Real-Time Performance and Middleware for Multiprocessor and 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 adequate support for ensuring the timing constraints of soft real-time tasks on multicore platforms, and thus is dependent on (potentially inadequate) support from the underlying operating system. 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 of vanilla Linux primitives on multicore platforms. Second, it presents MC-ORB, the first real-time object request broker (ORB) designed to address the nuances of multiprocessor (and especially multicore) platforms with a novel core-aware middleware thread architecture and allocation service for soft real-time tasks. Third, it evaluates MC-ORB's performance on a Linux multicore testbed, the results of which demonstrate its efficiency and effectiveness.**

***Keywords*-real-time middleware; multi-threading; multiprocessor and multicore Linux; performance evaluation**

I. Introduction

Processors with multiple cores on each chip are poised to dominate the real-time and embedded systems space. 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 do not effectively schedule real-time workloads on such platforms. Although many real-time systems have been based on specialized hardware and operating systems, the *vanilla* Linux common-off-the-shelf (COTS) OS is commonly used in practice (e.g., in multimedia products) to reduce costs. Therefore, benchmarking the real-time performance of Linux primitives is essential for developing predictable real-time applications on multicore platforms.

Requirements for increased software productivity and quality motivate the use of *distributed object computing (DOC) middleware*, such as CORBA [1], rather than building applications entirely from scratch. The use of CORBA middleware has increased significantly in domains such as aerospace, telecommunications, medical systems, and distributed interactive simulations, that are characterized by stringent quality of service requirements.

While traditional real-time DOC middleware (e.g., Real-Time CORBA [2]) has shown promise for distributed systems with soft real-time constraints, existing middleware lacks support for multiprocessor (and especially multicore) platforms. For example, existing task allocation (TA) services and object request broker (ORB) concurrency architectures do not consider thread CPU affinity and migration issues that arise with a multicore architecture. Instead, hosts are the minimum granularity for task assignment in existing middleware, even though 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, beyond the control of existing middleware and at the mercy of the particular kind (and version) of operating system upon which it runs. Any admission control (AC) based on such an imprecise assumption necessarily loses its applicability. To support soft real-time tasks on multicore platforms, portably across a variety of operating systems (and their versions), real-time middleware must *explicitly* manage task allocation not only among hosts, but also among each host's cores.

Research contributions. To address the limitations of current real-time middleware in supporting soft real-time tasks on multiprocessor platforms, we have: (1) conducted an experimental analysis of the real-time performance of key Linux features on a multicore platform, the results of which are valuable for both our middleware design presented in this paper and as guidance for other real-time system developers alike; (2) developed what is to our knowledge the first real-time DOC middleware designed for multicore platforms, called MC-ORB (for *MultiCore Object Request Broker*); and (3) performed an empirical evaluation of MC-ORB, the results of which demonstrate the efficiency and effectiveness of our middleware on a multicore platform.

Section II introduces background information on Linux and on real-time object request brokers (ORBs), and describes related work. Section III presents an experimental analysis of the real-time performance of key Linux features on a multicore platform. Section IV presents the architecture and services of MC-ORB, the first real-time ORB specifically designed for multicore platforms. Section V evaluates the performance of MC-ORB and characterizes the overheads it introduces, and Section VI offers conclusions.

II. Background and Related Work

Linux kernel. The Linux 2.6 kernel [3] introduced a new O(1) scheduler with better support for SMP and multicore

platforms. Prior to the 2.6.23 kernel, a runqueue was maintained for each core and at regular intervals, the kernel would try to redistribute threads to maintain a balance in the number of running threads per processor, across the processor complex. In the 2.6.23 kernel a new "completely fair scheduler" (CFS) [4] was introduced, based on fair scheduling of threads across processors.

While these advances move Linux closer to being an efficient soft real-time operating system on multiprocessor platforms, the level of support depends on the version of Linux installed. In practice, distributed real-time systems tend to have long upgrade cycles imposed by re-certification requirements and other constraints and thus real-time middleware must be designed for *portability across operating system versions as well as across different kinds of operating systems*. This paper therefore focuses (1) on characterizing Linux features on multicore platforms for a median (2.6 but pre-2.6.23) version of the OS that is reasonably representative of the platforms in current use, (2) on a novel middleware architecture that explicitly reduces thread migrations (and thus dependence on particular platform versions to minimize their cost), and (3) on providing a core-aware middleware task allocation (TA) service.

Linux performance. Bryant et al. [5] have developed Linux kernel enhancements to reduce multiprocessor thread management overheads on SGI Altix platforms. Calandrino et al. [6] improve Linux support for soft real-time systems, including adding admission control (AC) for arriving soft real-time periodic tasks in the Linux kernel to guarantee schedulability, and modifying processor affinities to prevent Linux from migrating real-time tasks from the processors where their utilizations were guaranteed by AC. Brandenburg et al. [7] evaluate scheduling algorithms on a multicore platform with hardware multi-threading, based on their LITMUS^{RT} Linux extensions.

