Supporting Per-processor Local-allocation Buffers Using Multi-processor Restartable Critical Sections

David Dice, Alex Garthwaite, and Derek White

Supporting Per-processor Local-allocation Buffers Using Multi-processor Restartable Critical Sections

David Dice, Alex Garthwaite, and Derek White

SMLI TR-2004-126 March 2004

Abstract:

One challenge for runtime systems like the Java™ platform that depend on garbage collection is the ability to scale performance with the number of allocating threads. As the number of such threads grows, allocation of memory in the heap becomes a point of contention. To relieve this contention, many collectors allow threads to preallocate blocks of memory from the shared heap. These per-thread local-allocation buffers (LABs) allow threads to allocate most objects without any need for further synchronization. As the number of threads exceeds the number of processors, however, the cost of committing memory to local-allocation buffers becomes a challenge and sophisticated LAB-sizing policies must be employed.

To reduce this complexity, we implement support for local-allocation buffers associated with processors instead of threads using multiprocess restartable critical sections (MP-RCSs). MP-RCSs allow threads to manipulate processor-local data safely. To support processor-specific transactions in dynamically generated code, we have developed a novel mechanism for implementing these critical sections that is efficient, allows preemption-notification at known points in a given critical section, and does not require explicit registration of the critical sections. Finally, we analyze the performance of per-processor LABs and show that, for highly threaded applications, this approach performs better than per-thread LABs, and allows for simpler LAB-sizing policies.

microsystems

M/S MTV29-01
2600 Casey Avenue
Mountain View, CA 94043

email addresses:
dave.dice@sun.com
alex.garthwaite@sun.com
derek.white@sun.com

Supporting Per-processor Local-allocation Buffers Using Multi-processor Restartable Critical Sections

Dave Dice, Alex Garthwaite, and Derek White

1 Introduction

One central challenge of implementing scalable multi-threaded programs is how to efficiently manage shared resources, such as memory. The traditional way to manage shared resources is to use one of the blocking synchronization operations provided by the operating systems, such as mutexes. But if the shared resource is frequently used by many threads, the use of blocking synchronization can quickly become a bottleneck. One common solution is to partition the resources among threads, so each thread has a thread-local portion of the resource which may managed without synchronization. The main issue then becomes devising the most efficient partitioning of the global resource, so that resources are available to the threads that need them, and not wasted on the threads that do not. Another solution on uniprocessors is to use kernel-assisted non-blocking synchronization, such as restartable atomic sequences. These schemes don't prevent several threads from starting a transaction for a shared resource at the same time, but they detect contention, and cause interrupted transactions to either roll-forward or roll-back to a consistent state.

In this paper, we describe the effects of replacing the thread-local object allocation scheme in a Java™ virtual machine (JVM) with a processor-local allocation scheme. This processor-local scheme is based on recent work with multi-processor restartable critical sections (MP-RCS). Compared to thread-local management, it enables us to bound the partitioning of the global resource to at most the number of processors. Compared to other, possibly non-blocking, synchronization schemes that either map threads to some bounded partitioning of the global resource or allow all threads to interact with a single resource, our association of the partitioned resources directly with processors eliminates contention for those partitioned resources.

Multi-processor restartable critical sections allow user-level threads to manipulate processor-local data safely, without using atomic instructions. The key to these processor-local transactions is the ability to notify a thread executing in a critical section that it may have been preempted. To avoid complicating the interface and contract with the scheduler, our implementation of MP-RCS relies only on notification as threads begin running on processors (ON-PROC notification). Through ON-PROC notification, the flow of control in the thread may be redirected to a recovery routine rather than allowing the MP-RCS transaction to commit.

The purpose of this report is to investigate a new and simple mechanism for implementing the MP-RCS service, to apply this mechanism to a new set of challenges, and to show how MP-RCS transactions may be written in high-level languages.

