Timeslice Donation in Component-Based Systems

Udo Steinberg
Technische Universität Dresden
udo@hypervisor.org

Alexander Böttcher
Technische Universität Dresden
boettcher@tudos.org

Bernhard Kauer
Technische Universität Dresden
bk@vmmon.org

***Abstract*—An operating system that uses a priority-based scheduling algorithm must deal with the priority inversion problem, which may manifest itself when different components access shared resources. One solution that avoids priority inversion is to inherit the priority across component interactions. In this paper we present our implementation of a timeslice donation mechanism that implements priority and bandwidth inheritance in the NOVA microhypervisor. We describe an algorithm for tracking dependencies between threads with minimal runtime overhead. Our algorithm does not limit the preemptibility of the kernel, supports blocked resource holders, and facilitates the abortion of inheritance relationships from remote processors.**

I. Introduction

Priority inversion [1] occurs when a high-priority thread *H* is blocked by a lower-priority thread *L* holding a shared resource *R* as illustrated in Figure 1. Priority inversion can be unbounded if a medium-priority thread *M* prevents the low-priority thread from running and thus from releasing the resource. A lock that protects a critical section from concurrent access is a typical example for a shared resource that can cause priority inversion. In component-based systems the shared resource may also be a server thread that is contacted by multiple clients.

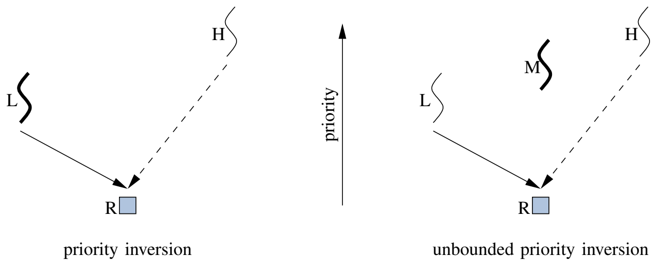

Figure 1. Example of priority inversion: The currently active thread is marked bold.

Resource Access Protocols

Several solutions for circumventing the priority inversion problem have been proposed. They range from disabling preemption to using complex protocols to control resource access. Disabling preemption while holding a shared resource is prohibitive in systems with real-time or low-latency requirements. Protocols such as the priority ceiling protocol (PCP) and the priority inheritance protocol (PIP) [2] avoid priority inversion by defining rules for resource allocation and priority adjustment that guarantee forward progress for threads holding shared resources.

The priority ceiling protocol prevents deadlocks that arise from contention on shared resources. However, PCP requires a priori knowledge about all threads in the system. At construction time every shared resource is assigned a static ceiling priority, which is computed as the maximum of the priorities of all threads that will ever acquire the resource. Priority ceiling is therefore unsuitable for open systems [3] where threads are created and destroyed dynamically or where the resource access pattern of threads is not known in advance. Because priority ceiling relies on static priorities it is not applicable to scheduling algorithms with dynamic priorities, such as earliest deadline first (EDF) [4].

When using the priority inheritance protocol, the priority of a thread that holds a shared resource is temporarily boosted to the maximum of the priorities of all threads that are currently trying to acquire the resource. Priority inheritance works in systems with dynamic priorities and does not require any prior knowledge about the interaction between threads and resources.

Bandwidth inheritance (BWI) [5] can be considered an extension of the priority inheritance protocol to resource reservations. Instead of inheriting just the priority, the holder of a shared resource inherits the entire reservation of each thread that attempts to acquire the resource. Bandwidth inheritance reduces the blocking time for other threads when the resource holder's own reservation is depleted.

Resource reservations in our system are called timeslices and consist of a time quantum coupled with a priority. Timeslices with a higher priority have precedence over those with a lower priority. The time quantum facilitates round-robin scheduling among timeslices with the same priority.

In this paper, we describe and evaluate the timeslice donation mechanism of the NOVA microhypervisor [6]. This mechanism allows for an efficient implementation of priority and bandwidth inheritance in an open system with many threads. We discuss issues that arise when threads block or unblock while holding shared resources and explore how blocking dependencies can be tracked with minimal overhead.

