Exploiting Multicores to Optimize Business Process Execution

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

***Abstract*—While modern CPUs offer an increasing number of cores with shared caches, prevailing execution engines for business processes, workflows, or Web service compositions have not been optimized for properly exploiting the abundant processing resources of such CPUs. One factor limiting performance is the inefficient thread scheduling by the operating system, which can result in suboptimal use of shared caches. In this paper we study performance of the JOpera business process execution engine on a recent multicore machine. By analyzing the engine's architecture and by binding threads that are likely to access shared data to cores with a common cache, we achieve speedups up to 13% for a variety of workloads, without modifying the engine's architecture and implementation, apart from binding threads to CPUs. As the engine is implemented in Java, we provide a new Java library to manage thread bindings and hardware performance counters. We also leverage hardware performance counters to explain the observed speedup in our performance analysis.**

***Keywords*-Business process execution engines; multicores; performance optimization; thread–CPU bindings; hardware performance counters**

I. Introduction

Business process execution engines have become crucial components within modern service-oriented architectures [1], [2]. 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 [3]. Business processes are typically themselves delivered as services to clients [4], requiring process execution engines to be able to scale to handle a potentially high and unpredictable number of concurrent client requests [5].

In this paper, we target JOpera [6], a Java-based business process execution engine featuring a visual composition environment for the Eclipse platform. The JOpera engine can be deployed as a standalone server on dedicated machines. JOpera executes service compositions defined in a visual modeling language which are then compiled to Java code for efficient execution. The kernel of its engine features a scalable architecture which was originally designed to run on parallel execution environments such as on computer clusters [7].

Since it has become difficult to further increase the clock rate of processors, nowadays chip manufacturers are delivering more processing power by increasing the number of cores available in CPUs [8]. Recent multiprocessors combine several cores with a hierarchy of caches on a single chip. Typically, each core has its own small L1 and L2 caches, while several or all cores on a chip share a larger L3 cache. Examples include Intel Nehalem processors and AMD Opteron processors. Caches shared by cores on CPUs allow for faster communication between threads that are executing on different cores of the CPU, avoiding passing the data through inter–CPU connections or through main memory. Therefore, threads that frequently access the same shared data can benefit from executing on cores of the same CPU, since the accessed data may be kept in the shared cache.

While any multi-threaded application using thread pools, such as JOpera, may deliver better performance when running on a modern multicore machine, the abundant hardware resources of such machines are often underutilized unless the application has been tuned to exploit the specific hardware architecture. In addition, when two communicating threads are executing on cores that do not have a shared cache, extra overhead is incurred [9].

In order to make better use of modern multicores, in this paper we explore how to exploit thread–CPU bindings in the context of the JOpera engine. Modern operating systems offer the ability to constrain the set of cores on which a given thread may run, and thus bind any thread to any core. This lightweight technique is a promising approach, as it does not require to modify the existing architecture of the JOpera engine. Instead, we just extended the engine's kernel to control particular thread–CPU bindings. Our approach is easily applicable to other service-oriented middleware built using thread pools, as it does not require any redesign or major refactoring of the engine's code.

As JOpera is implemented in Java, we need an API to specify thread–CPU bindings. Since the standard Java class library does not offer any API to this end, we developed a new library, called OverHPC, which supports this feature. In addition, OverHPC enables to perform precise measurements with hardware performance counters (HPCs) directly from Java.

We explore different policies for thread–CPU bindings in JOpera and investigate performance for a variety of different workloads. On a multicore machine with modern micro-architecture, carefully chosen thread–CPU bindings yield a performance improvement up to 13%.

The scientific contributions of this paper are two-fold. First, we provide a new library to manage thread–CPU bindings in Java-based service-oriented middleware and show how it helps tune an existing business process

execution engine for modern multicores. Second, we present detailed evaluation results that demonstrate the possible performance gains with appropriately set thread–CPU bindings. We also show that an inappropriate use of this mechanism deteriorates performance. Our results are confirmed by collecting measurements from specific hardware performance counters, which help to explain the performance improvement.

This paper is structured as follows: Section II presents JOpera, which we use as a case study in this paper. Section III discusses our library for accessing HPCs and setting thread–CPU bindings. Sections IV and V present the results of our performance investigations. Section VI discusses related work, and Section VII concludes.

