOURCE_STALLS	8,687,997,291	
Opteron-12	DISPATCH_STALL_WAITING_FOR_ALL_QUIET	109,194	73,548
Nehalem-24	RESOURCE_STALLS	8,144,458,517	4,946,470,009
P7-32	PM_CMPLU_STALL_DCACHE_MISS	3,811,559,957	965,369,119

(c) HPC summary for the random test

Computer	hardware performance counter	*random*	
		remote	local
Xeon-16	RESOURCE_STALLS	245,895,004,685	
Opteron-12	DISPATCH_STALL_WAITING_FOR_ALL_QUIET	3,730,706	2,502,472
Nehalem-24	RESOURCE_STALLS	110,051,287,543	90,840,446,862
P7-32	PM_CMPLU_STALL_DCACHE_MISS	442,596,280,637	326,702,174,576

(d) Correlation time/counters

Computer	hardware performance counter	correl.
Xeon-16	RESOURCE_STALLS	N/A
Opteron-12	DISPATCH_STALL_WAITING_FOR_ALL_QUIET	0.986
Nehalem-24	RESOURCE_STALLS	0.999
P7-32	PM_CMPLU_STALL_DCACHE_MISS	0.999

Table II. Linear and random array processing using remote versus local NUMA node settings, tested by performing a series of simple operations on an 1 GB array. Speedups in (a) refers to local over remote settings. Time results are expressed in seconds. The prefetcher efficiency is computed as the ratio between the best random and linear runs. Xeon-16 does not feature a NUMA architecture. Tables (b) and (c) report the HPCs measured during the experiments. Table (d) contains the correlation between execution times and HPCs.

that both tests execute the same amount of operations, the access path order is stored in a separate array. In the linear case, this additional array contains values sequentially sorted from 1 to 2^{30}. In the random case, the array is first filled with sequential values that are then shuffled until there are no two consecutive sequential accesses in a row. In this way, the main loop of the test is identical for both the linear case and the random case: only the order in which memory is accessed is changed. To guarantee repeatability of the experiment, randomization uses a fixed seed.

Since most of the computers in our testing environment have large last-level caches, the usage of a large memory array makes sure that caching is only partially compensating for the slow-down incurred when remote memory is used. Some machines also feature hardware memory prefetching, which typically improves sequential data access by preloading adjacent memory locations into CPU caches. To bypass these optimizations and to have a more precise idea of how each computer deals with non-linear memory access, we added the previously described random pattern as an artificial worst-case to exclude hardware prefetching contributions. The linear or random tests are started by passing a specific command-line parameter to the benchmark: in this way, the code is compiled once and we make sure that the main-loop code executed is exactly the same in both cases.

To measure the role of prefetching and the impact of remote versus local memory, we started both tests by forcing the OS to allocate all the data on a specific NUMA node and to execute the program on a single core only. Execution times are reported in Table II. The *remote* setup allocates memory on a NUMA node while it executes the application on a core belonging to a different

DOI: 10.1002/cpe

NUMA node, thus making all memory access through the interconnect. The *local* configuration, instead, refers to the benchmark executed on a core belonging to the same NUMA node where all the memory is allocated (100% local memory accesses). The Xeon-16 computer does not use a NUMA architecture, thus a single result is reported to give a comparison of its performance with the other machines. The information provided by these experiments is completed through the measurement of memory latency-related hardware performance counters to have a more precise idea on what is happening inside each machine.

The results are very stable among multiple runs under the same configuration: the standard deviation after 5 runs is below 1%.

5.2. Discussion

This experiment shows how two important aspects of modern processor architectures have strong influence on performance: hardware prefetching and data locality. To measure the latency introduced by inefficient prefetching and remote memory usage, we used several performance counters. On Intel x86 architectures, we used the RESOURCE_STALLS counter. According to Intel's documentation[‖]: "This event counts the number of cycles when the number of instructions in the pipeline waiting for retirement reaches the limit the processor can handle. A high count for this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, and other instructions that depend on these cannot execute until the former instructions complete execution). In this situation new instructions cannot enter the pipe and start execution". Since HPCs are architecture-specific, we used other (similar) ones on Opteron-12 (DISPATCH_STALL_WAITING_FOR_ALL_QUIET) and P7-32 (PM_CMPLU_STALL_DCACHE_MISS): like RESOURCE_STALLS, a high number of events means higher latency times and less efficiency of memory-related operations.

Linear tests show that *local* memory accesses are faster than *remote* by about 24.1% on average (21.1% on Opteron-12, 21.4% on Nehalem-24, and 29.8% on P7-32). Linear tests are also performed very quickly: hardware prefetch keeps CPU caches filled with useful new data while cores process the information already available. Latency-related HPC events (see Table II(b)) are lower on *local* settings, since the CPU pipeline spends less time waiting for its cache to be filled with data coming from near memory.

Things become more complex when a less obvious access pattern is used (random test). Hardware prefetching is no longer able to provide a continuous stream of useful data to the CPUs, which have to wait for the information to be retrieved from memory into caches. Cache memory is filled with unnecessary adjacent data that is evicted by the following prefetching. Random tests are slower than linear ones by large factors (reported in Table II(a) as Prefetcher Efficiency). While such a difference might surprise at first glance, it is important to remember that our benchmark is explicitly conceived as a worst-case scenario to break the efficiency of the hardware prefetcher. In fact, with purely random memory accesses, the hardware prefetcher is downgrading the performance of the system, since unnecessary data is retrieved and stored in caches, wasting time and resources. The explosion of HPC stall events measured during the random experiment (Table II(c)), when compared to the low values measured in the linear case (Table II(b)), clearly show how stressed the hardware is by the aleatory pattern. Even in this case, *local* settings bring lower but still relevant speedups over *remote* by about 15.4% on average (13.9% on Opteron-12, 16.96% on Nehalem-24, and 25.6% on P7-32).

Since the difference between the linear and random tests is very large, we made an additional measurement using the PERF_COUNT_HW_INSTRUCTIONS generic event to count the total number of CPU instructions executed during each run. This value is constant independently of the option being used (linear or random) and the memory allocated (*local* or *remote*). This confirms that the differences in the execution times are exclusively coming from the latency between CPU and memory and not from a different code path being executed.

[‖] `http://software.intel.com/`

DOI: 10.1002/cpe

As a matter of fact, the performance of our benchmark is significantly improved or degraded by simply changing an OS-level parameter such as the way NUMA node bindings are managed. These results are in line with what we expected, since memory latency on remote NUMA nodes is always higher than on local nodes [31].

Despite the synthetic nature of this benchmark, our measurements show that there is a significant benefit when an application is executed under a proper NUMA configuration. Unfortunately, most of the default OS distributions and software available do not explicitly care about this hardware characteristic, which is often limited to server machines equipped with several CPUs. Default kernels may start an application on a CPU and then migrate its threads several times to different CPUs, invalidating benefits coming from local NUMA nodes and from shared CPU caches. Since memory is typically allocated on the local node connected to the CPU where the application is started, after migration of a thread to a CPU on a different NUMA node, all memory accesses become remote, decreasing performance and responsiveness at runtime. Human intervention is required to manually decide how to tune the OS settings for a specific application. Because of the heterogeneity of multicore architectures (aggregating cores and memory into a variable number of NUMA nodes with characteristics that are different on each computer), optimal settings are often machine-dependent and not always trivial to identify.

From this point of view, the best candidate for such kind of decisions is usually the application developer, who precisely knows how the software is behaving. The developer can allow the application to automatically scan the hardware and decide how to tune itself for optimal performance. In this article we explore this approach in detail.

Thanks to this test, we can conclude that hardware prefetching and memory allocation plays a critical role on modern machines. While with a synthetic benchmark written in C it is easy to stress/tune the system according to these settings, when developers deal with more complex and realistic applications (such as JOpera) it is difficult to reproduce similar results. In addition, in Java it is more difficult to predict how the JVM will compile and optimize some portions of code. For these reasons, in the following sections we will focus on techniques that boost JOpera's performance thanks to improved locality.

6. BASELINE

While the previous experiment was a synthetic benchmark to quantify the impact of remote versus local memory accesses on multicore architectures, from this section on we focus on real middleware by optimizing JOpera. JOpera is a well established software in its community, targeting both the academic and the industrial world, and thus is a good example of a complex multi-threaded application requiring high scalability.

6.1. Experiment

In this experiment we measure the performance baseline with a simple execution "as is" of JOpera on all the computers in our testing environment, as it would perform once downloaded from the official web site and executed. The baseline shows the performance that is automatically achieved out-of-the-box, trusting the way OS schedulers work by default.

