View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCSE-2008-10

2008-01-01

Real-Time Performance and Middleware on Multicore Linux Platforms

Yuanfang Zhang, Christopher Gill, and Chenyang Lu

An increasing number of distributed real-time applications are running on multicore platforms. However, existing real-time middleware (e.g., Real-Time CORBA) lacks support for scheduling soft real-time tasks on multicore platforms while guaranteeing their time constraints will be satisfied. This paper makes three contributions to the state of the art in real-time system software for multicore platforms. First, it offers what is to our knowledge the first experimental analysis of real-time performance for vanilla Linux primitives on multicore platforms. Second, it presents MC-ORB, the first real-time object request broker (ORB), designed to exploit the features of multicore platforms, with admission control... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
Zhang, Yuanfang; Gill, Christopher; and Lu, Chenyang, "Real-Time Performance and Middleware on Multicore Linux Platforms" Report Number: WUCSE-2008-10 (2008). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/221

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/221

Real-Time Performance and Middleware on Multicore Linux Platforms

Yuanfang Zhang, Christopher Gill, and Chenyang Lu

Complete Abstract:

An increasing number of distributed real-time applications are running on multicore platforms. However, existing real-time middleware (e.g., Real-Time CORBA) lacks support for scheduling soft real-time tasks on multicore platforms while guaranteeing their time constraints will be satisfied. This paper makes three contributions to the state of the art in real-time system software for multicore platforms. First, it offers what is to our knowledge the first experimental analysis of real-time performance for vanilla Linux primitives on multicore platforms. Second, it presents MC-ORB, the first real-time object request broker (ORB), designed to exploit the features of multicore platforms, with admission control and task allocation services that can provide schedulability guarantees for soft real-time tasks on multicore platforms. Third, it gives a performance evaluation of MC-ORB on a Linux multicore testbed, the results of which demonstrate the efficiency and effectiveness of MC-ORB.

Washington University in St. Louis

School of Engineering & Applied Science

Real-Time Performance and Middleware on Multicore Linux Platforms *

Yuanfang Zhang, Christopher Gill and Chenyang Lu
Department of Computer Science and Engineering
Washington University, St. Louis, MO, USA
{yfzhang, cdgill, lu}@cse.wustl.edu

Abstract

An increasing number of distributed real-time applications are running on multicore platforms. However, existing real-time middleware (e.g., Real-Time CORBA) lacks support for scheduling soft real-time tasks on multicore platforms while guaranteeing their time constraints will be satisfied. This paper makes three contributions to the state of the art in real-time system software for multicore platforms. First, it offers what is to our knowledge the first experimental analysis of real-time performance for vanilla Linux primitives on multicore platforms. Second, it presents MC-ORB, the first real-time object request broker (ORB), designed to exploit the features of multicore platforms, with admission control and task allocation services that can provide schedulability guarantees for soft real-time tasks on multicore platforms. Third, it gives a performance evaluation of MC-ORB on a Linux multicore testbed, the results of which demonstrate the efficiency and effectiveness of MC-ORB.

1 Introduction

As evidenced by recent products from major CPU vendors, multicore processors (which include several cores on a single chip) are poised to dominate the real-time and embedded systems space. Dual-core chips are popular in today's market, and many CPU vendors have released designs with more than two cores. Applications that process large numbers of transactions with soft real-time constraints are likely deployed on multicore platforms even today. However, standard operating systems such as Linux can not effectively schedule soft real-time workloads on such platforms. Moreover, the real-time performance of Linux primitives on multi-core platform has not been systematically evaluated. Benchmarking the real-time performance of Linux primitives is essential for developing predictable real-time applications on multicore platforms.

While traditional real-time middleware such as Real-Time CORBA [10] ORBs have shown promise for distributed systems with soft real-time constraints, existing middleware lacks support for guaranteeing such constraints on multicore platforms. For example, existing admission control (AC) and task allocation (TA) services do not consider thread CPU affinity and migration issues that arise with a multicore architecture. Hosts are the minimum granularity for task assignment in existing middleware. However, on a multicore platform once a task is assigned to a host, it could be executed on any core or even migrated among cores in that host, which is not controlled by existing middleware. Any AC based on such an imprecise assignment necessarily loses its reliability. To support soft real-time tasks on multicore platforms, real-time middleware should therefore be able to provide AC and TA services not only among separate hosts, but also among cores within each host.

Research contributions. To address the limitations of current generation real-time middleware in supporting soft real-time tasks on multicore platforms, we have: (1) provided an experimental analysis of the real-time performance for vanilla Linux on multicore platforms, the results of which are valuable for both real-time middleware developers and real-time application developers alike; (2) developed what is to our knowledge the first real-time ORB for multicore platforms (MC-ORB), with AC and TA services that enforce real-time task constraints on multicore platforms; and (3) performed an empirical evaluation of MC-ORB, the results of which demonstrate the efficiency and effectiveness of our middleware on multicore platforms.

Section 2 introduces background information on the Linux kernel and real-time ORBs, and describes related work. Section 3 presents an experimental analysis of the real-time performance of vanilla Linux on multicore platforms. Section 4 presents the design of MC-ORB, the first real-time ORB specifically intended for multicore platforms. Section 5 evaluates the performance of MC-ORB and characterizes the overheads it introduces. Finally, we

*This work was supported in part by NSF grant CCF-0615341 and NSF CAREER award CNS-0448554.

offer concluding remarks in Section 6.

2 Background and Related Work

Linux kernel. Kernel support for symmetric multiprocessors (SMP) was introduced in Linux 2.0, but it wasn't until the 2.6 kernel that the power of SMP was fully realized. The pre-2.6 scheduler used a single runqueue for all processors in a SMP system. This meant that a task could be scheduled on any processor, which can be good for load balancing but could disrupt memory caches. The pre-2.6 scheduler also used a single runqueue lock, so that in an SMP system even the selection of which task to execute locked out any other processors from manipulating the runqueues, resulting in idle processors awaiting release of the runqueue lock and accordingly decreasing efficiency.

