The Mechanics of In-Kernel Synchronization for a Scalable Microkernel

Volkmar Uhlig*
IBM T.J. Watson Research, Yorktown Heights, NY

ABSTRACT

Systems with minimal kernels address the problem of ever-increasing system software complexity by strict separation of resource permission management and resource policies into different trust domains. Lately, such system structure has found wide attention in the research community and industry in the form of hypervisors and virtual machines.

With an increasing number of processors, these systems face a scalability problem. The separation eliminates semantic information about the expected parallelism for individual resources, such as memory pages or processors. Hence, the kernel is unable to optimize its synchronization primitives on a case-by-case basis—a precondition for a scalable, yet well-performing system.

In this paper we present an adaptive synchronization scheme, one of the core building block for scalable microkernels. Herewith, unprivileged components (like virtual machines) can express the degree of concurrency at the granularity of individual resources. The kernel can safely adapt and optimize its internal synchronization regime on a case-by-case basis as we show exemplary for inter-process communication and the memory management subsystem of an L4 microkernel.

1. INTRODUCTION

Minimal kernels address the problem of ever-increasing complexity of monolithic systems by strict separation of *resource permission management* and *resource policies*. The resource permissions are managed and enforced in the privileged kernel while policies are removed from the system's critical part and placed in unprivileged mode. This separation provides better fault containment, allows for coexistence of different systems and system personalities, and enables simple and safe system extensibility. Such system structure has been investigated by a large body of research, leading to solutions like microkernels [1, 11, 21, 27, 29], exokernels [8, 10] and recently gained popularity for hypervisors hosting virtual machines [4, 15, 20, 28].

The reduction of kernel functionality to sole permission management naturally leads to a simplified set of unified kernel-provided resources: memory pages, processors, and interrupts. The management interface to these resources is as basic as delegation and revocation of access permissions to pages, creation and deletion of virtual address spaces, some form of processor scheduling, and basic routing of messages and asynchronous events. The high-level semantics of operating system abstractions (such as files and tasks) is then constructed out of manipulations of these basic resources.

While the reduction of kernel abstractions of a microkernel simplifies the kernel implementation it also has drawback backs. The unified view on system resources eliminates the option for code-path specialization. This becomes a problem on large-scale shared-memory multiprocessors where the synchronization strategy needs to be optimized on a case-by-case basis. This has been shown by Chaves and others [7] who differentiate between ten alternative kernel synchronization mechanisms and conclude that the selection of a particular primitive depends, among others, on frequency of access, read-to-write ratio, and the amount of parallelism.

With the elimination of high-level abstractions, a microkernel (from now on referred to as *kernel*) lacks the semantic information that is usually available to an operating system (OS). For example, the abstraction of a memory mapped file in a monolithic OS carries the additional information about inter-dependencies between physical pages, the number of potential readers and writers, the maximum number of processors that may access the file, kernel vs. application memory, etc. Defaulting to one synchronization scheme will ultimately lead to either poor performance or poor scalability; for example, overly fine-granular synchronization will induce a large runtime overhead while coarse-granular synchronization will create contention points and limit scalability.

When considering the overall system, like a scenario with multiple virtual machines running on a hypervisor, the required resource usage patterns are readily available—just not to the hypervisor.

In this paper we present a novel approach that enables unprivileged resource policy managers to express resource concurrency to the kernel and control and safely re-adjust the kernel's synchronization mechanisms. The solution comprises of two aspects: We use a scalable fixed-size representation of processors to express concurrency that is independent of the overall number of processors in the system. Such a space efficient representation is a requirement for optimizations down to the granularity of individual resources. Second, *dynamic locks*, is a synchronization mechanism which enables the kernel to dynamically and safely adjust the synchronization granularity and primitive without compromising kernel integrity and security.

We validate our approach for the L4 microkernel on a (moderately) parallel multiprocessor machine with 16 processor contexts. We show that dynamic adaptation can maintain the overhead of kernel operations close to those

*This work was done at University of Karlsruhe, Germany.

of a uniprocessor and scale.

2. KERNEL SCALABILITY

The choice for an efficient, low-overhead concurrency control is a function of the degree of concurrency, placement, and contention—which all can change depending on the resource use over time. A microkernel, providing an abstract low-level machine interface, is disadvantaged over a monolithic OS which can specialize concurrency control on its manifold alternative code paths. The monolithic OS carries implicit high-level semantics in the synchronized code and data paths and thus can use alternative concurrency control schemes based on typical access patterns. For example, typical OS structuring naturally differentiates between memory types, like buffer cache pages, heap, or device control registers. In the abstract machine interface of a microkernel such differentiation does not exist, yet each resource type has a different concurrency control sweet spot.

The microkernel uses concurrency control for two purposes: (*i*) for concurrency control explicitly requested by applications (such as a serialization function) and (*ii*) concurrency control on kernel data structures to ensure correctness and safety. The second class is most critical for system performance; it includes all key system resources abstracted by the kernel: memory permissions, processor resources, and asynchronous events (i.e., hardware interrupts).

The key idea presented in this paper is to provide similar flexibility and choice of the concurrency control mechanisms as available to monolithic OSes by *dynamically adapting* the in-kernel concurrency control mechanisms at runtime in response to dynamic application behavior using *application-provided feedback*.

We provide explicit application-level control over the in-kernel concurrency control mechanisms, placement, and adaptation through the kernel system call interface.[1] Based on McKenney's study on in-kernel synchronization [23], we adapt the concurrency mechanisms for locality (primarily local vs. primarily shared access), granularity (coarse- vs. fine-granular locks), and contention likelihood (synchronization primitive). The adaptability is based on three contributions: (*i*) a space efficient encoding of locality information which allows for individual tagging of each kernel-managed resource, (*ii*) a dynamic locking primitive which can be activated and deactivated at runtime without compromising correctness, and (*iii*) the safe adaptation of the synchronization granularity for critical section fusing built upon the dynamic lock primitive.

