Mostly Lock-Free Malloc

Dave Dice
Sun Microsystems, Inc.
Burlington, MA
dice@computer.org

Alex Garthwaite
Sun Microsystems Laboratories
Burlington, MA
alex.garthwaite@sun.com

Abstract

Modern multithreaded applications, such as application servers and database engines, can severely stress the performance of user-level memory allocators like the ubiquitous malloc subsystem. Such allocators can prove to be a major scalability impediment for the applications that use them, particularly for applications with large numbers of threads running on high-order multiprocessor systems.

This paper introduces Multi-Processor Restartable Critical Sections, or MP-RCS. MP-RCS permits user-level threads to know precisely which processor they are executing on and then to safely manipulate CPU-specific data, such as malloc metadata, without locks or atomic instructions. MP-RCS avoids interference by using upcalls to notify user-level threads when preemption or migration has occurred. The upcall will abort and restart any interrupted critical sections.

We use MP-RCS to implement a malloc package, LFMalloc (Lock-Free Malloc). LFMalloc is scalable, has extremely low latency, excellent cache characteristics, and is memory efficient. We present data from some existing benchmarks showing that LFMalloc is often 10 times faster than Hoard, another malloc replacement package.

1 Introduction and Motivation

Many C and C++ applications depend on the malloc dynamic memory allocation interface. The performance of malloc can be crucial and can limit the application. Many malloc subsystems were written in a time when multiprocessor systems were rare. They use memory efficiently but are highly serial and constitute an obstacle to throughput for parallel applications. Evidence of the importance of malloc comes from the wide array of after-market malloc replacement packages that are currently available.

Our goals are to provide a malloc implementation with the following characteristics:

- **Low latency.**
- **Excellent scalability**. It should provide high throughput even under extreme load with large numbers of threads and processors.
- **Memory-efficiency**. The allocator should minimize wastage
- **Minimal interconnect traffic.** Interconnect bandwidth is emerging as the limiting performance factor for a large class of commercial applications that run on multiprocessor systems. We reduce traffic by increasing locality.
- **Generality.** Our implementation should be independent of threading models, and work with large numbers of threads in a preemptive multitasking environment, where threads are permitted to migrate freely between processors.

In addition to the raw performance of malloc itself, the secondary effects of malloc policies take on greater importance. A good malloc implementation can reduce the translation lookaside buffer miss rate, the cache miss and the amount of interconnect traffic for an application accessing malloc-ed blocks. These effects are particularly important on large server systems with deep memory hierarchies.

Malloc implementations exist in a wide spectrum. At one extreme we find a single global heap protected by one mutex. The default "libc" malloc in the Solaris™ Operating Environment is of this design. This type of allocator can organize the heap with little wastage, but operations on the heap are serialized so the design doesn't scale well. At the other extreme a malloc allocator can provide a private heap for each thread. Malloc operations that can be satisfied from a thread's private heap don't require synchronization and have low latency. When the number of threads grows large, however, the amount of memory reserved in per-thread

heaps can become unreasonable. Various solutions have been suggested, such as adaptive heap sizing or trying to minimize the number of heaps in circulation by handing off heaps between threads as they block and unblock.

In the middle, we find Hoard [1], which limits the number of heaps by associating multiple threads with a fixed number of local heaps. A thread is associated with a local heap by a hashing function based on the thread's ID. The number of heaps in Hoard is proportional to the number of CPUs. Since the local heaps are shared, synchronization is required for malloc and free operations. To avoid contention on heaps, Hoard provides more heaps than processors.

LFMalloc is similar to Hoard version 2.1.0 - it uses many of the same data structures and policies as Hoard but needs only one heap per CPU. Malloc and free operations on shared per-CPU heaps don't usually require locking or atomic operations. Allocating threads know precisely which processor they are running on, resulting in improved data locality and affinity. Local heap metadata tends to stay resident in the CPU, even though multiple threads will operate on the metadata.

2 Related Work

Much research has been done in the area of kernel-assisted non-blocking synchronization. In [4] and [5] Bershad introduces *restartable atomic sequences*. Mosberger et al. [2][3] and Moran et al. [6] did related work. Johnson and Harathi describe *interruptible critical sections* in [29]. Takada and Sakamura describe *abortable critical sections* in [28]. *Scheduler activations*, in [7] and [8], make kernel scheduling decisions visible to user programs.

A primary influence was the paper by Shivers et al. [9] on atomic heap transactions and fine-grained interrupts. In that work, the authors describe their efforts to develop efficient allocation services for use in an operating system written in ML, how these services are best represented as restartable transactions, and how they interact well with the kernel's need to handle interrupts efficiently. A major influence on their project is the work done at MIT on the ITS operating system. That operating system used a technique termed PCLSR [10] that ensured that, when any thread has examined another thread's state, the observed state was consistent. The basic mechanism was a set of protocols to roll a thread's state either forward or backward to a consistent point at which they could be suspended.

Hudson et al. [11] propose clever allocation sequences that use the IA64's predicate registers and predicated instructions to annul interrupted critical sections.

Similarly, the functional recovery routines [12] and related mechanisms of the OS/390 MVS operating system allow sophisticated transactions of this type.

We also draw on a large body of malloc research, including that of Hoard. Hoard version 2.1.0 is built on top of Lea's malloc package [13].

Finally, a number of recent papers emphasize allocators that are friendly to multiprocessor systems. These include Bonwick's Vmem allocator [14], McKenney's kernel memory allocator [16], Nakhimovsky's mtmalloc [25], Larson and Krishnan's LKMalloc [23], Golger's PTMalloc [26] and Vee and Hsu's allocator [27].

