Fast Component Interaction for Real-Time Systems

Udo Steinberg, Jean Wolter, Hermann Härtig
Technische Universität Dresden
Department of Computer Science
01062 Dresden, Germany
{steinberg, wolter, haertig}@os.inf.tu-dresden.de

Abstract

Open real-time systems provide for co-hosting hard-, soft- and non-real-time applications. Microkernel-based designs in addition allow for these applications to be mutually protected. Thus, trusted servers can coexist next to untrusted applications. These systems place a heavy burden on the performance of the message-passing mechanism, especially when based on microkernel-like inter-process communication.

In this paper we introduce capacity-reserve donation (in short Credo), a mechanism for the fast interaction of interdependent components, which is applicable to common real-time resource-access models. We implemented Credo by extending L4's message-passing mechanism to provide proper resource accounting and time-donation control, thereby preserving desired real-time properties.

We were able to achieve priority inheritance and stack-based priority-ceiling resource sharing with virtually no overhead added to L4's message-passing implementation. By providing a mechanism that does not impose performance penalties, while still guaranteeing correct real-time behaviour, Credo allows for the usage of microkernels in general-purpose but also in specialized systems.

1. Introduction

In the last years a significant amount of work has been put into the integration of requirements for different computing domains into real-time environments. The resulting systems are called open systems because they are no longer restricted to hard real-time applications. Examples for such open systems are "open secure systems" which integrate trusted subsystems and untrusted legacy components into one system, and open real-time systems [4] which co-host hard-, soft- and non-real-time-systems, for example a GSM stack, a banking application and games running in parallel on a cell phone.

All these applications place various constraints on the underlying system. To be able to run these applications on one platform the following mechanisms for separation are required:

- Temporal separation to provide real-time capabilities
- Spatial separation to ensure protection and fault containment
- Communication control to prevent "security leaks"

Microkernels provide an ideal platform to host open systems. They provide address spaces for spatial protection and thereby isolate faults and protect trusted components from attacks and faults in untrusted components. Information flow can be controlled by restricting inter-process communication (IPC) [10], the only means of communication available in microkernel-based systems. Because all components heavily rely on IPC, it has to be fast. Current systems are mostly based on synchronous IPC, which achieves a high performance due to its short code path and low pressure on hardware resources like cache and TLB.

Synchronous IPC implementations differ in how they reduce IPC overhead. They either migrate threads from one component to another like in Pebble [6] or they block the sender and directly switch to the receiver like in L4 [12]. Because in the latter approach the sender blocks until the receiver sends a reply, IPC creates a dependency between two schedulable entities, which can lead to scheduling anomalies.

Most real-time kernels that focus on minimality use a simple fixed-priority scheduling algorithm, which on one hand does not provide temporal isolation and on the other hand is prone to priority-inversion problems when component interaction creates a dependency between two schedulable entities. In this paper we introduce a reservation-based approach to provide temporal isolation. We discuss the problems related to component interaction, such as priority inversion and correct accounting of consumed time, and demonstrate how our mechanism can be used to implement

Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)
0-7695-2400-1/05 $20.00 **IEEE**

priority inheritance and stack-based priority ceiling. Finally we discuss performance issues and show that the described mechanism does not degrade IPC performance much while providing a predictable behaviour for component interaction.

1.1. Related Work

Various approaches have been proposed to temporally isolate applications in an open system thereby protecting them against the misbehaviour of other applications. Hierarchical approaches [4, 11, 21] isolate the different application classes in a high-level scheduler that in turn hands down CPU guarantees that can be used by the lower-level schedulers to execute their application class. Especially cross-class communication requires a decision of all schedulers involved during the communication. While this overhead can possibly be neglected for monolithic systems, the typically one to two orders of magnitude faster inter-process communication in microkernel-based systems cannot tolerate the performance penalty.