2.1 Safety Requirements

In order to preserve the security guarantees of the microkernel, dynamic adaptation of the concurrency mechanism must be safe. We postulated the following requirements:

- The adaptation of the concurrency mechanism must not allow an application to elevate its privileges (e.g., corrupt kernel-internal data structures).
- Adaptation of the concurrency mechanism must be local. Independence of applications must be preserved.
- For transparency reasons, incorrect configuration of the concurrency mechanism should not influence the correctness of an application beyond timing and scalability characteristics.

[1]Note that adaptation could also be achieved by monitoring access patterns and predicting future behavior. However, such a prediction would introduce a kernel-level policy and thus violate the minimality criterion of a microkernel [21].

2.2 Parallelism vs. Locality

Multiprocessor optimizations represent a scalability vs. performance trade-off between alternative schemes which are inherently architecture specific and, to a large degree, depend on the peculiarities and physical limitations of the memory interconnect.

When choosing a particular concurrency control mechanism, the kernel needs to consider locality (which processors may access a resource), the concurrency (how many processor may access the resource), and the topology of the hardware (latency, overhead, and fairness of the synchronization primitives). As mentioned before, a microkernel cannot predict resource usage pattern a-priori and thus needs to be able to perform well for a spectrum of access pattern. The scenarios range from frequent accesses from a small set of processors to less frequent accesses but by a large number of processors for highly shared resources. Furthermore, access patterns may change over time.

The widely used processor sets, a bitmap with a bit per processor, has a prohibitively high memory footprint with increasing number of processors. Furthermore, determining the total number of processors selected in a set has a complexity of $O(n)$. Both properties render a bitmap representation not useful for the intended purpose of fine granular adaptation.

When taking the structure of the memory interconnect into consideration, one can reduce the footprint as following: Non-uniform memory has a higher cost for remote than for local accesses. A system should therefore decrease the frequency of accesses to remote resources with increasing cost; otherwise the system will ultimately saturate on resource stalls. The most common and relevant approaches for reducing the remote access frequency are (*i*) to explicitly favor local over remote resources (and reallocate resources on migration), (*ii*) minimize resources that are shared across many processors (e.g., by replication), and (*iii*) co-locate tasks that have a high degree of sharing on neighboring processors.

We developed a fixed-size compressed processor bitmap targeting the spectrum of frequent local vs. less frequent but parallel access. Similarly to floating point encoding we trade accuracy for space and allow to group neighboring processors and represent them with a single bit in the bitmap. As we argued above, such clustering is natural due to the hardware structure of the cache hierarchy of multi-cores and memory interconnects.

The encoding of the *cluster mask* is a structure with three fields: a bitmap, an order, and a window offset. The order denotes how many processors a bit represents; the offset denotes the starting processor ID of the bitmap. Figure 1 shows an example of processors encoded as alternative cluster masks.

On a 32-bit machine, our encoding allows to represent from 16 individual processors up to 2^{256} processors in a value of the size of a register. Testing for an individual processor in the mask or computing the overall number of processors are all simple arithmetic and logic functions.

We use the cluster mask in three scenarios: for per-kernel

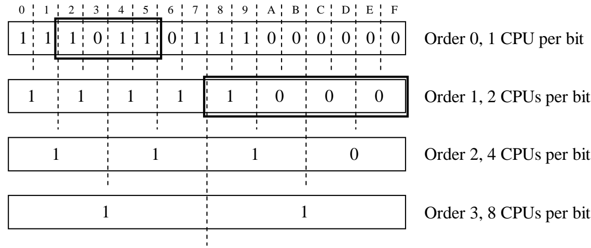


Figure 1: Cluster-mask encoding for different orders. The sample values show the reduction of accuracy with increasing order. The encoding for the two highlighted bitmaps as triplets $(\text{order}, \text{offset}, \text{bitmap})$ **are** $(0, 2, 11)$ **and** $(1, 4, 8)$**.**

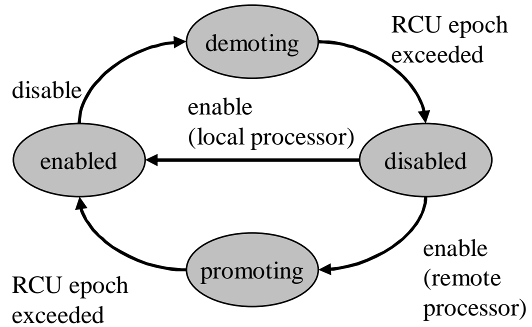


Figure 2: State diagram for promotion and demotion of spin locks

object tracking of locality information, for communicating locality information between application and kernel (in a processor register), and for tracking of stale TLB entries. Only with available and detailed locality information can the kernel adapt its concurrency regime. In Section 3 we describe the usage scenarios in more detail.

2.3 Adaptive Synchronization

The second building block for adaptive concurrency control is the ability to modify the in-kernel synchronization mechanism in a safe manner. Most operations of a modern microkernel are extremely short, in most cases less than a few hundred instructions. More heavy-weight synchronization mechanisms (like blocking) are exposed to the applications. We found that standard spin locks are an efficient mechanism for in-kernel synchronization, however, with overheads of more than a hundred cycles on a Pentium 4 architecture. For example for our critical inter-address space communication path, a spin lock incurs an overhead of 15 percent per lock.

Depending on the expected degree of parallelism and access patterns as expressed (and enforced) in a per-kernel object processor cluster mask, we either use memory-based spin locks for concurrent accesses or fall back to message based synchronization via inter-processor interrupts. Both synchronization schemes have performance advantages and disadvantages based on the specific access patterns [7]. When accesses to a kernel object are primarily on one processor, message based synchronization eliminates the cost of a memory-based lock from the critical path. Whereas, when the kernel accesses an object from multiple processors concurrently, the overhead and latency for message based synchronization becomes disadvantageous.

Note, that the adaptive scheme can be trivially extended to other synchronization primitives such as MCS locks [22, 25], but will not be discussed here further.

2.3.1 Dynamic Locks

In order to accomplish dynamic adaptability, we extend the memory-based lock primitives (e.g., spin locks) by a *lock state*. The lock state denotes whether or not the lock primitive is active. In disabled state, the lock operation (e.g., an atomic operation) is not executed thus incurring no (or only marginal) overhead.