The 2.6 kernel [1, 2] introduced a new O(1) scheduler that included better support for SMP systems. Since the 2.6 kernel maintains a runqueue for each CPU, the number of running threads on each of the CPUs in the system can be balanced. At regular intervals, the kernel tries to redistribute threads to maintain a balance in the number of running threads per processor, across the processor complex. In addition, with a runqueue per processor, a thread generally shares affinity with a CPU and can better utilize the CPU's hot cache.

The better support for multiprocessor architectures in Linux 2.6 moves it closer to being an efficient soft real-time operating system on those platforms. However, Linux still can not fulfill important real-time requirements, such as guaranteeing system schedulability. This paper therefore focuses on characterizing the timing of Linux features on multicore platforms, and then on providing necessary AC and TA services in middleware for real-time systems.

Linux deficiencies. Calandrino et al. [4] offer reasons why current Linux support is inadequate for soft real-time periodic tasks. One deficiency is that Linux has no mechanisms to ensure that real-time deadlines are met. They added an AC mechanism in the Linux kernel to perform schedulability analysis for arriving soft real-time periodic tasks. Another deficiency is that Linux itself may migrate real-time tasks, which can cause deadline misses since tasks are no longer on the cores where their utilization was guaranteed by AC. To prevent real-time task migration, they modified the processor affinities of real-time tasks so that each thread may only run on a specific core. Instead of changing the Linux kernel to correct these deficiencies, we address them in middleware, i.e., in our new real-time ORB (MC-ORB), which is a more flexible approach since it does not depend on modifying the kernel and can be easily adapted to other operating systems.

Real-time ORBs. The OMG's Real-Time CORBA specification [10] provides standard policies and mechanisms that support quality-of-service requirements end to end, which enhances the effectiveness of distributed object computing middleware for performance-sensitive systems. ORBs send requests between clients and servers transparently. A real-time ORB end-system provides standard interfaces so that applications can specify their resource requirements. The policy framework defined by the CORBA Messaging specification lets applications configure ORB end-system resources, such as thread priorities, buffers for message queuing, transport-level connections, and network signaling, to control ORB behavior. TAO [6] is a real-time CORBA ORB that is compliant with the Real-time CORBA specification [10]. nORB [13] is a light weight real-time ORB for memory-constrained networked embedded systems. nORB is developed and maintained at Washington University and achieves comparable real-time performance to TAO, while reducing footprint significantly. We developed our MC-ORB implementation, and its AC and TA middleware services for multicore platforms by extending nORB.

Middleware services. In previous work we developed the first instantiation of a middleware AC service [16] supporting both aperiodic and periodic tasks, on top of TAO [6], and the first configurable component middleware services [15] for AC and load balancing of aperiodic and periodic tasks on top of CIAO [5]. However, all our previous middleware services were designed for uniprocessor platforms, and do not consider the nuances of multicore platforms.

3 Experimental Study of Linux on Multicore Platforms

In this section, we describe experiments we conducted to evaluate the real-time performance of vanilla Linux on multicore platforms. Since the reliability of the experiments is based on the quality of CPU timing information, we first did a clock calibration check on our testbed. After verifying that our testbed can deliver suitably precise timing measurements, we then used the testbed to evaluate the performance (1) of load balancing and thread migration mechanisms in vanilla Linux in this section, and (2) of MC-ORB in Section 5. All experiments presented in this section were performed on a dual-core Pentium-IV 3.4GHz machine with 2G RAM and 2048K cache, running Linux version 2.6.17. All real-time periodic tasks are implemented by real-time Portable Operating System Interface (POSIX) threads [3]. Periodic timeouts are generated by POSIX timers. Each timeout is sent as a POSIX signal to a particular thread. The performance information in this section is also provided as a guideline for the MC-ORB design and empirical evaluation discussions in Sections 4 and 5.

3.1 Clock Calibration Check

The x86 processor architecture has a 64 bit counter that is incremented once per clock cycle. The RDTSC instruction [7] puts the time-stamp counter (TSC) in registers edx:eax. The returned 64 bit value represents the count of ticks since the most recent processor reset. The RDTSC instruction previously has been an excellent high-resolution, low-overhead way of getting CPU timing information. With the advent of multicore/hyperthreaded CPUs and systems with multiple CPUs, our first challenge was to determine whether RDTSC can still provide reliable results. The issue has two components: rate of tick and whether all cores have identical values in their TSCs. On multicore platforms, there is no longer any promise that the TSCs of the multiple CPUs will be synchronized, and you can no longer simply assume reliable TSC values, unless you lock your program to using a single CPU.

Since in practice real-time applications may need to rely on the consistency of TSCs of multiple cores, we ran the following experiment to measure the offsets of TSC values between two cores. In this experiment, one POSIX thread runs on each core. Signals are sent back and forth between the two threads repeatedly for 5 minutes. As soon as one thread receives the signal from the other thread, it records its core's TSC value, then sends back an ACK signal to the other thread on the other core.

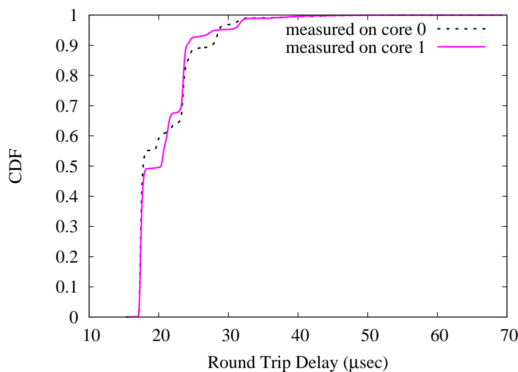

Figure 1. Round Trip Delay

Clock frequency difference check. The time interval from when a thread sends a signal until it receives an ACK signal from the other thread measures the round trip delay (RTD). We calculated the time by using the CPU frequency (obtained from the /proc directory) to divide the difference of TSC values at the beginning and end of each round trip. We also measured the round trip delay from the perspective of each core. As Figure 1 shows, the round trip delays measured by the TSCs on the different cores were approximately the same, which alleviates the potential concern about clock frequency differences in our test bed.