II. The JOpera Business Process Execution Engine

In this section we present the architecture of the latest version (2.5.0) of the JOpera[1] engine. 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 snippets or Java method calls) as well as remote service invocations (e.g., through RESTful HTTP calls). In fact, JOpera is fully compatible with BPEL process execution, but also enables the engine to run atypical business processes, composed for example of WS-* and RESTful web services.

At runtime, a JOpera server responding to client requests may run several process instances in parallel, each having its own independent state. The execution of processes is broken down into two main steps: process *navigation* and tasks *dispatching*. Navigation is about determining which task should be executed next, based on the current state of a process instance, while dispatching a task requires to carry out the actual task. This could involve a call to a local Java method or a remote invocation of a web service.

The recent version of the JOpera engine[2] uses two thread pools: N and D. N is dedicated to run processes (Navigator worker threads), while D runs tasks (Dispatcher worker threads). Each thread of the N and D pools exchanges task execution requests and task execution results by directly submitting jobs into the other pool's queue. As we are going to show, the code and memory access patterns of the two kinds of workers are sufficiently different to warrant grouping them in separate thread pools to exploit locality. This approach has the potential to make better usage of the available hardware resources.

III. The OverHPC Library

In this section we describe OverHPC, a Java library for accessing platform-specific functionalities such as HPCs, low level thread scheduling methods, and hardware architecture discovery. The OverHPC library is part of the Overseer suite freely available on our website[3].

Table I
A brief description of the main OverHPC API methods.

OverHPC API overview	
getSupportedEvents()	Returns a list with all the HPCs available on the current platform.
initEvents()	Initializes a list of HPC events to monitor.
bindEventsToCore()	Binds the initialized events to a specific core ID.
bindEventsToThread()	Binds the initialized events to a specific thread PID.
start()	Starts acquiring HPC measurements.
stop()	Suspends acquisition of HPC measurements.
getEventFromCore()	Returns the current value about a specific HPC being monitored on a core.
getEventFromThread()	Returns the current value about a specific HPC being monitored on a thread.
getThreadId()	Returns the PID of the thread this method is called from.
setThreadAffinity()	Applies a custom affinity mask to bind a specific thread to a core.
getThreadAffinity()	Returns the current affinity mask of a specific thread.
getAffinityInfo()	Returns information about the amount of CPUs, cores per CPU, and how they are mapped.

A. Hardware Performance Counters

HPCs are registers directly embedded into microprocessors to keep track of hardware-related activities and requiring low level accesses to be read. From a Java perspective, this leads to the implementation and interfacing of native methods to bring this information from inside a processor to the application. Each microprocessor architecture family has different and specific counters that make it difficult to deliver a portable solution. Our approach is based on three elements, each acting at different levels inside the operating system.

At the lowest level the library is based on the libpfm4 API[4]. The Linux kernel supports direct HPC information retrieval since version 2.4.30: unlike other existing solutions, libpfm4 allows to directly gather information without specific patches or additional modifications to the OS kernel. The abstraction level provided by libpfm4 and the availability of HPC-interfaces embedded into last generations of kernels are key advantages promoting the use of HPCs as real-time *in vivo* profilers.

The libpfm4 API is written in C and is interfaced by our library through a wrapper, using JNI and native methods to make the required functionalities accessible from Java. OS calls for retrieval and modification of thread–CPU bindings are also brought to Java through native methods.

OverHPC hides the complexity of libpfm4 and native code by offering a set of basic high-level methods to the user (see Table I). Native resources are automatically deallocated when no longer necessary, obeying to the

[1] http://www.jopera.org

[2] As opposed to having a single thread dedicated to run the process and a single thread dedicated to run tasks as described in earlier publications [6].

[3] http://sosoa.inf.unisi.ch

[4] http://perfmon2.sourceforge.net

Java programming conventions. In this way, HPCs and basic kernel scheduling operations can be easily accessed from the Java runtime in a few lines of code and by requiring minimal external dependencies, keeping all the native aspects hidden behind OverHPC.

B. OverHPC Functionalities

OverHPC allows to enumerate all hardware and software performance counter events supported by the system running the application. Selected counters can be initialized and assigned to a specific core (to profile in a hardware-oriented way) or to a specific thread (to measure them in a software-oriented manner).

Monitoring can be halted and resumed at will, in order to gather events only within relevant portions of code. Performance measurements about threads can be gathered both from inside the same thread or from another one, enabling to build external supervising and profiling agents. The amount of counters that can be monitored at the same time is determined by the hardware architecture and limited by the available system resources.