When switching between enabled and disabled states there is a transition period, when the lock state update is not fully propagated to all processors. We avoid kernel data corruption due to an inconsistent view on the lock state by enforcing the *old state's behavior* until all processors in the system have a consistent view.

For common kernel lock primitives, such as spin locks, it is impossible to derive whether remote processors actively try to acquire the lock. There is no upper bound for the time it takes to propagate a state update to all processors. Even by adapting the spin loop such that it explicitly checks for a state change, it is not possible to predict memory access latencies in large multiprocessor systems.

Read-copy update (RCU) successfully addresses a similar problem with its epoch scheme [24]. The typical implementations of RCU pass a token between the processors; passing the token ensures that the processor executes a *safe* code path (within the kernel) that is guaranteed to hold no data references. The RCU token thus also guarantees that a processor is not actively trying to acquire a lock. Hence, a full token round-trip—an epoch—guarantees that all outstanding lock attempts have completed and that all processors are aware of the update of the lock primitive.

Each dynamic lock carries an epoch counter that denotes when the state switch is safe. The lock code first tests whether the lock is in a transition period; if that is the case and the stored epoch counter denotes that the epoch expired, the locking code finalizes the state switch. In addition to the *enabled* and *disabled* state, the lock has two intermediate states: *promoting* and *demoting* (see Figure 2). When the lock is in the disabled state, the code path can safely skip the actual lock acquisition.

Figure 3 illustrates the transition from an enabled to a disabled lock state initiated on CPU A including the required transition period for the epoch. When enabling a disabled lock on the processor that has exclusive access we avoid the delay of the RCU cycle, but enable the lock immediately. This is safe if the enabling code does not depend on the lock for its correctness.

The additional functionality induces a higher overhead on the lock primitive due to the higher cache footprint for the state and epoch counter, and also higher execution overhead for the additional checks. One can minimize the overhead by carefully placing the state and epoch data (e.g., the colocation of the two state bits with other read data on the hot path has no cache impact).

The following code fragment shows sample code for a nor-

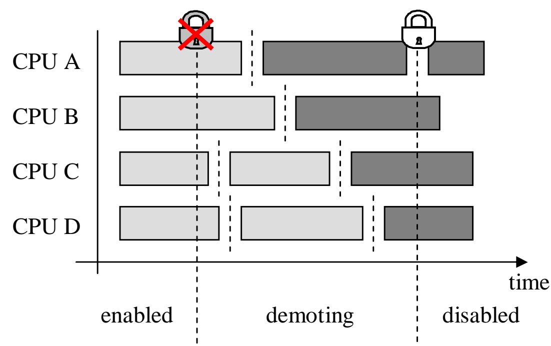


Figure 3: Delayed lock demotion based on RCU epochs. After CPU A disabled the lock, it has to remain active for one complete epoch to ensure no more outstanding lock attempts by other processors.

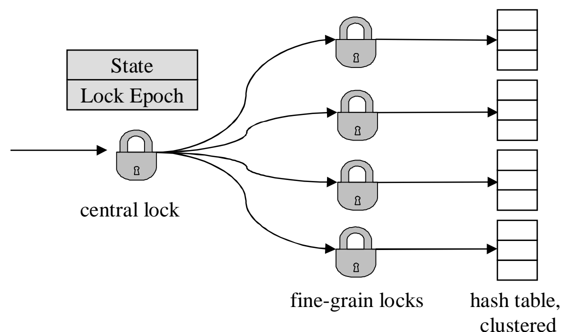


Figure 4: Cascaded spin locks to control locking granularity. Only one of the locks—coarse or fine—is enabled at a time. Reconfiguration at run time addresses contention vs. overhead.

mal spin-lock primitive with a value of zero in the lock variable indicating a free lock. The overhead for a disabled lock using the specific encoding is a single jump instruction.[2]

```
if (Lock.State != disabled) {
  while TestAndSet(Lock.Spinlock) { /* wait */ }
  if ( Lock.State != enabled &&
       Lock.Epoch+1 < GlobalEpoch )
    { /* perform lock state manipulation... */ }
}
/* critical section */
Lock.Spinlock = 0; /* lock release */
```

2.3.2 Lock Granularity

Built upon the dynamic locks we derive a method for dynamically adjustable lock granularities. The decision for a coarse-grained vs. a fine-grained locking strategy depends on the specifics of the algorithm, that is *where* to lock and *what* gets protected by locks.

Fine-granular locking requires a divisible data structure where each part can be individually locked without compromising the overall structural integrity. The data structure is divided into multiple parts and each part is protected by an individual lock. The lock granularity depends on the number of independent parts of the overall data structure.

We make the lock granularity adjustable by introducing *multiple locks* to the same object—one for each locking granularity. In order to lock an object, *all* locks have to be acquired, while some locks can be disabled. The disabled locks incur marginal runtime overhead and only require access to read-shared data (i.e., the lock state). Acquiring all locks ensures that all processors synchronize on the same locks, whichever lock may be active at the time. We avoid deadlocks by simple lock ordering, for example, coarse-grain locks are always acquired before fine-grain locks. The lock release has to take place in reverse order of the acquisition. Figure 4 shows an example of multiple cascaded locks protecting a hash table and individual clusters of the table.

Switching the lock granularity for a dividable object takes place in multiple stages. To preserve correctness, at no point in time should two processors use a different lock granularity (i.e., one uses coarse grain, the other fine grain locks). A safe transition between the lock modes therefore requires an intermediate step, where *both* locks—coarse and fine grain—are active. After that transition period, when all processors are aware of the new locking strategy, the lock code can disable the previous lock. The completion is tracked with the RCU epoch counter, similar to dynamic locks. Note that the additional memory footprint for a two-level lock cascade is identical to dynamic locks: two bits for the state and the epoch counter.