We do not yet setup any replication system (as discussed in Section 3) nor NUMA node bindings, but instead we entirely rely on the OS scheduler and on the Java runtime. To see how well the engine scales, we repeat the same test with different Kernel and Invoker thread pool sizes, ranging from 1 to 64 threads by adding a thread to each pool at each step. Each point in Figure 3 is computed by measuring the time required (y axis) to process 128000 instances of the Benchmark workflow introduced in Section 4.3, starting by using 1 thread, then 2, 3, ... up to 64 (x axis). This number of instances (128000, kept constant for all experiments from now on) is a value satisfying several constraints at the same time: it is large enough to ensure robust, repeatable measurements, it allocates a reasonable amount of memory on all the machines, and it is a number that can easily be divided by powers of two, useful for our forthcoming experiments with replicas. To avoid noise

Prepared using cpeauth.cls
DOI: 10.1002/cpe

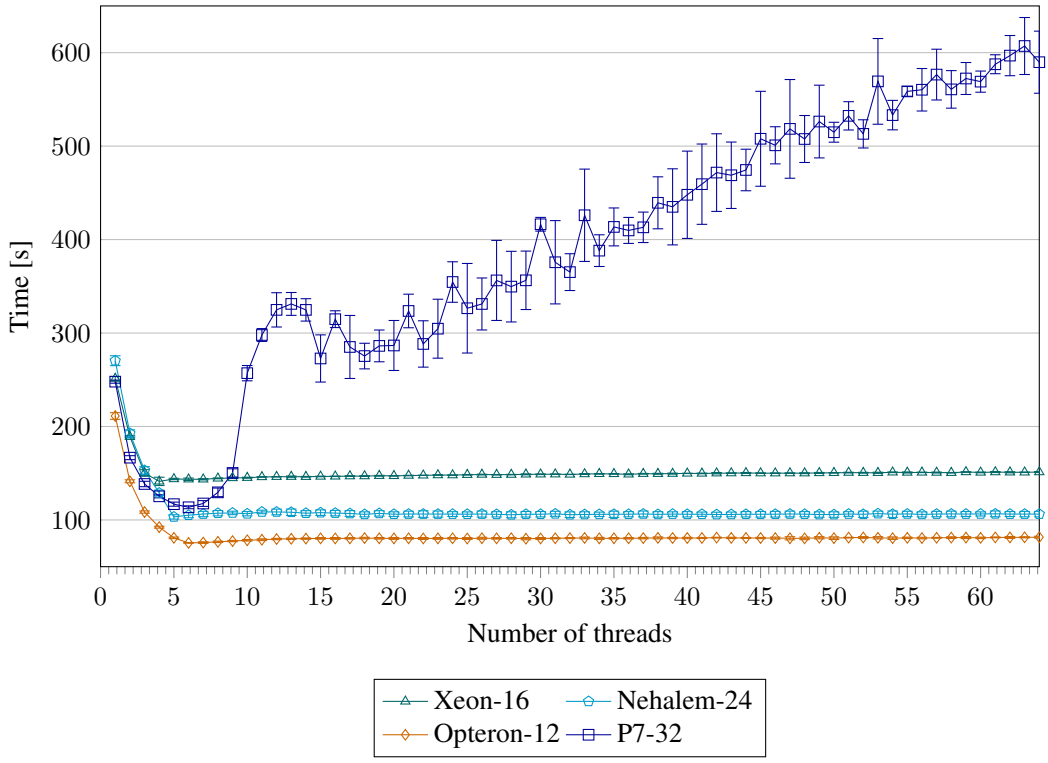


Figure 3. Naïve scalability test resizing JOpera's Kernel and Invoker thread pools from 1 to 64 threads with a step of one thread. At each step, we present the average of 5 runs with standard deviation processing 128000 instances of the Benchmark workflow.

introduced by the Java runtime (e.g., just-in-time compilation or garbage collection), we repeat each run 5 times and we plot the arithmetic mean and standard deviation of the measurements.

6.2. Discussion

Our measurements show the scalability limit hit by JOpera using an increasing number of threads for its thread pools. Despite the large amount of memory and cores available on the machines in our testbed, a larger number of threads does not automatically translate into any significant speedup. Significant improvements are observed only within a limited range of thread pool sizes: 4 for Xeon-16, 6 for Opteron-12, 5 for Nehalem-24, and 6 for P7-32.

Over this threshold, additional threads are only increasing the amount of resources used without giving any significant additional speedup. On P7-32, which is running on a different CPU architecture, OS, and Java environment than the other machines, the system even becomes unstable when larger thread pools are used.

P7-32 is the computer with the most sophisticated hardware architecture, using daughterboards for each NUMA node and shared caches both at the L2 and L3 levels (as discussed in Section 4). The IBM J9 JVM also differs from the Oracle HotSpot JVM used on the other machines: J9, by default, uses all the idle machine cores available to schedule its internal threads for code optimization and garbage collection (even for single-threaded Java applications). If we allow the OS scheduler to freely schedule all the J9 JVM threads over all the available cores, performance rapidly decreases because of the mixed usage of local and remote memory reducing the cache efficiency. As reported in Table II, P7-32 is the fastest machine when memory is properly accessed but also the slowest when less predictable patterns are used.

Measurements show that slightly over-sized thread pools are not significantly detrimental to the application: the only overhead paid is a bit more memory consumed but there is no major

DOI: 10.1002/cpe

performance penalty. The only exception is P7-32, where over-sized thread pools immediately increase execution time and introduce jittering.

This chart also shows that the fastest machine on an execution "as is" is not always the most recent one, nor the one with the highest number of cores. The fastest execution times have been recorded on Opteron-12 with 6 threads per pool. Opteron-12 is the server with the smallest number of PUs of the whole testing environment. Nevertheless, it performs similarly to a machine like Nehalem-24, which is one generation more recent, features more cores, and uses faster memory. As a matter of fact, a comparison of the specifications discussed in Section 4 and results obtained through this test reveals a gap between the kind of performance expected from certain classes of hardware and what is concretely measured.

Obviously, hardware is not the only factor that has an impact on performance. The question is how well the software is designed to take advantage of the hardware. For these reasons, in the next sections we will improve the scalability of JOpera by applying two optimization techniques. The first one is based on server consolidation through the creation of several replicas of JOpera. The second uses hardware-awareness to match replicas to specific components of the underlying hardware, thus improving locality through optimal machine settings.

7. OPTIMIZED SYSTEM

At this point we have a more precise idea about the performance of the computers in our testing environment and a first confirmation that a real application executed "as is" is not scaling as desired. In the following experiments, we address this issue by applying the optimization strategies described in Section 3 to JOpera. We perform two experiments: the first one uses a fixed amount of memory and threads to be divided among an increasing number of replicas. This experiment shows how to optimally share some fixed computational resources. In the second experiment, we allocate new resources for each replica until the system saturates, to show to which extent the replication process improves performance on a machine entirely dedicated to the execution of JOpera.

This section is structured as follows: in 7.1 and 7.2 we describe and discuss our experiments using replicas to improve locality through partitioning of a fixed amount of resources. In 7.3 we investigate the role of garbage collection on our results. In 7.4 and 7.5 we measure the efficiency of the replication mechanism using resources until system's saturation, while in 7.6 and 7.7 we evaluate replicas on a non-NUMA hardware architecture.

7.1. Scaling JOpera: Replication Using a Fixed Amount of Resources

In the previous section we have shown that a simple approach based on just increasing the number of threads is not enough to efficiently exploit the computational power offered by multicore machines. Despite the multi-threaded nature of the JOpera engine architecture (as described in Section 2), sequential code sections are limiting the benefit of parallelization and scalability of the whole application according to Amdahl's law [32].

To mitigate these limitations, in this experiment we use an approach based on server consolidation. Instead of using a single instance of JOpera to process 128000 requests, we create several instances (replicas) and distribute requests among them. In this test we fix the total number of threads of JOpera's internal pools to a constant value and replicate the whole engine (including the JVM) by keeping the total sum of these threads constant among all replicas. For example, if the total number of threads is 32 and we use a single replica, its Kernel and Invoker thread pools will have 16 threads each. Two replicas will use 8 threads per pool, four replicas 4 threads, etc. This approach keeps the same total number of threads but changes the way the threads are running in different JVM processes instead of in a single one. For each machine, we allocate a maximum number of threads corresponding to twice the total number of cores to be able to run at least one thread of both the Kernel and Invoker on each PU. The total number of threads corresponds to 32 on Xeon-16, to 24 on Opteron-12, to 48 on Nehalem-24, and to 64 on P7-32. The same procedure is

DOI: 10.1002/cpe

applied to allocate the memory, where the total amount is fixed at 16 GB on all machines and then divided according to the number of concurrent replicas.