While HPCs are mainly passive instruments to observe the internal behavior of applications, OverHPC also features active instruments to control and experiment with thread–CPU bindings.

OverHPC can be used to acquire the OS kernel process ID (Unix PID) of the different threads being executed by the Java runtime and allows developers to modify the list of processors where these threads can be executed.

OverHPC analyzes the underlying system to determine how cores are mapped on the available CPUs (in the case of multiprocessor machines). This information is an extension to what Java natively offers through the `Runtime.availableProcessors()` method. OverHPC informs the user not only about the total number of cores, but also about *how* they are distributed over different CPUs, giving information such as the number and IDs of the cores available in each CPU.

This information permits the creation of affinity groups based on a list of core IDs physically sharing a hardware resource, such as a common cache (L2, L3) or a privileged Non-Uniform Memory Access (NUMA) connectivity. For example, binding the execution of threads accessing and reusing shared software data structures to cores within a shared hardware resource may leverage locality to reduce the number of operations such as thread migrations, cache evictions, or ineffective prefetches.

IV. Evaluation Settings

In this section we present and discuss the experiments we performed to benchmark the JOpera multicore engine, tuned through thread pools configured to use different thread–CPU bindings. We first introduce the hardware and software configurations used for testing. We then present results including both performance and HPC measurements which confirm the validity of our approach based on thread–CPU bindings.

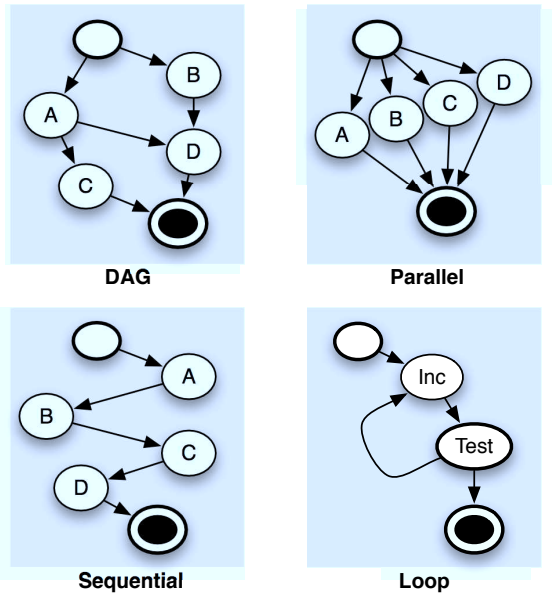


Figure 1. Benchmark workflows used for the performance evaluation.

A. Business Process Benchmarks

To evaluate the JOpera engine performance, we measure the execution times of four different generic kinds of business processes: *DAG*, *Parallel*, *Sequential*, and *Loop*. These processes have been chosen because they represent fundamental workflow patterns and are commonly found in most business process modeling languages. All tasks belonging to the first three processes perform the same invocation to a remote service. We adopted this solution to put more stress on the engine, thus minimizing noise factors coming from networking issues. The *Loop* example invokes simple Java snippets (to increment the loop counter and test for the loop exit condition) that are directly executed by the navigator threads. Also, we measure the execution time of a batch of thousands of process instances of the same kind. These are all executed concurrently and started within a very short time window.

The processes have the following control flow structures (see Figure 1):

- *DAG* — Each task is connected to others with a simple direct acyclic graph (DAG) structure. In terms of BPEL, this corresponds to a `<flow>` block with control links.
- *Parallel* — Each task is completely independent from the others and therefore can be run in parallel. This is equivalent to a BPEL `<flow>` block without any control flow dependencies between its child elements.
- *Sequential* — Each task depends on the previous one, thus the workflow contains a linear sequence of tasks. This is equivalent to a BPEL `<sequence>` block.
- *Loop* — The control flow of this process simply executes a loop, which iterates for a fixed number of times (100 iterations). It corresponds to a BPEL `<while>` block. The loop body is kept empty.