Since dynamic locks only access the lock variables when the locks are enabled, the memory required for the lock variable can be dynamically allocated (i.e., when first switching to fine-grain locking). The following code fragment shows pseudo code for a two-level lock cascade including code that handles dynamic adjustment.

```
enum State = {Coarse, Fine, ToCoarse, ToFine};
if (State != Fine) {
  lock (CoarseLock);
  if (State != Coarse AND LockEpoch+1 < GlobalEpoch) {
    if (State == ToCoarse)
      State = Coarse;
      /* potentially release fine lock memory */
    else
      State = Fine;
  }
}
for index = 0 to NumberObjects {
  if (State != Coarse) lock (FineLock[index]);
  /* critical section */
  if (State != Coarse) unlock (FineLock[index]);
}
unlock (CoarseLock);
```

A special case of the scheme occurs when the fine-granular objects themselves are not locked but modified with an atomic operation (e.g., atomic compare-and-swap). In coarse-grain lock mode, all objects are protected by a single lock and the data members can be modified with non-atomic memory operations whereas in fine-granular synchronization mode, manipulations are performed with multi-processor safe atomic operations.

The overhead for the special handling is minor on most architectures. For example, IA32's `cmpxchg` instruction requires an additional instruction prefix to differentiate between the synchronized multiprocessor and the unsynchronized version of the instruction. A conditional jump over the one-byte instruction prefix, depending on the lock state,

[2] The solution is directly transferable to architectures that realize atomic operations with a pair of load-linked, store-conditional instructions.

induces only marginal overhead and code complexity, while reducing the cost from 136 to 12 cycles (Pentium 4 2.2GHz). Itanium's predicates allow for a similarly efficient implementation.

3. IMPLEMENTATION IN L4

We integrated the described mechanisms in an L4 microkernel with multiprocessor support [30]. We revisited all core kernel primitives and explicitly expressed concurrency. While alternative systems abstractions (e.g., virtual processors and page tables of a hypervisor) may lead to slightly different solutions, we believe many of the mechanisms are directly transferable to other microkernels and hypervisors.

We set the following design goals:

Scalability: Scalability is the primary goal. Following Unrau's design principles [33] for scalable systems, the kernel has to preserve parallelism, all operations must have a bounded overhead, and preserve locality.

Performance: The execution overhead must be minimal. For processor-local operations we evaluate the performance based on the uniprocessor implementation. For operations that are either cross-processor or require serialization due to concurrency, we optimize on the macro-operation level. A macro operation comprises multiple smaller, potentially independent operations.

Policy freedom: New mechanisms should not introduce kernel policies.

Transparency: When operations violate isolation restrictions we favor transparent completion at higher cost over explicit failure notification to the application.

We extended the system call interface to the first-class kernel abstractions with a cluster mask parameter: the locality domain of a thread, the locality domain of an address space, the locality domain of an interrupt. Accesses from processors within the locality domain denoted in the cluster mask use spin locks and outside of the locality domain fall back to message-based synchronization. When the locality domain is limited to a single processor, the kernel disables the spin locks. Also, we made the granularity and inter-dependency of page mappings explicit and adaptable.

Following, we describe both solutions in more detail.

3.1 Inter-process Communication

In L4, the key kernel mechanism for system composition is inter-process communication (IPC). All higher level services are constructed out of the base IPC primitive: communication for remote procedure calls, synchronization between threads, on-demand paging, interrupt delivery, and exception handling. Thus minimizing the overhead of the IPC primitive is key to good overall system performance. The overhead of simple memory based synchronization increases the cost for an IPC by as much as 83 percent on a Pentium 4 system, making it the key candidate for an adaptive synchronization scheme.

IPC takes place between two threads, is synchronous, unbuffered, and uses hand-off scheduling to remove unnecessary overhead from the critical path [21]. Two scenarios are most common and relevant: (*i*) client-server interaction and (*ii*) IPC-based signaling and notification (semaphores, queues, etc).

We make the assumption that the programmer has detailed knowledge on application behavior and communication patterns and aim at maximizing performance for those applications. Furthermore, we assume that the overheads for cache migration, synchronization, and signaling are so high that frequent cross-processor remote-procedure calls are unfeasible. This assumption stems from (*i*) the overhead for asynchronous inter-processor signaling, (*ii*) the cost for cache-line migrations relative to the frequency of remote invocations and the potentially achievable speedup. For upcoming tightly-coupled heavily-threaded multi-core architectures some of the assumption may need to be reevaluated.

Client-Server IPC

Client-server communication is the most performance-critical operation, because it replaces all simple kernel-invocations in a monolithic OS by two IPCs. In most cases, the client depends on the results of the server invocation and needs to block until completion of the remote operation. The IPC invocation thus follows the instruction stream of the application similar to a normal system-call invocation in a monolithic kernel.

IPC in the client-server scenario is therefore a mechanism to (*i*) guarantee that the remote server is in a consistent state (signaled by waiting for an incoming message) and (*ii*) to transfer control into the server's address space. It is important to note that client-server relations usually do not make use of the parallelism threading would provide, but *explicitly avoid* it. In order to minimize the IPC overhead, the scheduler is not invoked when switching from client to server thread and the server executes on the client's time slice instead [5].

In multiprocessor systems the client request could theoretically be handled by a server thread on a remote processor. The overhead of cache line migration for parameters and data working set, latency and overhead for remote signaling, and overhead due to scheduling renders a cross-processor model unfeasible. We therefore decided to strictly favor the performance of processor-local over cross-processor IPC.

System services (e.g., file systems) need to be aware of the parallelism and have to provide *at least* one handler thread per processor to scale. Threads have to be distributed across all processors such that clients can interact with a local representative.

Synchronization and Notification

Synchronization is a latency-sensitive operation. The time from the release operation of a synchronization primitive until a previously blocked thread starts executing influences the responsiveness and increases the overall runtime of a task. The design goal is to minimize that latency. The cost for an IPC that unblocks a thread on the same processor is at least as low as alternative high-level kernel synchronization primitives (such as semaphores or mutexes).

Signaling across processor boundaries requires three operations: (*i*) test and modify the thread's state from blocked to ready, (*ii*) enqueue the thread into its home-processor's scheduling list, and (*iii*) notify the remote processor to reschedule. These three steps can be realized in two different ways. Firstly, the data structures are manipulated directly and therefore require some form of synchronization (e.g., locks). Secondly, the operation is initiated via an in-kernel message and performed by the thread's home processor it-

self. In that case no locks are required because operations are serialized by the remote CPU.