This experiment is also executed with an additional variable: NUMA node bindings. In the simplest case (referred to as *default*), we run each replica without specifying any kind of explicit bindings to a NUMA node, that is, each replica is executed by using the OS default NUMA policy. The OSs used in our testing environment, by default, allocate memory locally to the same NUMA node where the allocation request has been generated but do not prevent thread migrations from being scheduled on all the PUs available.

In the other case (*local*), we force the execution of each replica to a specific NUMA node. The *local* configuration makes sure that memory allocations and code execution happen always and only within the same NUMA node, removing any remote memory access (although only a subgroup of PUs is used for execution). This goal is achieved by preventing the OS scheduler from migrating threads belonging to the same application on cores not being part of the initial NUMA node. The whole application lifetime happens within the NUMA node (memory and cores) it has been initially assigned to. Since Opteron-12, Nehalem-24, and P7-32 feature large caches shared among all the cores of the same CPU, and since the hardware architecture of these machines has one CPU per NUMA node, the *local* configuration also improves last-level cache usage.

This comparison is not performed on Xeon-16 because the machine does not have a NUMA architecture. In Section 7.6 we present a specific strategy to improve performance on Xeon-16 by exploiting shared CPU caches only.

7.2. Results and Discussion

Results are reported in Figure 4. Each point is computed as the average value after 5 independent runs. For each run, the value retained is measured after two warmup runs to give enough time to the JVM to apply its optimization policies. Results measured in this way are very stable with a standard deviation below 3% (Xeon-16, Opteron-12, and Nehalem-24) and 5% (P7-32). When closer to the fastest deployment configuration, standard deviation is also lower and below 2%.

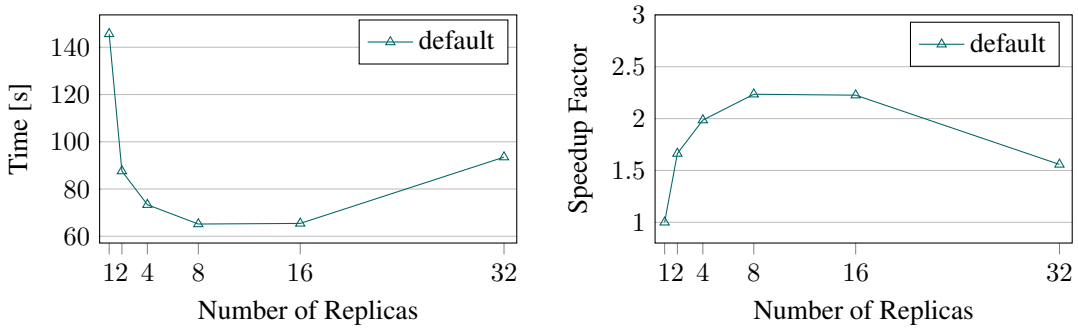
Speedup factors are computed on the four machines by using the execution time scored by a single replica with *default* settings as baseline. Measurements reported in the charts show that the mechanism of replication is bringing a concrete speedup for all the targeted computers. If we just consider the *default* configuration, the usage of replicas over a single instance leads to speedup factors of 2.2 (Xeon-16 with 8 replicas), 1.7 (Opteron-12 with 8 replicas), 2.8 (Nehalem-24 with 16 replicas), and 10.5 (P7-32 with 32 replicas).

The results are even better when we consider the additional boost in performance given by hardware-aware NUMA node bindings. The *local* configuration always reduces execution times over *default*, scoring speedup factors of 1.9 (Opteron-12 with 8 replicas), 3.4 (Nehalem-24 with 8 replicas), and 13.8 (P7-32 with 16 replicas). Proper *local* NUMA bindings translate then into an additional speedup over *default* of about 11% on Opteron-12, 21% on Nehalem-24, and 31% on P7-32.

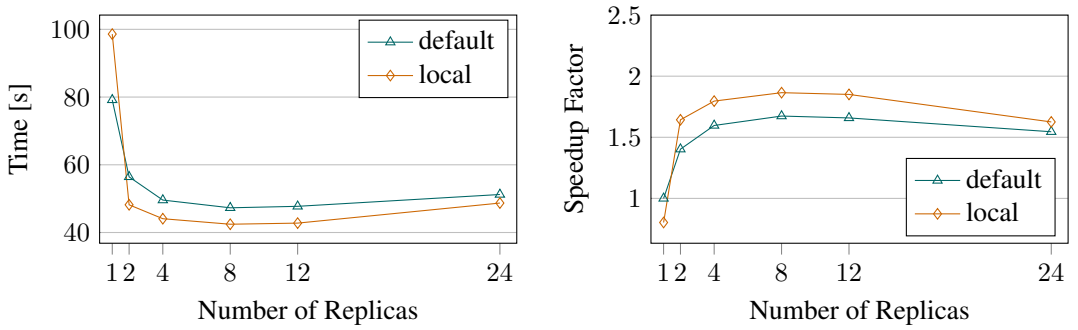
The only exception where the *local* configuration is not performing faster than *default* happens when a single replica is used. In this case, the *local* configuration limits the execution of the whole JOpera engine to a single NUMA node, both for what concerns memory and cores used. For example, a single instance on Nehalem-24 runs on all the 24 cores available in the *default* configuration, but only on 6 cores when the *local* policy is used. This means that the benefit obtained from the adoption of one replica optimally running only on a subset of the available hardware is outperformed by an inefficient default configuration being executed on all the available hardware. For this reason, the gap between 24 and 6 cores is too large to be offset by improved memory locality. This drawback disappears by running two instances with *local* settings over two replicas using *default* parameters: in that case, 12 cores interacting with their local memory perform better than 24 cores freely managed by the OS scheduler.

Figure 5 illustrates in a single chart the results obtained through the *local* configuration on all computers. If we compare the baseline (using a single instance) with a point where all the machines

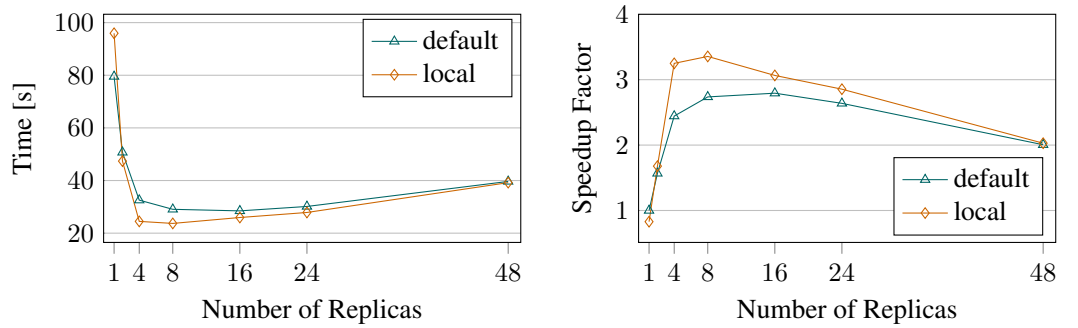
DOI: 10.1002/cpe



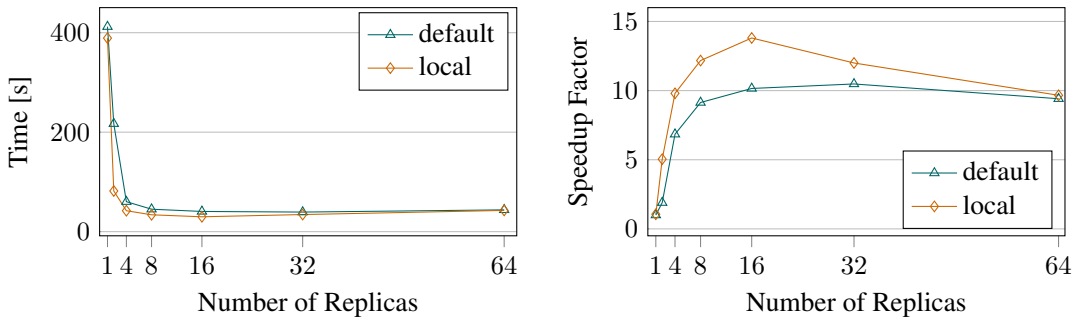
(a) Xeon-16



(b) Opteron-12



(c) Nehalem-24



(d) P7-32

Figure 4. Execution time and speedup given by processing the same total number of requests by one or more replicas. Each replica is executed either "as is", without any explicit binding (default), or by using optimal NUMA settings (local).