Instead of extending Linux itself, we address thread migration and utilization issues in our new MC-ORB middleware. Our use of per-core thread pools (described in Section IV) resembles the clustered scheduling of hardware threads in LITMUS^{RT}, though our middleware approach is more flexible since it can be easily ported to various other platforms. Moreover, we characterize performance of additional Linux features, including non-negligible tick offsets between cores and *vanilla Linux* load balancing and thread migration overheads on a multicore testbed, observations which we have used in the design of MC-ORB.

Real-time ORBs. The OMG's Real-Time CORBA specification [2] provides standard policies and mechanisms that support quality-of-service requirements end to end, including standard interfaces that let applications specify their resource requirements and configure object request broker (ORB) end-system resources, such as thread priorities, message buffers, connections, and network signaling, to control ORB behavior. TAO [8] is a full-featured Real-time CORBA [2] ORB. nORB [9] is a light-weight real-time ORB for memory-constrained networked embedded systems, which achieves comparable real-time performance to TAO, while reducing footprint significantly. MC-ORB uses nORB as a starting point, but introduces a new concurrency architecture and TA service to ensure task feasibility.

In previous work we developed the first instantiation of a middleware AC service [10] supporting both aperiodic and periodic tasks, on top of TAO [8], and the first configurable component middleware services [11] for AC and load balancing of aperiodic and periodic tasks on top of CIAO [12]. However, those middleware services were designed for single-core platforms, and did not consider the characteristics and requirements of multicore platforms, such as task allocation among cores.

III. Real-Time Performance of Linux

In this section, we present an experimental analysis of three aspects of Linux on multicore platforms that are crucial for many real-time applications: (1) clock differences between cores, (2) the overhead of thread balancing between cores, and (3) the latency of thread migration between cores. The results presented in this section provide key guidelines for the development of both real-time applications and middleware on multicore platforms, which we apply to the design and evaluation of MC-ORB in Sections IV and V. All experiments in this section were performed on a dual-core Pentium-IV 3.4GHz (constant rate on both cores) machine with 2GB RAM and 2MB cache, running Linux 2.6.17. We do not introduce lock contention or concurrent rebalancing of multiple cores in our experiments, though our approach could be extended to consider those issues as future work. While the quantitative results of our performance study are specific to this particular hardware architecture and OS version, the observations from our study provide important *insights* for real-time system design on multiprocessor (including multicore) platforms. Furthermore, our measurement *methodology* can be applied to benchmark the real-time performance of other multiprocessor platforms. All real-time periodic tasks are implemented by real-time POSIX threads [13]. Periodic timeouts are generated by POSIX timers. Each timeout is sent as a POSIX signal to a particular thread.

A. Clock Differences between Cores

The x86 processor architecture has a 64 bit counter that is incremented once per clock cycle. The RDTSC instruction [14] puts the TSC in registers edx:eax. The returned 64 bit value represents the count of ticks since the most recent processor reset. The RDTSC instruction has been an excellent high-resolution, low-overhead way of getting timing information on single-core processors. Intel synchronizes the cores' TSCs at boot time, but does not guarantee TSC values on different cores are exactly the same

at any time. Linux also tries to synchronize the TSCs across the cores at boot time, though a small skew between TSCs may still remain. Since real-time applications may rely on the consistency of TSCs of multiple cores, it is important to evaluate the differences between the TSC values acquired by the RDTSC instruction on different cores. Specifically, we look for (1) any differences in the observed clock *frequencies* and (2) any *offset* of the TSC values between cores, whether due to hardware or software. 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 a signal to the thread on the other core.

Clock frequency. The interval between when a thread sends and receives a signal measures the round trip delay (RTD), based on the difference of TSC values at the beginning and end of each round trip. We measured the RTDs from the perspective of each core. As the cumulative distribution functions (CDFs) in Figure 1 show, the RTDs measured by the TSCs on the different cores are similar, with negligible differences between the mean values (on core 0 20.715 μs with standard deviation 4.141 μs, and on core 1 20.716 μs with standard deviation 4.138 μs). These results indicate that the observed clock frequencies of different cores are sufficiently close to each other for most soft real-time applications. Although the variance is moderately high relative to the small RTD values, these results are a suitable basis for measuring clock offsets (see next).