II. Background

Component-based operating systems achieve additional fault isolation by running device drivers and system services in different address spaces. Communication between these components must use inter-process communication (IPC) instead of direct function calls in order to cross address-space boundaries. When multiple clients contact the same server, threads in the server are a shared resource and therefore prone to cause priority inversion.

Lazy scheduling was originally introduced as a performance optimization in the L4 microkernel family to bypass the scheduler during inter-process communication [7]. Figure 2 illustrates the communication between a client and a server thread. Because threads and timeslices are separate kernel objects, the kernel can switch them independently. During IPC, the kernel changes the current thread from the client C to the server S and back, without changing the current timeslice. The effect is that the client donates its timeslice to the server.

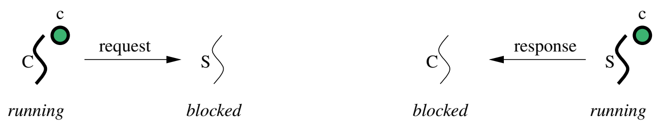

Figure 2. Synchronous communication between a client and a server in a component-based system. Left side: During the request the client thread C donates its timeslice c to the server thread S. Right side: When the server responds, the kernel returns the previously donated timeslice c back to the client.

Timeslice donation can be used to implement priority inheritance, but only if the kernel correctly resumes the donation after a preemption. For this purpose the kernel must track dependencies between threads so that it can determine the thread to which a timeslice has been most recently donated. Most versions of L4 do not implement dependency tracking. Therefore, priority inversion may occur when a server thread is preempted and afterwards uses its own, potentially low-priority, timeslice. This problem is described in more detail in [8].

Timeslice Donation and Helping

The NOVA microhypervisor implements priority and bandwidth inheritance using the following two closely related mechanisms:

Donation: In the left example of Figure 3, a high-priority client thread C sends an IPC to a low-priority server thread S. By donating the client's timeslice c to the server, the priority of S is boosted to that of C and the kernel can directly switch from the client to the server without having to check for the existence of ready threads with priorities between the client and the server, such as the medium-priority thread T. Without timeslice donation, S would use its own low-priority timeslice s and T would be able to preempt S, thereby causing priority inversion for C. During IPC, the kernel establishes an explicit donation dependency from C to S, which we denote by a solid arrow. When the scheduler selects the client's timeslice c, it follows the donation dependency and activates S instead of C, thereby resuming the donation.

Helping: Donation boosts the priority of a server to that of its current client and ensures that, for as long as the server works on behalf of the client, it can only be preempted by threads with a higher priority than the client. Helping augments donation by boosting the priority of the server even further when higher-priority clients try to rendezvous with the server while it is busy handling a request. In the right example of Figure 3, the server thread S is handling the request of a client thread C and initially uses the client's timeslice c. Another thread H with a higher priority than C can preempt the server and attempt to rendezvous with S. Because the rendezvous fails, H switches directly to S in order to help S finish its current request, thereby elevating the priority of S to that of H. Unlike donation, the kernel does not establish an explicit dependency from H to S. Upon selecting the timeslice h, the scheduler activates H, which simply retries its operation. We denote such an implicit helping dependency by a dashed arrow.

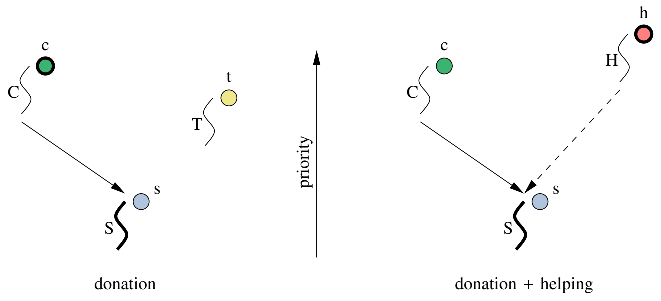

Figure 3. Example of timeslice donation and helping during client-server communication. The currently active thread and timeslice are marked bold.

Threads in a realtime system typically obtain only a limited time quantum in each period of execution. If a server exhausts the time quantum of its current client during the handling of a request, the server becomes stuck until the client's time quantum has been replenished. In such cases other clients cannot rendezvous with the server and therefore make use of the bandwidth inheritance property of the helping mechanism to allow the server to run the request to completion.

