Scaling software on multi-core through co-scheduling of related tasks

Srivatsa Vaddagiri Bharata B Rao Vaidyanathan Srinivasan Anithra P Janakiraman
Balbir Singh Vijay K Sukthankar

IBM India Software Labs, Bangalore
{vatsa, bharata, svaidy, janithra, balbir, vksuktha}@in.ibm.com

Abstract

Ever increasing demand for more processing power, coupled with problems in designing higher frequency chips are forcing CPU vendors to take the multi-core route. IBM® introduced the first multi-core processor with its POWER4® in 2001, that had two cores in a chip and also 4 chips in a package. Other CPU vendors have followed the trend with dual and quad-core processors becoming increasingly common. It is estimated that by year 2021, there will be chips with 1024 cores on them [6]. Such platforms pose huge challenge on how software effectively utilizes so many cores. One problem of interest is how tasks are scheduled on such platforms. The existing Linux scheduler attempts to distribute tasks equally among all CPU chips. It does not optimize this task placement, taking into consideration that all tasks need not be equal with respect to their use of shared CPU resources (like L2 cache). In this paper, we look at how misplacement of tasks across CPU chips can significantly affect performance and how existing Linux interface to solve that problem is inflexible. We present a new interface which can be used by applications to hint which threads share data closely and thus should be co-scheduled on *neighbouring*[1] CPUs to the extent possible by OS scheduler. We present several results showing the inflexibility of existing interface and how the suggested interface solves those problems.

1 Trends in modern system architecture

Modern multi-core processors have innovative and complex cache hierarchy design in order to hide memory access latency and optimize bandwidth on various intra-chip and inter-chip interconnect buses. With faster CPUs, application performance is now becoming bound on the availability of its working data set in *local* CPU cache.

Table 1 aptly illustrates this point using *c2cbench* [1], a benchmark that measures the cost of data transfer between two caches. The benchmark was used to measure throughput for transferring 256KB of data between a producer and consumer thread.[2,3] By controlling the CPUs on which two threads run, the benchmarks measures cost of cache-to-cache transfer. Best throughput is seen when both threads are co-scheduled on sibling cores (which share the same L2/L3 cache). The throughput drops by a factor of 4-6 when the threads are forced to run on cores that don't share the cache hierarchy. An interesting data point from Table 1 that represent typical system cache topology is that co-scheduling producer/consumer tasks on sibling hardware threads gives best performance since they share most of the cache hierarchy. The benefit of cache sharing is outweighing the cost of contention for shared execution resources in the core.

Although memory and inter-chip interconnect bandwidth has been increasing in each generation of processors, the trend seems to indicate that the ratio of access latency between remote and local cache will continue to be significant. Thus we can conclude that task placement can significantly affect performance, especially for scenarios where two or more tasks work closely on shared data.[4] Co-scheduling such related tasks on *neighbouring* CPUs can improve performance by making best use of shared cache hierarchy.

[1] *Neighbouring* CPUs are those that share some or all of a cache hierarchy.

[2] c2cbench -P0 -C1 -prw -crd -d4096 -b256 -s8 -k1 -K0 -I1000

[3] The terms *thread* and *task* are used interchangeably throughout the paper.

[4] The term *thread cluster* is used to refer to a group of tasks that work closely on some shared data.

Relative throughput for data sharing (GB/sec)	*Sibling hardware threads*	*On-chip cores*	*Off-chip cores*
IBM POWER5®	3.9a	4.3a	1a
IBM POWER6®	6.4b	1.4b	1b
Intel® Xeon Quad Core	N/A	6.5c	1c
Intel Core i7	4.1d	2.1d	1d

Table 1: Producer-consumer throughput for 256KB transfer

Scenario	*No-co-scheduling case* (million records/sec)	*Co-scheduling case* (million records/sec)	*Impact of co-scheduling*
Two instances	8.76	9.71	**+10.84%**
Single instance	15.73	9.74	**-38%**

Table 2: Co-scheduling *ebizzy* instances

Scenario	*No-co-scheduling case* (seconds)	*Co-scheduling case* (seconds)	*Impact of co-scheduling*
Two instances	209.44	207.79	**+0.78%**
Single instance	107.42	203.92	**-89.8%**