Adaptive IPC Operation

Instead of favoring one synchronization method for IPC, the kernel supports both: message-based and lock-based synchronization. Based on its communication profile, each thread can derive a set of *preferred remote processors* that are expressed in a cluster mask. Based on the mask, the kernel fine-tunes the IPC path. Processors that are specified in the mask use a lock-based IPC scheme, while all others use message-based synchronization. The kernel may further fine-tune the memory lock primitive itself (i.e., use spin locks or MCS locks).

When the cluster mask restricts accesses to a single processor, the home processor of a thread, the kernel *disables* the spin lock on the IPC path using the dynamic lock demotion scheme as described in Section 2.3. The deactivated lock state removes the overhead of two atomic operations from the critical path.

Remote Scheduling for Lock-based IPC

After completion of a remote IPC operation, the partner thread becomes runnable and has to be considered by the remote scheduler. Rescheduling is achieved by sending an inter-processor interrupt to interrupt the current execution. If the currently executing remote thread has a higher priority than the activated one, the thread can simply be enqueued into the ready list and the normal scheduling mechanism will activate it at some later point in time.

In order to scale, the kernel has per-processor structures for the scheduler. These structures include one ready list per scheduling priority and a list to handle timeouts. To lower the overhead for the processor-local case, scheduling lists are processor specific and unsynchronized (thus avoid the cost of a spin lock on the critical path) but must not be accessed from remote processors.

We use a lazy remote enqueing scheme with each scheduler having one or more remote requeue lists that enable remote processors to use memory-based queues. Requeue lists are protected by per-list locks; multiple lists per scheduler further reduce concurrency from many remote processors and thus the potential for lock contention.

3.2 Memory Management

L4's abstraction of a page mapping is a *flexpage* [14] describing a set of contiguous virtual pages of an address space. Flexpages are mapped hierarchically between address spaces. The kernel uses a flat resource management model: there is no differentiation between specific properties of individual resources such as RAM and device memory.

The kernel maintains an internal database of page mappings which represents the memory map hierarchy. For each physical page frame the kernel maintains a sorted n-ary tree that is rooted from the root address space (σ_0). The tree is realized via a sorted doubly-linked list where each map node contains a depth field representing the level in the map hierarchy.

Since individual entries in the list represent actual resource permissions, a relaxed consistency model (like RCU) could ultimately lead to resource corruption on multiprocessor systems, like one user-level thread overwriting a page marked as read-only. We realize a strict consistency

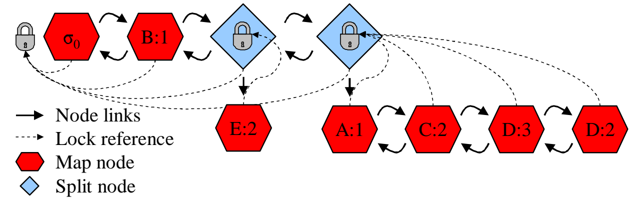


Figure 5: Mapping tree structure including split nodes. Concurrent operations are possible within independent subtrees (σ_0, E and A).

model by explicitly synchronizing neighboring nodes in the list.

However, a simple list-based synchronization model introduces an inter-dependency between potentially unrelated mapping nodes. We address the problem by three algorithmic changes: (*i*) explicit control of synchronization domains by partitioning the mapping tree, (*ii*) dynamic adaptation of the synchronization granularity for super pages (as described in Section 2.3.2) and (*iii*) TLB dirty tracking via the cluster mask which enables aggressive TLB shoot-down coalescing while maintaining the ability for parallel out-of-band TLB updates.[3] Figure 5 depicts the kernel data structure for the n-ary tree with two synchronization sub-domains.

We extended the flexpage management interfaces such that the pages the granularity and locality is a parameter to the management system call. The interface reflects the common usage scenarios for stackable user-level pagers which allocate large pools of memory and hand them out individually. The life span of a page mapping ranges from very short (e.g., for fork/exec) to the life time of the system; the mapping granularity ranges from a single page to a multi-megabyte memory object for an on-demand pager.

Figure 5 illustrates the kernel data structure for the mapping tree of one page frame. The split nodes are synchronization and TLB tracking objects which get inserted into the list structure at application request. Individual subtrees can be further subdivided allowing for arbitrary synchronization domains for the kernel data structure and for TLB coherency. TLB dirty tracking is achieved using a per-processor version; the relevant processor set is tracked in a cluster mask in the split node.

4. EVALUATION

In this section we evaluate the overhead and scalability of individual kernel operations in a set of microbenchmarks. We compare the cost of individual kernel operations for different multiprocessor workload configurations, intra- vs. inter-processor operations and low vs. high concurrency. Furthermore, the benchmarks evaluate the *overhead* of multiprocessor primitives to the baseline performance given by the uniprocessor kernel, which we assume is the achievable minimum for the specific operation.

Benchmarks were performed on an IBM xSeries 445 server with eight Pentium 4-based 2.2 GHz Intel Xeon processors.[4]

[3]Due to space constraints we have to refer to [31] for a detailed description of the algorithm.

[4]Each physical processor (Intel family 15, model 2, stepping 6) had an 8KB L1 data cache, a 12KB μop trace cache,

Each processor had two logical threads enabled. The system consisted of two processor boards with four processors each. Each board had separate memory (2 GB each) and both boards were interconnected via IBM's NUMA interconnect.

IA-32 has a relatively strong memory ordering model, defined as *write ordered with store-buffer forwarding* [9]. The strong write ordering has implications on speculative execution and induces a particularly high overhead on atomic instructions. The cost for atomic exchange (xchg) is 125 cycles and atomic compare-exchange (lock cmpxchg) is 136 cycles.

The test system had only eight processors with a total of sixteen processor contexts. Literature suggests that scalability problems in many cases only become visible for systems of more than 32 processors. While we were limited by the hardware constraints, however, we argue that the results are still representative because our primary focus is on preserving full algorithmic independence and minimizing the execution overhead.