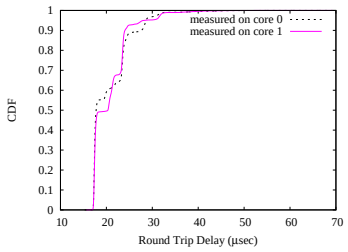

Figure 1. Round Trip Delay

Clock offset. We then evaluate the offset between the clocks of different cores. We represent (i) the TSC offset between core 1 and core 0 ($TSC_1 - TSC_0$) as δ, (ii) the TSC value on core 0 when it sends a signal as x, (iii) the TSC value on core 1 when it sends a signal as y, and (iv) the TSC value on core 0 when it receives a signal as z. Figure 2 illustrates the timing order. We then have (1) $y - x = \delta + RTD/2$ and (2) $z - x = RTD$.

Note that the one-way latency from one core to the other is approximated by half of the round-trip latency in (1), and the round-trip latency is measured on the same core in (2). Solving (1) and (2), we get $\delta = 2 * y - x - z$. We use this equation to estimate the TSC offset on core 1. We also get the TSC offset between core 0 and core 1 ($TSC_0 - TSC_1$)

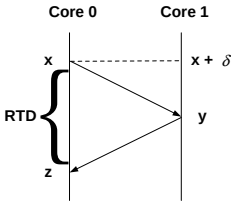

Figure 2. Clock Offset Measurement

by swapping their roles in the above measurement.

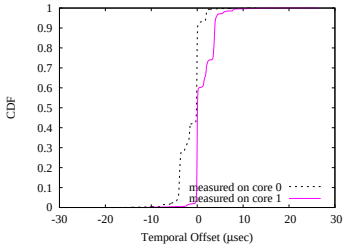

Figure 3. Offset Between Cores

Figure 3 shows CDFs of temporal offsets between cores on our dual-core testbed machine. To ensure that the offsets observed on different cores are consistent, we then reversed the signs of all offset values from core 0 and compared them with the offset values from core 1, as is shown in Figure 4.

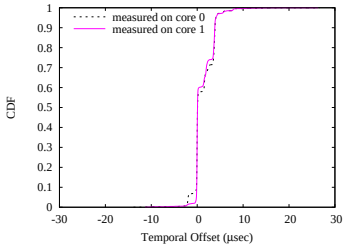

Figure 4. Offset Between Cores After Reversing Signs Of Values Collected On Core 0

They are closely matched, which validates consistency of the measured offsets on both cores. Core 0 saw a mean of 1.295 μs, a maximum of 26.645 μs, and a standard deviation of 2.228 μs, while on core 1 they were 1.294 μs, 26.335 μs, and 2.091 μs, respectively. These experiments show that offsets of up to 27 μs may exist between the TSC values on different cores. For high-resolution timing measurements, these offsets may affect the accuracy of results.

Insight 1: *Clock frequencies of different cores are close to each other. However, clock offsets between cores are non-negligible. Therefore for all the timing measurements in this section and Section V, we always record the start and stop TSCs on the same core, and use half of the round trip time between cores to approximate the one way time from one core to the other. This same guideline should be applied to other timing measurements of activities across cores.*

B. Load Balancing Between Cores

To deal with varying workloads on multiprocessor platforms, Linux dynamically balances the numbers of threads located on different cores. On pre-2.6.23 Linux platforms, the rebalancing operation first checks whether the other core is busier than the local core. If so, it then invokes the move_tasks() function to try moving threads from the other runqueue to the local one. The move_tasks() function scans all threads in the other runqueue. For each thread, the function invokes can_migrate_task(), which returns 1 if the local core is included in the thread's CPU affinity mask. If can_migrate_task() returns 1, move_tasks() moves the candidate thread to the local runqueue. If the cores are essentially balanced or all threads on the busier core are bound to that core, the rebalancing operation tunes the *balance_interval* parameter in the scheduling domain descriptor to delay the next rebalancing operation invocation.

While the thread balancing mechanism in Linux may improve the performance of general-purpose applications on multiprocessor platforms, it is unsuitable for real-time applications because it is not cognizant of task schedulability [6]. Furthermore, dynamic thread balancing may severely affect the predictability of real-time performance. Therefore, real-time applications and middleware must prevent real-time threads from being migrated by Linux. Specifically, a thread can be *bound* to a core by setting its CPU affinity mask using the sched_setaffinity() system call provided by Linux. However, even when every thread is bound to a core, the Linux thread balancing mechanism is still invoked by the kernel. We are interested therefore in quantifying the overhead introduced by Linux thread balancing when no threads can be moved.[1]

The move_tasks() function tries to move threads between unbalanced cores, so it costs the most when (1) all threads are bound to specific cores and (2) the number of them on each core is unbalanced. We measured the execution time of the move_tasks() function and the overhead *per invocation* because of kernel level buffer size limitations.

We conducted four experiments to measure the overheads associated with the Linux thread balancing mechanism. The goal of these experiments is to characterize the frequency and overhead of the thread 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. Each periodic task does simple computation for different iterated times, so that cache behavior has very little influence on the overhead of thread balancing. The periods of the tasks are uniformly distributed between 50 ms and 1 s. All tasks in each set are bound to one core, by modifying the CPU affinity of each task. The 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 insert RDTSC instructions 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 the modified kernel, we ran the four experiments described in this section on it.