Offset check. We represent the TSC offset between core 1 and core 0 ($TSC_1 - TSC_0$) as δ, the TSC value on core 0 when it sends a signal as x, the TSC value on core 1 when it sends the ACK signal after receiving the signal from core 0 as y, and the TSC value on core 0 when it receives the ACK signal as z. We then have the following equations:

$$y - x = \delta + RTD/2 \quad (1)$$
$$z - x = RTD \quad (2)$$

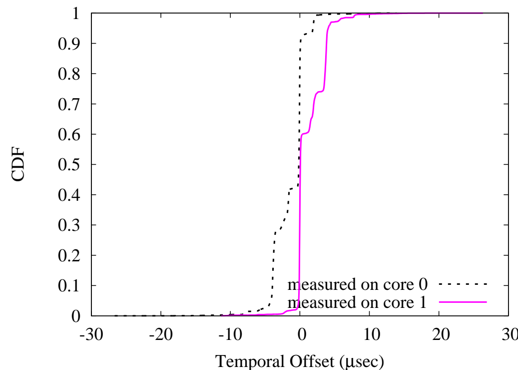

Figure 2. Offset Between Cores

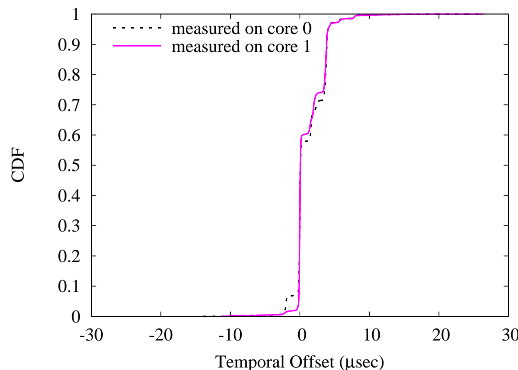

Figure 3. Reversed Offset Between Cores

After substituting $z - x$ for RTD in equation (1), we get $\delta = 2 * y - x - z$. We use this equation to measure the TSC offset on core 1, then convert it to a temporal offset by using the CPU frequency to divide it. We also get the TSC offset between core 0 and core 1 ($TSC_0 - TSC_1$) by swapping the roles of core 0 and core 1 in the above measurement. Figure 2 shows the cumulative distribution function (CDF) of temporal offsets between cores on our dual-core testbed machine. Assuming a non-zero temporal offset exists in our testbed, to avoid its influence on all measurements in this section and Section 5, the start and stop TSCs are recorded on the same core, and half of the round trip time between cores is used to approximate the one way time from one core to the other.

To ensure that these offset results are consistent and reliable, we then reversed all offset values from core 0 and

compared them with the offset values from core 1 as shown in Figure 3. They are closely matched with each other, which validates the accuracy of our observations. All time measurements in this paper are then based on similar use of the RDTSC to get high-resolution CPU timing information from each core.

3.2 Load Balancing Between Cores

When tasks are released in an SMP system, Linux places each one in a given CPU's runqueue. Linux does not know whether a task will be short-lived or will run for a long time. Therefore, the initial allocation of running tasks to CPUs is likely suboptimal. To maintain a balanced workload across CPUs, work can be redistributed, taking work from an overloaded CPU and giving it to an underloaded one. The Linux 2.6 scheduler approximates the needed functionality by balancing the *number* of threads in each runqueue, which is helpful if workloads are similar across threads.

At regular intervals, the kernel checks to see whether the thread counts per CPU runqueue are unbalanced; if they are, the kernel performs a cross-CPU re-balancing of running tasks. A drawback of this process is that the new CPU's cache is *cold* for a migrated task (needing to pull its data into the cache), but cache effects are not the main research issue of this paper. We are more concerned with the overhead introduced by such load balancing in a real-time system. Especially, when all real-time tasks are pinned to specific cores according to a global task allocation plan, the load balancing approach used by Linux can be both useless and troublesome.

We conducted four experiments to measure the overheads associated with the Linux load balancing mechanism. The goal of these experiments is to characterize the frequency and overhead of the load balancing mechanism in Linux. Four task sets are randomly generated, one for each of the four experiments. Two of the task sets contain 10 real-time periodic tasks each, and the other two contain 30 each. The periods of the tasks are uniformly distributed between 50 msec and 1 sec. All tasks in each set are bound to one core, which is accomplished by modifying the CPU affinity of each task. The CPU utilization for that core is either 0.6 or 1.0, which is randomly divided among all tasks on that core. Each experiment runs its task set for 5 minutes. We put the RDTSC instruction before and after the move_tasks() function in the sched.c file of the 2.6.17 Linux kernel source code, and write the TSC offsets into a static kernel level buffer. After re-compiling this modified kernel, we ran the four experiments described in this subsection on it.