DOI: 10.1002/cpe

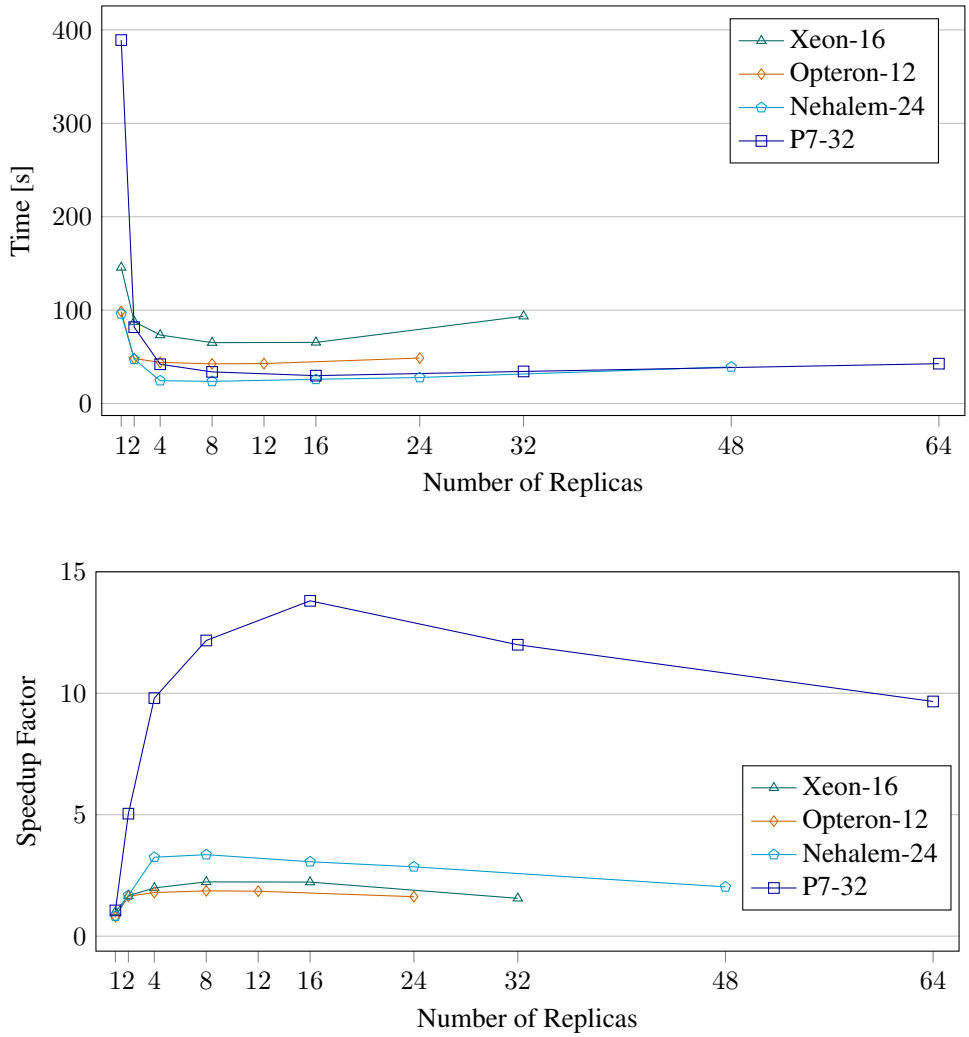


Figure 5. Summary of the *local* results (*default* for Xeon-16) obtained on the targeted machines. The speedup factors are computed relatively to the execution time measured with a single replica and *default* settings.

are close to their best performance (for example using 16 replicas), we observe how the machines with higher specifications (reported in Section 5) are finally performing better.

By using a single instance, the best results are not always obtained on the best computer (as seen in Figure 3), while the replication mechanism coupled with the hardware-aware configuration outperforms the baseline and unleashes the computational power of each multicore machine.

With 16 replicas, Nehalem-24 becomes the fastest computer, followed by P7-32 showing similar execution times. Opteron-12 is third, initially performing like Nehalem-24 but fading with more than 2 replicas (as expected, since this computer features 2 CPUs and 2 NUMA nodes, while both Nehalem-24 and P7-32, equipped with 4 CPUs and 4 NUMA nodes, scale well until 4 replicas). Xeon-16 is the slowest machine of the testing environment, also showing the lowest speedup factors. Its behavior is interesting because Xeon-16 is the machine getting less benefit from replication, despite its 4 CPUs and a higher number of cores when compared to Opteron-12. The reason is that Xeon-16 is not using a NUMA architecture and thus all the cores are competing for memory access on the same bus, while other machines have 2 (Opteron-12) or 4 (Nehalem-24, P7-32) independent NUMA nodes. When JOpera runs with the proper configuration, we measure benefits coming not only from the pure computational power (that is, the number of cores), but also from the way these resources are packaged into the hardware. When the software uses replicas to mimic the distributed

DOI: 10.1002/cpe

nature of the underlying multicore hardware, and when these replicas use NUMA node bindings to accurately match the hardware architecture, we obtain the highest peak performance.

Another additional information given by this experiment is the correlation between the number of replicas and the number of NUMA node available on each machine. The speedup factor curve starts to flex once one replica per NUMA node is allocated. That is, after 2 replicas on Opteron-12, respectively after 4 replicas on Nehalem-24 and P7-32. This is interesting because there are several hardware metrics involved, like the total number of cores, of CPUs, etc., but the only one correlated with the speedup factor curve is the number of NUMA nodes. In fact, our experiments are making an intense usage of memory, achieving peaks of 14/15 GB, justifying the weight that proper memory settings play in our scenario.

Finally, the behavior shown by P7-32 requires some additional considerations. The speedup factors obtained on this machine may seem surprising at a first glance, but they are the consequence of its poor baseline. If we consider the execution times scored with 1 replica reported in Figure 5, P7-32 takes almost 400 seconds to perform the same task that Opteron-12 and Nehalem-24 complete in only 100 seconds. This makes P7-32's baseline the worst of our testing environment, by a factor of 4. If we consider how P7-32's performance improves with the usage of replicas, we see (in Figure 5) that its performance is more in line with the other machines, ending up with results close to Nehalem-24. This very different performance can be explained by the fact that P7-32 features a different processor family (Power versus x86), a complex hardware architecture (with daughterboards for each processor and a more complex CPU shared-cache system), a different OS (Red Hat versus Ubuntu), and a different Java VM (IBM J9 versus Oracle Hotspot).

So far, our observations would suggest a rule of thumb in the form of "allocate at least one replica for each NUMA node available". We will see later how to refine this guideline into a more general rule that works for non-NUMA hardware as well.

7.3. Impact of Garbage Collection

Here we explore the impact of Garbage Collection (GC) on the results previously discussed. Each replica uses its own memory manager, thus reducing the total amount of available and used memory.

We measure the impact of GC on execution time with two metrics: the total number of GC events and the total cumulative time spent by the JVM during GC operations. This information is directly provided by the Java runtime through the GarbageCollectorMXBean management interface. We repeated the experiment described in Section 7.1 on each JVM used in our testing environment (on Opteron-12 with Oracle HotSpot and on P7-32 using IBM J9), configured to log the GC metrics.

Since GC fine-tuning is beyond the scope of this work, we use default settings. These settings correspond to the parallel GC on both Oracle HotSpot (i.e., *-XX:+UseParallelGC*) and IBM J9. We do not activate the NUMA-aware GC (*-XX:+UseNUMA*) because this is not a default setting, even when the *-server* flag is used on a NUMA machine. Using a NUMA-aware GC can improve the baseline (as showed in [31]) but requires additional testing, since memory objects could be copied to the wrong node and affect performance.

Results are reported in Table III and illustrate two significant aspects: first, the total number of GC events per replica is almost identical, independently of the number of replicas, the JVM being used, and execution times (wall times). As in the previous experiment, we progressively reduce the amount of memory assigned to each replica, in order to keep a constant total amount of memory (16 GB) across all replicas. A higher number of replicas means less heap memory assigned to each one, but also a smaller number of Benchmark workflow instances to execute and smaller Kernel and Invoker thread pools. GC events alone cannot be correlated with the speedup observed through the replication mechanism. Also, the use of *local* memory is not influencing the total number of GC events in comparison with the *default* configuration.