Table 3: Co-scheduling *kernbench* instances

Metric	*No co-scheduling* (million records/sec)	*Co-scheduling* (million records/sec)	*Impact of co-scheduling*
VM1 Throughput	5.85	5.95	**+1.7%**
VM2 Throughput	3.64	5.65	**+55.22%**
VM3 Throughput	7.67	7.27	**-5.22%**

Table 4: Co-scheduling KVM VMs

Scenario	*No co-scheduling* (seconds)	*Co-scheduling* (seconds)	*Impact of co-scheduling*
Two instances	1x	0.8846x	**+11.54%**
Single instance	1x	1.2339x	**-23.39%**

Table 5: Co-scheduling Trade6 application

2 Co-scheduling opportunities

In this section, we look at few opportunities that exist in real world where we can co-schedule related threads on *neighbouring* CPUs for improving performance.

2.1 Multiple instances of same applications

In many cases, multiple instances of the same application are launched. For example, multiple users launching same compiler program to compile their program, multiple application servers launched on the same machine as a vertical cluster [2] etc. Probability of data sharing between threads of an instance is higher than between threads across instances. Co-scheduling threads of an instance on *neighbouring* CPUs could potentially yield better performance, *provided* the opportunity exists to utilize remaining CPUs for other work.

Table 2 shows the results of co-scheduling for *ebizzy* [3] benchmark, a workload resembling web application server. The benchmark creates several threads that search for a random key from the same memory region. The memory region thus is shared between all threads of the benchmark.

In first scenario, two instances of *ebizzy* are launched simultaneously on a machine having two dual-core Intel Xeon® CPUs (with 4MB shared L2 cache). In no-co-scheduling case, they were not bound to any CPU and in co-scheduling case, each instance was bound to a separate dual-core CPU. Co-scheduling gives good results in this scenario. In the second scenario, only one instance is launched. Co-scheduling that single instance, which means binding that instance to a single dual-core CPU, does not give good results in this scenario. This is because the single instance, being *hard*-bound to single dual-core CPU, is not effectively making use of all the available (idle) CPUs in the system.

Table 3 shows the results of co-scheduling for *kernbench*, a Linux kernel compilation benchmark. On the same machine described above, two instances of *kernbench* are launched simultaneously in first scenario. Each instance spawns 11 threads for compiling different source files in parallel. Each of those 11 threads will compile its own source file and hence there is very little data sharing between threads of an instance. Co-scheduling in this scenario will not give any benefit and in the second scenario of single instance is actually *hurting* performance.

2.2 Virtualization

Power, cooling and real-estate constraints in data centers are forcing customers to consolidate their applications on fewer and powerful machines. Advanced virtualization capabilities of modern processors are being fully utilized to carve several virtual machines (VM) out of a single machine. Each VM gets the illusion as if it has its own set of hardware resources (CPUs, memory etc). The mapping of virtual resources of a VM to underlying physical resources is managed by a hypervisor software. For example, in case of CPUs, the hypervisor will schedule the different virtual CPUs (VCPU) of a VM on different physical CPUs.

Typically each VM hosts a single application, say a database server or webserver. In such a case, data sharing is more likely to occur between threads belonging to the same VM rather than between threads of different VMs. Thus it makes sense to consider co-scheduling different VCPUs of a VM on *neighbouring* CPUs, *provided* the opportunity exists to utilize remaining CPUs for other work.

In an experiment involving KVM based virtualization, 3 VMs, VM1, VM2 and VM3, were launched on a machine having 2 quad-core Intel Xeon CPUs. *ebizzy* benchmark was started simultaneously on all three VMs. In the first case, VMs were not bound to any CPU. In the next case, VM1 and VM2 were bound to two different quad-core CPUs and VM3 was not bound to any CPU. The results shown in Table 4 shows that co-scheduling helps improve the performance of *ebizzy* benchmark running inside VM1 and VM2.

2.3 Application Server

Java application servers like WebSphere® Application Server (WAS) are used to host business applications written in J2EE. The same application server can host multiple applications or multiple application instances on the same node. Probability of data sharing is higher between threads of the same application (instance) and hence an application (instance) could form the basis for co-scheduling threads. In case of applications like YouTube or online gaming, it is possible to group threads at a even much finer granularity. For example, all threads serving the same video/photo-album or all threads serving players of the same game instance could be grouped together to form a cluster.