The first column of Table 1 indicates the experiment, the second column shows the number of tasks, and the third column shows the utilization. The 4rd column shows the total number of times in 5 minutes when Linux found an imbalance of running threads between cores and attempted to move tasks for balancing. However, since each task is bound to a particular core, no actual task migration happens in these experiments. The time delay of actually moving a task between cores is discussed in Section 3.3. The rightmost columns of Table 1 show the overhead: Linux checks each thread in the busier runqueue to see if it can be moved to the other core before actually attempting to move it. Since threads are bound to their current cores, the check fails for each task, so the overheads for load balancing in these experiments are minima. The normal overheads of load balancing are these plus the actual task migration overheads (discussed in Section 3.3).

expt.	**tasks**	**util.**	**imbal.**	**overhead per imbal. (ns)**		
				min	mean	max
1	10	0.6	211	405	983	1899
2	30	0.6	210	566	1178	2120
3	10	1.0	588	536	854	1463
4	30	1.0	596	671	1124	2069

Table 1. Overhead of Load Balancing Checks

Comparing the overheads for task sets with 10 tasks vs. 30 tasks, there is a small but noticeable difference. Although the scheduler is required to check every thread in the runqueue to see whether it should be moved from an overloaded core to another core, the difference in overhead remains between 12% and 41% as the number of tasks in the system increases from 10 to 30. Since the number of tasks on any CPU in many real-time systems is usually less than 30, the task count is not expected to affect the overhead of load balancing significantly in real-time systems, especially given the μsec scale overheads involved for each check.

When we compare task sets with different CPU utilizations, we see a more significant increase in overhead. When the CPU utilization is 0.6 for a core, that core has idle time, so the load is "balanced" between the two cores and no task migration is required. When CPU utilization is 1.0 for one core, that core is overloaded. The load is always unbalanced between the cores and the kernel thus is always eager to move tasks.

In each of these four experiments, the total overhead of load balancing in 5 minutes was less than 1.3 msec which is acceptable for most soft real-time applications. Moreover, it is relatively easy to turn off the load balancing flag (SD_LOAD_BALANCE) in the Linux kernel if necessary, though as we noted previously such modifications may diminish portability.

3.3 Thread Migration Between Cores

Real-time tasks may be moved from one overloaded core to another underloaded core in order to admit more tasks while still guaranteeing their deadlines. Moving tasks can be done either automatically by Linux load balancing or manually by invoking a system call. In Linux, the system call sched_setaffinity() can be used to trigger thread migration among cores. It also can be used to bind threads to a particular core.

The sys_sched_setaffinity() function first looks for the descriptor of the target thread, then updates the thread's CPU affinity mask. Moreover, this function has to check whether the thread is included in a runqueue of a CPU that is no longer present in the new affinity mask. In that case, the thread has to be moved from one runqueue to another one. To avoid problems due to deadlocks and race conditions, this job is done by special kernel threads (there is one such *migration* thread per CPU). Whenever a thread has to be moved from a runqueue rq1 to another runqueue rq2, the kernel wakes up the migration thread of the processor associated with rq1, which in turn removes the thread from rq1 and inserts it into rq2. Before completing its work, the migration thread also checks whether (1) the migrated thread has higher priority than the currently running thread on the target CPU, and (2) the TIF_POLLING_NRFLAG flag of the rq2's currently running thread is clear (the target CPU is not actively polling the status of the TIF_NEED_RESCHED flag of the thread). If both conditions are true, the migration thread raises an Inter-processor Interrupt (IPI) and forces rescheduling on the target CPU. If only the first condition is true, it sets the TIF_NEED_RESCHED flag to 1 for the target CPU which then reschedules tasks at the next polling time. The migration thread runs at the highest priority, so the migration procedure is not interrupted by any other real-time threads.

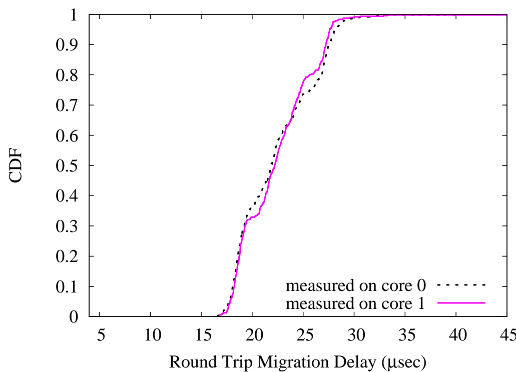

Figure 4. Self Migration

The delay of thread migration must be considered if such migration is used in a real-time system (e.g., to move threads between cores to balance utilization as we discuss in Section 4). We conducted experiments to measure this delay under different possible conditions, using 10 periodic real-time tasks. The period for each task is randomly sampled from a uniform distribution between 50 msec and 1 sec. Tasks are uniformly assigned to the two cores and the CPU utilization of each core is 0.5. The CPU utilization is randomly divided among all tasks assigned to that CPU. All tasks are scheduled with the RMS policy. All tasks are pinned to their assigned cores at creation time, except for one task which is migrated from one core to the other, and then immediately moved back (by itself or by another thread). The round trip migration delay is calculated by reading the begin and end TSCs on the same core, then subtracting the begin value from the end value. To avoid the migrated task being blocked by other higher priority tasks (mixing migration delay and scheduling delay), the migrated task always has the shortest period. We ran these experiments under three different scenarios. To ensure reliable results, in each scenario we ran the task set twice with the migrated thread starting on different cores. Each run lasted 5 minutes and the CDFs of all round trip migration delays under three different scenarios are showed respectively in Figures 4, 5 and 6.