Second, the total amount of time taken by GC is only marginally affected by the number of replicas. With Oracle HotSpot JVM (Opteron-12), there is a significant time decrease only from 1 to 2 replicas (by 30%, from about 13.7 to 9.5 seconds). GC times are then very similar while wall times always decrease or remain stable until 24 replicas are used. With 24 replicas, GC time significantly decreases again (by about 6-10% when compared to the previous table entry), but wall

Prepared using cpeauth.cls
DOI: 10.1002/cpe

(a) Results obtained on Opteron-12 (Oracle HotSpot JVM)

Number of replicas	RAM per replica	*default*			*local*		
		wall time	GC events	GC time	wall time	GC events	GC time
1	16 GB	79.1	43	13.7	98.6	43	18.3
2	8 GB	56.4	43	9.5	48.2	43	9.6
4	4 GB	49.6	43	11.1	44.1	43	10.5
8	2 GB	47.3	43	10.1	42.5	43	9.1
12	1,37 GB	47.7	41	9.4	42.8	41	8.6
24	682.7 MB	51.2	39	7.6	48.7	39	7.0
Avg.				10.2			10.5

(b) Results obtained on P7-32 (IBM J9 JVM)

Number of replicas	RAM per replica	*default*			*local*		
		wall time	GC events	GC time	wall time	GC events	GC time
1	16 GB	412.2	18	21.5	389.2	18	27.6
2	8 GB	217.1	18	14.1	81.7	18	26.3
4	4 GB	60.2	18	15.9	42.1	18	14.9
8	2 GB	45.1	17	11.7	33.9	17	11.9
16	1 GB	40.6	19	9.7	29.9	19	11.4
32	512 MB	39.3	20	10.1	34.4	20	14.9
64	256 MB	43.8	25	10.7	42.7	25	16.1
Avg.				13.4			17.6

Table III. Gargabe Collection test performed on Opteron-12 (top) and P7-32 (bottom). Times are expressed in seconds.

time increases. In addition, the GC does not seem to take a significant advantage from the execution on *local* settings: average *local* GC times are similar to *default* GC times, while *local* wall times are always faster than *default* ones. The main exception happens with a single replica and *local* settings, but (as explained in the previous section) this is a consequence of the number of PUs used: 1 *local* replica runs on a single NUMA node with 6 PUs, while 1 *default* replica uses all the 12 cores available on Opteron-12. Despite the improved locality, less cores are used by the *local* configuration to perform the same amount of computation.

For the sake of completeness, we repeated this experiment by extending parallel GC also to the old generation collections (using the special option *-XX:+UseParallelOldGC*). On average, this setup gives an additional 15–20% speedup of the GC times measured in both the *default* and *local* settings. Both configurations run slightly faster in absolute terms (since GC is taking less time), but show similar behavior and relative performance as discussed previously.

The GC times measured on IBM J9 (P7-32) are only marginally correlated with the measured wall times. Like in the previous case, there is a significant GC time difference between using a single replica and 2 or more. GC on IBM J9 is less influenced by *local* deployment than on Oracle HotSpot: for example, *local* settings using 4 replicas outperform the *default* counterpart, but both record similar GC times. Average GC times on P7-32 are higher when *local* deployment is used, while wall times are shorter: this shows that GC performance is negatively affected by *local* settings, while the rest of the execution is taking a significant speedup from improved locality and optimal NUMA bindings. The reason is that by default, IBM J9 uses all available PUs to run its internal threads to perform GC, even if the running Java application is single-threaded. With *local* settings, we reduce the number of PUs that the IBM J9 process can use on P7-32 from 32 PUs to the number of PUs available within a specific NUMA node. Despite the improved locality, GC is executed by less PUs and its performance is decreased.

7.4. Scaling JOpera: Replication Using an Increasing Amount of Resources

This experiment uses the same workload and methodology of the previous experiments but allocates resources to replicas in a different way. Before, we first fixed the total number of threads (according to the number of cores available) and the total amount of memory to distribute over one or more

DOI: 10.1002/cpe

replicas. That approach was limiting resources and identifying a more efficient way to organize them.

Here, we use a different strategy. We first define a basic replica, with a fixed number of threads and memory, and we scale in the number of replicas by cloning. While the total number of threads per replica in the previous experiment was different on each machine (corresponding to the number of cores available), in this test we keep this value constant to run JOpera replicas with exactly the same parameters on all computers. With this test, we eventually saturate each machine.

For each replica we reserve 8 GB of memory and assign 8 threads to both the Kernel and Invoker thread pools. We chose 8 threads because this value gives on average good results on the four computers in our testing environment, according to the measurements reported in Figure 3. To make things easier, we adopted this value as the fixed pool size for a single replica on all the machines, although they reach saturation with a different number of threads. Since slightly oversized thread pools are not affecting performance, we can ignore this aspect, apart from some small additional costs in terms of memory resources (which are not critical in our experiment).

In this case, too, we execute the whole test with two different configurations: the first time "as is" with *default* settings, the second with NUMA node bindings exploiting hardware locality (*local*). Since we allocate 8 GB of RAM to each replica, this test has been skipped on Xeon-16 because it does not have enough memory to allow more than 2 replicas.

7.5. Results and Discussion

Results are shown in Figure 6. The measurements gathered through this test confirm what we have discussed in the previous experiment. In addition, this test also shows resource saturation points that should not be exceeded to avoid performance degradation. These saturation points are easily identified in the charts of Figure 6 where execution times start to grow again after an initial positive trend.

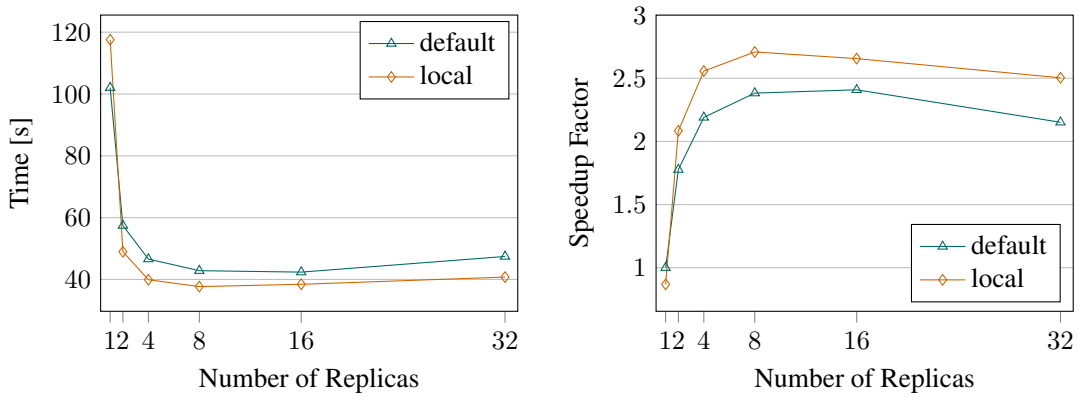
The second information given by this test is that the adoption of the *local* configuration reduces the number of replicas necessary to saturate the machine. If we consider the example of Opteron-12, in the *default* settings, peak performance is obtained with 16 replicas yielding a speedup factor of 2.4. The same test executed with *local* settings only requires 8 replicas to reach a speedup of factor 2.7. Nehalem-24 reaches peak performance with a speedup factor of 5.2 using 32 replicas in *default* setup, versus a factor of 5.5 obtained using 16 replicas with *local* settings.

P7-32 shows a different behavior: on that machine, peak performance is reached with 32 replicas giving speedup factors of 16.2 (*default*) and 22.5 (*local*). This makes the *local* configuration faster than *default* by about 38%, again highlighting the importance of proper NUMA node bindings on a computer with such a particular hardware topology. As explained in the previous experiment, P7-32 performs badly using a single replica (see Figure 6(c)). For this reason, speedups are much higher on P7-32 than on the other machines: this is due to the poor baseline execution, taking close to 400 seconds, whereas Opteron-12 and Nehalem-24 take only about 100 seconds.

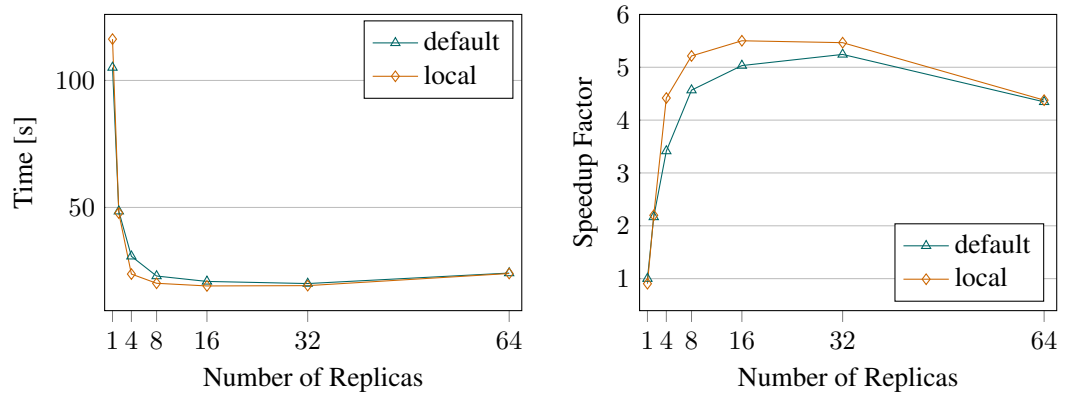
This observation leads to an additional remark. On the one hand, P7-32 is the newest and most expensive machine of our testing environment, but also the one with the most complex hardware architecture. According to its specification, P7-32 is also the most powerful computer of our testbed. On the other hand, all this power comes at the cost of additional care that software developers, either at the level of the OS or application, have to take to fully exploit its resources. Despite the different hardware architecture, OS, and Java VM, our approach clearly show how P7-32 goes from the last position (1 replica, *default* settings, 387.4 seconds) to the first place (32 replicas, *local* settings, 17.3 seconds). For comparison, the fastest execution times on the other machines are 37.7 seconds on Opteron-12 (8 replicas, *local* settings) and 19.1 seconds on Nehalem-24 (16 replicas, *local* settings).