B. Measurement Environment

Our targeted hardware is a Dell PowerEdge M605. The M605 is a blade server equipped with 64GB RAM and two

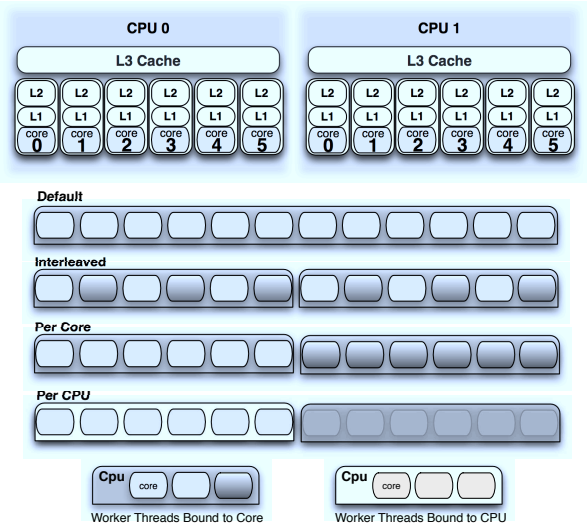


Figure 2. Overview of the processor and cache architecture of the M605 used in the evaluation (top). Schematic summary of the different thread–CPU binding policies (bottom).

2.6GHz AMD Six-Core Opteron processors (for a total of 12 cores). Each CPU comes with a large last-level cache (6MB L3) shared by all cores. Each core also features 512KB L2 and 64KB L1 caches (see Figure 2 for the detailed CPU architecture). This machine exploits a cache-coherent NUMA architecture: each CPU is optimally connected to a dedicated RAM slot and can efficiently access its data with minimal latency. When a CPU requires to access data stored on another memory slot, request times increase significantly.

Web services invoked by workflows are deployed on a separated machine used as a standard web server. In order not to influence the overall execution time with delays caused by the web server, each service responds the same message to any request within a fixed time interval. The size of each request and response message is negligible. In this way, we can ensure that the measured performance is not affected by this component.

The two computers are connected through a private 100MBit LAN, with an average message round-trip time of 0.5 milliseconds.

We executed all experiments under Ubuntu Linux 64 bit Server Edition version 10.04, running on kernel 2.6.31-20. As JVM, we used the Oracle Sun Java 64 bit Server version, build 1.6.0_20-b02 with Hotspot build 16.3-b01. We used Apache as web server on the second machine.

C. Thread–CPU Binding Policies

We extended the JOpera's kernel to use thread pools with customizable thread–CPU binding settings and tuned configuration parameters for the considered multi-core machine. In particular, we consider the impact of different thread–CPU binding policies and scheduling settings on performance. Hence, we test four different thread–CPU binding policies affecting the way threads are scheduled on the cores of the two available CPUs. These policies describe different methods for assigning two distinguished groups of threads to two different CPUs.

The four policies are schematically depicted in Figure 2, and feature the following characteristics:

- *Default* — The first policy, used as the baseline reference, is the "Default" configuration: the JVM and the operating system handle scheduling automatically without any explicit intervention.
- *Per CPU* — The second policy gives the OS scheduler freedom to dispatch a pool of threads within the bounds of a single CPU (for example: thread x can execute on any core of CPU0 but never on cores of CPU1).
- *Per core* — The third policy is a stricter version of the previous one, not only assigning threads of a same thread pool to a specific CPU but also constraining their execution to a single core (for example: thread 1 to CPU0–core0, thread 2 to CPU0–core1, thread 3 to CPU0–core3, etc.).
- *Interleaved* — This last policy is used to artificially simulate inefficient scheduling. It works like the reversed version of the previous policies, by spreading worker threads of a same pool over different CPUs and by also fixing them to run on interleaved cores, thus reducing per-processor locality (for example: thread 1 assigned to CPU0–core0, thread 2 to CPU1–core1, thread 3 to CPU0–core2, thread 4 to CPU1–core3, etc.).

V. Measurement Results

In order to observe speedups in the JOpera kernel optimized through different thread–CPU binding policies, we execute an exhaustive set of measurements with four different business processes run in batches with a growing number of concurrent process instances.

In more detail, we execute batches with 5'000, 10'000 and 30'000 instances. The charts in Figures 3a, 4a, and 5a show the relative speedup factor (computed from the elapsed wall time for each batch) of the "Per CPU", "Per core", and "Interleaved" policies over the "Default" one.

Data obtained from experiments confirm our assumptions as follows. First, thread–CPU bindings have a relevant performance impact (either positive or negative) when applied to the JOpera engine. Their effect on performance remains consistent across different runs, workflows, and lighter or heavier workloads. According to the kind of test and thread–CPU binding policy, we observe speedups of up to 13% ("Per core", DAG, 5'000 instances), but also slowdowns of -8% ("Interleaved", DAG, 30'000 instances). These variations are significantly higher with respect to the standard deviation of our measurements, which is under 2%.