The Resource-Reservation framework [17, 20], which is based on Bandwidth Isolation [1, 7], successfully isolates applications by reserving hardware resources and guaranteeing each application a time C in every period T for which the application can use the resource. Resource-reservation techniques have also been applied to dynamic-priority schemes [1] and ported to Linux [19]. Constant-Bandwidth Servers [1] use deadline postponing to provide bandwidth isolation. Extensions for resource reclaiming [15, 3] have been proposed for and added to CBS. However, these approaches only support independent tasks.

Resource Containers [2] provide a flexible mechanism for servers to execute using the resources of their clients. Niz et al. [18] apply resource-container techniques for resource sharing in reservation-based systems. Server processes switch reservations depending on which client's request they process. However, the overhead for changing the resource container to be used and for delegating a container between processes is unduly large to be performed on every interaction.

[7] showed theoretically that donation-based systems are viable. Early work in L4 [12] implemented a version of donation that, while being fast, lacked generality due to unpredictability. The opposite path with a predictable and general model as proposed by [5] came with a prohibitive cost for microkernel systems. Credo in turn is both predictable and efficient because it extensively removes scheduling from the critical communication path, making capacity-reserve donation applicable for high-performance inter-component communication schemes as they are provided by the L4 microkernel [13].

The remainder of this paper is structured as follows: Section 2 discusses how message passing and capacity-reserve donation can be combined to provide predictable real-time interaction between interdependent components and describes our solution to priority-inversion and accounting problems. In Section 3 we show how common resource-access protocols, such as priority inheritance and stack-based priority ceiling, can be built on top of our mechanism. We evaluate our approach in Section 4 and conclude the paper in Section 5.

2. Fast Component Interaction

2.1. Message Passing in Microkernel-Based Systems

To interact with each other, components must communicate. In this section we examine the problems that arise when message passing is applied to real-time systems with interacting components and illustrate how our approach overcomes these problems.

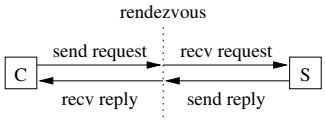

Figure 1. Message passing between a client and a server

Figure 1 shows a typical component interaction between a client and a server thread in a system with synchronous message-passing, such as in L4. A successful message transfer requires a rendezvous between a sender and a receiver thread. Initially the client acts as sender and sends a request to the server. The server acts as receiver and must agree to receive the message from the client – by waiting for a message from a particular client or from any client. Once the server has received the request, it works on behalf of the client to handle the request and computes a reply. During the message transfer from the server back to the client the server acts as sender and the client is the receiver. The client blocks after sending the request until the server replies or a specified timeout triggers. The server unblocks from its receive operation at the moment it receives the request and blocks again in another receive operation after sending the reply back to the client. L4 achieves its efficient message-passing performance by completely eliminating scheduling from the critical path by directly switching from the sender to the receiver of a message so as to avoid scheduling decisions and by lazily updating the ready queue. Applying the invariant that all ready threads except the currently executing thread must be in the ready queue and that blocked

threads may remain enqueued until the next invocation of the scheduler, this technique removes the need to manipulate the ready queue during message passing for the common case that the server sends its reply back to the client before the next scheduling decision is required.

Many scheduling models assume that all components of a real-time system are independent of each other. In a system with frequent component interaction through message passing this assumption is often not true. Threads providing a service can be viewed as a resources and all client threads communicating with servers contend for these resources. In combination with fixed-priority scheduling of threads this leads to priority-inversion problems as shown in the left-hand example of Figure 2.

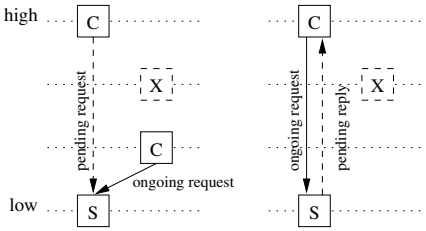

Figure 2. Priority inversion during message passing

In the right-hand example a client thread C sends a request to the low-priority server thread S. Before S can respond it is preempted by a medium-priority thread X, which may cause unbound priority inversion in the uniprocessor case. Therefore, the basic message-passing mechanisms must be designed such that well-known methods to avoid priority inversion, namely priority inheritance and stack-based priority ceiling, are supported without sacrificing performance.