Table 5 shows the result of co-scheduling for Trade6 application on a server with two dual-core Intel Xeon CPUs. Time taken to complete the benchmark is shown on a relative scale, with the *No co-scheduling* case forming the baseline to compare against. In first scenario, two Trade6 instances are launched. Co-scheduling each instance on a separate dual-core CPU results in better performance compared to not co-scheduling any instance. In the second scenario, a single instance is launched. Co-scheduling that single instance (which mean *hard*-binding it to a single dual-core CPU) is actually *hurting* performance in this case, as it does not utilize fully all the available CPU resources.

The key observations from these experiments are:

1. Co-scheduling helps improve performance for certain workloads, where high degree of data sharing exists between threads.
2. Co-scheduling should not be at the cost of idling CPUs. In other words, its better to *break* co-scheduling in favor of utilizing as many required (idle) CPUs.

3 Detecting co-scheduling opportunities

In Section 2, we saw that opportunities exists in real-world for improving performance on multi-core systems by co-scheduling related threads. How do we detect such opportunities? In most cases, it is done with manual intervention—after carefully studying the workload and the platform behavior. Co-scheduling is achieved using existing interfaces like *sched_setaffinity* and *cpuset*. Beyond providing the raw support to co-schedule tasks, Linux doesn't have any capability to *automatically* detect co-scheduling opportunities and co-schedule selective tasks based on that.

3.1 Automatic detection

[8] describes one mechanism to automatically determine co-scheduling opportunities on IBM Power5-based multi-core platform, based on observing certain HPCs (Hardware Performance Counter) related to cache-miss events. The algorithm described is however quite complex and it remains to be seen how easily it can be adapted to a general purpose operating system like Linux.

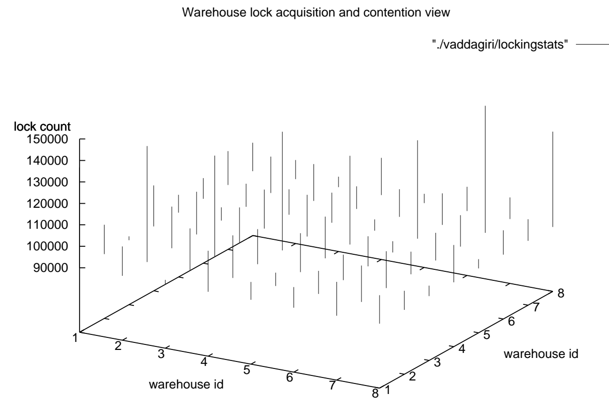

Figure 1: Warehouse lock acquisition and contention view

We present below a more simpler approach which could form the basis of *automatic* co-scheduling. The approach is based on the fact that data sharing between threads generally involves them acquiring the same locks guarding shared data access. Analyzing lock acquisitions can give us a clue on threads that are closely working on shared data. Once such threads groups are detected, we could automatically co-schedule them on *neighbouring* CPUs using the interface described in Section 4.2.

3.2 Workload

We ran SPECjbb2000 [4] and modified the default configuration of SPECjbb so that multiple threads (terminals) can simultaneously access the warehouse. In our experiments we used 8 warehouses with 4 threads per warehouse. We instrumented the benchmark to collect information about threads and which warehouse they belonged to.

3.3 Results

Figure 1 shows a plot depicting the lock acquisition and contention count for each of the threads by their warehouse ID. The same data is show in numerical tabular form below, Table 3.1.

As can be seen from Table 3.1, the highest locking was seen between the threads belonging to the same warehouse. Figure 1 displays the same graphically. The data was obtained by instrumenting the mutual exclusion paths on a per thread and a per mutex basis. This

Warehouse Id	1	2	3	4	5	6	7	8
1	**136774**	103674	91826	109088	98283	99615	105770	103254
2	103674	**143964**	109358	109848	96172	100722	106946	98890
3	91826	109358	**136828**	106294	108150	101856	107154	94878
4	109088	109848	106294	**145206**	109430	104000	100534	107342
5	98283	96172	108150	109430	**131296**	95266	102882	94316
6	99615	100722	101856	104000	95266	**135796**	104676	101312
7	105770	106946	107154	100534	102882	104676	**149144**	100070
8	103254	98890	94878	107342	94316	101312	100070	**134370**

Table 6: Warehouse to warehouse lock acquisition and contention count

data was then summed to extract thread to thread locking statistics by summing lock acquisition counts for each mutex and thread pairs. The warehouse data was obtained by summing the lock statistics for all threads belong to the warehouse.