Second, there is one winning thread–CPU binding policy that always improves performance over the default JVM scheduling. Among the four compared policies, the "Per core" policy is giving robust results, as it is always faster than the "Default" one, and almost always the one with the highest performance. The advantages coming

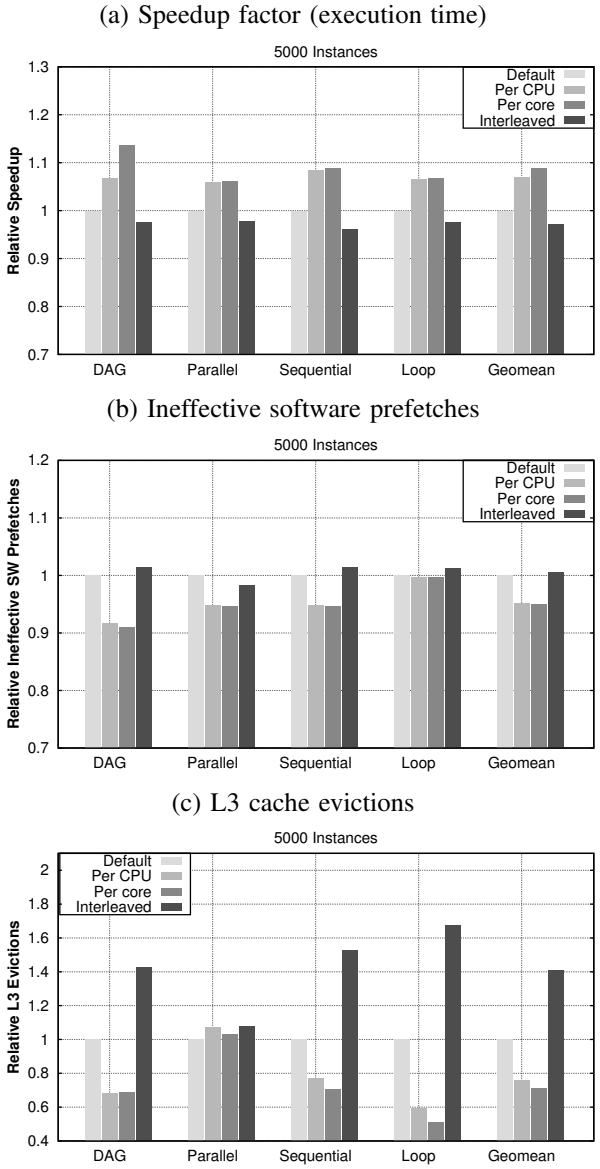


Figure 3. Performance comparison with 5'000 instances (test results shown relative to the "Default" policy).

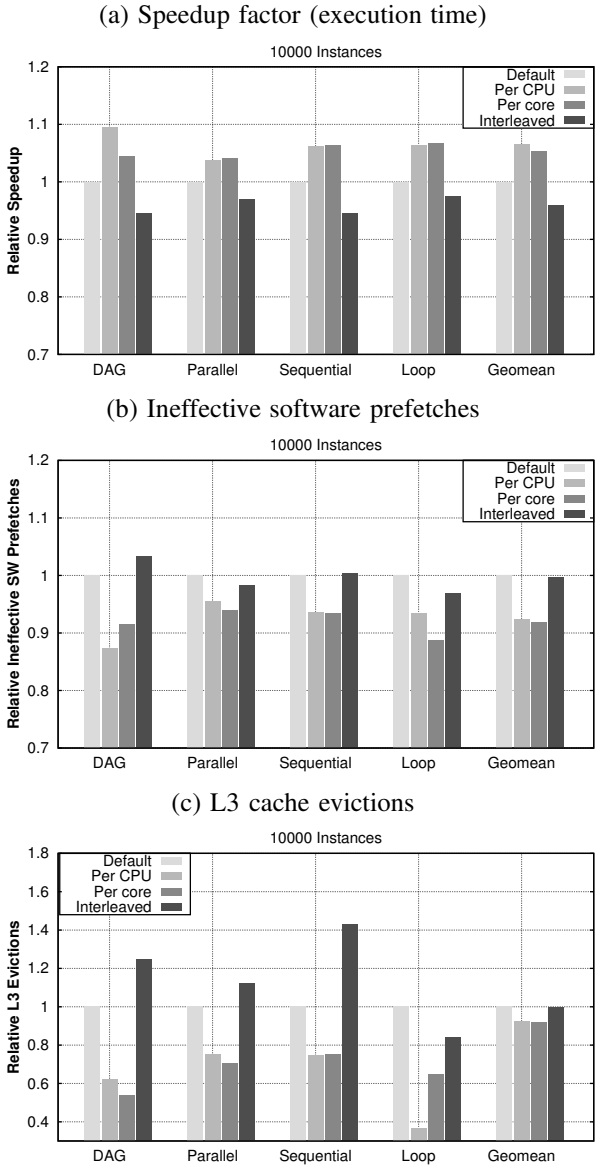


Figure 4. Performance comparison with 10'000 instances (test results shown relative to the "Default" policy).

from the adoption of this policy can be explained considering additional measurements done with some HPCs.