The last information obtained with this test confirms our previous statement about allocating at least one replica for each NUMA node available. The speedup factors grow very quickly up to the point where more replicas than the available NUMA nodes are used. Then, more resources are being used to still achieve performance gains, but in a progressively less effective way. For example, on Opteron-12, 4 additional replicas are required to go from a speedup factor of 2.6 (4 replicas, *local* settings) to 2.7 (8 replicas, *local* settings). That is, 4 additional replicas consuming 32 GB of RAM

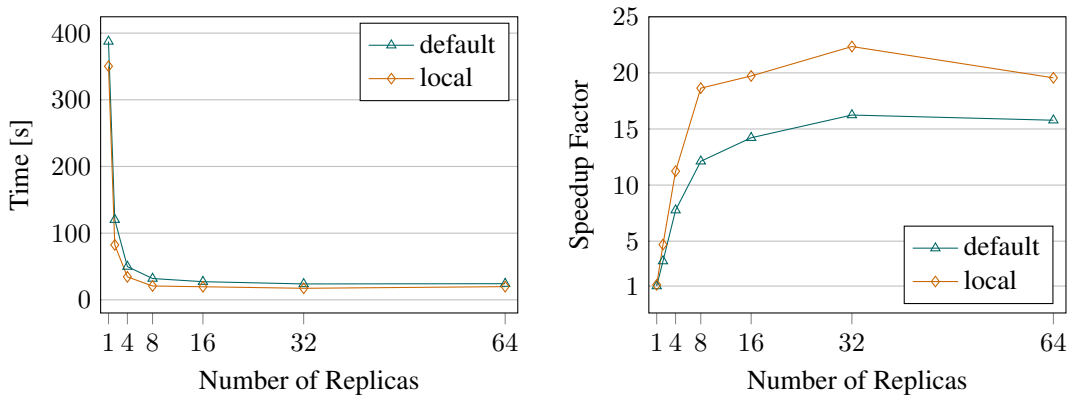
Prepared using cpeauth.cls
DOI: 10.1002/cpe



(a) Results obtained on Opteron-12



(b) Results obtained on Nehalem-24



(c) Results obtained on P7-32

Figure 6. Execution time and speedup given by replicating the engine with a fixed number of threads (8) and a fixed memory size (8 GB). Each replica is executed either without any explicit binding (default) or by using optimal NUMA settings (local).

and other computational resources are needed to get a speedup of less than 4%. Considering that in this test we double the number of threads and the amount of memory consumed at each step, the performance gains achieved may not be worth the resources consumed for that. While exploiting all the available resources makes perfectly sense on a machine dedicated to the execution of JOpera

DOI: 10.1002/cpe

solely, a smaller number of replicas could be used on computers with other applications running concurrently. In such a case, the allocation of one replica for each NUMA node available assures an optimal usage of the available resources.

7.6. Scaling JOpera: Replication on Non-NUMA Hardware

So far, we performed experiments by using replication and (whenever possible) two different NUMA configurations: *default* and *local*. One of the machines used for our tests (Xeon-16) does not feature NUMA hardware. In this section we target exclusively this computer and perform a variant of the test described in Section 7.1, by reusing the same memory and thread pool sizing settings.

Instead of tuning replicas by binding them to a specific NUMA node, we use the way CPU caches are organized to identify an optimal distribution of replicas over several cores. Xeon-16 features 4 CPUs, each one with 4 cores. Within each CPU, cores are aggregated into two groups with 4096 kB of L2 cache in common (refer to Section 4 for more details on Xeon-16 hardware topology). That is, Xeon-16 has in total eight couples of cores under a shared L2 cache. Since extra overhead is incurred if two communicating threads are executing on cores that do not have a shared cache [33], in this test we measure how optimal cache-oriented bindings lead to a concrete benefit, similar to the one obtained through proper NUMA settings. We confirm our observations with measurements based on HPCs to show that metrics such as cache misses are reduced as performance improves.

We use also an additional configuration, for a total of three different settings. The *default* configuration executes replicas "as is", entirely relying on the OS scheduler. The *shared* configuration forces the execution of replicas to cores sharing a common cache. In contrary, the *non-shared* configuration intentionally schedules threads of the same replica to cores without any cache in common. The *non-shared* policy is used to artificially simulate a worst case scenario.

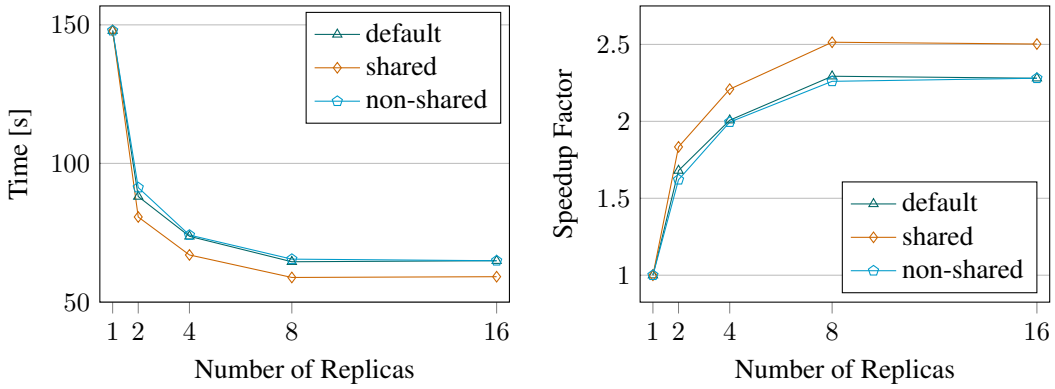
7.7. Discussion and Results

Results are reported in Figure 7. Despite of the lack of a NUMA architecture with several nodes to deploy replicas on, execution times and speedup comparisons among the three different configurations show a behavior similar to our previous observations. While on NUMA machines efficient NUMA settings bring an additional performance gain of about 20–30% over a *default* configuration, shared cache optimizations only result in a speedup of 10% (see the comparison of peak performance using 8 replicas). The *shared* deployment with 8 replicas perfectly fits the Xeon-16 hardware, where each replica runs on a couple of cores under a common L2 cache. In this way, the number of replicas perfectly matches the number of shared caches at the hardware level. With more replicas, there are no further performance gains and performance starts to degrade (charts report more cache misses and slightly higher execution times with 16 replicas than with 8 replicas). We can now refine our statement as "allocate at least one replica for each NUMA node respectively each shared cache".

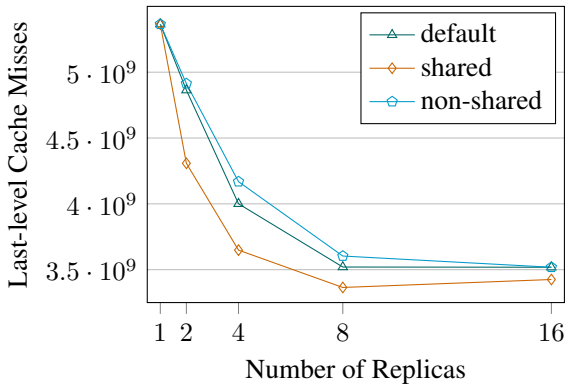
The performance gain of the *shared* configuration is achieved by a better exploitation of processor caches. Each step of the Benchmark workflow is executed by threads of either the Kernel or of the Invoker pools. When a thread from one pool passes its information to another thread of the second pool, the likelihood that this information is already stored in cache is higher when these threads are executed by cores with a common cache. Thanks to this improved locality, data is less often accessed in memory, since the information is already available in CPU caches. For this reason, the number of cache misses during the execution of the test is reduced when the *shared* configuration is used. The correlation (Table IV) between cache misses and both execution times and speedups confirms our assumptions. Conversely, the *non-shared* policy has the worst performance and the highest number of cache misses.

It is interesting to mention that there is no big difference between the result for the *default* and for the *non-shared* configuration. This comparison shows that cache tuning is a delicate operation nullifying most of its benefits when not addressed through a dedicated approach (as it happens in the *default* case): a scheduler not taking care of these hardware characteristics performs similarly to an intentionally misconfigured setup.