3 Design and Implementation

LFMalloc consists of a malloc allocator and the MP-RCS subsystem. The allocator, which shares many characteristics with Hoard, uses MP-RCS to safely operate on CPU-local malloc metadata. The MP-RCS subsystem includes a kernel driver, a user-mode notification routine, and the restartable critical sections themselves. The kernel driver monitors migration and preemption, and posts *upcalls* to the user-mode notification routine. The notification routine (a) tracks which processor a thread is running on, and (b) aborts and recovers from any interrupted critical sections. A restartable critical section is a block of code that updates CPU-specific data. Collectively, we refer to the driver and the notification routine as the critical execution manager. The initial implementation of LFMalloc is on SPARC™ Solaris.

3.1 MP-RCS

MP-RCS permits multiple threads to operate on CPU-specific data in a consistent manner - without interference. MP-RCS doesn't presume any particular thread model - threads may be preemptively scheduled and may migrate between processors at any time.

In the general case, interference can come from either preemption or from threads running concurrently on other CPUs. Since restartable critical sections operate only on CPU-specific data, we need only be concerned with preemption. For a uniprocessor system, interference is solely the result of kernel preemption.

Techniques to avoid interference fall into two major categories: mutual exclusion-based or lock-free [30] [17][18]. Mutual exclusion is usually implemented with locks. It is pessimistic – mutual exclusion techniques prevent interference. If a thread holding a traditional lock is preempted, other threads will be denied access to the critical section, resulting in convoys, excessive context switching, and the potential for priority inversion.

Lock-free operations, like compare-and-swap (CAS), load-locked/store-conditional (LL-SC) and MP-RCS are optimistic – they detect and recover from interference. A transaction fetches data, computes a provisional value and attempts to *commit* the update. If the update fails, the program must retry the transaction. Unlike locking techniques, lock-free operations behave gracefully under preemption. If a thread is interrupted in a critical section, other threads are permitted to enter and pass through the critical section. When the interrupted thread is eventually rescheduled it will restart the critical section. Since the thread is at the beginning of a full quantum, it will normally be able to complete the critical section without further quantum-based preemption. Lock-free operations are also immune to deadlock. Note that the term *critical section* is usually applied to mutual exclusion, but in this paper we mean it in the more general sense of *atomic sequence*.

Under MP-RCS, if a thread executing in a restartable critical section either (a) migrates to another CPU, or (b) is preempted, and other related threads run on that processor, then the interrupted operation may be ”stale” and must not be permitted to resume. To avoid interference, the kernel driver posts an upcall to a user-mode recovery routine. The upcall, which runs the next time the thread returns to user-mode, aborts the interrupted operation and restarts the critical section by transferring control to the first instruction in the critical section. A thread that has been preempted and may hold stale state is not permitted to commit.

Figure 2 describes the control flow used by restartable critical sections. Table 1 provides a detailed example of the actions that take place during an interrupted critical section.

3.1.1 Restartable Critical Sections

A restartable critical section is simply of block of code. The start, end, and restart addresses of all critical sections are known to the notification routine.

A restartable critical section proceeds without locks or atomic instructions, using simple load and store instructions. Restartable critical sections take the following form: (a) fetch the processor ID from thread-local storage and locate CPU-specific data, (b) optionally fetch one or more CPU-specific variables and compute a provisional value, (c) attempt to store that new value into a CPU-specific variable. As seen in Figure 2, restartable critical sections always end with a store instruction – we say the store *commits* the transaction.

3.1.2 Kernel Driver

The kernel driver delivers *selective* preemption notification. If a thread is preempted, and other unrelated threads run, then when the original thread is scheduled onto a CPU there is no risk of interference, so the kernel does not post an upcall. Also, by convention, we don’t permit blocking system calls within a critical section, so the kernel can forego notification to threads that voluntarily context switch.

The driver makes kernel scheduling decisions, which are normally hidden, visible to user-mode threads. The driver doesn’t change scheduling policies or decisions in any manner but instead informs the affected threads. It doesn’t change which threads will run or when they will run but only where they will resume: in a notification routine or at the original interrupted instruction. Upcalls are deferred until the thread runs – at the next transition from kernel to user-mode.

Referring to Figure 2, upcalls are delivered at the *ONPROC* transition. ONPROC is a Solaris term for “on processor” – it is the thread state transition between *ready* and *running*. Solaris internals are described in detail in [19]. Running the notification routine at the *OFFPROC* “off processor” transition is more intuitive and natural, but would violate Solaris scheduling invariants.

The kernel driver posts an upcall by saving the interrupted user-mode instruction pointer, and substituting the address of the user-mode upcall routine. The next time the thread returns to user-mode, it will resume in the upcall routine instead of at the interrupted instruction.

The current driver is conservative; it posts an upcall if there is any potential for interference. The driver is currently unable to determine if a thread was executing within a critical region. Interference will result in an upcall, but not all upcalls indicate true interference. For ease of construction we partitioned responsibility: the kernel driver detects migration and preemption while the notification routine detects and restarts interrupted critical sections.

The kernel driver uses a set of preexisting context switch hooks present in Solaris. Among other things, these hooks are used to load and unload performance counters and hardware device state that might be specific to a thread. The driver merges multiple pending upcalls which can result when a thread is preempted and then migrates. The kernel driver passes the current processor ID and the original interrupted instruction pointer (IP) as arguments into the notification routine. The notification routine will examine the interrupted IP to see if it was within a restartable critical section.

3.1.3 User-mode Notification Routine

The user-mode notification routine aborts and restarts interrupted critical sections, as well as tracks which CPU a thread is currently executing on. Critically, the notification routine executes *before* control could pass

back into an interrupted critical section. This gives the routine the opportunity to vet and, if needed, to abort the critical section.