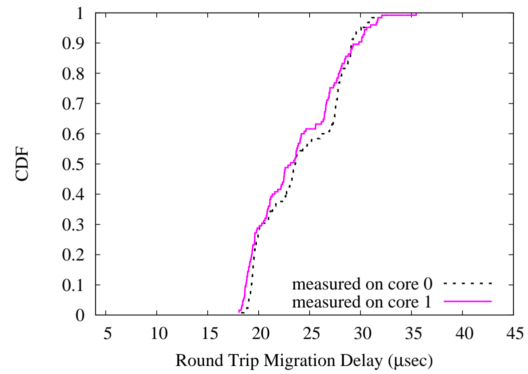

Figure 5. Running Thread Migration

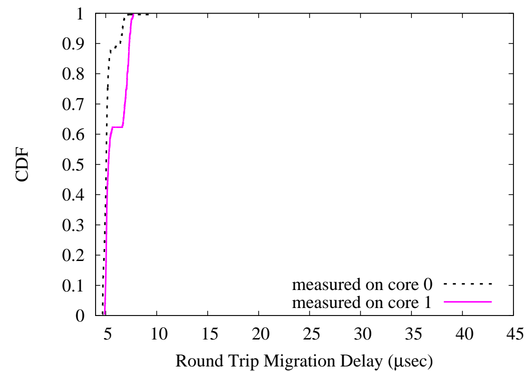

Figure 6. Sleeping Thread Migration

Task thread migrates itself (Figure 4). In this scenario,

the migrated task did a round trip migration itself at the beginning of each release. Figure 4 shows the self migration delay, which ranges between 16 and 45 μsec.

Manager thread migrates a running task thread (Figure 5). In this scenario, an extra manager thread (with the highest priority) is added. It migrates the shortest period task thread from one core to the other, and then immediately back, every 50 msec. Figure 5 shows the migration delay if the manager thread migrates the task thread when it is *running*, which ranges between 18 and 36 μsec.

Manager thread migrates a sleeping task thread (Figure 6). Figure 6 shows the delay in the third scenario, when the manager thread migrates the task thread when it is *waiting* for the next release signal, which ranges between 4 and 10 μsec. The delay shown in Figure 5 is significantly larger than in Figure 6, because when the task thread is not running, the manager thread does not need to wake up the migration kernel thread, and only needs to reset the CPU affinity mask of the task thread. Moreover, we do not see a significant difference between the results in Figure 4 and Figure 5, which means that when a task is running, migrating it to another core either by itself or by another thread incurs similar delay. These performance results were important considerations in our design and evaluation of MC-ORB which are discussed in Sections 4 and 5.

4 Middleware Design

We developed MC-ORB for multicore platforms, with the following three new features: (1) MC-ORB contains new layers in the ORB-level server-side request processing structure to include necessary real-time task support; (2) MC-ORB provides a new AC service to perform schedulability analysis for each arriving soft real-time task on a multicore system and to accept or reject the task accordingly; and (3) MC-ORB provides a new TA service that can allocate each real-time task to an appropriate core, and also may re-allocate previously admitted real-time tasks to make room for a newly arriving one.

4.1 Design Challenges and Solution Approach

Design challenges. In open real-time applications, new tasks may arrive dynamically and simultaneously on multicore servers, and require AC and TA services to ensure satisfaction of their timing requirements. The number of potential concurrent server tasks is equal to the product of the number of server operations, the number of different priorities, and the number of potential clients. The number of potentially concurrent server tasks may vary from application to application, so that it is not easy to create the proper number of threads with the proper priorities at system initialization time, and a more reasonable approach may be to handle those tasks as they arrive. However, dynamically creating and destroying threads add significant overhead, and should be limited to the extent possible.

Solution approach. To address these challenges, we use a two-layer architecture in MC-ORB. In the lower *connection* layer, static connections are created at system initialization time. There is a connection for each different priority for each potential client. Each client thus has its own pre-connected connection for each particular priority. In the upper *application* layer, we pre-create all the threads which are going to process server operations and keep them in distinct pools. All threads in one pool are bound to one particular core. This approach bounds the cost of the resources that the server is going to use. The number of threads is equal to the maximum number of requests the server can handle simultaneously, which is also the server's capacity.

Contrast this with the thread-per-request model, in which a new thread is created for each request. If it receives a large spurt of requests in a short period of time, the server will spawn a large number of threads to handle the load. This would degrade service for all requests and may cause resource allocation failures as the load increases. In the thread pool model, when a request arrives, an existing thread is chosen from one of the pools to handle the request.

4.2 MC-ORB Architecture

Communication between MC-ORB endsystems is per the CORBA model: a client stub marshals the parameters of a remote call into a request message and sends it to a remote server, which demarshals the request and calls the appropriate servant object; the reply is then marshalled into a reply message and sent back to the client, where it is demarshalled and the result is returned to the caller.

On the client side, each thread is associated with a reactor-level timer that dispatches periodic timeout events to that thread. When the timer fires, the thread sends the request to the server through a pre-connected priority *lane* [11] according to its priority. On the server side, a *reactor* [12] is associated with each lane, and connection threads wait on the reactor for the requests from clients at each particular priority. The number of connection threads for each priority is equal to the number of potential clients in the system, and their priorities are equal to the priority of that lane. MC-ORB server side architecture contains four layers. The steps in this architecture, shown in Figure 7, are: (1) connection threads read the incoming client requests from the network, demultiplex the requests to connection handlers that perform general Inter-ORB protocol (GIOP) processing, and insert the connection handlers into priority queues; (2) a manager thread processes the requests

in order according to their priorities; (3) the manager thread invokes the AC and TA services for each task; (4) the manager thread dequeues the first application thread from the proper thread pool, changes the priority of this application thread according to the request priority, and then passes a connection handler to it; (5) the application thread collaborate with an Object Adaptor to dispatch requests to servant operations using the connection handler.