2.2. Approach

To achieve fast component interaction while preventing priority inversion and misaccounting, we combine message passing with a donation scheme. While usually the scheduler keeps thread state in a single data structure, our approach maintains two types of context information for each thread: an execution context (CPU/FPU registers, thread state) and a scheduling context. The scheduling context represents the capacity reserve of the thread, a time quantum coupled with a priority, and is used to sort threads in the ready queue. Due to the separation of the contexts the kernel can now switch execution context and scheduling context indepedently, which facilitates the implementation of capacity-reserve donation from one thread to another.

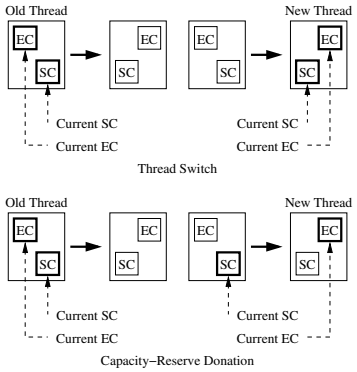

Figure 3. Context separation and switching

The kernel tracks references to the currently running execution context and to the currently active scheduling context. The current scheduling context defines the priority of the current execution context and provides the time quantum for executing the thread represented by that execution context. During a regular thread switch from one thread to another the kernel switches to both the execution context and the scheduling context of the selected thread. The newly dispatched thread then consumes its own time quantum and runs with its own priority as shown in the left-hand example of Figure 3.

In contrast to regular thread switches, the kernel switches only the execution context when sending a request from a client to a server thread as illustrated in the right-hand example of Figure 3. By not switching the scheduling context, the client effectively donates its time quantum and priority to the server for the time the server works on behalf of the client. Upon sending the reply, the server switches back to the client's execution context and the client reobtains its donated scheduling context. Donation of scheduling contexts is transitive. If the server needs to contact another server to process the request, it acts as a client itself and further donates the scheduling context it received. Combining lazy queueing and capacity-reserve donation with message passing leads to:

- Fast message passing – no ready queue manipulations, priority changes or scheduling decisions required on the critical kernel path
- Correct accounting – the consumed CPU time is always accounted to the client requesting a service

For an intuitive understanding of the application of these mechanisms to priority inheritance and stack-based priority ceiling the reader may temporarily jump to Section 3. However, some ramifications needed for a complete understanding will only be provided in the following subsections.

2.3. Donation Algorithm

In a fully preemptive kernel asynchronous thread wakeups due to expired send- or receive timeouts can cause the preemption of the current execution context. In the uncommon case that such a wakeup preempts a client-server scenario with a donation dependency, the scheduler must recognize and correctly resume the donation. We now devise an algorithm for the scheduler to select the new current scheduling context (SC) and the new current execution context (EC) after a preemption by tracking the donation path.

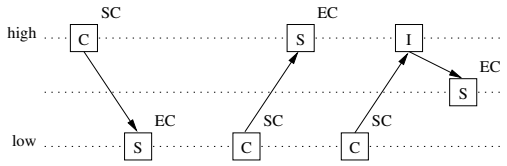

Figure 4. Donation scenarios (downward, upward, mixed)

Figure 4 shows three different scenarios of donation. When the scheduler selects a new thread to run, it traverses the ready queue starting at the highest priority level. We distinguish the following three cases of capacity-reserve donation:

1. Downward donation – The client that donates its scheduling context has a higher priority than each server involved in the handling of the client's request. Therefore the scheduler finds the blocked client first, selects the client's scheduling context as current scheduling context and then follows the message-passing partners from the client to the thread to which the client's scheduling context had been donated prior to the preemption of the request handling. The execution context of the last thread in the donation chain becomes the current execution context.