Similar issues arise when a client aborts its request before the server can reply, when the client is deleted, or when the communication channel between the client and the server is destroyed. Such cases leave the server in an inconsistent state that is similar to the state when the server is preempted, except that the old client will no longer provide the time quantum for the server to complete the request. Instead, subsequent clients use the helping mechanism to bring the

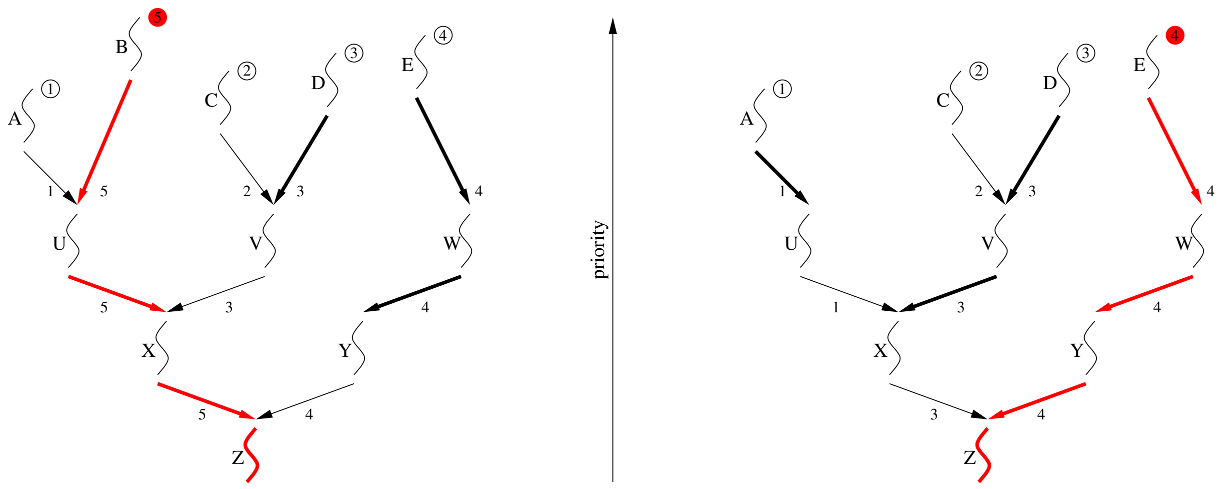

Figure 4. Dependency tracking: The highest-priority incoming edge of each node and the currently active thread and timeslice are marked bold. Changes to nodes in the priority inheritance tree may require updates along the path from the changed node to the root node. In this example the incoming edges of U, X, and Z must be updated when B leaves the priority inheritance tree.

server back into a consistent state where it can accept the next request. Because synchronous communication between threads on the same CPU always uses timeslice donation and helping, server threads that can only be contacted on their local CPU do not need a timeslice of their own.

The donation and helping mechanisms are transitive. If a server needs to contact another server to handle a client request, it further donates the current timeslice to the other server for the duration of the nested request. Therefore, the kernel must be able to handle large dependency tracking trees.

Multiprocessor Considerations

Helping and donation cannot be easily extended to multiprocessor systems and we are currently aware of only one proposal that describes a multiprocessor priority inheritance protocol [9].

One observation is that priorities of threads on different CPUs are not directly comparable. Additionally, the result of any comparison would quickly become outdated when other processors reschedule. Another observation is that a client cannot donate time from its CPU to help a server on another CPU. Such an operation would cause time to disappear on one processor and to reappear on another. Donating additional time to an already fully loaded CPU causes overload and can potentially break real-time guarantees.

The overload situation can be avoided if the client pulls a preempted server thread over to its CPU to help it locally. However, such an approach requires the address space of the server to be visible and identically configured on all processors on which clients for this server exist. In cases where client threads from different CPUs attempt to help the same server thread simultaneously, the kernel would need to employ a complex arbitration protocol among all helping client threads to ensure that each server thread executes on one processor only at a time. Furthermore, migrating the working set of the server thread to the CPU of the client and then back to the original CPU can result in a significant amount of coherence traffic on the interconnect.