In prior work [DG02], we described an alternate implementation of the MP-RCS service: the *upcall* mechanism. This mechanism had several good properties:

- overhead is only incurred for those threads in transactions at the time they come back on-processor, and
- the upcall mechanism is general enough that threads may use it to directly maintain thread-specific references to processor-local data.

However, this mechanism also required the maintenance of a mapping between processors and threads in the kernel, as well as a somewhat complicated upcall mechanism—including a downcall to recover saved state. Most importantly, this mechanism becomes difficult to manage when multiple services wish to use MP-RCS transactions for different purposes. Our current approach is meant to address all of these issues by hiding some details behind a new interface, and exposing other details to applications and libraries to allow better control over MP-RCS transactions.

We used this mechanism to implement a highly scalable malloc library and demonstrated that the resulting library outperformed other implementations by several orders of magnitude. Much of the performance improvement arose from the fact that we eliminated the use of memory-barriers, atomic instructions, and mutexes from most of the commonly executed paths in the malloc interface. In this study, we will apply the MP-RCS service to a new problem area: the implementation of local-allocation buffers (LABs) in a garbage-collected system. We will describe local-allocation buffers in more detail in the next section.

Local-allocation buffers represent more of a challenge in two senses. First, because of their effect on the scaling of highly threaded applications, existing implementations have been highly tuned. In particular, existing LAB sizing policies take into account factors like the number of allocating threads, the amount of memory available for allocation, and the rate of allocation to adjust the sizes of LABs given to threads. In doing so, these policies balance the need to avoid contention during allocation with the cost of wasting space in unused portions of local-allocation buffers. Second, the allocation of memory to a local-allocation buffer is typically achieved with a single atomic instruction. Subsequent allocations of objects from that LAB are then achieved simply by bumping a pointer with a store instruction. As such, it is difficult to improve on the performance of allocation. Where the deployment of LABs remains a challenge, however, is in the sizing policies one may choose.

Our goal in this study is to show that local-allocation buffers may efficiently be associated with processors rather than threads and that doing so removes much of the need to carefully tune LAB sizing policies. This reduction in complexity means both that a wider range of collection techniques may be employed with highly threaded applications on multiprocessors and that the behavior of the Java™ platform is more resilient to the availability of a wider range of computational resources. One insight gained from this study has been the fact that depending on the number of threads and the number of processors, per-thread and per-processor local-allocation buffers each offer a different set of benefits. From this insight, we implemented the capability of flipping between these two modes of mapping local-allocation buffers.

To demonstrate the effectiveness and efficiency of per-processor local-allocation buffers, we examine the performance of several applications written in the Java™ programming language that cover a spectrum of behavior. Using these results, we argue that MP-RCS may be effectively applied to the implementation of local-allocation buffers and that doing so does have benefits.

1.1 Roadmap

In the next section, we outline the problem we have chosen to address with MP-RCS in this report: the sizing of local-allocation buffers in a garbage-collected Java virtual machine. We then describe our new implementation of MP-RCS using the `%asi` register and present a library of MP-RCS primitive operations—the critical-section interface—that allow MP-RCS transactions to be written in high-level languages. Following this section, we examine how the critical-section interface may be used to implement per-processor LABs. We then offer performance results demonstrating the efficiency of per-processor LABs. Finally, we conclude and offer future areas of application for MP-RCS transactions.

2 Local-allocation Buffers

One of the services of a garbage collector is the allocation of properly initialized memory from the heap [Jon96]. Depending on the collection technique employed, free memory is distinguished from allocated memory either by maintaining the former in free-lists—typically found in non-moving collectors—or by maintaining a pointer—a *free-pointer*—to a boundary between allocated and free memory. Further, the garbage-collected heap may be organized into several subheaps, each with its own mechanisms for allocating memory. Because the memory available for allocation is a shared resource, care must be taken to allow multiple, independent threads to perform allocations concurrently.