DOI: 10.1002/cpe



(a) Results obtained on Xeon-16



(b) Last-level cache misses counter for Xeon-16

Figure 7. Execution times, speedup, and cache misses measured by distributing the same requests to a growing number of replicas. Each replica is executed without explicit bindings (default), by using shared processor caches (shared), or by intentionally interleaving caches to nullify their benefit (non-shared).

correlation	affinity settings		
performance/cache misses	*default*	*shared*	*non-shared*
Time	0.90	0.97	0.91
Speedup	0.97	0.99	0.98

Table IV. Correlation between L2 cache misses and execution times resp. speedup factor.

8. RELATED WORK

Most currently available PEEs do not consider the underlying hardware as a privileged source for performance optimization. Most PEEs rely on replication and distribution-based techniques to increase performance, but such techniques do not yet take into account the underlying hardware topology. Some examples are the engine proposed by Li et al. in [34], the OSIRIS middleware by Brettlecker et al. [35], or previous versions of JOpera. This article complements existing approaches by showing the potential of low-level optimizations for complex parallel applications running on multicore machines.

Moreover, to the best of our knowledge, the idea of considering modern shared-memory multicore machines as special cases of distributed environments for composite service execution has not been addressed by others. Some strong motivating arguments for this vision have been identified by Baumann et al. in [2], where the implications of the distribution are discussed with respect to the

DOI: 10.1002/cpe

design of complex software infrastructures. With this work, we propose a concrete application in the field of high performance composite service execution, and we propose a rigorous approach to hardware-awareness optimization techniques.

8.1. PEEs and SOA Performance

PEEs and middleware have become critical components in modern SOAs [36]. Key aspects for such middleware are performance and scalability. For this reason, many research efforts have been focused on building middleware for high throughput service composition, and several software architectures have been proposed for PEEs.

In [37], Lu et al. propose an architecture based on an event-driven reactive machine with message-passing interaction, implemented using the Concurrency and Coordination Runtime (CCR) available in the .Net framework. The architecture is able to scale and improves performance. However, the evaluation provided does not include any comparison with parallel architectures based on thread pools such as the one presented in this article. We think that hardware-aware thread pool-based architectures may represent a valid alternative to event-based solutions, mainly because of their efficient usage of different machines' memory layout. Other interesting considerations are addressed in [38], where a parallel data-intensive application is tested with different implementations, namely CCR, C+MPI, and Java+MPI. The paper shows that thread-level parallelism issues have to be carefully tuned in order to obtain good performance, but still demonstrates that such thread-based solutions may represent an optimal architectural choice.

In [39], Lin et al. propose a Quality of Service (QoS) management architecture for coordinating services deployed in a common virtualized environment. The research is done considering multicores and virtualization as performance enhancement technologies, and proposes an adaptive QoS-aware architecture able to provide guarantees for QoS contracts. Another interesting approach to service composition and business process orchestration is presented in [40]. The paper introduces a complex, service-oriented architecture for streaming service interactions that extends BPEL to support data-intensive applications. The authors also provide tools for scaling the system by means of dynamic allocation and replication of cloud resources.

8.2. Multicore Performance

Modern chip multi-processors present non-uniform cache hierarchies. Such systems rely on several cores clustered on separate chips, and such cores do not share a uniform amount of cache memory. This asymmetry is a major issue for performance optimization. In [14] the authors explore the impact of cache sharing on multi-threaded programs. While cache sharing can reduce the communication latency between threads, it increases cache contention. For many concurrent programs, cache sharing has insignificant performance impact because of large working sets. The authors point out that it is essential to transform programs in a cache-sharing-aware way in order to benefit from shared caches. In addition, choosing appropriate thread CPU binding can help improve performance.

Another relevant issue for multi-threaded applications on multicore machines is thread migration. In fact, scheduling policies not optimized for specific multicore topologies are commonly implemented in the operating system and impact performance due to an increased number of cache misses [41]. In [12] the authors explore the performance impact of thread migration for concurrent Java workloads. They show that the impact of three factors (migration frequency, the number of migrations that cross L2 cache boundaries, and the working set size) need to be taken into consideration in order to measure the overhead coming from thread migration. For Java workloads, the authors argue that migration frequency is relatively low, but of course it depends on the way multi-threaded applications manage thread pools. The result is furthermore confirmed by the experiments presented in this paper.

In [42] the authors show that some Java benchmarks, such as SPEC JBB 2005 as well as several benchmarks in the SPEC JVM2008 suite, are "partially" scalable. These benchmarks scale well with a smaller number of cores, but the scalability degrades when more cores are enabled. The authors argue that for these benchmarks, scalability is limited by object allocation that consumes

DOI: 10.1002/cpe

the available memory write bandwidth on several multicore platforms. A similar analysis is also performed and extended in [43]. Thanks to our approach, the JOpera engine is not affected by this allocation barrier issue, as objects are optimally allocated according to the underlying hardware architecture.

Similar to our approach, Autopin [44] is a framework for multi-threaded OpenMP applications performance enhancement. It dynamically searches for the optimal thread-CPU binding maximizing a given cost function. The Autopin tool automatically searches among a given set of fixed thread-to-core bindings (defined by the user) for the best configuration in order to exploit thread locality. Each binding layout is called a *pinning*, and the framework adaptively selects the best one by accessing a predefined set of hardware performance counters at runtime. In this way threads monitored by the Autopin framework are migrated to the best available CPU/core. The Overseer API presented in this article provides a rich set of interfaces which could be used to implement some auto tuning techniques similar to the one adopted by Autopin. Developers can use it to build custom binding strategies and to monitor a customized set of hardware performance counters in the same way Autopin does. Indeed, it is possible to see our approach as a *lower level* one, on top of which it is possible to build some kind of self-tuning binding strategy.

Another comparable approach is the one proposed by Tam et al. in [45]: an OS-level thread scheduler for multi-processor multi-cores machines able to identify at runtime the best allocation strategy for each executed thread. Like for Autopin, the scheduler is based on constant hardware performance counter feedback, but unlike Autopin the allocation strategy is done entirely by a smart scheduler able to monitor stall breakdowns and consequently to detect so-called *sharing patterns* in order to obtain a realistic thread clustering aimed at forcing CPU bindings of related threads (i.e., threads influencing the same performance counter, for example, by sharing the same memory). The approach differs from our one in that all the sharing patterns in JOpera are defined by its internal architecture. Furthermore, our approach is more at the application-level.

9. CONCLUSION

Modern multiprocessor 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 new architectures offer high computational power through increased explicit parallelism but also require specific software optimizations to realize performance gains.

In this article we explore how JOpera, an existing, Java-based PEE, can benefit from customized replication and memory/CPU binding mechanisms. By restricting the cores on which certain threads may execute and explicitly allocate memory to selected NUMA nodes, we are able to force threads that are likely to access shared data to execute on cores that share a common memory node and/or CPU cache, resulting in improved thread communication. This approach works particularly well for applications executing parallel and independent requests that can easily be dispatched over several replicas and concurrently executed.

Our design allows the PEE to adapt to the different hardware topologies upon startup. For optimal results, the self-configuration on startup is performed taking into account the number of available NUMA nodes, CPUs, and the way cores and caches are physically mapped. This hardware analysis allows the engine to automatically decide if and how many replicas should be created.

Our extensive performance evaluation on recent and heterogeneous multicore machines confirms that properly defined replication and NUMA node/cache bindings improve performance by up to a factor of 20, according to the underlying system. However, the tuning of the number of replicas and bindings must be done carefully: inappropriate settings can result in a waste of resources and performance deterioration. We have shown that the simple approach following the idea of “just add more execution threads to improve system performance” is not enough. Instead, it is important to consider *how* threads of the engine’s components are mapped to NUMA nodes and cores. Our experimental validation suggests to create at least one replica for each NUMA node, respectively for each shared cache.

DOI: 10.1002/cpe

In our ongoing research work we are exploring fundamentally new middleware designs to further improve performance on recent multicore machines, by considering modern hardware architectures like a distributed system. We are exploring auto-tuning mechanisms inside the engine to refine various performance-relevant configuration parameters, such as thread pool sizes or distribution of replicas at runtime, leveraging monitoring information from HPCs to be used as QoS metrics to dynamically measure the efficiency of the application and to apply optimization policies on the fly. An interesting possible extension of our approach should go beyond self-configuration on startup. It should be possible to dynamically allocate and deallocate replicas as the system load conditions change. A slightly modified architecture can also reuse the replication strategy to improve system reliability, by automatically creating a new replica when another one fails.

ACKNOWLEDGMENTS

This work is funded by the Swiss National Science Foundation with the SOSOA project (SINERGIA grant nr. CRSI22 127386). P7-32 has been kindly provided by IBM as a Shared University Research (SUR) award. The Benchmark workflow has originally been proposed by Dieter Roller.