2. Upward donation – If the last server involved in the message-passing operation has a higher priority than the client, the scheduler finds that ready server first and selects it as the current execution context. However, the scheduler must not select the server's own scheduling context as the current scheduling context and instead traverse the message-passing partners backwards to find the source of the donation. The beginning of the donation chain is the client whose scheduling context becomes the current scheduling context.

3. Mixed donation – If a server has to contact another server to handle the client's request, a scenario can occur where both the client and the last server have a lower priority than an intermediate server that was involved in the handling of the request. The scheduler then finds the blocked intermediate thread first. Traversing the donation chain backwards leads to the client and yields the current scheduling context. Following the donation chain from the intermediate server to the ready server leads to the current execution context.

We can now summarize the algorithm for resuming a preempted message-passing operation by the scheduler as follows:

- When the scheduler finds a blocked thread in the ready queue, traversing the donation chain backwards yields the source of the capacity-reserve donation – the current scheduling context, which may also be the blocked thread itself. Traversing the donation chain forward leads to the most recent recipient of the capacity-reserve donation, the thread representing the current execution context.

- When the scheduler finds a ready thread, no forward traversal of the donation chain is necessary – the ready thread represents the current execution context. However, the scheduler must still find the current scheduling context, which may be provided by a different thread.

Although this original algorithm promised zero overhead on the critical message-passing path of the kernel and pushed the entire fixup cost for the preemption case into the scheduler, we found a number of problems that we needed to address in a revised version.

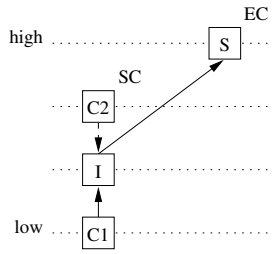

Figure 5. Reverse traversal problem

Figure 5 illustrates a scenario that makes reverse traversal of donation chains impossible. A low-priority client thread C_1 sends a request to an intermediate server thread I. In order to provide the service, I has to contact a high-priority server S. However, before I can contact S, another client thread C_2 preempts I, attempts to send a request to I as well and blocks because I is not receiving. After S has received the request from I it is preempted by an asynchronous thread wakeup and the scheduler now has to resume the donation. It finds the high-priority thread S in the

ready queue first and selects it as execution context because S is ready. In order to determine the scheduling context that S had been preempted on (C_2), the scheduler traverses the donation chain backwards and finds I. The problem is now that thread I has received capacity-reserve donations from multiple threads. Although a forward traversal always produces a donation chain, a backward traversal can produce a complex donation tree.

A different problem can occur when the scheduler finds a blocked thread in the ready queue and performs a forward traversal of the donation chain to determine the current execution context. If the thread at the end of the donation chain is not ready but blocked, it cannot be dispatched. Such "dead ends" can be caused by threads that have no dependency on another thread, for example threads in the following states:

- Sleeping (receiving from a special id with a timeout)
- Receiving from any other thread
- Receiving from a hardware-interrupt source

Note that a send operation can never cause a dead end, because it always produces a dependency on another thread, namely on the send target. To handle dead donation chains the scheduler can either skip over them or remove them from the ready queue. The first option heavily degrades the performance of dispatch decisions, because potentially the scheduler has to traverse several donation chains and skip over many threads. The second option requires that the kernel reenqueues the entire donation tree in an atomic fashion when the dead end becomes ready, which is difficult to implement.

Another problem is that donation-chain traversal makes preemption decisions expensive. When an asynchronous thread wakeup occurs, the kernel should be able to quickly decide if the awakened thread should preempt the current thread or not. In a mixed-donation scenario as shown in Figure 4 a thread with a priority between that of I and S could be woken up and should not preempt S, because I could always donate its higher-priority scheduling context to S instead of passing along the scheduling context of C. Dispatch decisions require knowledge of the maximum priority of all threads along the donation path. We discuss solutions to these problems in the following subsection.