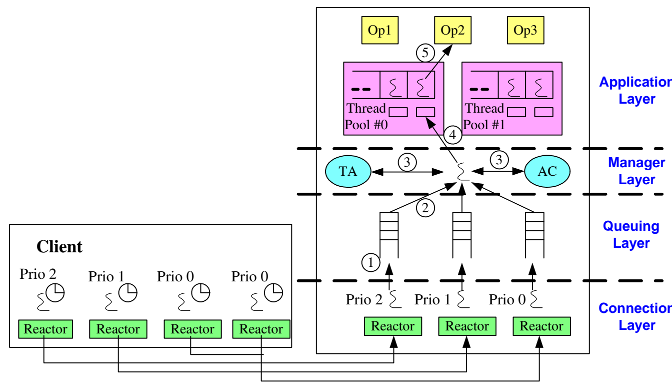

Figure 7. Single Manager Thread Architecture

Priority-based connection layer. The connection layer is statically configured with different priorities, e.g., Prio 0, Prio 1 and Prio 2. In this architecture, each client maintains a map of pre-established connections to servers. One lane is maintained for each connection priority in the server ORB (as shown in Figure 7 on the client side multiple threads at the same priority may feed into one server-side priority lane). Once a client connects, the server side ORB creates a new connection handler for each priority and registers it with the reactor for that priority lane, and also creates a connection thread with the appropriate real-time priority to wait on the reactor. All connections are pre-allocated during MC-ORB initialization, which minimizes the latency between client invocation and servant operation execution. Once a request is sent, the connection thread reads it from the network, stores it, and inserts the point to the connection handler into the proper priority queue in the queuing layer.

Queuing layer. This layer provides the mechanism for communicating between the connection layer and the manager layer. The queuing layer thus allows the connection layer and the manager layer to interact in a decoupled "producer/consumer" manner. To avoid priority inversion, the queuing layer consists of multiple queues, one for each priority. Requests in the highest non-empty priority queue are processed preferentially by the manager layer. Queuing is necessary, since our system does not bound the number of requests, so simultaneously arriving requests can be buffered in the queue while waiting for further processing. Moreover, the queues are only used to pass pointers to connection handlers, which is very efficient.

Manager layer. A single manager thread (which has highest priority) blocks on a condition variable. Once it is notified, it will process queued requests according to their priorities. After dequeuing a request, if it is not from an admitted task, the manager thread first decides whether to accept this request and to which core to allocate this request if so, by invoking the AC and TA services. Otherwise, the manager thread allocates it to the previous assigned core. The manager thread then dequeues the first waiting application thread from the thread pool on the proper core, changes the priority of that application thread to the request's priority and passes the connection handler to it.

The connection threads in the lower level can not do AC and TA directly, because the AC and TA services require exclusive access. An application thread with medium priority could preempt a connection thread with low priority when it was accessing the AC and TA services, while other connection threads with high priority were waiting to access the AC and TA services. This priority inversion could occur if low priority threads are allowed to enter this critical section. By using a single manager thread with the highest priority to access the AC and TA services, we preclude such a priority inversion. Although many RTOS [14] kernels include support for either priority inheritance or priority ceilings to adjust the priorities of threads while they access a critical section, we developed MC-ORB for operating systems like vanilla Linux which is widely used for multicore platforms but does not have such support.

At system initialization time, applications can use interfaces provided by the AC and TA services to specify the processing requirements of their operations in terms of various parameters, such as execution time, period, deadline or priority. Our AC service can support any schedulability based admission test corresponding to a static-priority uniprocessor scheduling algorithm. The AC service performs the admission test for each arriving task. The TA service assigns incoming tasks and re-allocates previously admitted tasks to the proper cores by exhaustively searching for an optimal partition that can satisfy the schedulability utilization bound according to the static-priority scheduling algorithm running on the server. Since the numbers of cores and tasks are limited in our testbed, exhaustive search can be used on-line by our TA service, though the focus here is not on the AC and TA algorithms themselves: our middleware services can be easily implemented using other AC and TA algorithms according to each application's needs.

Application layer. Each application thread waits in a queue until it is notified by the manager thread, and then obtains the passed connection handler. The target servant is found, the request is demarshalled, and the application-level method is executed. When the application thread completes

it work, it enqueues itself back into the thread pool. We have one thread pool for each core, all threads in each pool are bound to that particular core, and all application threads are pre-created and inserted into pools at MC-ORB initialization time. At first, the numbers of threads in all pools are equal and fixed. The total is equal to the number of requests that the server handles simultaneously. The thread pool model avoids thread migration overhead (discussed in Section 3.3), if there are waiting application threads in the proper pool. If no threads are available in the proper pool, perhaps because they are all busy processing requests, but threads are available in other pools, we can migrate a thread from another core onto the proper core and then execute the request. If no application threads are available in the whole system to process the new request, we can either reject the request because it exceeds the server's capacity, or expand the server's capacity by spawning more threads at run time.

4.3 Leader/Followers Variation

For the manager layer, instead of using a designated manager thread to invoke the AC and TA services, another design we investigated is to use a Leader/Followers architecture. Any application thread may become the leader thread which in turn invokes the AC and TA services. The steps in this approach, shown in Figure 8, are: (1) the leader thread invokes the AC and TA services, and then puts the priority and CPU affinity information of the current request into a special shared variable; (2) the leader thread picks one thread on a different core than its own as the new leader, wakes up the new leader, then blocks itself on a condition variable; (3) the new leader reads the information from the shared variable, changes the priority and CPU affinity of the old leader thread according to the information, then wakes up the old leader thread; (4) the old leader thread executes the requested operation. In step 3, we only allow the new leader to change the blocked thread's CPU affinity instead of the old leader itself changing, because of the overhead difference shown in Section 3.3.

Comparing this architecture to the single manager thread architecture described in Section 4.2, for each arriving request the single manager thread architecture introduces fewer context switches (1 vs. 2) in the manager layer and less frequent thread migration (in the Leader/Followers architecture there is a 50% chance of migration on *each request*), and has acceptable message passing cost. We therefore implemented only the single manager thread architecture in MC-ORB. MC-ORB implemented a new ORB-level server-side request processing structure to include necessary real-time task support by extending nORB [13] small-footprint real-time ORB, and its performance was evaluated in Section 5.