REFERENCES

1. Hill MD, Marty MR. Amdahl's law in the multicore era. *IEEE Computer* 2008; **41**(7):33–38.
2. Baumann A, Peter S, Schüpbach A, Singhania A, Roscoe T, Barham P, Isaacs R. Your computer is already a distributed system. Why isn't your OS? *Proc. of the 12th conference on Hot topics in Operating Systems (HotOS)*, USENIX Association: Berkeley, CA, USA, 2009; 12–12.
3. Gottschalk K, Graham S, Kreger H, Snell J. Introduction to Web services architecture. *IBM Systems Journal* 2002; **41**(2):170–177.
4. Weske M. *Business Process Management: Concepts, Languages, and Architectures*. Springer, 2007.
5. Dustdar S, Schreiner W. A survey on Web services composition. *International Journal on Web and Grid Services (IJWGS)* August 2005; **1**(1):1–30.
6. Weerawarana S, Curbera F, Leymann F, Storey T, Ferguson D. *Web Services Platform Architecture*. Prentice Hall, 2005.
7. Baresi L, Di Nitto E, Ghezzi C. Towards open-world software. *Computer* October 2006; **39**:36–43.
8. Deelman E, Singh G, Su MH, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman GB, Good J, *et al.*. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 2005; **13**(3):219–237.
9. Schuler C, Weber R, Schuldt H, Schek HJ. Peer to peer process execution with OSIRIS. *Proc. of the Service-Oriented Computing Conference (ICSOC 2003)*, 2003; 483–498.
10. Gu X, Nahrstedt K, Yu B. Spidernet: an integrated peer-to-peer service composition framework. *Proc. of the 13th IEEE symposium on High Performance Distributed Computing (HPDC)*, 2004; 110–119.
11. Yu W. Scalable services orchestration with continuation-passing messaging. *Proc. of the 1st conference on Intensive Applications and Services (INTENSIVE)*, 2009; 59 –64.
12. Teng Q, Sweeney PF, Duesterwald E. Understanding the cost of thread migration for multi-threaded Java applications running on a multicore platform. *Proc. of the IEEE Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009; 123–132.
13. Rajagopalan M, Lewis B, Anderson T. Thread scheduling for multi-core platforms. *Proc. of the 11th USENIX workshop on Hot topics in Operating Systems (HotOS)*, 2007; 1–6.
14. Zhang EZ, Jiang Y, Shen X. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? *Proc. of the 15th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM: Bangalore, India, 2010; 203–212.
15. Pautasso C, Heinis T, Alonso G. Autonomic resource provisioning for software business processes. *Information and Software Technology* January 2007; **49**:65–80.
16. Pautasso C, Alonso G. JOpera: a toolkit for efficient visual composition of Web services. *International Journal of Electronic Commerce (IJEC)* 2004; **9**(2):107–141.
17. Peternier A, Bonetta D, Pautasso C, Binder W. Exploiting multicores to optimize business process execution. *Proc. of the IEEE conference on Service-Oriented Computing and Applications (SOCA)*, 2010; 1–8.
18. Bonetta D, Peternier A, Pautasso C, Binder W. A multicore-aware runtime architecture for scalable service composition. *Proc. of the IEEE Asia-Pacific Services Computing Conference (APSCC)*, IEEE Computer Society: Washington, DC, USA, 2010; 83–90.
19. Bonetta D, Peternier A, Pautasso C, Binder W. Towards scalable service composition on multicores. *Proc. of the conference on On The Move to meaningful internet systems (OTM)*, Springer-Verlag: Berlin, Heidelberg, 2010; 655–664.
20. Pautasso C. RESTful Web service composition with BPEL for REST. *Data Knowl. Eng.* September 2009; **68**:851–866.
21. Keller MS. The cutting edge: moving to SMP. *Linux Journal* March 2000; **2000**.

DOI: 10.1002/cpe

22. LaRowe RP Jr, Ellis CS, Kaplan LS. The robustness of NUMA memory management. *Proc. of the thirteenth ACM Symposium on Operating Systems Principles (SOSP)*, ACM: New York, NY, USA, 1991; 137–151.
23. Liberman J, Kochhar G. Optimal BIOS settings for high performance computing with Poweredge 11G servers. *Technical Report*, Dell 07 2009.
24. Gepner P, Kowalik MF, Fraser DL, Wackowski K. Early performance evaluation of new six-core Intel Xeon 5600 family processors for HPC. *Proc. of the 9th International Symposium on Parallel and Distributed Computing (ISPDC)*, IEEE Computer Society: Washington, DC, USA, 2010; 117–124.
25. Gillmann M, Mindermann R, Weikum G. Benchmarking and configuration of workflow management systems. *Proc. of 7th conference on Cooperative Information Systems (CoopIS)*, Eilat, Israel, 2000; 186–197.
26. Silver B. *BPMN Method and Style: a levels-based methodology for BPM process modeling and improvement using BPMN 2.0*. Cody-Cassidy Press, 2009.
27. Pautasso C, Wilde E. Push-enabling RESTful business processes. *Proc. of the 9th International Conference on Service Oriented Computing (ICSOC)*, Springer: Paphos, Cyprus, 2011.
28. Kleen A. A NUMA API for Linux. *Technical Report* Apr 2005.
29. Du J, Sehrawat N, Zwaenepoel W. Performance profiling of virtual machines. *Proc. of the 7th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE)*, ACM: New York, NY, USA, 2011; 3–14.
30. Peternier A, Bonetta D, Binder W, Pautasso C. Overseer: low-level hardware monitoring and management for Java. *Proc. of the 9th international conference on the Principles and Practice of Programming in Java (PPPJ)*, Denmark, 2011.
31. Ogasawara T. NUMA-aware memory manager with dominant-thread-based copying GC. *Proc. of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, ACM: New York, NY, USA, 2009; 377–390.
32. Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. *Proc. of the April 18-20, 1967, spring joint computer conference (AFIPS)*, ACM: New York, NY, USA, 1967; 483–485.
33. Constantinou T, Sazeides Y, Michaud P, Fetis D, Seznec A. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News* 2005; **33**(4):80–91.
34. Li G, Muthusamy V, Jacobsen HA. A distributed service-oriented architecture for business process execution. *ACM Trans. Web* 2010; **4**(1):1–33.
35. Brettlecker G, Milano D, Ranaldi P, Schek HJ, Schuldt H, Springmann M. ISIS and OSIRIS: a process-based digital library application on top of a distributed process support middleware. *Digital Libraries: Research and Development*, *Lecture Notes in Computer Science*, vol. 4877, Thanos C, Borri F, Candela L (eds.). Springer, 2007; 46–55.
36. Tabatabaei S, Kadir W, Ibrahim S. A comparative evaluation of state-of-the-art approaches for Web service composition. *In Proc. of the 3rd International Conference on Software Engineering Advances (ICSEA)*, 2008; 488–493.
37. Lu W, Gunarathne T, Gannon D. Developing a concurrent service orchestration engine in CCR. *Proc. of the 1st International Workshop on Multicore Software Engineering (IWMSE)*, ACM, 2008; 61–68.
38. Qiu X, Fox G, Yuan H, Bae S, Chrysanthakopoulos G, Nielsen H. Performance of multicore systems on parallel datamining services. *Computational Grids Laboratory: Indiana University* 2007; .
39. Lin K, Liao S. Service monitoring and management on multicore platforms. *Proc. of the IEEE International Conference on e-Business Engineering (ICEBE)*, 2006; 623–630.
40. Heinzl S, Seiler D, Juhnke E, Stadelmann T, Ewerth R, Grauer M, Freisleben B. A scalable service-oriented architecture for multimedia analysis, synthesis and consumption. *International Journal of Web and Grid Services* 2009; **5**(3):219–260.
41. Kazempour V, Fedorova A, Alagheband P. Performance implications of cache affinity on multicore processors. *Proc. of the 14th Euro-Par conference on Parallel Processing (Euro-Par)*, Springer-Verlag: Berlin, Heidelberg, 2008; 151–161.
42. Zhao Y, Shi J, Zheng K, Wang H, Lin H, Shao L. Allocation wall: a limiting factor of Java applications on emerging multi-core platforms. *Proc. of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, ACM: New York, NY, USA, 2009; 361–376.
43. Kalibera T, Mole M, Jones R, Vitek J. A black-box approach to understanding concurrency in DaCapo. *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012.
44. Ott M, Klug T, Weidendorfer J, Trinitis C. Autopin-automated optimization of thread-to-core pinning on multicore systems. *Proc. of 1st workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2008.
45. Tam D, Azimi R, Stumm M. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. *Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, Lisbon, Portugal, 2007; 47–58.

DOI: 10.1002/cpe