2.4. Implementation

The problems of the original algorithm are caused by missing or lost information about the donated scheduling context. At the time of the preemption the kernel knew which execution context and scheduling context were active, but did not save the information. If we take one step back and relax our requirement of an unmodified message-passing path, we can come up with a less problematic algorithm that is still fast. Recall that we have to traverse the donation chain to find the current scheduling context and the current execution context, and to compute the maximum priority of all threads along the chain during preemption decisions. A closer look at the donation graphs reveals that

- each execution context to which capacity reserves have been donated forms the root of a donation tree
- the leaves of the donation tree are the scheduling contexts that have been donated to the execution context at the root of the tree
- there is a unique path from each leaf node to the root node
- the leaf node provides the time quantum and the highest-priority thread along the path provides the priority for the execution context at the root node
- at any point in time there is exactly one path that provides time quantum and priority to the root node because it has the highest priority of all paths

For the improved algorithm we add the following additional attributes to each node of a donation tree to locally cache information in order to reduce the overhead for recomputation of these attributes by traversing donation chains.

- a reference to the last scheduling context that has been donated to this node (donated scheduling context)
- the maximum priority along the donation chain from the origin of the donated scheduling context to this node (path priority)
- a reference to the node that donated the scheduling context to this node (donating thread)

Initially the donated scheduling context of each thread points to the thread's own scheduling context, the path priority is the priority of that scheduling context and the donating thread pointer is a null pointer.

When a new dependency is added or removed between a donator and a donatee, the kernel traverses the donation chain from the donatee towards the root node until it reaches the root node or encounters a node with a higher path priority. For each traversed node, the kernel updates the donation attributes if necessary.

When adding a dependency, the kernel

- updates the donating-thread pointer of all traversed nodes to point to the node's donator, thereby rewriting the donation path

- sets the path priority of all traversed nodes to the maximum of the donator's path priority and the donatee's own priority, thereby caching the ceiling priority of the path
- sets the donated scheduling context of all traversed nodes to the donated (current) scheduling context, thereby caching the scheduling context that has been most recently donated

When removing a dependency, the kernel checks the message-passing partner list of the donatee to find the highest-priority thread that still has a dependency on the donatee and, if such a thread exists, uses that thread's donation attributes to update the traversed nodes. Otherwise it uses the donatee's own donation attributes and sets the donating-thread field of the donatee to a null pointer.

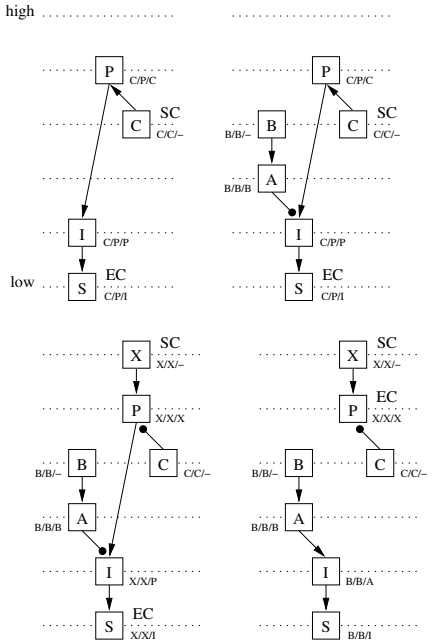

Figure 6. Example dependency tree (donated scheduling context/path priority/donating thread)

Figure 6 shows four example donation trees and the additional attributes added to each node. In the first example the client thread C donated its scheduling context to S via P and I. For clarity all threads are shown with their original priority and not with their path priority. When the scheduler is invoked it traverses the ready queue. The scheduler removes all blocked threads it comes across (P and I) from the queue until it finds a ready thread (S). It then selects that thread as current execution context and the thread's last donated scheduling context as current scheduling context. The second example illustrates that, if S blocks, a scenario can occur where a donation chain joins an already existing donation tree. As soon as A donates to I, the kernel notices that A's path priority is lower than that of I and therefore no update is required between the donatee (I) and the root node (S). In the third example a high-priority thread X joins an already existing donation tree for S. As soon as X donates its scheduling context to P, the higher path priority of X triggers an update of the donation chain from the donatee (P) to the root node (S).