Due to these drawbacks our algorithm does not include cross-processor helping. However, it supports IPC aborts from remote CPUs.

III. Related Work

In our previous work on capacity-reserve donation [10], we described an algorithm for computing the effective priority of a server as the maximum of the effective priorities of its current and all pending clients. The algorithm performs the tracking of dependencies and priorities by storing priority information inside the nodes and along the edges of a priority inheritance tree. For each node in the tree, the outgoing edge is marked with the maximum of the priority along all incoming edges of that node as shown in Figure 4.

Unfortunately, changes to nodes of the inheritance tree may require numerous updates to the edges of the tree as shown on the right side. When thread B leaves the priority inheritance tree (because it experiences an IPC timeout or is deleted), the kernel must recompute the priorities along the edges from the changed node down to the root node. In this example, the kernel must update the incoming edges of threads U, X, and Z to determine that the timeslice of thread E has become the highest-priority timeslice donated to Z. Depending on the nesting level of IPC, the number of updates to the priority inheritance tree can become very large, resulting in long-running kernel operations that must be executed atomically. Protecting the whole tree with a global lock for the duration of the update is undesirable because it disables preemption and limits the scalability of the algorithm in multiprocessor environments.

A more efficient version of the bandwidth inheritance protocol [11] has been implemented in the Linux kernel. It also uses a tree structure to track the dependencies between tasks and resources. When a task blocks on a shared resource, all tasks that previously inherited their bandwidth to that task must be updated to inherit their bandwidth to the holder of the shared resource instead.

OKL4 is a commercially deployed L4 microkernel, which is derived from L4Ka::Pistachio. OKL4 tracks IPC dependencies across preemptions and implements a priority inheritance algorithm. The kernel grabs a spinlock during updates to the inheritance tree in order to guarantee atomic updates.

With the realtime patch [12] series, support for priority inheritance was introduced to the Linux kernel. Because the realtime patch made the kernel more preemptible, the need arose to avoid unbounded priority inversion when threads are preempted while holding kernel locks [13]. Further research based on the Linux realtime patches, especially in the context of priority inheritance, is conducted by the KUSP [14] group. Their research focus is on supporting arbitrary scheduling semantics using group scheduling [15] in combination with priority inheritance.

IV. Implementation

Dependency tracking algorithms that store priority information along the edges of the priority inheritance tree share the problem that updates to a node in the tree require a branch of the tree to be updated atomically. For example, when a client with a high-priority timeslice joins or leaves an existing priority inheritance tree, it must rewrite the priority information along the edges from the client to the server at the root of the tree as shown in Figure 4. The update of the tree cannot be preempted because the scheduler must not see the tree in an inconsistent state. Therefore, the duration of the update process defines the preemptibility of the kernel. In an open system, a malicious user can create as many threads as his resources permit, arrange them in a long donation or helping chain and then cause an update in the priority inheritance tree that will disable preemption in the kernel for an extended period of time. Therefore, we devised a new dependency tracking algorithm that does not affect the kernel's preemptibility and at the same time keeps the dependency tracking overhead low. Before we describe this algorithm in detail, we present our requirements.

Requirements

To prevent malicious threads from being able to cause long scheduling delays in the kernel, we require updates in the priority inheritance tree to be preemptible. Furthermore, we demand that each operation is accounted to the thread that triggered it. Our goal is to move all time-consuming operations from the performance-critical paths in the kernel into functions that are called infrequently. For example, we strive to move as much dependency tracking as possible out of the IPC path into the scheduler and into functions that handle deletion of threads and communication aborts. The new dependency tracking algorithm works for an arbitrary number of threads and is not limited to small-scale systems or systems where all communication patterns must be known in advance.