3.4 Observations

The results obtained from the experiments above indicate that

1. Although all threads in a process share the same address space, the working data set could be be different for each thread.
2. A group of threads could share the same working set to form a thread cluster.
3. Co-scheduling such thread clusters on *neighbouring* CPUs should help improve performance (as proven in this case by Figure 6 and Figure 7).

4 Co-scheduling interface

Once co-scheduling opportunities are determined, either manually or automatically, co-scheduling related tasks together on *neighbouring* CPUs is accomplished using interfaces such as *sched_setaffinity* or *cpuset*.

4.1 *Hard* affinity interface

Both *sched_setaffinity* and *cpuset* provide the ability to control where tasks execute. Using these interfaces it is possible to co-schedule threads of a cluster on *neighbouring* CPUs. The biggest drawback with these interfaces is the *hard*-affinity it creates between tasks and CPUs, because of which it can actually *hurt* performance sometimes (as highlighted by *Single instance* scenario of Table 2). What would be better is a *soft*-affinity interface, which would allow threads to be *soft*-bound to CPUs.

4.2 *soft* affinity interface

The *soft*-affinity interface allows applications or administrators to register thread clusters. The CPU scheduler would then *automatically* co-schedule threads of a cluster on *neighbouring* CPUs, *provided* other CPUs can be used for executing other work. In case no other work exists, then scheduler would break co-scheduling of a thread cluster in favor of utilizing all required CPUs for the cluster.

The interface to register thread clusters is built on top of the cgroup process-grouping feature of Linux kernel [7]. A new cgroup subsystem, called *co-scheduler*, was written to mediate between user space and scheduler (Figure 2). The *co-scheduler* subsystem provides a filesystem based API (with help of *cgroup* subsystem) for thread clusters to be registered. The API allows creation/deletion of thread clusters or movement of threads from one cluster to another (Figure 3). The *co-scheduler* subsystem closely tracks the load of each cluster across various CPUs (Figure 4), based on which it will *automatically* co-schedule threads of few clusters on *neighbouring* CPUs. Co-scheduling of threads is accomplished by manipulating their CPU affinity. A high-level flowchart for the working of *co-scheduler* subsystem is shown in Figure 5.

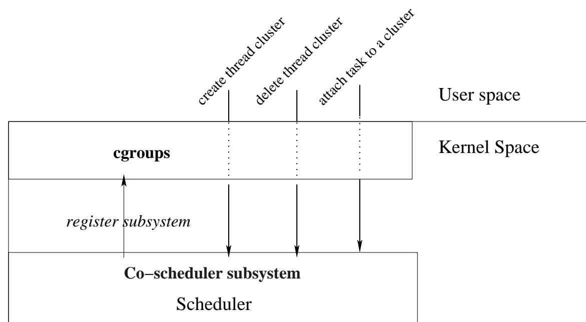

Figure 2: Co-scheduler subsystem

```
# mkdir /cgroup
# mount –t cgroup –o coscheduler none /cgroup
# cd /cgroup
# mkdir cluster1
# mkdir cluster2
# /bin/echo pid1 > cluster1/tasks
# /bin/echo pid2 > cluster2/tasks
```