A. HPC Measurements

While the previous experiments measure the execution times, showing only which policy is performing faster than others, more data is required to precisely identify which factors contribute to the speedup. With the additional information provided by HPCs, we can accurately observe the effect of the various policies at the hardware level.

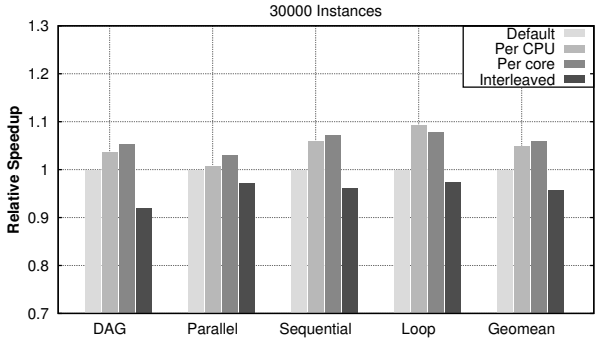
We used OverHPC to embed observation spots directly within the JOpera engine's kernel. In this way, we instrumented the engine's thread pools with counters tracking the system behavior precisely during the execution of worker threads code. This observation allows us to explain our results through two counters: ineffective software prefetches and L3 cache evictions (reported in the middle (b) and bottom (c) charts at Figures 3, 4, and 5).

Ineffective software prefetches occur when a prefetch requests a portion of memory that is already cached. This request is useless but forces the processor to check for the availability of the memory specified, spending several cycles in the operation.

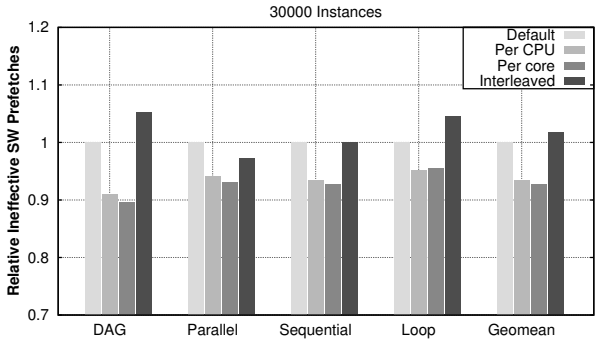
L3 evictions happen when the amount of data that needs to be stored in the cache is bigger than the available cache. When a new element is added, a cache replacement policy determines which previously stored information can be replaced with the newer data.

Our results show that the amount of ineffective prefetches and L3 evictions is reduced by the adoption of binding policies improving locality, such as "Per CPU" and "Per core". This observation is confirmed by the increase of the same counters when the "Interleaved" policy binds threads across CPUs thus voiding the benefits

(a) Speedup factor (execution time)



(b) Ineffective software prefetches



(c) L3 cache evictions

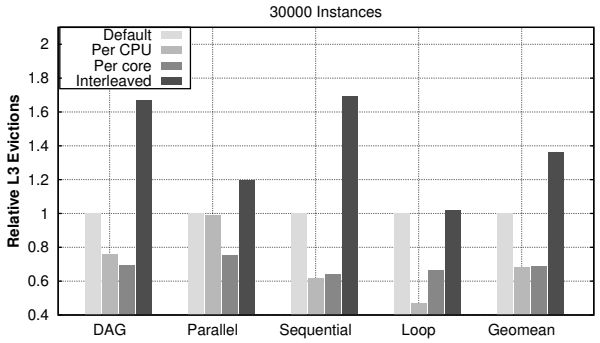


Figure 5. Performance comparison with 30'000 instances (test results shown relative to the "Default" policy).

brought by the L3 caches. As summarized in Table II, for all workload sizes, there is a strong correlation between both of the chosen counters and the system's performance measured in terms of the batch execution time.