Improved Algorithm for Dependency Tracking

Our new algorithm is based on the idea of storing no priority information whatsoever in nodes of the tree, which obviates the need for updating the priorities when threads join or leave the priority inheritance tree. Furthermore, priority information in the tree cannot become stale. However, this approach requires the kernel to restore the missing information during scheduling decisions, which works as follows:

When invoked, the scheduler selects the highest-priority timeslice from the runqueue and then follows the donation links to determine the path that the timeslice has taken prior to a previous preemption. When the scheduler finds a thread that has no outgoing donation link, it switches to that thread. In the left example of Figure 4, the scheduler selects the timeslice with priority 5, which belongs to thread B, and then follows the donation links from B via U and X to Z. Because Z has no outgoing edge, Z is dispatched. When thread B leaves the tree as shown in the right example of Figure 4, the scheduler selects the timeslice with priority 4, which belongs to thread E and then follows the donation links from E via W and Y to the server Z.

Traversing the donation links from a client's timeslice to the server at the root of the priority inheritance tree is a preemptible operation. If a higher-priority timeslice is added to the runqueue while the scheduler is traversing the tree, the kernel restarts the traversal, beginning with the higher-priority timeslice instead. The benefit of this algorithm is that whenever nodes in the inheritance tree are added or removed, no priority information must be updated. Algorithms that store priorities in all nodes of the tree can quickly determine the highest-priority timeslice donated to a thread by checking the highest-priority incoming edge of that thread. In contrast, our algorithm must compute this information by traversing the priority inheritance tree after a preemption. We quantify the cost for this operation in Section V.

Blocking

An interesting scenario occurs when the server thread at the root of a priority inheritance tree blocks. This can happen when the server waits for an interrupt that signals completion of I/O or when it waits for the reply from a cross-processor request for which timeslice donation cannot be used.

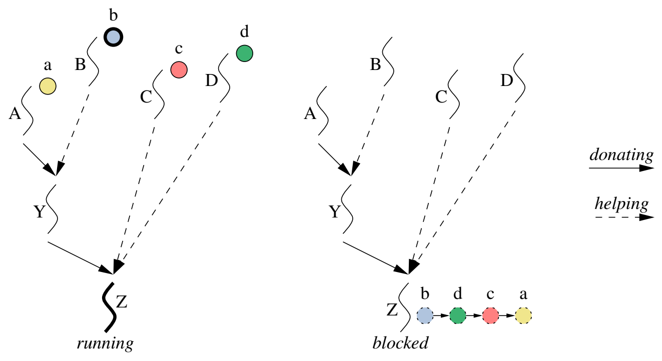

Figure 5. Blocking of threads when the holder of a shared resource is blocked.

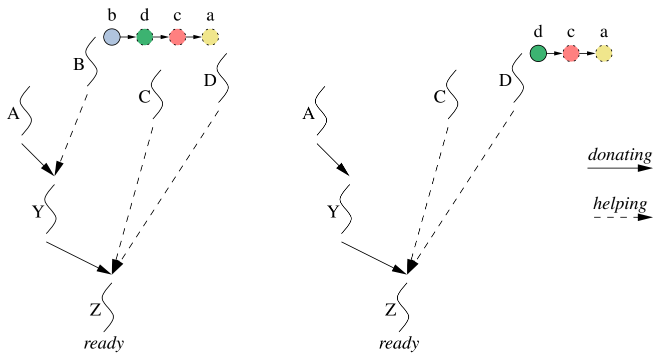

Figure 6. Staggered wakeup of threads when the holder of a shared resource unblocks.

In the left example of Figure 5, a server thread Z blocks while using timeslice *b*. In that case the kernel removes *b* from the runqueue and enqueues it in a priority-sorted queue of timeslices that are blocked on Z. During the subsequent reschedule operation, the scheduler selects timeslice *d* and traverses the priority inheritance tree down to *Z*. When it finds that *Z* is still blocked, *d* is also added to the queue of blocked timeslices. The right side of Figure 5 illustrates that all other timeslices that have been donated to *Z* are gradually removed from the runqueue and become blocked on *Z* when they are selected by the scheduler.