Figure 3: Registering thread clusters

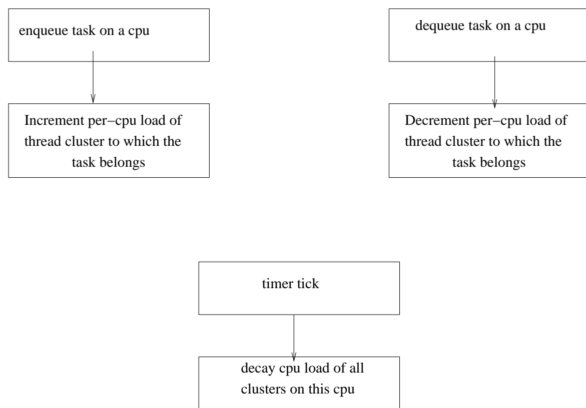

Figure 4: Tracking cluster load

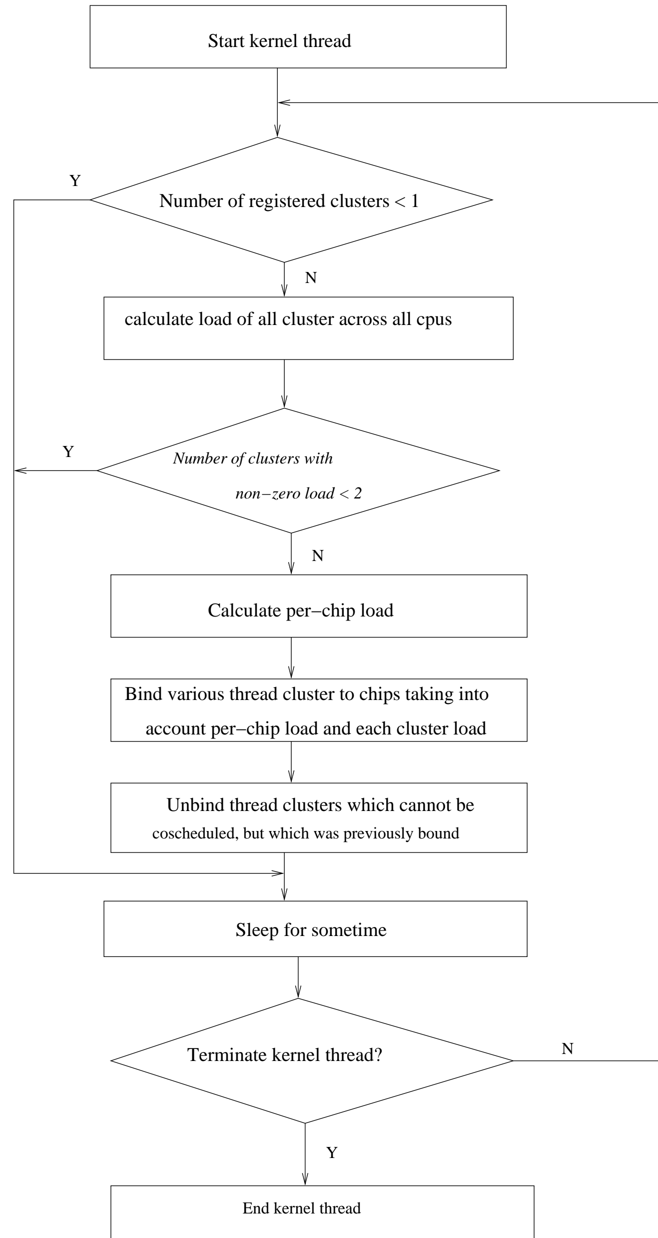

Figure 5: Co-scheduler operation

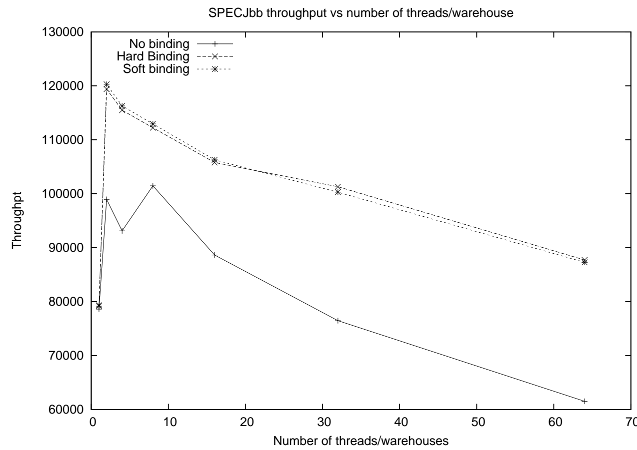

Figure 6: SPECJbb2000—Absolute throughput

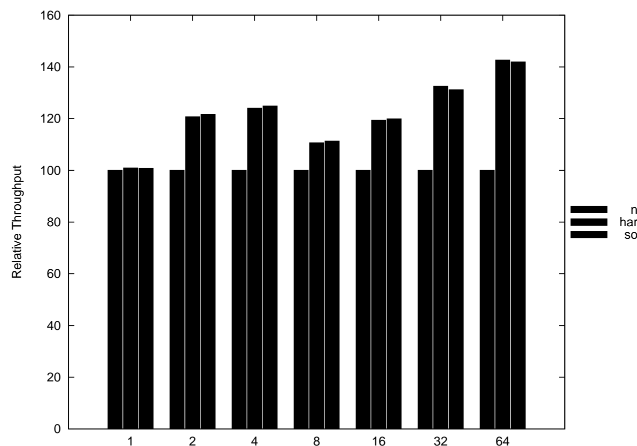

Figure 7: SPECJbb2000—relative throughput

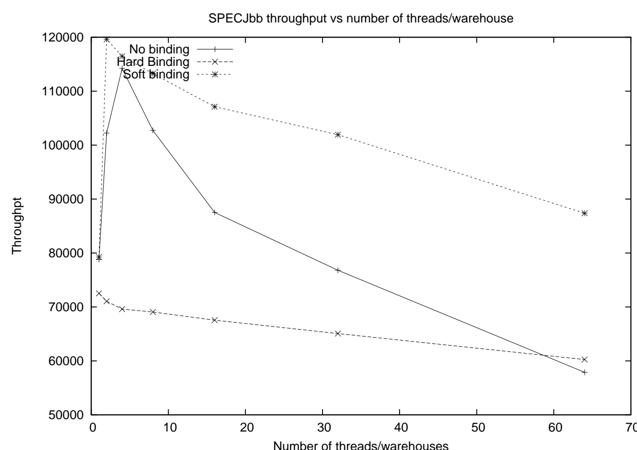

Figure 8: SPECJbb2000—Two warehouses with varying number of threads

4.2.1 Results

Some results comparing *hard*- and *soft*-affinity are provided below:

1. SPECJbb
   SPECJbb [4] is a Java benchmark used to evaluate Java performance. The benchmark creates several warehouses and several threads (or terminals) per warehouse. Threads associated with the same warehouse will very likely access the same data that is associated with the warehouse. The benchmark was modified to bind threads using both the *hard*- and *soft*-affinity interfaces. Figure 6 shows the results of using the interfaces on a system having two dual-core Intel Xeon CPUs. Two warehouses were created and the number of threads per warehouse was varied from 1 to 64. In case of *hard*-affinity, threads belonging to first warehouse were bound (using *sched_setaffinity*) to first dual-core CPU while threads belonging to second warehouse were bound to the second dual-core CPU. In case of *soft*-affinity, threads of the both warehouses were registered as separate clusters. The results show that binding, through either *soft*-affinity or *hard*-affinity, provides better results. Also *soft*-affinity is giving equally good results as *hard*-affinity. Figure 7 shows the same results on a relative scale (with reference to the results obtained without binding any threads).

   Figure 8 shows some results which exposes the weakness with *hard*-affinity. In this case, the number of warehouse was kept constant at 2, while the number of threads/warehouse was varied from 1 to 64. For the *hard*-affinity case, threads of both warehouses were bound to first dual-core CPU, which causes a gross under-utilization of resources. For the *soft*-affinity case, threads of each warehouse were registered as a separate cluster. The results show that *hard*-affinity gives poorer results compared to not binding any threads. Also *soft*-affinity is giving best performance compared to no-binding or *hard*-affinity by deciding to schedule threads of two warehouses on separate dual-core CPUs.

2. Java application server
   IBM Trade Performance Benchmark Sample [5] for WebSphere Application Server or Trade6 is the fourth generation of WebSphere end-to-end benchmark and performance sample application, which

simulates a real-world workload. To study the impact of co-scheduling threads of the same JVM instance, we used up to 5 WebSphere Application Server profiles each running its own installation of Trade6 on a machine having two dual-core Intel Xeon CPUs. Each of the Trade6 instances was configured to use its own DB2 instance as the backend. The Trade6 application was stressed using the WebSphere Studio Workload Simulator engine (iwlengine) which generates a set of requests continuously till a particular runtime is reached.