The kernel passes the current processor ID to the notification routine which stores the value in thread-specific storage. Subsequent restartable critical sections fetch the processor ID from thread-specific storage and locate the appropriate CPU-specific data, such as malloc metadata. Upcalls give us a *precise* processor ID – used inside a restartable critical section, the value will never be stale.

We originally implemented upcalls with UNIX signals, but signals had specific side-effects, such as expedited return from blocking system calls, that proved troublesome. Still, upcalls are similar in spirit to signals.

Finally, to avoid nested upcalls, MP-RCS inhibits notification while an upcall is executing. The upcall hand-shakes with the kernel to indicate that it is finished. Also, upcalls have precedence over asynchronous signals.

3.1.4 Transactional Layer

To make it easier to employ MP-RCS we have developed a small transactional layer written in assembler, but callable from C. The layer is built on top of small restartable critical sections. It supports starting a transaction, conditional loads, and conditional stores. Each thread has a private "interference" flag. Starting a transaction clears the flag. The load and store operators, protected by restartable critical sections, test the flag, and then execute or annul the memory operation, accordingly. The notification routine sets the flag, indicating potential interference.

3.1.5 Discussion

Upcalls and restartable critical sections are time-efficient On a 400 MHz UltraSPARC-II system the latency of an upcall is 13 μsecs. Restarting a critical section entails reexecution of a portion of the critical section. These are wasted cycles, but in practice critical sections are short and upcalls are infrequent. An important factor in the upcall rate is the length of a quantum. If the duration of a restartable critical section approaches the length of a quantum, the critical section will fail to make progress. This same restriction applies to some LL-SC implementations, like that found in the DEC Alpha.

MP-RCS confers other benefits. Restartable critical sections are not vulnerable to the A-B-A problem [18] found in CAS. More subtly, reachable objects are known to be live. Take the example of a linked list with concurrent read and delete operations. We need to ensure that a reader traversing the list doesn't encounter a deleted node. The traditional way to prevent such a race is to lock nodes – such locks are usually associated with the reference, not the object. A restartable critical section can traverse such a list without any locking. If a reader is preempted by a delete operation the reader will restart and never encounter any deleted nodes. [22], with the *Read-Copy Update* protocol, and [15] describe other approaches to the problem of existence guarantees.

MP-RCS does not impose any particular thread model. MP-RCS works equally well with a 1:1 model or a multi-level level model (where multiple logical threads are multiplexed on fewer "real" threads). In addition, MP-RCS operates independently of the relationship between threads and processors; under MP-RCS threads may be bound to specific processors or allowed to migrate freely.

3.2 Malloc and Free

This section briefly describes the Hoard allocator, contrasts Hoard with the LFMalloc allocator and describes the operation of malloc and free in LFMalloc.

3.2.1 Hoard

Hoard has a single global heap and multiple local heaps. Each heap contains a set of superblocks. A superblock consists of a header and an array of fixed size blocks. The superblock header contains a LIFO free list of available blocks. A block contains a pointer to the enclosing superblock, alignment padding, and the data area. The malloc operator allocates a block from a free list and returns the address of the data area.

All superblocks are the same length – a multiple of the system page size - and are allocated from the system. All blocks in a superblock are in same *size-class*. The maximum step between adjacent size-classes is 20%, constraining internal fragmentation to no more than 20%. Hoard handles large requests by constructing a special superblock that contains just one block – this maintains the invariant that all blocks must be contained within a superblock.

3.2.2 Comparing LFMalloc and Hoard

LFMalloc shares all the previous characteristics with Hoard but differs in the following:

1. In LFMalloc a local heap may "own" at most one superblock of a given size-class while the global heap may own multiple superblocks of a given size-class. Hoard permits both global and local heaps to own multiple superblocks of a size-class. Hoard will move superblocks from a local heap to the global heap if the proportion of free blocks to in-use blocks in the local heap passes the *emptiness threshold*. Because LFMalloc limits the number of superblocks of a given size to one, it does not need to take special measures to prevent excessive blocks from accumulating in a local heap. .

2. LFMalloc requires only *C* local heaps (*C* is the number of processors), where Hoard must over-provide heaps to avoid heap lock contention. Hoard works best then the number of threads is less than the number of local heaps.
3. In Hoard, a set of threads, determined by a hash function applied to the thread ID, will share a local heap. During its tenure a thread will use only one local heap. Hoard has the potential for a class of threads, running concurrently on different processors, to map to a single "hot" local heap. In LFMalloc, threads are associated with local heaps based on which processor they are executing on. Concurrent malloc operations are evenly distributed over the set of local heaps. The association between local-heaps and CPUs is fixed in LFMalloc while in Hoard the association varies as threads migrate between CPUs.
4. Since LFMalloc has fewer superblocks in circulation, we increased the superblock size from 8192 to 65536 bytes without appreciably affecting the peak heap utilization.
5. In LFMalloc, each superblock has two free lists: a local free list which is operated on by restartable critical sections and a remote free list protected by a traditional system mutex. The two lists are disjoint. Malloc and free operations on a local heap manipulate the local free list without locks or atomic instructions. A thread allocates blocks from its local heap but can free a block belonging to any heap. Blocks being freed in a non-local heap are added to the remote free list.
6. In malloc, Hoard computes the size-class with a loop. LFMalloc uses a radix map which operates in a constant number of instructions.
7. Superblocks are size-stable. They remain of a fixed size-class until the last in-use block is released. At that time, LFMalloc will return the superblock to the operating system or it will make the superblock available to be "reformatted" – it can be changed to any needed size-class. Hoard never returns superblocks to the operating system.