Staggered Wakeup

When *Z* eventually becomes unblocked, all timeslices that have previously been blocked on *Z* must be added back to the runqueue, effectively reversing the operation of blocking from Figure 5. Because an arbitrary number of timeslices can potentially be blocked at the root of a priority inheritance tree, releasing all of them at once contradicts our requirement of avoiding long scheduling delays. Based on the observation that only the highest-priority timeslice from the blocked queue will actually be selected by the scheduler, releasing the other timeslices can be deferred. The left side of Figure 6 illustrates that, when *Z* becomes unblocked, the kernel adds *b*, the highest-priority timeslice blocked on *Z*, back to the runqueue. The other timeslices that were blocked on *Z* remain linked to *b* and are not added to the runqueue yet. When *b* lowers its priority or is removed from the runqueue, the kernel adds *d*, the first timeslice linked to *b*, back to the runqueue and leaves the remaining timeslices linked to *d* as shown in the right of Figure 6. The benefit of this approach is that when Z unblocks, only a single timeslice needs to be added to the runqueue. The other blocked timeslices will be released in a staggered fashion.

Direct Switching

Recall from Figure 2 that the kernel implements timeslice donation by directly switching from one thread to another while leaving the current timeslice unchanged. When a server responds to its client, the kernel must check whether it can undo the timeslice donation by directly switching back to the client. Switching back to the client is wrong in cases where the server is currently using the timeslice of a high-priority helper and the client and the helper do not share the same incoming edge in the priority inheritance tree of the server. For example, when *Z* responds to *Y* in the left example of Figure 5, the kernel can only switch from *Z* to *Y* if *Z* is running on timeslice *a* or *b*. If *Z* is running on timeslice *c* or *d*, the kernel cannot return the timeslice to *Y*, because the timeslice was not donated to *Z* via *Y*. The kernel must instead switch to thread *C* or *D* so that they can retry their rendezvous with *Z*. Because our algorithm does not store any information along the edges of the priority inheritance tree, the kernel uses the following trick: When the scheduler selects a new timeslice and starts traversing the tree, the kernel counts the number of consecutive donation links along the path in a CPU-local *donation counter*. At the beginning of a new traversal and every time the kernel encounters a helping link, the donation counter is reset to zero. The donation counter indicates how often the kernel can directly switch from a server back to its client. When *Z* replies to *Y* in the left example of Figure 5, the current timeslice is *b* and the donation counter is 1, indicating that the kernel can directly switch from *Z* to *Y*, but not from *Y* to *A*. When a client donates the current timeslice to a server, the kernel increments the donation counter. When a server responds to its client, the kernel decrements the donation counter. The update of the donation counter is the *only* overhead added to the performance-critical IPC path by our dependency tracking algorithm.

Livelock Detection

Communication in component-based systems can lead to deadlock when multiple threads contact each other in a circular manner. In Figure 7, a client thread *C* contacts a server *Y*, which in turn contacts another server *Z*. Deadlock occurs when *Z* tries to contact *Y*. In our implementation, *Y* and *Z* would permanently try to help each other, thereby turning the deadlock into a livelock.

In NOVA, the kernel can easily detect such livelocks during the traversal of the priority inheritance tree by counting the number of consecutive helping links in a *helping counter*. When the value of the helping counter exceeds the number of threads in the system, the kernel can conclude that the current timeslice is involved in a livelock scenario. It can then remove the timeslice from the runqueue and print a diagnostic message.

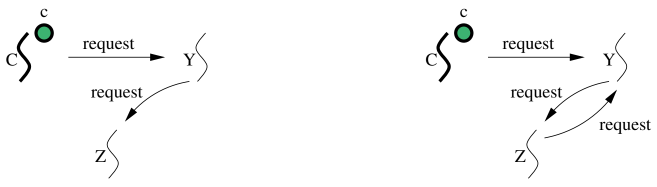

Figure 7. Development of a Livelock

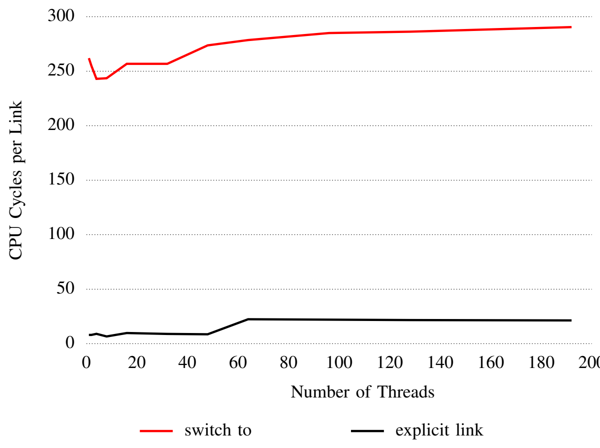

Figure 8. Overhead for traversing a helping link: The average cost is independent of the number of threads along the path. The graph compares an implementation in which the kernel simply switches to the destination thread with an implementation that reduces the overhead by tracking helping links explicitly.