Table II
CORRELATION COEFFICIENTS

Workload Size (Number of Instances)	Ineffective Software Prefetches	L3 Cache Evictions
5'000	0.9842	0.9456
10'000	0.9125	0.9883
30'000	0.9661	0.9946

VI. RELATED WORK

Currently available process execution engines do not consider the underlying hardware as a privileged source of performance-related opportunities. Most SOAs system performance are based on replication and distribution techniques, like the BPEL engine proposed by Li et al. in [10], the OSIRIS middleware by Brettlecker et al. [11], or previous versions of JOpera. This paper complements existing approaches by showing the potential of such low level optimizations for complex parallel applications running on multicore machines.

A. Service Composition Engines and SOA Performance

Service composition engines and middleware are becoming critical components in modern SOAs [12]. 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.

In [13], Lu et al. propose an architecture based on event-driven pattern and message passing interactions, using the CCR runtime available in the .Net framework. They evaluate the performance of their design, but do not provide a comparison to parallel architectures based on thread pools such as the one presented in this paper. The same approach is followed in [14], where a composite application for data mining is tested with several service technologies, from CCR, C+MPI to Java+MPI. The paper shows how thread-level parallelism issues have to be considered in order to obtain good performance.

In [15], Lin et al. propose a QoS management architecture for coordinating services deployed on 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 processes orchestration is presented in [16]. The paper introduces a complex, service-oriented architecture for streaming service interactions that extends BPEL to support for data-intensive applications. They also provide tools for scaling the system by means of dynamic allocation and replication of cloud resources.

B. Multicore Performance

Modern chip multi-processors provide non-uniform cache sharing; cores on separate chips usually do not share caches in the same way as cores on the same chip. In [17] 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 affinities helps improve performance particularly for such transformed programs.

Thread migration, resulting from load balancing in the operating system, may impair performance due to increased L2 cache misses [18]. In [19] 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 explain thread migration overhead. For Java workloads, the authors argue that migration frequency is relatively low, such that Java applications do not suffer from high thread migration penalties on current multicores.

In [20] 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 the available memory write bandwidth on several multicore platforms.

Similar to our approach, Autopin [21] 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 checks among a given set of fixed thread-to-core bindings (defined by the user) for the best available configuration in order to exploit thread locality. Each binding layout is called a *pinning*, and the framework adaptively selects the best one by accessing at runtime a predefined set of hardware performance counters. In this way, automatically, threads monitored by the Autopin framework are migrated to the best available CPU/core. The OverHPC library presented in this paper allows to implement 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 any kind of self-tuning binding strategy.

Another comparable approach is the one proposed by Tam et al. in [22]: an OS-level thread scheduler for SMP-CMP-SMT 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 to force affinity migrations of related threads (i.e., threads influencing the same performance counter, for example by sharing the same memory).

C. Hardware Performance Counters

Hardware performance counters have recently emerged as one of the privileged sources of information to improve software performance. HPCs are widely adopted and integrated in the software development cycle [23] and an increasing number of tools for accessing and manipulating performance counters has been proposed. PAPI, the Performance Application Programming Interfaces library [23] is probably the most widely adopted tool for measuring hardware performance counters. The library offers a high-level, platform-independent access to CPU counters, providing developers with a standard way to access specific platform related counters as well as generic platform independent counters. Recently, a new version of the library, called Component-PAPI (PAPI-C [24]) has been announced in late 2009. The new library extends the standard PAPI framework with the possibility to obtain informations not only from CPU related events, but also from other sources. Examples of such sources are GPUs, memory interface chips, network interface cards, as well as BIOS, ACPI and LM related sensors.

VII. Conclusion and Future Work

Prevailing middleware for business process execution has not been optimized for running on recent processors with modern multicore micro-architectures. While multi-threaded middleware can generally benefit from an increasing number of available cores, thread migrations due to operating system scheduling and inefficient access to shared data often limit performance.

In this paper, we explore how JOpera, an existing, Java-based business process execution engine, can benefit from customized thread–CPU bindings. By restricting the cores on which certain threads may execute, we are able to force threads that are likely to access shared data to execute on cores that share a common cache, resulting in improved thread communication. This approach works particularly well for middleware using multiple thread pools, where the threads of each pool are likely to access common data, whereas threads of different pools require less frequent communication.

Our extensive performance evaluation on a recent multicore machine with different business processes confirms that properly defined thread–CPU bindings improve performance by up to 13%, taking a manually tuned engine configuration as baseline. However, the tuning of thread–CPU bindings must be done carefully: inappropriate bindings can result in performance deterioration. With the aid of hardware performance counters, we explore the reasons for the measured performance impact of thread–CPU bindings.