4.1 Communication

We evaluated the overhead of the IPC primitive for client-server and cross-processor communication. Note that beside standard communication, L4's IPC primitive is used for a number of system operations, including page fault and exception handling, resource delegation, scheduling notification, and interrupt delivery [15] and thus runtime overhead reflects on a large set of OS subsystems such as IO and paging.

Client-server scenarios use IPC to transfer request and response payloads; the operation is completely executed on one processor. The cost is dominated by the inherent cost of TLB and cache invalidations on the P4. In Figure 6 we compare the IPC overhead for message-based and lock-based synchronization against the base line uniprocessor primitive (numbers are given for inter- and intra-address-space IPC).

The minimal multiprocessor overhead via message-based in-kernel synchronization is 2.9% for inter-AS IPC and 8.6% for intra-AS IPC. Compared, the overhead for lock-based synchronization amounts to 27.6% for inter-AS IPC and 81.2% for intra-AS IPC. Besides the lock, processor-local IPC only touches per-processor data structures and thus scales optimally.

For cross-processor IPC we evaluated a common parallel programming barrier scenario, where multiple worker threads get a notification message from a coordinator. This setup is comparable to the OpenMP microbenchmark for the PARALLEL directive as proposed by Bull [6]. We measured the notification latency of all workers for an increasing number of processors. Figure 7 compares the overhead for message- and lock-based synchronization. For cross-processor IPC lock-based synchronization clearly outperforms the message-based scheme by a factor of 2.5 for 2 processors up to 3.6 for 16 processors, respectively. This high overhead is partially incurred by the multiple inter-processor messages the kernel sends. Furthermore, for lock-based synchronization the control thread can continue sending IPCs while inter-processor interrupts are in-flight whereas for message-based synchronization the control thread stalls.

a shared 512 KB L2 cache, and a shared 2 MB L3 cache. It further featured a 64 entry data TLB and a 64 entry instruction TLB.

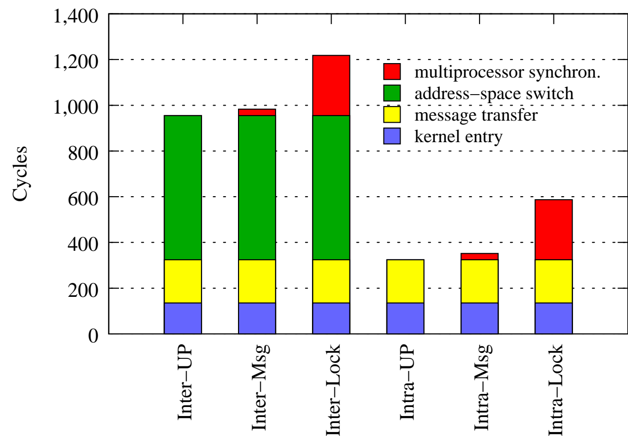


Figure 6: Break-down of IPC cost for inter- and intra-address-space communication for uniprocessor (UP), message-based (msg) and lock-based (lock) synchronization. All on same processor.

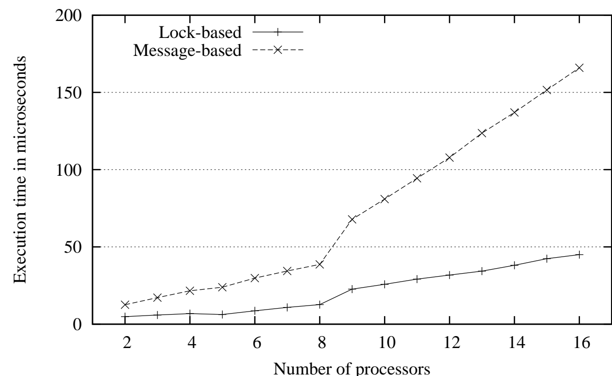


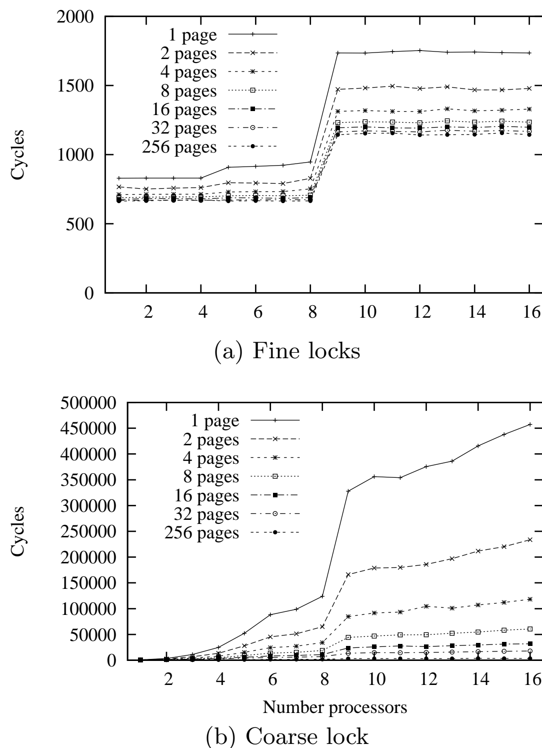
Figure 7: Signaling latency for fork-join programming model as used in OpenMP. The signaling latency is directly related to the overall execution time and therefore limits the speedup that can be achieved via a parallelized workload.

4.2 Memory Management

We evaluated the overhead and scalability of a repeated write permission revocation. This operation is typical for copy-on-write operations such as fork and identical to complete revocation except for the final memory release. Since the kernel data representation remains unmodified, the benchmark is repeatable. We ran all benchmarks 200 times and report the average over all runs.

The runtime overhead per mapping for fine vs. coarse granular synchronization is 185 cycles per mapping. The benchmark showed a 24 percent overhead for a moderately populated address space with 128 pages and 32 percent overhead for 256 pages (in Linux, a Bash maps 185 pages, Emacs 1513 pages, and MySQL 4453 pages). Additionally, individual locks incur a higher cache footprint of slightly more than one cache line per mapping.