3.2.3 LFMalloc: Malloc and Free Algorithms

In LFMalloc a superblock is either *online* or *offline*. An online superblock is associated with a processor – it belongs to a local heap. An offline superblock is in the global heap. We say a block is *local* to a thread if the block belongs to a superblock that is associated with the processor on which the thread is running. The block is *remote* if the enclosing superblock is associated with some other processor. Finally, the block is *offline* if it belongs to a superblock in the global heap. A local heap and local free lists of the superblocks it owns are CPU-specific data. A local heap is permanently fixed to its processor. A superblock's local free list is CPU-specific while the superblock remains online.

LFMalloc uses restartable critical sections to operate on local free lists and to attach and detach superblocks from local heaps.

Pseudo-code for malloc (sz)

```
[1]   // RCS {...} delineates a restartable
[2]   // critical section
[3]   if sz is large
[4]       construct and return a large superblock
[5]   i ← SizeClass [sz/8]
[6]   self ← find thread-local storage block
[7]       for the current thread
[8]   Retry:
[9]   RCS {  // try to allocate locally
[10]     H ← LocalHeap[self.processorID]
[11]     S ← H[i]
[12]     b ← pop a block from S.LocalFreeList
[13]  }
[14]  if b != null, return b // fast path
[15]  // Superblock is empty - replace it
[16]  search the global heap for a
[17]      non-full superblock R of size-class i
[18]  if none found,
[19]     construct a new superblock R
[20]  else
[21]     lock R
[22]     Merge R.RemoteFreeList into
[23]         R.LocalFreeList
[24]     unlock R
[25]  // R goes online, S goes offline
[26]  Try to install R in H, replacing S
[27]  If failed,
[28]      move R to global heap
[29]  else
[30]      move S to global heap
[31]  Goto Retry
```

Following the pseudo-code above, malloc finds the local heap *H* associated with the current processor using the processor ID supplied by the notification routine. Using the local heap, malloc locates the superblock *S* of the appropriate size-class and attempts to unlink a block *b* from the local free list. If no superblock of the size-class exists, or if the superblock's local free list is empty, malloc will find or construct a new superblock and then attempt to install it in the local heap. Finally, the previous superblock, if it exists, is returned to the global heap – it transitions offline.

Pseudo-code for free (b)

```
[1]   if the block is large,
[2]      destroy the superblock and return
[3]   S ← superblock containing b
```

```
[4]   RCS {
[5]      If block is not local, goto Slow
[6]      add b to S.LocalFreeList
[7]   }
[8]   return   // fast path
[9]   Slow:
[10]  lock S  // S is remote or offline
[11]  add b to S.RemoteFreeList
[12]  unlock S
[13]  if the number of in use blocks in S
[14]    reaches 0 and the number of entirely free
[15]    superblocks in global heap exceeds a
[16]    limit,
[17]        destroy the superblock s
[18]  return
```

To free a local block, the free operator pushes the block onto the superblock's local free list. To free a non-local block, the free operator must lock the superblock, add the block to the remote free list, and then unlock the superblock.

A malloc request that can be satisfied from the local heap is lock-free. Likewise, freeing a local block is lock-free.

Local heap metadata is accessed exclusively by one processor. The association between processors and local heaps remains fixed even as various threads run on the processors. This supports our claim of good locality. Because superblocks consist of a contiguous set of blocks, and a superblock stays associated with a processor until it becomes empty, our design minimizes false sharing.

Malloc and free operations tend to occur on the same processor. The only exceptions are when a thread migrates between the malloc and free, or when the allocating thread hands off the block to be freed by some other thread. This means that most free operations are on local blocks and take the "fast path", show in free, above.

We believe the arguments in [1] regarding Hoard's maximum fragmentation and bounded blowup apply to LFMalloc as well.

4 Results

In this section we describe our results. We compare the scores of a set of malloc benchmarks, varying the number of threads used in the benchmarks, and the underlying malloc package. We also pick a single micro-benchmark, and, holding the number of threads constant, perform a detailed analysis of the execution of that program with various malloc implementations.

4.1 Methodology and Environment

All tests were performed on a 16-processor Sun E6500 system, with 400 MHz UltraSPARC-II processors. The system had 16 Gb or RAM and was running Solaris 8. Each processor had a 16 Kb L1 cache and a 4 Mb direct-mapped L2 cache. All tests were run in the timeshare (TS) scheduling class. All components, except those provided in binary form by the system, were compiled in 32-bit mode with GCC 2.95.3. Hoard was built at optimization level 6 (-O6), and the remainder at optimization level 4 (-O4).

We ran all tests with the new *liblwp* libthread package (liblwp can be found in /lib/liblwp in Solaris 8). Liblwp uses a single-level threading model where all user-level threads are bound 1:1 to LWPs (an LWP is a light-weight process, or kernel thread). In Solaris 9, this 1:1 threading model will become the default. Hoard version 2.1.0 hashes the current thread's LWP ID to locate a local heap. Hoard queries a thread's LWP ID by calling the undocumented lwp_self function. In the default libthread on Solaris 8 and lower, lwp_self simply fetches the current LWP ID from thread-local storage. In the liblwp, lwp_self calls the _lwp_self system call, which is significantly slower. We adapted Hoard to call thr_self instead of lwp_self. This is benign, as under liblwp or Solaris 9 libthread, a thread's ID is always the same as its LWP ID.

We built all C++ benchmarks with mallocwrap.cpp from Hoard. Mallocwrap replaces the *__builtin new* and *delete* operators with simple wrappers that call malloc and free, respectively.

All malloc packages, including LFMalloc, were built as dynamically linked libraries.

We compared LFMalloc to the following: Hoard 2.1.0, the default Solaris libc malloc, and mtmalloc. Mtmalloc, described as an "MT hot memory allocator", is delivered in binary form with Solaris. Libc malloc is highly serialized – all operations are protected by a single mutex.