[1] An alternative approach to disable the thread balancing mechanism is to change the load balancing flag and re-compile Linux. Since we are interested in understanding the impact of thread balancing on real-time applications, we did not disable thread balancing in our experiments.

tasks	util.	imbal.	overhead per imbal.(ns)			overhead
			min	mean	max	(total μs)
10	0.6	211	405	983	1899	207
30	0.6	210	566	1178	2120	247
10	1.0	588	536	854	1463	509
30	1.0	596	671	1124	2069	670

Table I
OVERHEAD OF LOAD BALANCING CHECKS

Each row of Table I indicates one experiment, the first column shows the number of tasks, and the second column shows the utilization. The third column shows the total number of times in 5 minutes when Linux found an imbalance between the cores and attempted to move tasks. 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 III-C. The 4th, 5th and 6th columns of Table I characterize the overheads when the move_tasks() function 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 all threads in the busier runqueue have to be scanned. The rightmost column shows the total overhead during each 5 minute run. The normal overheads due to thread balancing are these plus the actual task migration overheads (discussed in Section III-C).

As expected thread balancing incurs higher overhead with more threads because the move_tasks() routine must iterate through all tasks upon invocation. Thread balancing also incurs a higher overhead when the busy core has a higher utilization (the other core is always idle in our experiments). When the utilization is 0.6 for the busy core, the core may be idle when rebalance_tick() is invoked by the kernel. Since the other core is always idle, the runqueues of both cores are then empty and hence do not need to be balanced. In contrast, when the utilization is 1.0 for a core, it is overloaded, the load is always unbalanced between the cores, and the kernel attempts to move tasks more frequently. In each of these four experiments, the total overhead of thread balancing in 5 minutes was 670 μs or less, which is negligible for most soft real-time applications.

Insight 2: *Linux load balancing is useless for real-time applications with a task allocation service that binds threads to cores, but its overhead is insignificant, making it unneces-*

sary to re-compile the kernel to disable the load balancing mechanism for most soft real-time applications.

C. Thread Migration Costs

To deal with dynamic arrivals and departures of real-time service requests, real-time middleware needs to reallocate threads at run time. However, in contrast to the thread balancing mechanism of the Linux kernel, real-time middleware must reallocate threads based on their schedulability. In Linux, the sched_setaffinity() system call can be used to trigger thread migration among cores. It also can be used to bind threads to a particular core.

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

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 IV). 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 ms and 1 s. Tasks are uniformly assigned to the two cores such that the utilization of each core is 0.5. The utilization is randomly divided among all tasks assigned to that core. Each task is executed by a separate thread assigned a real-time priority based on the Rate Monotonic policy. All tasks are bound to their assigned cores at creation time, except for one task that is migrated from one core to the other, and then immediately moved back at the beginning of each release. 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. Here, half of the round trip time between cores is used to approximate the one way time from one core to the other to avoid the offset influence between cores, which is non-negligible as we observed in Section III-A. Furthermore, to avoid other higher priority tasks interfering with the migrated task (thus 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 shown respectively in Figures 5, 6 and 7 and are compared in Table II.

Self migration. In this scenario, a thread migrates itself round trip at the beginning of each release. Figure 5 and the second row of Table II show the self migration delay, which ranges between 16 and 45 μs.

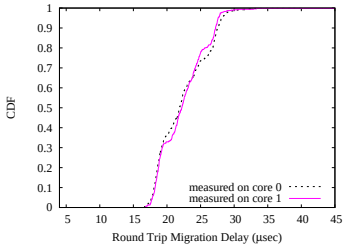

Figure 5. Self Migration

Manager thread migrates a running thread. In this scenario, an extra (highest Linux kernel priority for real-time processes) manager thread migrates the thread running the task with the shortest period from one core to the other and then immediately back, every 50 ms. Figure 6 and the third row of Table II show the migration delay if the manager thread migrates the task thread when it is *running*, which ranges between 18 and 36 μs.

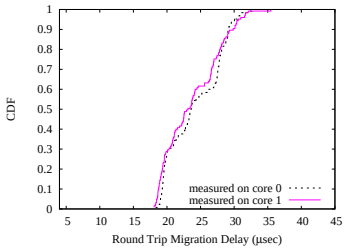

Figure 6. Running Thread Migration

Manager thread migrates a sleeping thread. Figure 7 and the fourth row of Table II show 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 μs. The delay shown in Figure 6 is significantly larger than in Figure 7, 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 5 and Figure 6, which means that when a task is running, migrating it to another core either by itself or by another thread incurs similar delays. These results were important considerations in the design and evaluation of MC-ORB as discussed in Sections IV and V.