In the next benchmark we evaluated the scalability of concurrent unmap operations with increasing number of processors. The same set of pages was mapped to threads on dif-



(a) Fine locks

(b) Coarse lock

Figure 8: Unmap performance for one processor repeatedly unmapping a set of pages with (a) fine granular and (b) coarse granular locking.

ferent processors. One thread revoked the write permissions to an increasingly large set of pages while the other threads revoked the permissions to exactly one page. We then measured the average execution time with a coarse and a fine granular locking scheme.

With fine-grain locks, the cost per mapping remained almost constant, independent of the number of processors (see Figure 8a). The graph shows two anomalies, one between 4 and 5 processors and a second between 8 and 9 processors. In the benchmark we placed the parallel unmap operations first on physical processors (1 to 8) followed by logical processors (9 to 16). The first knee at 5 processors is when the unmap operation hit the first NUMA processor (5 to 8). The overhead reduced with an increased number of unmapped pages, since the cache-line transfer was only necessary for one page out of the overall set of pages. The second, more drastic overhead increase was incurred by parallel execution of multiple processor threads of the same physical processor (SMT). The two processor threads on each core competed for execution resources and interfered with each other. While the overhead per mapping increased by a factor of two, the cost per mapping remained constant for 9 to 16 in parallel operating unmaps.

A benchmark comparing the cost for unmaps of independent pages showed perfect scalability for individual locks compared to linearly growing cost per processor for coarse locks (Figure 8b).

4.3 Memory and Cache Footprint

The additional flexibility for concurrency control comes at a cost in memory footprint and thus have implications on performance due to secondary effects such as cache pollution and available memory.

The thread control block (TCB) was extended by a cluster mask field and the RCU epoch (a spin lock was already required beforehand). However, since the kernel stack is part of the TCB the overall size did not increase. The state field of the dynamic lock was encoded in two spare bits of another field accessed on the critical path and therefore had no additional memory or cache footprint.

The descriptor size for memory mappings was extended by an additional word allowing to use a lazy reclamation scheme based on RCU, growing the data structure by 25 percent. We introduced an intermediary descriptor type which allows to isolate two mapping hierarchies. This 20 byte intermediary descriptor is dynamically allocated at map time upon application request. A user-level pager can thus dynamically decide if subsequent mappings should be concurrently accessible or not.

The synchronization granularity of sub-pages of a larger superpage can be switched dynamically; the memory is allocated when the fine-granular synchronization becomes active. The memory footprint for the additional flexibility is limited to two state bits and an RCU epoch field which fitted into the already existing data structures.

5. RELATED WORK

Related work falls into three categories: scalable operating systems, locks and synchronization, and policy-free kernels.

Unrau [33] first addressed OS scalability in a systematic manner based on queuing theory. He proposed to consider the OS as a service center and evaluate scalability using the metrics throughput, utilization, and response time. He derived the three aforementioned design principles (preserving parallelism, bounded overhead, preserving locality). While highly relevant for this work, these principles ignore the base overhead of an operation. A system is still considered scalable even if the overhead is orders of magnitude higher than the achievable minimum.

Tornado [12] and K42 [3] introduced clustering as a structuring method for scalability, and was generalized in clustered objects [2]. A clustered object presents the illusion of a single object that is actually composed of multiple instances enabling a high degree of concurrency. Each instance handles calls from subsets of processors and software can adapt the object implementation according to specific access patterns and parallelism.

Synchronization mechanisms have been thoroughly investigated and lead to a variety of software and hardware solutions. Special processor locking primitives oftentimes optimize one particular synchronization operation [13,17,19] but not the synchronization strategy. MCS locks [25] reduce the cache coherency overhead and improve fairness for NUMA systems at higher runtime cost. Lim [22] developed reactive locks that dynamically switch between spin locks and MCS locks.

Unrau et al. [34] address the problem of overhead vs. parallelism via a hybrid coarse-grain/fine-grain locking strategy that has the low latency and space overhead of a coarse grain locking strategy while having the high concurrency of fine-grain locks. A coarse serializing lock protects multiple lightweight locks. However, the scheme does not eliminate the required cache line migration but only reduces the wait

time and further places restrictions on object allocations.

Speculative lock elision [26] reduces the runtime overhead of synchronization primitives by speculative execution and delayed write buffer commits after a successful lock release. The speculation unit detects conflicts that would violate correctness and restarts the operation, which limits the size of the critical section and still induces cache footprint.

Asymmetric locking primitives can greatly improve performance for read-mostly data structures. Read-copy update (RCU) [24] takes the idea to the extreme permitting read-side accesses with no synchronization. When all CPUs have passed through a quiescent state since a particular point in time, RCU guarantees that all CPUs see the effects of all changes made prior to the interval. RCU significantly simplifies many locking algorithms and eliminates many existence locks.

Previous work investigated the applicability of policy-free kernels for MP systems. Hydra [35] was specifically designed with the goal of separating mechanisms and policy and as an exploration testbed for multiprocessors. The Raven microkernel [27] also targets multiprocessor systems with a focus on minimizing the overhead for kernel invocations. While moving significant system parts to application level, Raven still uses a number of global data structures, thus limiting kernel scalability.

Chen [8] extended the uniprocessor Exokernel system [10] to multiprocessor systems. Similar to our approach, exokernel localizes kernel resources and uses in-kernel messaging to manipulate remote resources. However, the choice for synchronization primitives is ad-hoc and static and the specific implementation dictates the synchronization granularity. Other prominent projects that use a microkernel-based design, however, with less focus on policy-freeness are the already mentioned systems Tornado and K42.

6. CONCLUSION

In this paper we presented mechanisms to manage multiprocessor scalability for minimal, policy-free kernels. Preserving independence of unrelated—and thus parallelizable—operations requires bridging the semantic gap between the kernel and applications in order to scale.

The expected parallelism as well as the enforcement of access permissions can be efficiently expressed in the *processor cluster mask* which trades accuracy against space favoring locality. The *dynamic lock* primitive allows for dynamic activation and deactivation of kernel locks, and—when cascaded—enable dynamic adaptation of locking granularities.