One simple approach to coordinating this access is to use a lock. However, because application threads frequently allocate memory for objects from the heap, this allocation lock becomes a point of contention and the threads tend to exacerbate the contention through convoying effects.

Fortunately, most structures used to support allocation of free memory in the heap are relatively simple, consisting either of singly-linked lists or of a free-pointer. Since most modern processors provide atomic instructions like compare-and-swap (CAS) or load-linked (LL)/store-conditional (SC), these allocation structures are often designed in a manner so that one or two atomic instructions may be employed to safely allocate memory. Nonetheless, access to these data structures often becomes a scaling bottleneck in highly threaded applications as multiple threads contend to adjust them.

One mechanism commonly employed to reduce this contention is the *local-allocation buffer* or LAB. Typically, each thread is given one or more LABs from the heap. Once assigned to a thread, memory for individual objects may then be allocated by that thread without further synchronization with the other threads. Figure 1 shows how LABs may be used in a garbage-collected heap. In addition to reducing contention for allocation services, LABs provide a number of other benefits. By assigning local-allocation buffers to threads on an exclusive basis,

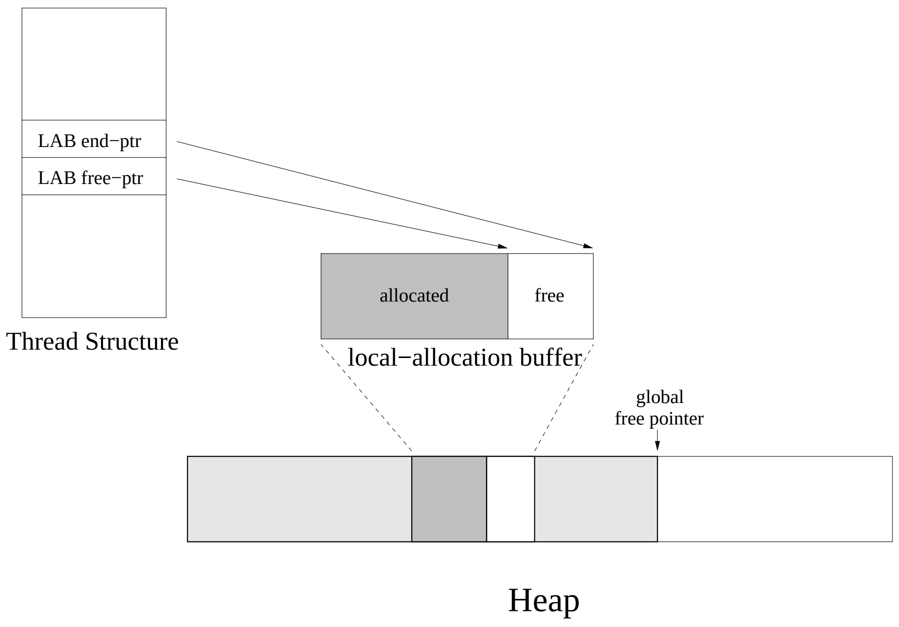


Figure 1: Partially filled per-thread local-allocation buffer

the objects allocated by those threads tend to be collocated in close proximity. Further, larger LABs reduce the cost of internal fragmentation due to an inability to allocate objects to fill each buffer. Finally, some collection strategies work best if threads may allocate out of relatively large local-allocation buffers—for example, as large as 64KB to 256KB per LAB.

Local-allocation buffers alleviate contention for allocating memory directly from the heap by reducing the frequency of such allocations. Clearly, the larger each LAB, the greater the effect. In the case of highly threaded applications, this overcommitment of memory to threads can lead to a different kind of scaling problem. If threads run frequently enough and are not able to allocate all of their LABs' memory before the heap is exhausted, the result may be more frequent garbage collections. The challenge is to balance these two effects, contention for allocation and frequency of garbage collection, to allow applications to scale as well as possible on the available processors and with the available memory.