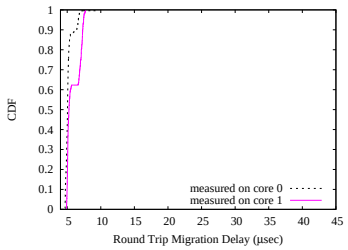

Figure 7. Sleeping Thread Migration

	core	min (μs)	mean (μs)	max (μs)
Self Migration	0	16.58	22.394	40.352
	1	16.765	22.418	44.988
Running Thread Migration	0	18.289	24.211	32.653
	1	18.078	23.785	35.475
Sleeping Thread Migration	0	4.676	5.207	9.523
	1	4.892	5.894	7.714

Table II
MIGRATION OVERHEAD COMPARISON

Insight 3: *Thread migration overhead can be non-negligible. To maintain task feasibility portably across different operating systems and operating system versions MC-ORB's concurrency architecture (described in Section IV) thus restricts threads to run on specific cores.*

IV. MIDDLEWARE DESIGN

MC-ORB features a novel server-side ORB architecture specifically designed for multiprocessor (and especially multicore) platforms. Important innovations of this architecture are: (1) a multicore-aware *task allocation* (TA) service designed to reduce the number of thread migrations among cores; (2) exclusive invocation of that TA service from within a single highest priority *manager thread* to avoid priority inversions; and (3) *handing off* client requests from that manager thread to *core-specific thread pools* that handle the requests at the appropriate priorities, again to reduce the number of thread migrations (in contrast to the *leader/followers* approach described in Section IV-C).

As we note in Section III-C, thread migration overhead may be non-negligible. An additional drawback of thread migration is that the new core's instruction cache is *cold* for a migrated thread (which could impose further costs comparable to those for migration). Therefore reducing unnecessary task migration can also minimize the cache performance penalty. Since operating systems generally do not expose cache states to the middleware layer, cache-aware scheduling [15] is not addressed in this paper. For MC-ORB we developed both a novel concurrency architecture and a task (re)allocation strategy, to reduce thread migrations.

There are two approaches to scheduling tasks on multiple cores: (1) partitioned, in which each task is assigned to a core more or less statically; and (2) global, in which tasks compete for the use of all cores. Although scalable OS-level global scheduling can be achieved [7], middleware-level global scheduling is unsuitable: global scheduling requires fine grain processor control, but middleware sits atop an operating system that may not expose fine grain scheduling information or control mechanisms. Therefore, MC-ORB adopts a partitioned scheduling approach.

A. MC-ORB Architecture Overview

A dynamic distributed real-time application implemented on MC-ORB consists of clients and servers, as Figure 8 illustrates. Each task corresponds to a client's periodic remote calls to an operation implemented on a server. A client may start or terminate a task dynamically, and provides a task's period and deadline information to the server when it starts the new task. The execution time of each operation is pre-registered with that server's TA service at system initialization time. The priority of a task is determined at run time by the server's scheduling policy.

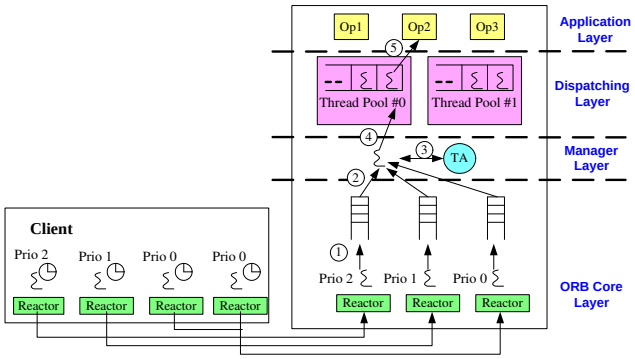

Figure 8. MC-ORB Middleware Architecture

On the client side, each thread is associated with a timer that generates periodic timeout events. When the timer fires, the thread sends the request to the server through a pre-connected priority *lane* [16]. On the server side, a *reactor* [17] is associated with each lane, and connection threads wait on the reactor for the requests from clients. 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. The server-side invocation steps, shown in Figure 8, are: (1) connection

threads read 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' addresses into priority queues; (2) a single highest priority manager thread processes the requests in order according to their priorities; (3) the manager thread invokes the TA service for each task; (4) the manager thread dequeues the first dispatching thread from the proper thread pool, changes the priority of this dispatching thread according to the request priority, and then gives a connection handler to it; (5) the dispatching thread collaborates with an Object Adaptor to dispatch requests to server operations using the connection handler.