V. Evaluation

We evaluated the performance of our priority-inheritance implementation using several microbenchmarks, which we conducted on an Intel Core2 Duo CPU with 2.67 GHz clock frequency.

In contrast to dependency tracking algorithms that store priority information in each node of the inheritance tree, our algorithm keeps the priority information only in the timeslices bound to the client threads that form the leaves of the inheritance tree. Whenever the current timeslice changes, the scheduler must follow the dependency chain to find the server thread at the root of the inheritance tree to which the timeslice has been donated. Fortunately such tree traversals are neither very frequent nor very expensive.

Frequency of Dependency Tracking

The kernel invokes the scheduler to select a new current timeslice when the current thread suspends itself and thereby removes the current timeslice from the runqueue. The scheduler is also invoked when the kernel releases a previously blocked thread and that thread adds a timeslice with a higher priority than the current timeslice to the runqueue. It should be noted that the scheduler need not be invoked during client-server communication (see Figure 2) because the current timeslice remains the same and the runqueue need not be updated.

The frequency of scheduler invocations depends on the number of preemptions in the system, which in turn depends on the length of timeslices and the frequency of higher-priority threads being released.

Cost of Dependency Tracking

The costs of each traversal depends on the depth of the priority inheritance tree and on the type of dependency encountered during the traversal. Donation dependencies are explicitly tracked by the kernel, which stores the IPC partner in the thread control block. Therefore, following a donation dependency is a pointer chasing operation, which can lead to cache and TLB misses. In NOVA, all timeslices and threads are allocated from slab allocators and thus likely to be in close proximity. Furthermore, the kernel uses superpages for its memory region to reduce the number of TLB misses. The traversal of a donation dependency typically only causes a cache miss.

A helping dependency indicates that a client thread did not manage to rendezvous with the server because the server was busy. In that case the client thread retries its rendezvous and thereby switches to the server thread. The thread switch is all that is required to traverse a helping dependency. The cost for the switch typically includes the cost for switching address spaces unless both threads happen to be in the same address space. The overhead can be reduced by tracking helping dependencies explicitly. There is a tradeoff between faster traversal of dependencies and having to store more information in the priority inheritance tree. We implemented and measured both variants. The CPU cycles required to traverse the priority inheritance tree are accounted to the newly selected current timeslice where the traversal started. Determining the thread at the root of the tree that will use the timeslice is part of the actual helping process.

Figure 8 shows that simply switching to the thread in order to help is much more expensive than following an explicit link. Furthermore, the costs of traversing a single helping link is nearly constant, irrespective of chain length. The step in the lower curve and the slight increase of 10–30 cycles in the upper curve can be attributed to additional cache usage when touching more threads.

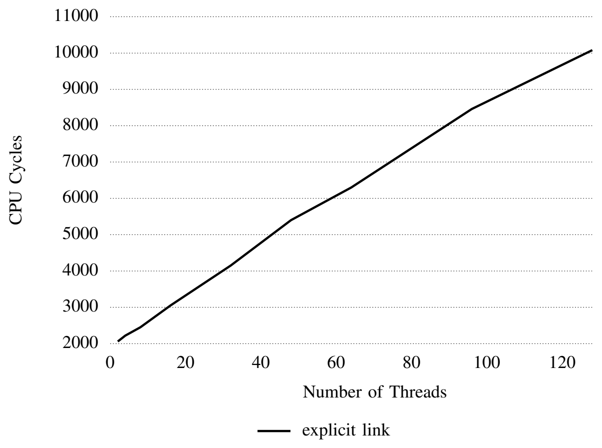

Figure 9. Cost for canceling an IPC: The cost of aborting an IPC operation scales linearly with the length of the path to the root node.