Since setting of thread–CPU bindings and the use of hardware performance counters are not supported in the standard Java class library, we provide the new library OverHPC that offers these missing features. The OverHPC library is publicly available.

While in this paper we demonstrate that existing service-oriented middleware using thread pools can be optimized for modern multicores by properly managing thread–CPU bindings, in our ongoing research work we are exploring fundamentally new middleware designs to further improve performance on recent multicore micro-architectures. Moreover, we are considering auto-tuning mechanisms inside the engine to refine various performance-relevant configuration parameters, such as thread pool sizes, at runtime, leveraging monitoring information from hardware performance counters. Finally, we

will validate our middleware design on a wide spectrum of modern multicore architectures, including Intel Nehalem and AMD Opteron CPUs.

ACKNOWLEDGMENT

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

REFERENCES

[1] K. Gottschalk, S. Graham, H. Kreger, and J. Snell, "Introduction to web services architecture," *IBM Systems Journal*, vol. 41, no. 2, pp. 170–177, 2002.

[2] M. Weske, *Business Process Management: Concepts, Languages, and Architectures*. Springer, November 2007.

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

[4] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson, *Web Services Platform Architecture*. Prentice Hall, March 2005.

[5] L. Baresi, E. Di Nitto, and C. Ghezzi, "Towards Open-World Software," *Computer*, vol. 39, pp. 36–43, October 2006.

[6] C. Pautasso, T. Heinis, and G. Alonso, "Autonomic resource provisioning for software business processes," *Information and Software Technology*, vol. 49, pp. 65–80, January 2007. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2006.08.010

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

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

[9] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec, "Performance implications of single thread migration on a chip multi-core," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 80–91, 2005.

[10] G. Li, V. Muthusamy, and H.-A. Jacobsen, "A distributed service-oriented architecture for business process execution," *ACM Trans. Web*, vol. 4, no. 1, pp. 1–33, 2010.

[11] G. Brettlecker, D. Milano, P. Ranaldi, H.-J. Schek, H. Schuldt, and M. Springmann, "Isis and osiris: A process-based digital library application on top of a distributed process support middleware," in *Digital Libraries: Research and Development*, ser. Lecture Notes in Computer Science, C. Thanos, F. Borri, and L. Candela, Eds. Springer, 2007, vol. 4877, pp. 46–55.

[12] S. Tabatabaei, W. Kadir, and S. Ibrahim, "A comparative evaluation of state-of-the-art approaches for web service composition," in *The Third International Conference on Software Engineering Advances, 2008. ICSEA'08*, 2008, pp. 488–493.

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

[14] X. Qiu, G. Fox, H. Yuan, S. Bae, G. Chrysanthakopoulos, and H. Nielsen, "Performance of multicore systems on parallel datamining services," *Computational Grids Laboratory: Indiana University*, 2007.

[15] K. Lin and S. Liao, "Service monitoring and management on multicore platforms," in *IEEE International Conference on e-Business Engineering, 2006. ICEBE'06*, 2006, pp. 623–630.

[16] S. Heinzl, D. Seiler, E. Juhnke, T. Stadelmann, R. Ewerth, M. Grauer, and B. Freisleben, "A scalable service-oriented architecture for multimedia analysis, synthesis and consumption," *International Journal of Web and Grid Services*, vol. 5, no. 3, pp. 219–260, 2009.

[17] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2010, pp. 203–212.

[18] V. Kazempour, A. Fedorova, and P. Alagheband, "Performance implications of cache affinity on multicore processors," in *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 151–161.

[19] Q. Teng, P. F. Sweeney, and E. Duesterwald, "Understanding the cost of thread migration for multi-threaded Java applications running on a multicore platform," in *ISPASS '09: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2009, pp. 123–132.

[20] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao, "Allocation wall: a limiting factor of Java applications on emerging multi-core platforms," in *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. New York, NY, USA: ACM, 2009, pp. 361–376.

[21] M. Ott, T. Klug, J. Weidendorfer, and C. Trinitis, "Autopin-Automated Optimization of Thread-to-Core Pinning on Multicore Systems," in *Proceedings of 1st Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2008.

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

[23] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, August 2000.

[24] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Proceedings of the 3rd Parallel Tools Workshop*. Dresden, Germany: Springer (to appear), 2010.