A standard suite of multithreaded malloc benchmarks does not yet exist. Instead, we recapitulate some of the micro-benchmark results reported in previous papers. We use *Larson*, from [23] which can be found in the Hoard source release. Larson simulates a server. We discovered that Larson scores are sensitive to scheduling policy. Worker threads start and make progress outside the timing interval – before the main threads starts timing. If the main thread stalls, as can happen with a large number of worker threads, the Larson scores will be biased upward. To remedy the problem we added a barrier in the beginning of the worker's code path. The main thread starts timing and only then drops the barrier. All Larson scores reported in this paper reflect the modified version.

We also report results for *linux-scalability* [24] and Hoard's own *threadtest*. Finally, we introduce our own

micro-benchmark, *mmicro*. In writing mmicro, we factored out influences such as scheduling decisions and fairness. The program measures the throughput of a set of threads – it does not measure latency. Given a fixed time interval, it measures the aggregate amount of work (malloc-free pairs) accomplished in the allotted time. No threads are created or destroyed in the timing interval. The benchmark reports malloc-free pairs/second.

We use Solaris' *cputrack* utility to collect data from the processor's performance counters. Cputrack perturbs scheduling decisions. Mmicro scores are stable under cputrack – they don't change. However, Larson, linux-scalability and threadtest are not stable under cputrack.

4.2 Speedup and Scalability

We ran Larson, threadtest, linux-scalability and mmicro on libc, mtmalloc, Hoard and LFMalloc, varying the number of concurrent threads from 1…16, 18, 20, 22, 24, 26, 28, 30, 32, 64, 128 and 256. The results are presented in Figure 1 (a)-(d). All scores are normalized to libc – the Y-axis is given as the speedup multiple over libc. For all data points, LFMalloc was faster than libc, mtmalloc and Hoard.

The performance drop for Larson, Figure 1(a), at higher numbers of threads may be due to scalability obstacles outside the allocator. As an experiment, we modified Larson and removed the malloc-free operations from the loop of the worker thread. The modified program had a peak score of 153 times libc at 24 threads. At higher thread counts the performance dropped off.

In Figure 1 (e), we show the scalability characteristics of the various malloc packages by plotting the Y-axis as the speedup relative to one thread. An ideal benchmark running on a perfectly scalable allocator would have Speedup(t) = t, for t up to the number of processors. Both Hoard and LFMalloc show excellent scalability – concurrent allocator operations don't impede each other.

4.3 Execution Characteristics

To better understand why LFMalloc performs well, we picked a single micro-benchmark, mmicro, and set of parameters and then ran on libc, mtmalloc, Hoard and LFMalloc, using cputrack and other tools to gather detailed data. We present that information in Table 2, Execution Characteristics. As can be seen in the table, programs running on LFMalloc have excellent cache and interconnect characteristics, supporting our previous claims. Notable is that mmicro on LFMalloc runs at .909 cycles per instruction, realizing the potential of the UltraSPARC's superscalar design. Care should be taken when examining the libc results. Under libc, mmicro was only able to achieve 14.1% CPU utilization – it could only saturate about two of the 16 processors. Threads in the libc trial spent most of their time blocked on the single heap mutex.

The raw translation lookaside buffer (TLB) miss rate was higher under LFMalloc than Hoard, but in the timing interval LFMalloc executed nearly 10 times more malloc-free pairs. The TLB miss rate per allocation pair is lower for LFMalloc.

In Table 3, we explore which aspect of our algorithm – fast synchronization or perfect CPU-identity – contribute the most to performance. As seen in the table, CPU-identity is critical to scalability. MP-RCS-based synchronization is helpful, but only improves performance about 2 times as compared to a traditional Solaris mutex.

Finally, the heap "footprint" for LFMalloc is generally very close to that of Hoard's. We ran threadtest at 1…16, 18, 20, 22, 24, 26, 28, 30, 32, 64, 128 and 256 threads. Hoard's peak heap utilization ranged from 6.591 Mb to 7.430 Mb while LFMalloc's peak heap utilization ranged from 4.980 Mb to 5.898 Mb