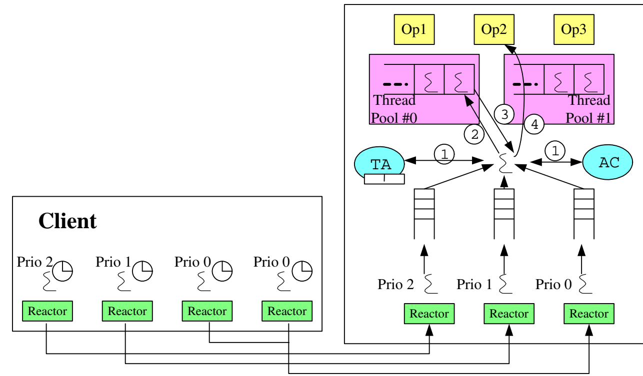

Figure 8. Leader/Followers Architecture

5 Middleware Evaluation

The experiments we conducted to evaluate MC-ORB were performed on a testbed consisting of two dual-core Pentium-IV 3.4GHz machines with 2G RAM and 2048K cache each. Both machines ran Linux version 2.6.17. One machine is used as the client, which sends requests to the other (server) machine. In the following experiments, each task set consists of periodic real-time tasks. Task periods are uniformly distributed between 50 msec and 1 sec. The deadlines of periodic tasks are equal to their periods. All tasks are scheduled by the RMS policy. The AC service does the admission test using the time-demand analysis [8] for RMS. The TA services provides the optimal partition under the schedulability utilization bound for RMS [9] by exhaustive search.

5.1 Overhead Measurement

We measured the extra overhead that MC-ORB introduced on the server side in processing each request to guarantee its real-time constraints. We measured the delay from when the connection thread receives the request until the proper application thread is notified to process the request or the task is rejected. In this experiment, we always bound the connection thread and application thread for the measured requests to same core to avoid the influence of clock offsets between cores. This experiment used one task set with 11 periodic real-time tasks. The total CPU utilization for the 2 cores was 1.6 and was randomly divided among all tasks. To avoid a connection thread or an application thread being blocked by executing other higher priority tasks and thus mixing true overhead with the scheduling delay, we always used the requests from the task with the highest priority in this measurement, and we forced MC-ORB to invoke the AC and TA services for every request from this highest priority task. MC-ORB may handle a request in 5 different

ways, so we ran the same task set 5 times under 5 different scenarios to measure the overheads, the results of which are shown in Table 2. Each run lasted 5 minutes. The scenarios were: (1) a task is allocated to the same core as the manager thread; (2) a task is allocated to a different core than the manager thread; (3) all threads on the proper core have been used up, and one thread is moved from the other core to execute the new task; (4) running tasks are reallocated to allocate the new task; (5) the new task is rejected by the AC service.

scenario no.	minimum	mean	maximum
1	43 μs	55 μs	109 μs
2	42 μs	58 μs	111 μs
3	50 μs	64 μs	121 μs
4	222 μs	235 μs	289 μs
5	39 μs	50 μs	107 μs

Table 2. Overhead of MC-ORB Extensions

The overheads for scenarios 1 and 2 are close to each other, which means that allocating tasks to either core does not make much difference in overhead. The difference between scenarios 1 and 3 is almost equal to the system overhead of migrating a sleeping thread which is measured in Section 3.3. The overhead for scenario 4 is larger than all other scenarios, because migrating one running thread costs about 15 μsec (as we measured in Section 3.3). In the run for scenario 4, we always forced MC-ORB to reallocate all other running tasks when the requests from the highest priority task arrives, so the overhead in Table 2 was the maximum (in practice the actual overhead is proportional to the number of reallocated tasks). The overhead when rejecting a task is the smallest of all scenarios. The extra overhead introduced by MC-ORB per request under any scenario was less than 0.3 msec, which is acceptable for most soft real-time applications.

5.2 Deadline Miss Ratio Comparison

The Linux operating system only considers the numbers of running threads when it does load balancing between cores. When the running threads require significantly different CPU utilization, only balancing the numbers of running threads can not balance the workloads well between the cores, which may cause some tasks that require high CPU utilization to miss their deadlines.

In the experiments presented in this section, each task set contains two groups of periodic real-time tasks. The total utilization of a task set is equally divided between the two groups. One group always contains a fixed number of periodic real-time tasks ($n_1 = 10$). The other group contains a variable number of periodic real-time tasks, but the number is no greater than 10 ($n_2 \leq 10$). These experiments thus have two changeable factors. One is the total utilizations of all tasks in a task set. The other is the balance factor between the two groups ($N = n_2/n_1$). The smaller the balance factor, the greater the utilization difference of individual tasks in the two groups. We randomly generated 10 task sets for each pair of changeable factors, and ran each task set for 5 minutes on both nORB and MC-ORB. The performance metric is the *fraction of workloads with any deadline miss* in the 10 task sets. The results are shown in Figures 9, 10 and 11 when the total utilization is increased from 1.4 to 1.5 and to 1.6. We validated the performance of our AC service by running this experiment when the AC and TA services in MC-ORB are enabled. There was no deadline miss for any workloads in that experiment: all admitted tasks met their deadlines, although some tasks were rejected by the AC service. Then, to focus only on the effect of the TA service in MC-ORB, in the following experiments, we disabled the AC service in MC-ORB.