When a donation dependency is removed, for example the dependency between P and I, as illustrated in the fourth example, the kernel checks the partner list of the donatee (I) and finds A as highest-priority donator. It then sets I's donating thread pointer to A and the path priority and donated scheduling context of I and S to B.

3. Resource-Access Models

In microkernel-based systems the kernel usually does not provide any high-level resources at all. Objects like files, devices or sockets are provided by user-level servers. To invoke an operation on such an object a client just sends a request to the server providing the object, using the message-passing facilities. From a timing perspective this invocation looks like a local function call with a slightly higher overhead if the thread donates its capacity reserve.

If resources are shared, we are faced with priority-inversion issues and have to take this into account while designing the system. We can solve the priority-inversion problem by either designing the system for priority inheritance or for stack-based priority ceiling.

3.1. Design based on Priority Inheritance

To construct a system using priority inheritance we have to assign priorities such that threads always request services from threads with a lower priority. Capacity-reserve donation then always results in priority inheritance, which solves the priority-inversion problem. In contrast to normal priority inheritance, where the holder of the resource inherits the priority of the thread requesting the resource, in our case the thread representing the resource inherits the priority of the requesting thread, which results in the expected behaviour. After sending the reply to the resource holder the donating thread may acquire the resource.

If we cannot directly represent the resource as a thread but instead have to protect a more general critical section, we can implement a binary semaphore with priority inheritance. The thread providing the semaphore either waits for incoming requests if the semaphore is free or waits for a release reply from the semaphore holder, thereby creating

a dependency on it. Other incoming requests block on the semaphore thread donating their reservations along the dependency chain to the semaphore holder.

3.2. Design based on Stack-Based Priority Ceiling

To implement a system with stack-based priority ceiling we associate each resource with a high-priority thread and construct the system such that requests to claim a resource are always made from a lower-priority thread to the higher-priority thread representing the resource. While a thread uses the resource, it effectively provides its own time but runs with ceiling priority of the resource thread.

3.3. Priority-Ceiling Protocol

We are not able to implement the priority-ceiling protocol using the proposed mechanism. The priority-ceiling protocol consists of scheduling rules, allocation rules and priority inheritance rules. We are able to implement the scheduling rules. We could implement the allocation rules by using a central server which keeps track of priorities of threads, priority ceilings of all resources and the current ceiling of the system. The server would allocate resources to threads according to the allocation rules. But the allocation rules lead to a situation called avoidance blocking where a higher-priority thread blocks while requesting a free resource, because a thread has raised the system ceiling by allocating another resource. Both threads are independent from each other and our donation mechanism uses dependencies between threads to achieve priority inheritance. Therefore we are not able to implement the priority inheritance rules of the priority-ceiling protocol.

3.4. Resource Access and Capacity Reserves

In reservation-based systems with shared objects and mutual exclusion or client-server communication a scenario may occur where a capacity reserve is exhausted while a thread is inside a critical region or a server fulfils the request of a client while running on a donated capacity reserve. Other threads trying to enter the critical region or trying to send a request to the server could be blocked until the capacity reserve is replenished even if they have some budget left.

[3] suggests to allow the thread to overrun its budget to be able to leave the critical region. Since in our system the kernel does not know anything about critical regions we can not allow a thread to overrun its capacity reserve.