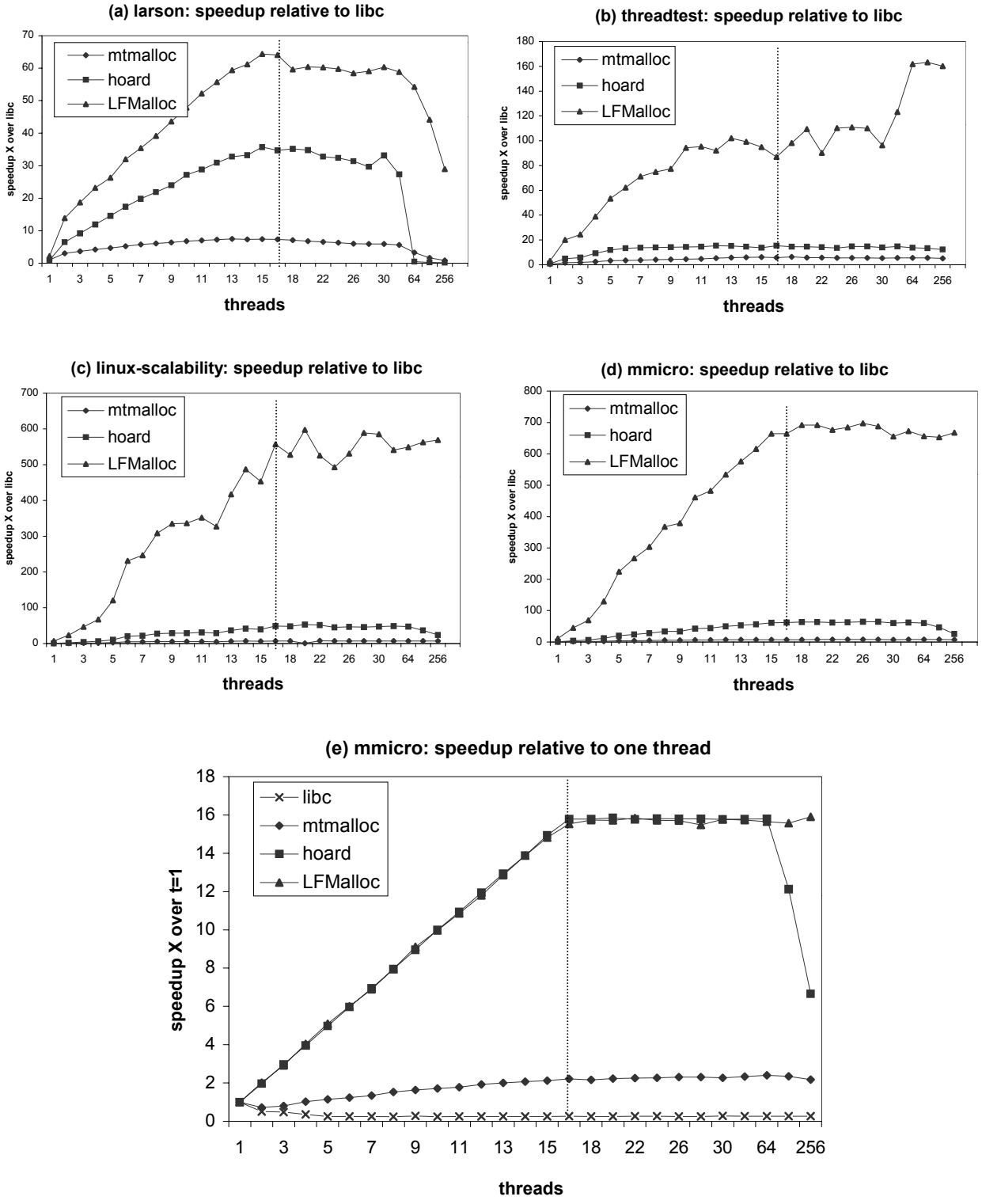

Figure 1: Scalability and Speedup

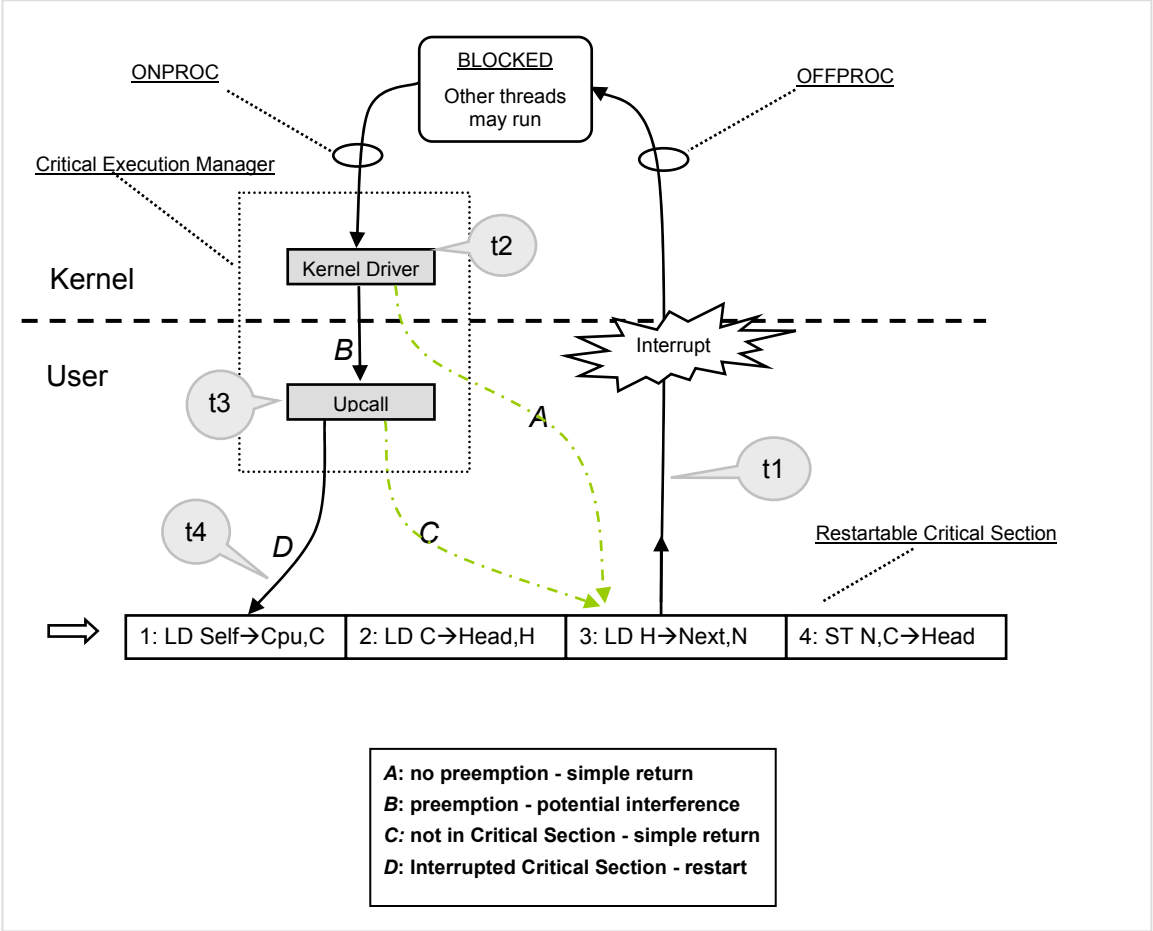

Figure 2: Control Flow for Restartable Critical Sections

Time	Action
t1	The thread enters the Restartable Critical Section. The critical section will attempt to pop a node off a CPU-specific linked-list. *Self* is a pointer to thread-local-storage and *Self→Cpu* is a pointer to CPU-specific-data. The thread executes instructions #1 and #2 and is then interrupted – the interrupted user-mode instruction pointer (IP) points to instruction #3. Control passes into the kernel. The kernel deschedules the thread – the thread goes *OFFPROC*. Other related threads run, potentially rendering the saved *H* and *C* register values stale.
t2	The thread is eventually rescheduled – it goes *ONPROC*. The kernel calls a routine in the driver, previously registered as a context switch callback, to notify the driver that a thread of interest is making the transition from *ready* state to *running* state. The driver checks to see if (a) the thread has migrated to a different CPU since it last ran, or, (b) if any related threads have run on this CPU since the current thread last went OFFPROC. We'll presume (b) is the case, so there is a risk of interference. The driver posts an upcall – the driver saves the interrupted IP and substitutes the address of the notification routine. The thread eventually returns to user-mode, but restarts in the notification instead of at instruction #3.
t3	The notification routine updates *Self->Cpu* to refer to the appropriate CPU (the thread may have migrated). Next, it retrieves the interrupted IP, which refers to instruction #3, and checks to see if the IP was within a critical section. In our example the routine adjusts the IP to point to the beginning of the critical section (#1). For convenience, we currently write critical sections in a non-destructive idiom – it does not destroy any input register arguments. If a critical section is interrupted the critical execution manager can simply restart it. The notification routine also checks for interrupted critical sections interrupted by signals and arranges for them to restart as well.
t4	The upcall unwinds, passing control to the start of the interrupted critical section, causing it to restart.

Table 1: Example of an Interrupted Critical Section

.

Table 2: Execution Characteristics

	libc	mtmalloc	Hoard	LFMalloc
# MP-RCS migration upcalls				473
# MP-RCS preemption upcalls				2705
# MP-RCS of interrupted critical sections				812
Peak heap size (Mb)	unknown	unknown	3.5 Mb	4.1 Mb
Malloc-free latency (ηsecs per pair)	166,525 ηsecs	19,230 ηsecs	2,251 ηsecs	227 ηsecs
Overall malloc-free throughput (pairs/sec)	102,111	844,454	6,589,446	70,749,108
User-mode CPU utilization (%)	14.1%	85.6%	98.6%	98.7%
Context switch rate (switches/second) Includes both voluntary switches caused by blocking and involuntary switches.	1043	8537	657	412
User-mode cycles per instruction (CPI)	1.985	3.106	1.505	.909