We validated the design and evaluated performance via microbenchmarks against an L4 microkernel with unrivaled uni-processor performance. The solutions were driven by our previous work for scalable multiprocessor virtual machines [32]. Due to the space limitations this paper primarily focused on the core kernel mechanics.

Future Work

We see three major areas for future work. First, this work primarily focuses on one specific architecture: the Pentium 4. Trade-offs for other hardware architectures are substantially different and need further investigation. Current architectural developments in the area of SMT and multicore systems change hardware properties. Tight integration of caches and improved IPI latencies may shift the cost points.

Second, the fundamental idea of alternative lock primitives that depends on the degree of parallelism is currently implemented as a software solution. The software-based approach requires additional code and thus incurs runtime overhead. The overhead could be eliminated by having explicit architectural support. This could include optimized operations on the cluster mask and support for dynamic locks.

Finally, the proposed kernel primitives require extensive testing and evaluation in a variety of real-world scenarios including large-scale databases. While we showed low overhead and independence for the individual kernel primitives in micro-benchmarks it remains open how the solutions behave in complex environments.

7. REFERENCES

[1] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition* (June 1986).

[2] Appavoo, J. *Clustered Objects.* PhD thesis, University of Toronto, 2005.

[3] Appavoo, J., Auslander, M., DaSilva, D., Edelsohn, D., Krieger, O., Ostrowski, M., et al. Memory management in k42. Whitepaper, Aug. 2002.

[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., et al. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating System Principles* (Bolton Landing, NY, Oct. 2003).

[5] Bershad, B. N., Anderson, T. E., Lazowska, L., and Levy, H. M. Lightweight remote procedure call. In *12th Symposium on Operating System Principles* (Oct. 1989).

[6] Bull, J. M., and O'Neill, D. A microbenchmark suite for openMP 2.0. In *3rd European Workshop on OpenMP* (Sept. 2001).

[7] Chaves, Jr., E. M., LeBlanc, T. J., Marsh, B. D., and Scott, M. L. Kernel-kernel communication in a shared-memory multiprocessor. In *The Symposium on Experiences with Distributed and Multiprocessor Systems* (Atlanta, GA, Mar. 1991).

[8] Chen, B. Multiprocessing with the Exokernel operating system. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2000.

[9] Corp., I. *IA-32 Intel Architecture Software Developer's Manual, Volume 1–3*, 2004.

[10] Engler, D. R., Kaashoek, M. F., and O'Toole, J. W. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the 15th ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, Dec. 1995), ACM SIGOPS, pp. 251–266.

[11] Gabber, E., Small, C., Bruno, J., Brustoloni, J., and Silberschatz, A. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference* (Monterey, CA, USA, June 1999).

[12] Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation* (1999).

[13] Goodman, J. R., Vernon, M. K., and Woest, P. J. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Boston, MA, Apr. 1989).

[14] Härtig, H., Wolter, J., and Liedtke, J. Flexible-sized page-objects. In *Proceedings of 5th International Workshop on Object-Orientation in Operating Systems (IWOOOS)* (Washington, DC, 1996), IEEE Computer Society, pp. 102–106.

[15] Heiser, G., Uhlig, V., and LeVasseur, J. Are virtual-machine monitors microkernels done right? *Operating System Review 40*, 1 (Jan. 2006), 95–99.

[16] Ho, C.-T., and Johnsson, L. Distributed routing algorithm for broadcasting and personalized communication in hypercubes. In *International Conference on Parallel Processing (ICPP 1986)* (1986), pp. 640–648.

[17] IEEE. *IEEE Std 1596–1992: IEEE Standard for Scalable Coherent Interface*. IEEE, Inc., Aug. 1993.

[18] Leierson, C. E. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers c-34* (Oct. 1985), 892–901.

[19] Lenoski, D., Laudon, J., Gharachorloo, G., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. S. The Stanford Dash multiprocessor. *IEEE Computer 25*, 3 (Mar. 1992).

[20] LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, Dec. 2004).

[21] Liedtke, J. On μ-kernel construction. In *Proc. of the 15th ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, Dec. 1995).

[22] Lim, B.-H. *Reactive synchronization algorithms for multiprocessors*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1995.

[23] McKenney, P. E. Selecting locking primitives for parallel programming. *Communications of the ACM 39*, 10 (1996), 75–82.

[24] McKenney, P. E., and Slingwine, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, Oct. 1998).

[25] Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (Feb. 1991), 21–65.

[26] Rajwar, R., and Goodman, J. R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture* (Austin, Texas, Dec. 1–5, 2001).

[27] Ritchie, S. The Raven kernel: a microkernel for shared memory multiprocessors. Tech. Rep. TR-93-36, University of British Columbia, Department of Computer Science, 30 Apr. 1993.

[28] Sailer, R., Valdez, E., Jaeger, T., Perez, R., van Doorn, L., Griffin, J. L., and Berger, S. sHype: Secure hypervisor approach to trusted virtualized systems. Tech. Rep. RC23511, IBM T.J. Watson Research Center, Yorktown Heights, NY, Feb. 2005.

[29] Shapiro, J. S., Smith, J. M., and Farber, D. J. EROS: a fast capability system. In *Proc. of the 17th ACM Symposium on Operating System Principles* (Kiawah Island, SC, 1999).

[30] System Architecture Group. *L4 X.2 Reference Manual*, 6th ed. University of Karlsruhe, Germany, Oct. 2005.

[31] Uhlig, V. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, Germany, May 2005.

[32] Uhlig, V., LeVasseur, J., Skoglund, E., and Dannowski, U. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium* (San Jose, CA, May 6–7 2004), pp. 43–56.

[33] Unrau, R. *Scalable Memory Management through Hierarchical Symmetric Multiprocessing*. Ph.D. thesis, University of Toronto, Toronto, Ontario, Jan. 1993.

[34] Unrau, R. C., Krieger, O., Gamsa, B., and Stumm, M. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Symposium on Operating Systems Design and Implementation* (Nov. 1994).

[35] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. HYDRA: The kernel of a multiprocessor operating system. *Commun. ACM 17*, 6 (June 1974).