This real-time middleware architecture is described in further detail in Section IV-B. In contrast to traditional (uniprocessor) real-time middleware architectures, it is specifically designed to avoid unpredictable and significant delays due to thread migration on multiprocessor and multicore platforms, by (1) restricting server-side thread pool boundaries to individual cores and (2) as a trade-off potentially requiring more hand-offs between server-side threads. Although this also could be achieved by direct modifications to the OS kernel, we focus instead on a portable approach targeted towards off-the-shelf operating systems.

B. MC-ORB Server Architecture

MC-ORB's server side architecture enforces a crucial separation of concerns among its four distinct layers, to promote optimization of each layer and to avoid inefficiency that can arise from entangling concerns. The *ORB core layer* handles transmission of client requests and responses. The *manager layer* manages task allocation while avoiding priority inversion and reducing thread migration. The *dispatching layer* receives requests from the manager layer and performs prioritized concurrent dispatching of client requests to server operations hosted in the *application layer*.

ORB core layer. The ORB core layer is statically configured with prioritized connections, and each client maintains a map of pre-established connections to servers. One lane is maintained for each connection priority in the server-side ORB (as shown in Figure 8, multiple client-side threads at the same priority may feed into a server-side priority lane). All connections are pre-allocated during MC-ORB initialization, which minimizes the latency between client invocation and server operation execution. Once a request arrives, a connection thread reads it from the network, stores it, and inserts a connection handler's address into the proper priority queue. Although an admission test is used in the Manager layer to enforce feasibility of their *processing*, queuing of requests in the ORB core is necessary since their *arrival* is unrestricted. The queues are only used to pass pointers to connection handlers, which is efficient.

Manager layer. To avoid priority inversion, requests in the highest priority non-empty queue are processed preferentially by the manager layer. On each host a single highest priority *manager thread* is responsible for all cores in that host. The manager thread blocks on a condition variable when all queues are empty. Once new requests arrive, the manager thread is notified by the lower-level connection thread delivering them, and then processes the queued requests according to their priorities. After dequeuing a request, if it is not from a previously admitted task, the manager thread first decides whether to accept this request and if so to which core to dispatch it, by invoking the TA service. Otherwise, the manager thread dispatches it to its previously assigned core. When a new task is admitted, the manager thread first looks for an available thread in the thread pool on the designated core. If there are available threads, the manager thread dequeues the first waiting dispatching thread from the thread pool on the proper core, then changes the priority of that thread to the request's priority and gives the connection handler to it. If the manager thread cannot find one on the designated core but there is an available thread in a different core's pool, the manager thread migrates the thread to the designated core using the sched_setaffinity() system call. If no thread is available in the server, the manager thread can either create a new thread or reject the new task.

We note that the connection threads in the lower layer cannot do task allocation directly, because the TA service needs exclusive serialized access to the admitted task allocation information. Consider also the case where a medium priority dispatching thread could preempt a low priority connection thread when it was invoking the TA service, and then other connection threads with high priority were forced to wait to invoke the TA service. This priority inversion could occur if low priority threads were allowed to enter this critical section. By using a single manager thread with the highest priority to invoke the TA service exclusively, we preclude such a priority inversion.

Task (re)allocation service. The TA service performs the admission test and allocates arriving tasks to cores. The key design goal is to reduce the number thread migrations and the associated overhead and cache penalty.

When a new task arrives, the server records the task's period and deadline information, and the TA service performs an admission test and allocation for the new task based on that information. The TA service also records the admitted task's allocation information. When a new task arrives, the TA service first tries to allocate the incoming task to any core on the server without re-allocating any admitted task and invokes the admission test for each such potential allocation. If any of them passes the admission test, the new task is admitted. Otherwise, the TA service searches for an allocation that passes the admission test by exploring potential re-allocations of previously admitted tasks to different cores. The manager thread then implements the task re-allocation by changing each affected thread's CPU affinity using the sched_setaffinity() system call. If no

schedulable allocation is found, the task is rejected.

Our TA service uses an exhaustive search that terminates at the first admissible allocation found, which is suitable for systems with a small number of cores. In principle our TA service can support any allocation algorithm suitable for static-priority uniprocessor scheduling, though our focus is not on the TA algorithms themselves.

Dispatching layer. The number of concurrently arriving requests may vary from application to application, so that it is not easy to determine the proper number of threads *a priori*. There are two traditional approaches for handling such dynamic requests: (1) creating threads on-the-fly in response to new requests (which often is too expensive [16]); and (2) statically creating the maximum possible number of threads (which often would consume too much memory). Existing real-time ORBs use *thread pools* at the granularity of hosts as a compromise between the two traditional approaches.

To reduce the number of thread migrations, we therefore again manipulate CPU affinities so that each thread pool is bound to a specific core. Each dispatching thread waits in a *core-specific thread pool* until it is notified by the manager thread, and then obtains the passed connection handler. The number of threads in each pool is equal to the server's capacity (how many requests it can handle at once).