User-mode snoops (cycles/second) - snoop invalidations caused by cache line migration	3,529,464/sec	22,484,855/sec	137,625/sec	41,452/sec
User-mode Snoops / malloc-free pair	33.382	26.112	0.021037	0.000583
System DTLB miss rate (exceptions/second)	4,965/sec	20,8114/sec	4,870/sec	10,028/sec
User-mode L1 miss rate	2.544%	6.668%	7.325%	1.699%
User-mode L2 miss rate	33.258%	24.139%	0.015%	0.008%
User-mode L1 Cache misses / malloc-free pair	139.087	36.022	10.333	.407

Execution characteristics of "mmicro 64 :10000 200" – a 10 second run with 64 threads, each thread loops, malloc()ing 64 blocks of length 200 and then free()s the blocks in same order in which they were allocated.

Table 3: Contributions of Synchronization and Perfect Processor Identity

Synchronization mechanism	Local heap selection	1 thread	16 threads	256 threads
Restartable Critical Sections	By processor ID via upcall	4548	69911	72163
Per-CPU Solaris mutex	By processor ID via upcall	2124	33020	30678
Per-CPU Solaris mutex	Hoard-style hash of thread ID	2012	23742	2060

"mmicro T :10000 64" scores - malloc-free pairs/msec – for 1, 16 and 256 threads showing the effect of different synchronization and heap selection mechanisms.

5 Conclusion

We have introduced Multi-Processor Restartable Critical Sections and demonstrated their utility by developing a scalable malloc allocator. Despite using shared heaps the allocator can usually operate lock-free. The allocator has excellent execution characteristics, particularly on multiprocessor systems.

MP-RCS is novel in that it works on multiprocessors but for CPU-specific data. Other kernel-assisted synchronization techniques work only on uniprocessors.

MP-RCS provides two facilities: processor location awareness and restartable critical sections. Restartable critical sections require processor location awareness, but the converse is not true. We have shown that of these, processor location awareness is more useful. Finally, MP-RCS operates without limiting kernel preemption policies, and is independent of threading models and applications.

5.1 Future Work

For uniprocessors and for a process with all related threads bound to one CPU, restartable critical sections can emulate CAS; all data is CPU-specific. Our results show the emulated form to be 1.8 times faster than the actual hardware instruction.

We intend to adopt *commit records* [29] to extend the MP-RCS facility to multi-word transactions.

By creating superblocks of size *B,* (*B* is a power of two*)* aligned on virtual addresses that are a multiple of *B*, we can eliminate the block header, saving 8 bytes per-block. Given a block, the free operator can then locate the enclosing superblock by simply masking the address of the block.

To reduce the cost and frequency of upcalls, we intend to establish a convention so the kernel can determine if an interrupted thread was executing in a critical section. We have also prototyped a simple free block cache "stacked" on top of the existing malloc subsystem.