For the purpose of this experiment the iwlengine script was modified to generate a fixed number of requests.The number of clients was fixed at 50. The results were first collected for 2 instances of Trade6 that were stressed simultaneously. Performance was measured using the iwlengine in terms of throughput and time taken. The threads of each WebSphere instance are likely to access the same data that is associated with the that WebSphere/Trade6 instance. This was exploited using both the *hard-* and *soft-* affinity interfaces. In case of *hard*-affinity, threads belonging to the first instance were bound (using *sched_setaffinity*) to the first dual-core CPU while threads belonging to second instance were bound to the second dual-core CPU. In case of *soft*-affinity, threads of both instances were registered as separate clusters.This experiment was repeated for 3, 4 and 5 application server instances.

Figure 9 shows the results of binding on a relative scale (with reference to the results obtained without binding any threads). The results shows that binding improves the throughput significantly. In some cases *soft-* affinity gives better results which could be attributed to the fact that *soft-* affinity gives priority to CPU utilization over co-scheduling.

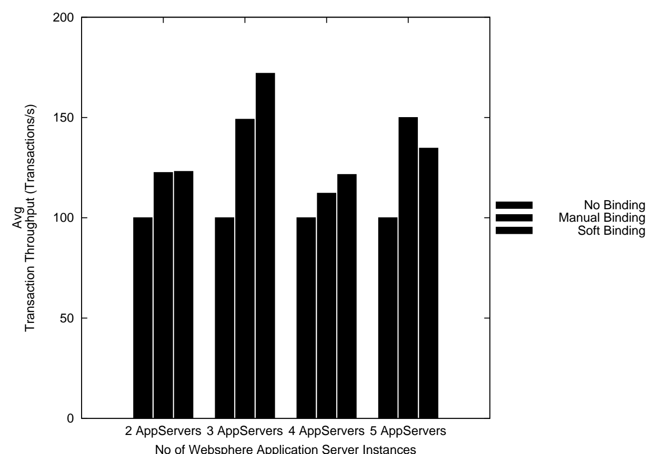

Figure 9: Trade6—Relative throughput

5 Acknowledgments

The authors thank IBM management (Premalatha M Nair, Naren A Devaiah, Naveen Kamat, Thomas Domin, Kalpana Margabandhu) for being supportive of this work. A special thanks goes to Manish Gupta (Associate Director, IBM India Research Labs) for prodding the authors to think about multi-core issues and who was instrumental in driving the idea of using lock-contention to form thread-clusters.

6 Legal Statement

©International Business Machines Corporation 2009.

Permission to redistribute in accordance with Linux Symposium submission guidelines is granted; all other rights reserved.

This work represents the view of the authors and does not necessarily represent the view of IBM. IBM, IBM logo, ibm.com, and WebSphere, are trademarks of International Business Machines Corporation in the United States, other countries, or both. Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Other company, product, and service names may be trademarks or service marks of others.

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates. INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you. This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

References

[1] Cache to cache producer-consumer benchmark. http://sourceforge.net/projects/c2cbench.

[2] Clustering with vertical cluster members. http://publib.boulder.ibm.com/infocenter/wchelp/v6r0m0/index.jsp?topic=%/com.ibm.commerce.admin.doc/tasks/tigvertcluster.htm.

[3] Ebizzy benchmark. http://sourceforge.net/projects/ebizzy/.

[4] Specjbb benchmark. http://www.spec.org/jbb2005/docs/WhitePaper.html.

[5] Trade performance benchmark for websphere application server. http://www.ibm.com/software/webservers/appserv/was/performance.html.

[6] F. Allen. Fran Allen talk on parallel computing. http://www.windley.com/archives/2008/02/fran_allen_compilers_and_parall%el_computing_systems.shtml.

[7] P. B. Menage. Resource control and isolation: Adding generic process containers to the linux kernel. http://ols.108.redhat.com/2007/Reprints/menage-Reprint.pdf.

[8] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *in EuroSys*, 2007.

Proceedings of the
Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.