All dispatching threads are pre-created, inserted into pools, and bound to a core at MC-ORB initialization time. At first, the numbers of threads in all pools are the same and fixed. When a new request arrives, if there are waiting dispatching threads in the thread pool on the designated core, one of them is dequeued from the pool and handles the request. If all threads in the proper pool are busy processing other requests but threads are available in other pools, as we noted previously the manager thread can migrate a thread from another core and then execute the request – if no dispatching threads are available at all the manager thread 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, as specified.

Application layer. The selected dispatching thread collaborates with an Object Adaptor to find the target server operation, demarshalls the request, executes the application-level method, and then puts itself back into the thread pool.

Implementation. MC-ORB is implemented using ACE 5.2.7 and was developed and evaluated on Linux 2.6.17. MC-ORB is based on nORB [9] with the major extensions including 7875 lines of C++ code (excluding ACE and IDL libraries). All the code can be downloaded as open-source from http://www.cse.wustl.edu/∼yfzhang/MC-ORB.html.

C. Leader/Followers Variation

Another alternative we investigated is the Leader/Followers architecture that is standardly used by existing real-time ORBs [8], [9] to reduce inter-thread communication and context switch costs *within a uniprocessor* [16] but which (as the following discussion explains) can increase the number of costly thread migrations if used on a multiprocessor platform. As shown in Figure 9, with the Leader/Followers architecture: (1) the current 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 a thread on another core as the new leader, wakes it up, 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 III-C. In contrast, the single manager thread architecture described in Section IV-B introduces fewer context switches (1 vs. 2) in the manager layer for each arriving request 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 adopted the single manager thread architecture for MC-ORB.

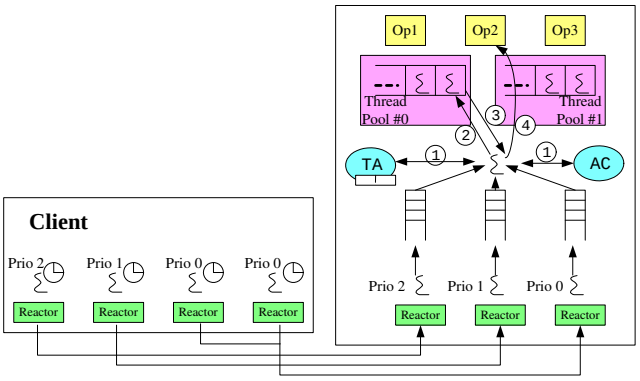

Figure 9. Alternative Leader/Followers Architecture

V. Middleware Evaluation

The experiments described in this section were performed on a testbed consisting of two dual-core Pentium-IV 3.4GHz machines with 2GB RAM and 2MB cache each. Both machines ran Linux 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 ms and 1 s. The deadlines of periodic tasks are equal to their periods. All tasks are scheduled by the RMS policy. The admission test is based on the RMS schedulable utilization bound [18]. Each core-specific thread pool contains 15 real-time threads.

A. Overhead Measurement

We measured the extra overhead that MC-ORB introduced on the server side in processing each request. We measured the delay from when the connection thread receives the request until the proper dispatching thread is notified to process the request or the task is rejected. In this experiment, we always bound the connection thread and dispatching thread for the measured requests to the same core to avoid the influence of clock offsets between cores as we observed in Section III-A. This experiment used one task set with 11 periodic real-time tasks. The total utilization for the 2 cores was 1.6 and was randomly divided among all tasks. To avoid a connection thread or a dispatching 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 TA service for every request that came 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 III. 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 are busy but a 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.

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 III
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 close to the overhead of migrating a sleeping thread measured in Section III-C. The overhead for scenario 4 is larger than all other scenarios, because migrating one running thread costs about 20 μs (as we measured in Section III-C), which reinforces our observation that thread migration overhead should be considered in real-time middleware design. In scenario 4, we always forced MC-ORB to reallocate all dispatching threads in the runqueues of two cores when the requests from the highest priority task arrives, so the overhead in Table III was maximal (in practice the actual overhead is proportional to the number of worker threads actually moved by the reallocation algorithm). The overhead when rejecting a task is the smallest of any scenario. The extra overhead introduced by MC-ORB under any scenario was less than 0.3 ms/request, which is acceptable for most soft real-time applications.

B. Real-Time Performance Comparison

MC-ORB is based on nORB [9] with major extensions for multicore platforms. Since nORB is designed for single-core processors, it does not perform task allocation among cores but is under the influence of kernel-level thread balancing in Linux. To evaluate the real-time performance advantages of MC-ORB on multicore platforms, we compare MC-ORB with nORB in the following experiments.

In the experiments in this section, each task set contains two groups of periodic real-time tasks. These task sets are designed to explore feasibility effects under different conditions of total load and load imbalance between cores. The total utilization of a task set is equally divided between 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 utilization* 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.