[7] extensively discusses situations in which tasks under budget control interfere with each other and derives formulas to calculate interference times for the bandwidth inheritance protocol. Credo is a similar approach. If a thread blocks on another thread it donates its capacity reserve to help the other thread out of its critical region and reobtains it when the other thread leaves that critical region. In a normal priority-based system threads always block on a thread with lower priority. With reservations threads may also block on threads with higher priority and an expired capacity reserve. We have to adapt our donation algorithm to handle this situation. While parsing the dependency chain we may encounter a node with a higher path priority but an expired capacity reserve. In this case the algorithm continues the dependency traversal.

In this paper our main focus was the donation mechanism. The reservation framework provided by the kernel [8] supports the following alternatives:

- *Adaption:* Threads are expected to voluntarily release their capacity reserves before they expire. If a capacity reserve expires, a notification is sent to an exception handler which may try to handle this situation by assigning additional time to the thread to help it out of its critical region and which may adjust the reservation of the thread to minimize the number of such events in the future.

- *Additional Reservation per Thread:* We may assign threads an additional reservation which they can use if their main reservation expires inside a critical region. But this leads to lower CPU utilisation since this reservation is seldomly used.

- *Reservation per Resource:* If we access a resource via stack-based priority ceiling we may give the thread representing the resource its own reservation. If the capacity reserve of a thread expires while using the resource the scheduler switches to the reservation of the thread representing the resource and the thread continues.

- *Reservation per Server:* In a client server scenario the server may have to reserve its own time when doing larger amounts of work for its clients.

4. Evaluation

A detailed analysis of L4 IPC costs has already been presented in [14]. [16] looked at the real-time behaviour of L4's message-passing system and [9] analysed the performance of a complete system running time-sharing applications, whereas [8] looked at a system running soft- and non-real-time applications in parallel.

In this paper we focused on the integration of capacity-reserve donation into a synchronous message-passing facility with preferably zero overhead added to the critical path. We developed a solution which adds a neglectible overhead

to each message-passing operation. Figure 7 shows an excerpt of the donation code in the IPC path. The function calls are generated inline and mostly only return or set a value. Context::donate_time translates to 16 instructions and typically only half of them is executed. Since this is difficult to measure with about 1380 cycles for an IPC on a 500MHz PIII we restrict ourself to a qualitative analysis. For the uncontended case the overhead comprises:

- Check whether we establish a dependency or release one
- Setting the thread's most recent donation partner
- Setting the thread's most recently donated scheduling context
- Computing the maximum of the donatee's and the donator's priority

These parameters are contained in the thread-control blocks (TCBs) of the donator and the donatee. Access to both TCBs can lead to two TLB misses, but introduces no additional overhead, because we have to touch both TCBs during message passing anyway. Referencing the additional attributes for donation tracking can cause an additional cache miss if these attributes are not properly aligned with other data that is accessed during message passing.

For the contended case we have to update the donation attributes for all other threads along the donation chain, so the total overhead depends on the number of threads involved in the donation scenario – the nesting depth of the message-passing system. For each of these threads we always incur an additional TLB miss and a cache miss because their TCBs are not used during the actual message-passing.

When the kernel preempts a thread with a path priority that differs from the thread's original priority, the thread must be enqueued in the ready queue according to the path priority and reenqueued according to its own priority when the donation ends. We expect the preemption of a message-passing operation to occur rather infrequently, which is why we update the ready queue in a lazy fashion.

5. Conclusion

We introduced Credo, a capacity-reserve donation scheme for fast component interaction in microkernel-based dependable open real-time systems.