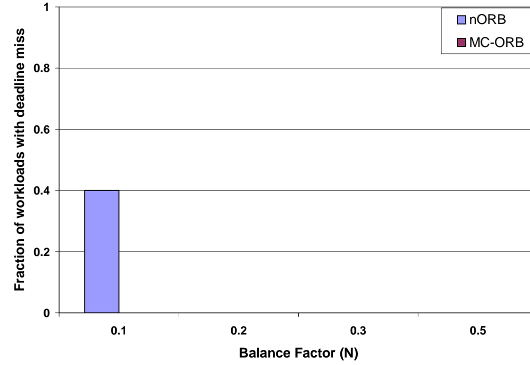

Figure 9. Misses when Total Utilization is 1.4

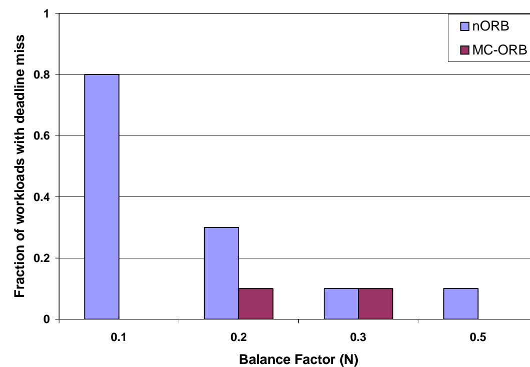

Figure 10. Misses when Total Utilization is 1.5

For light workload task sets, no bars are shown in Figure 9 for balance factors 0.2, 0.3 or 0.5, because there is no deadline miss for any task set in either nORB or MC-ORB under those 3 scenarios. However, when the balance

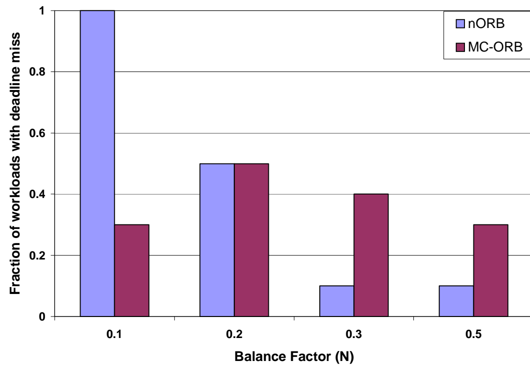

Figure 11. Misses when Total Utilization is 1.6

factor is 0.1, there is still no deadline miss in MC-ORB, while 4 task sets have deadline misses in nORB. This is because MC-ORB allocates tasks according to their utilizations, while nORB relies on the Linux load balancing mechanism which only counts the number of running tasks. The difference is clearer for a small balance factor, since tasks in each task set have extremely unbalanced workloads when the balance factor is 0.1. For medium workload task sets shown in Figure 10, there is no deadline miss in MC-ORB when the balance factor is 0.1 or 0.5. The performance of MC-ORB does not change much when the balance factor decreases, and always outperforms nORB, while the balance factor affects the performance of nORB significantly. For most of the randomly generated task sets in Figures 9 and 10, there may exist some partitions that can satisfy sufficient schedulability utilization bound for RMS, and if so our TA service in MC-ORB can find it through exhaustive search and can assign tasks according to this optimal partition to guarantee all tasks meet their deadlines. However, for the heavy workload task sets in Figure 11, there does not exist any partition that satisfies the sufficient schedulability utilization bound for RMS. Our TA policy in this case is still balancing the task utilization between cores. Since the "balancing" is only done at the task arrival time and is based on the estimated current utilization information, it can not be very precise, and the performance is not ideal, while the dynamic load balancing at short time intervals in Linux works better in some cases. However, the goal of our TA service on multicore platforms is to collaborate with the AC service and to provide an acceptable assignment which can guarantee all admitted tasks meet their deadlines. This paper does not focus on the problem of how to reduce the deadline miss ratio without the AC support.

6 Conclusions

Our work represents a promising step towards developing a new generation of real-time middleware for soft real-time tasks on multicore platforms. We first analyzed the real-time performance of vanilla Linux on multicore platforms. We then designed and implemented a novel real-time ORB, MC-ORB, which provides schedulability guarantees for real-time tasks on multicore platforms. Empirical evaluations showed that MC-ORB is highly efficient and effective on a multicore Linux platform.

References

[1] J. Aas. *Understanding the Linux 2.6.8.1 CPU scheduler*. Silicon Graphics, Inc., 2005.

[2] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd edition*. O'Reilly Publishers, 2005.

[3] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, Massachusetts, 1997.

[4] J. M. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. H. Anderson. Soft Real-Time Scheduling on Performance Asymmetric Multicore Platforms. In *RTAS*, 2007.

[5] Institute for Software Integrated Systems. Component-Integrated ACE ORB (CIAO). www.dre.vanderbilt.edu/CIAO/, Vanderbilt University.

[6] Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

[7] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2008.

[8] J. P. Lehoczky, L. Sha, J. Strosnider, and H. Tokuda. Fixed Priority Scheduling Theory for Hard Real-Time Systems. In *Foundations of Real-Time Computing, Scheduling, and Resource Management*, 1991.

[9] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1):46–61, Jan. 1973.

[10] Object Management Group. *Real-Time CORBA Specification*, 1.1 edition, Aug. 2002.

[11] I. Pyarali, M. Spivak, R. K. Cytron, and D. C. Schmidt. Optimizing Threadpool Strategies for Real-Time CORBA. In *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, 2001.

[12] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[13] V. Subramonian, G. Xing, C. Gill, C. Lu, and R. Cytron. Middleware Specialization for Memory-Constrained Networked Embedded Systems. In *RTAS*, 2004.

[14] A. Wellings, A. Burns, O. M. dos Santos, and B. M. Brosgol. Integrating Priority Inheritance Algorithms in the Real-Time Specification for Java. In *ISORC*, 2007.

[15] Y. Zhang, C. Gill, and C. Lu. Reconfigurable Real-Time Middleware for Distributed Cyber-Physical Systems with Aperiodic Events. In *ICDCS*, 2008.

[16] Y. Zhang, C. Lu, C. Gill, P. Lardieri, and G. Thaker. Middleware Support for Aperiodic Tasks in Distributed Real-Time Systems. In *RTAS*, 2007.