We validated the performance of MC-ORB's TA service by running this experiment and saw no missed deadlines for any workloads: all admitted tasks met their deadlines, although some tasks were rejected by the TA service. The average acceptance ratios when the total utilization is 1.6 and the balance factor is 0.1, 0.2, 0.3, or 0.5, were 94.97%, 95.36%, 96.44%, and 95.09%, respectively.

To focus only on the effect of the task allocation in MC-ORB, in the following experiments, we then disabled the rejection mechanism in MC-ORB. Then, when the TA service cannot find a schedulable allocation, the new task is still admitted, the TA service allocates the new task and may re-allocate the admitted tasks to balance utilization of the cores. The performance metric is the *fraction of workloads with a deadline miss*. The results are shown in Table IV when the total utilization is increased from 1.4 to 1.5 and to 1.6.

Even then, the task allocation provided by MC-ORB's TA service can effectively improve the server's ability to meet deadlines of task sets with unbalanced utilizations, and its benefit increases as the balance factor decreases indicating increasingly unbalanced task sets. As shown in Table IV, even with a total utilization of 1.4. nORB caused 4 unbalanced task sets (with a balance factor of 0.1) to miss deadlines, while MC-ORB met the deadlines of all task sets. The advantage of MC-ORB in meeting deadlines is more

Total Util.	ORB	Balance Factor			
		0.1	0.2	0.3	0.5
1.4	nORB	0.4	0	0	0
	MC-ORB*	0	0	0	0
1.5	nORB	0.8	0.3	0.1	0.1
	MC-ORB*	0	0.1	0.1	0
1.6	nORB	1	0.5	0.1	0.1
	MC-ORB*	0.3	0.5	0.4	0.3

Table IV
FRACTION OF WORKLOADS WITH DEADLINE MISS WHEN TOTAL UTILIZATION INCREASES (MC-ORB*: REJECTION MECHANISM IS DISABLED)

significant for unbalanced task sets with total utilization of 1.5: MC-ORB met the deadlines of all task sets with a balance factor of 0.1, while nORB missed deadlines in 80% of the task sets. However, the performance of MC-ORB degrades under higher load (with a total utilization of 1.6) along with that of nORB. This is because under overloaded conditions, balancing the utilization of all cores is not sufficient, and admission control must be employed to meet deadlines of a subset of the tasks. Our earlier experiments with MC-ORB's rejection mechanism activated showed that MC-ORB can effectively handle such overloaded conditions through on-line admission control.

VI. CONCLUSIONS

Our work represents a promising step towards developing a new generation of portable real-time middleware for soft real-time tasks on multiprocessor (and especially multicore) platforms. We first empirically evaluated the real-time performance of relevant vanilla Linux features on a multicore platform. The observations from our study provide important insights as design guidelines to the development of real-time applications and middleware: for example that clock offsets and thread migration overheads are non-negligible. We then designed and implemented a novel real-time ORB named MC-ORB, which is the first real-time middleware specifically designed to address the additional challenges (especially due to thread migration) posed by multicore platforms. We presented empirical evaluations showing that MC-ORB is highly efficient and effective on a multicore Linux platform, especially in comparison to a real-time ORB designed for uniprocessor platforms.

ACKNOWLEDGMENT

This work was supported in part by NSF grant CCF-0615341 (EHS) and NSF CAREER awards CCF-0448562 and CNS-0448554.

REFERENCES

[1] *The Common Object Request Broker: Architecture and Specifi cation*, 3.0.2 ed., OMG, Dec. 2002.

[2] *Real-Time CORBA Specifi cation*, 1.1 ed., OMG, Aug. 2002.

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

[4] I. Molnar, "Modular Scheduler Core and Completely Fair Scheduler," lwn.net/Articles/230501/, 2007.

[5] R. Bryant and J. Hawkes, "Linux Scalability for Large NUMA Systems," in *Proceedings of the Linux Symposium*, Ottowa, Ontario, Canada, Jul. 2003.

[6] J. M. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. H. Anderson, "Soft Real-Time Scheduling on Performance Asymmetric Multicore Platforms," in *13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '07)*, 2007.

[7] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson, "On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study," in *RTSS*, 2008.

[8] ISIS, "The ACE ORB (TAO)," www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

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

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

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

[12] ISIS, "Component-Integrated ACE ORB (CIAO)," www.dre.vanderbilt.edu/CIAO/, Vanderbilt University.

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

[14] *Intel 64 and IA-32 Architectures Software Developer's Manual*, http://www.intel.com/products/processor/manuals/index.htm, 2008.

[15] J. M. Calandrino and J. H. Anderson, "Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study," in *20th Euromicro Conference on Real-Time Systems (ECRTS '08)*, Prague, Czech Republic, Jul. 2008.

[16] 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.

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

[18] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.