Cost of Modifying the Inheritance Tree

Updates to the inheritance tree are required when the link between two threads in the tree is broken. Possible reasons include thread deletion, abortion of an ongoing IPC between two threads, or revocation of the communication channel between the client and its server.

Breaking a link in the inheritance tree requires the deletion of the IPC connection between the affected threads and a traversal of the inheritance tree down to the leaf to check for blocked timeslices. If blocked timeslices are found, the kernel performs a staggered wakeup for them.

To avoid taking any locks in the IPC path, the IPC connection is deleted on the CPU on which the client, server, and the priority inheritance tree are located. In case the deletion was initiated by a thread on a different processor, the remote CPU must send an inter-processor interrupt to break the link. However, the costly part of the tree traversal down to the root is performed by the initiating thread on the remote CPU.

Figure 9 shows the cost for an inheritance tree update. We measured the implementation where helping and donation links are explicitly tracked in the kernel and the update was triggered from a remote processor. A thread running on one CPU is aborted by a remote thread running on another CPU, so that an additional cross-processor synchronization is included in the overhead. The overhead depends on the length of the path from the aborted thread to the root node. The number of cycles required for breaking a link increases linearly with the length of path in the inheritance tree. The absolute duration to update the inheritance tree is less than $5\mu s$ for a path length of up to 64 threads and less than $8\mu s$ for up to 512 threads. To date we have not observed calling depths of more than 16 threads in real-world scenarios.

VI. Conclusion

We have designed a novel mechanism that implements priority and bandwidth inheritance in a component-based system. Our algorithm does not limit the preemptibility of the kernel, and keeps the runtime cost on the performance-critical IPC path minimal. The algorithm supports threads that block while holding shared resources, and can detect livelocks. Our evaluation shows that the performance overhead of the dependency tracking scales linearly with the number of threads in a call chain.

Acknowledgements

We thank Jean Wolter and the anonymous reviewers for their comments on an earlier version of this paper.

References

[1] B. W. Lampson and D. D. Redell, "Experience with Processes and Monitors in Mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, 1980.

[2] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[3] Z. Deng and J. W.-S. Liu, "Scheduling Real-time Applications in an Open Environment," in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 1997, pp. 308–319.

[4] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[5] G. Lipari, G. Lamastra, and L. Abeni, "Task Synchronization in Reservation-Based Real-Time Systems," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1591–1601, 2004.

[6] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-Based Secure Virtualization Architecture," in *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2010, pp. 209–222.

[7] J. Liedtke, "Improving IPC by Kernel Design," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 1993, pp. 175–188.

[8] S. Ruocco, "Real-Time Programming and L4 Microkernels," in *In Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2006.

[9] M. Hohmuth, "Pragmatic Nonblocking Synchronization for Real-Time Systems," Ph.D. dissertation, TU Dresden, Germany, 2002.

[10] U. Steinberg, J. Wolter, and H. Härtig, "Fast Component Interaction for Real-Time Systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, 2005, pp. 89–97.

[11] D. Faggioli, G. Lipari, and T. Cucinotta, “An Efficient Implementation of the Bandwidth Inheritance Protocol for Handling Hard and Soft Real-Time Applications in the Linux Kernel,” in *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2008, pp. 1–10.

[12] Linux Community, “Linux Realtime Patches,” 2010, April 2010. [Online]. Available: http://rt.wiki.kernel.org

[13] S. Rostedt, “RT-mutex Implementation Design,” 2010, Document shipping with the Linux 2.6 kernel sources, file: [Documentation/rt-mutex-design.txt].

[14] D. Niehaus and group, “Proxy Execution in Group Scheduling,” 2010, April 2010. [Online]. Available: http://www.ittc.ku.edu/kusp/kusp_docs/gs_internals_manual/index.html

[15] M. Friesbie, D. Niehaus, V. Subramonian, and C. Gill, “Group Scheduling in Systems Software,” in *In Workshop on Parallel and Distributed Real-Time Systems*, 2004.