We have adapted a Java™ virtual machine to operate on per-CPU heap allocation buffers with MP-RCS operators. In addition to heap allocation, we believe that MP-RCS can offer fast logging write barriers, and the potential for fast mutator thread suspension, needed by stop-the-world garbage collectors. (A thread targeted for suspension would block itself in the upcall).

We would like to collect data on NUMA systems – LFMalloc should perform well in that environment. We also intend to retrofit Hoard with MP-RCS components. In particular, we will replace Hoard's `hoardGetThreadID` thread-to-heap mapping function with one based on the precise processor ID.

5.2 Acknowledgements

We would like to thank Chris Phillips for pointing out IBM's use of function recovery routines and service requests. We would also like to thank Paula J. Bishop for helpful comments.

6 References

[1] Emery Berger, Kathryn McKinley, Robert Blumofe and Paul Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *ASPLOS-IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1997.

[2] David Mosberger, Peter Druschel and Larry L. Peterson. A Fast and General Software Solution to Mutual Exclusion on Uniprocessors. *Technical Report 94-07, Department of Computer Science, University of Arizona*. June 1994.

[3] David Mosberger, Peter Druschel and Larry L. Peterson. Implementing Atomic Sequences on Uniprocessors Using Rollforward. In *Software – Practice & Experience. Vol. 26, No. 1*. January 1996.

[4] Brian N. Bershad. Practical Considerations for Non-Blocking Concurrent Objects. In *Proc. International Conference on Distributed Computing Systems*, (ICDCS). May 1993.

[5] Brian N. Bershad. Fast Mutual Exclusion for Uniprocessors. In *ASPLOS-V: Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1992.

[6] William Moran and Farnham Jahanian. Cheap Mutual Exclusion. In *Proc. USENIX Technical Conference*. 1992.

[7] Christopher Small and Margo Seltzer. Scheduler Activations on BSD: Sharing Thread Management State Between Kernel and Application. *Harvard Computer Systems Laboratory Technical Report* TR-31-95. 1995

[8] T. Anderson, B. Bershad, E. Lazowska and H. Levy. Scheduler Activations: Effective Kernel Support for User-Level Management of Parallelism. *ACM Transactions on Computer Systems,* 10(1). 1992.

[9] O. Shivers, J. Clark and R. McGrath. Atomic Heap Transactions and Fine-grain Interrupts. In *Proc. International Conference on Functional Programming* (ICFP). 1999.

[10] Alan Bawden. PCLSRing: Keeping Process State Modular. Available at ftp://ftp.ai.mit.edu/pub/alan/pclsr.memo. 1993.

[11] Richard L. Hudson, J. Eliot B. Moss, Sreenivas Subramoney and Weldon Washburn. Cycles to Recycle: Garbage Collection on the IA-64. In Tony Hoskings, editor, ISMM 2000, *Proc. Second International Symposium on Memory Management, 36(1). of the ACM SIGPLAN Notices*. 2000.

[12] IBM OS/390 MVS Programming: Resource Recovery. 1998. GC28-1739-03.

[13] Doug Lea. A Memory Allocator. Available at http://g.oswego.edu/dl/html/malloc.html.

[14] Jeff Bonwick and Jonathan Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *Proc. USENIX Technical Conference*. 2001.

[15] Ben Gamsa, Orran Krieger, Jonathan Appavoo and Michael Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proc. of Symp. On Operating System Design and Implementation*. (OSDI-III). 1999.

[16] Paul McKenney, Jack Slingwine and Phil Krueger. Experience with a Parallel Memory Allocator. In *Software – Practice & Experience*. Vol. 31. 2001.

[17] Michael Greenwald. Ph. D. Thesis. Non-Blocking Synchronization and System Design. Stanford University, 1999.

[18] John Valois. Lock-Free Data Structures. Ph. D. Thesis, Rensselaer Polytechnic Institute, 1995.

[19] Jim Maura and Richard McDougall. Solaris™ Internals: Core Kernel Architecture. Sun Microsystems Press. Prentice-Hall. 2001.

[20] Hans-J. Boehm. Fast Multiprocessor Memory Allocation and Garbage Collection. *HP Labs Technical Report HPL-2000-165*. 2000.

[21] David L. Weaver, Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. SPARC International, Prentice-Hall, 1994.

[22] Paul McKenney and John Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *10th IASTED International Conference on Parallel and Distributed Computing Systems*. (PDCS'98). 1998

[23] P. Larson and M. Krishnan. Memory Allocation for Long-Running Server Applications. In *International Symp. On Memory Management* (ISMM 98). 1988.

[24] Chuck Lever and David Boreham. Malloc() Performance in a Multithreaded Linux Environment. In *USENIX Technical Conference*, 2000.

[25] Greg Nakhimovsky. Improving Scalability of Multithreaded Dynamic Memory Allocation. In *Dr. Dobbs Journal*, #326. July 2001.

[26] Wolfram Golger. Dynamic Memory Allocator Implementations in Linux System Binaries. Available at www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html. Site visited January 2002.

[27] Voon-Yee Vee and Wen-Jing Hsu. A Scalable and Efficient Storage Allocator on Shared-Memory Multiprocessors. In *International Symp. of Parallel Architectures, Algorithms, and Networks* (I-SPAN 99). 1999.

[28] Hiroaki Takada and Ken Sakamura. Real-Time Synchronization Protocols with Abortable Critical Sections. In *Proc. of the First Workshop on Real-Time Systems and Applications*. (RTCSA). 1994.

[29] Theodore Johnson and Krishna Harathi. Interruptible Critical Sections. *Dept. of Computer Science, University of Florida. Technical Report* TR94-007. 1994.

[30] Maurice Herlihy. A Method for Implementing Highly Concurrent Data Objects. ACM *Transactions on Programming Languages and Systems* (TOPLAS) 15(5), November 1993.