Credo, a donation-based mechanism, resolves priority-inversion and accounting problems in resource-sharing scenarios. The mechanism facilitates the construction of a variety of systems, among them systems based on priority inheritance and stack-based priority ceiling, so that existing mathematical models can be reused.

```
PUBLIC void
Context::donate_time (Context * d)
{
  Sched_context * s = d->sched();
  // someone donates time to me
  _path_prio = s->prio();
  set_sched_donatee(d);
  if (running_on_donated_time())
    {
      // revert potential priority bump
      sched()->set_prio(_path_prio);
      // replace donation
      set_sched(s);
      if (s->prio() < saved_sched()->prio())
        {
          // bump priority
          s->set_prio(saved_sched()->prio());
        }
    }
  else
    {
      if (s->prio() < sched()->prio())
        {
          s->set_prio(sched()->prio());
        }
      set_saved_sched(sched());
      set_sched(s);
    }
}
PUBLIC void
Context::add_dependency (Context* partner)
{
  if (!running_on_donated_time() ||
       partner != sched_donatee()) {
    if (sched()->prio() <=
                    partner->sched()->prio()) {
      if (!partner->running_on_donated_time()) {
        // ceiling scenario
        partner->donate_time(this);
      }
    }
    else {
      // priority inheritance scenario
      partner->donate_time(this);
    }
  }
}
```

Figure 7. Dependency code in IPC path

The design and implementation of this mechanism focused on preserving the high performance of the inter-component communication mechanism provided by the underlying microkernel (here L4). Decomposing a thread into execution context and scheduling context allowed us to integrate capacity-reserve donation into L4's IPC mechanism without noticeable performance overhead. Credo improves the applicability of microkernels for general-purpose as well as specialized systems such as dependable open real-time systems.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium*, Madrid, Spain, December 1998. IEEE.

[2] G. Banga and P. Druschel. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999. USENIX.

[3] M. Caccamo, G. Buttazzo, and L. Sha. Aperiodic servers with resource constraints. In *Real-Time Systems Symposium*, London, UK, December 2001. IEEE.

[4] Z. Deng and J. Liu. Scheduling real-time applications in open environment. In *Real-Time Systems Symposium*, San Francisco, December 1997. IEEE.

[5] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[6] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 267–282, June 1999.

[7] L. A. Guiseppe Lipari, Gerardo Lamastra. Task synchronization in reservation-based real-time systems. *IEEE Transactions on Computers*, 53(12):1591–1601, 2004.

[8] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig. Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, Dec. 2001.

[9] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, Oct. 1997.

[10] T. Jaeger, J. E. Tidswell, A. Gefflaut, Y. Park, K. J. Elphinstone, and J. Liedtke. Synchronous ipc over transparent monitors. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 189–194. ACM Press, 2000.

[11] M. Jones, J. Alessandro, F. Paul, J. Leach, D. RoOu, and M. RoOu. An overview of the rialto realtime architecture. In *Proceedings of the 7th ACM SIGOPS European Workshop*, pages 249–256, Connemara, Ireland, September 1996.

[12] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, Dec. 1993.

[13] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, Dec. 1995.

[14] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance (still the foundation for extensibility). In *6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Chatham (Cape Cod), MA, May 1997.

[15] G. Lipari and G. Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of Systems Architecture*, (46:327-338), 2000.

[16] F. Mehnert, M. Hohmuth, S. Schönberg, and H. Härtig. RTLinux with address spaces. In *Proceedings of the Third Real-Time Linux Workshop*, Milano, Italy, Nov. 2001.

[17] C. Mercer, S. Savage, and H. Tokunda. Processor capacity reserves for multimedia operating systems. In *International conference on Multimedia Computing and System*. IEEE, May 1994.

[18] D. Niz, L. Abeni, S. Saewong, and R. Rajkumar. Resource Sharing in Reservation-Based Systems. In *Real-Time Systems Symposium*, London, UK, December 2001. IEEE.

[19] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Fourth IEEE Real-time Technology and Applications Symposium (RTAS)*, Denver, Colorado, June 1998.

[20] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels. A resource-centric approach to real-time and multimedia systems. In *4th Real-Time Computing Systems and Application Workshop*. IEEE, November 1997.

[21] J. Regehr. *HLS: A Framework for Composing Soft Real-Time Schedulers*. PhD thesis, University of Utah, 2001.