To make this discussion more concrete, let us consider a particular Java platform, the ResearchVM.[1] Further, let us concentrate on a particular heap config-

[1]The ResearchVM was formerly known as the ExactVM and is available as the virtual ma-

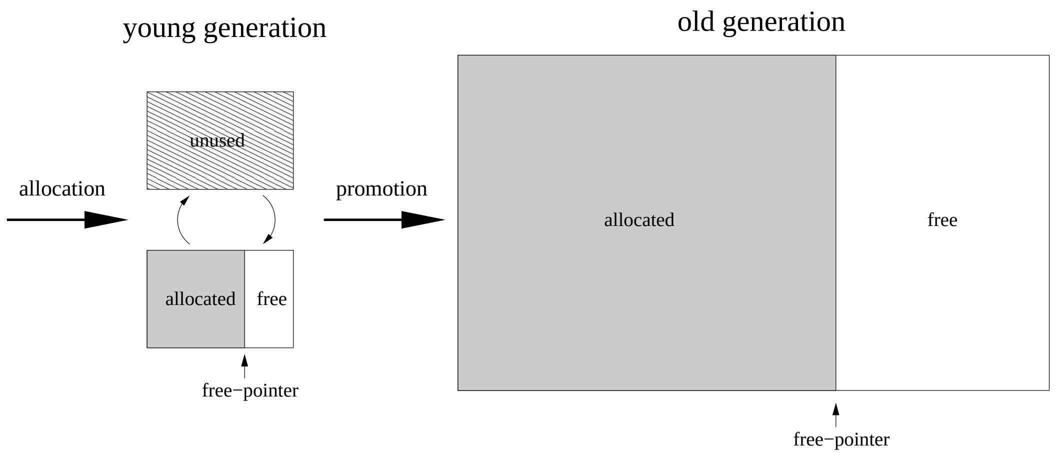


Figure 2: Two-generation heap

uration shown in figure 2: a two-generation heap with the younger generation organized as a pair of semispaces using copying collection and the older generation managed by a highly tuned mark-compact collector [Jon96]. Each of the two generations maintains a free-pointer which is atomically incremented to allocate memory—either for LABs or for individual objects—from a contiguous range of memory. Application threads typically allocate objects in the young generation. When this generation's free memory is exhausted, the application threads are suspended and a minor collection is performed. During this minor collection, surviving objects may be promoted to the old generation.

In the ResearchVM, application threads may allocate memory for objects in either generation. This approach allows most allocations to occur in the young generation while allowing some exceptions for known likely-long-lived objects to be placed directly in the old generation. Examples of such long-lived objects include `String` objects for interned strings. Because of this policy, each thread maintains for each generation a current local-allocation buffer from which it allocates small objects. Allocation of memory directly from the either generation is performed for one of three reasons:

- to reserve memory for a thread's LAB,
- to allocate an object too large to fit reasonably in a LAB, or

chine in the Java™ 2 SDK 1.2.2. An early description of the memory system in the ResearchVM is [WG98].

- to allocate a small object when not enough memory remains to allocate a suitably sized LAB.

By properly sizing the local-allocation buffers assigned to threads, most objects are allocated without synchronization in LABs, and the set of objects that are considered too large is reduced. The third reason to allocate directly from the heap impacts the efficiency of local-allocation buffers in an unexpected manner. When the heap is almost exhausted, threads that have full LABs must contend for access to the heap. This gives time to threads with unused LABs to allocate their memory. As local-allocation buffers increase in size, the time required in this phase leading up to the next collection also increases. So, the point at which the memory management ceases to hand out local-allocation buffers must trade off the desire to make use of already-reserved LABs and the increased costs of direct allocation. We will return to these trade-offs in the performance section.

The two operations on local-allocation buffers described so far are:

- allocation of memory for a new local-allocation buffer, and

- allocation of memory from a LAB for an object.

In the ResearchVM, the heap may support one or more LABs per generation. To allow this dynamic association of LABs and generations, each generation specifies a local-allocation buffer description outlining, *inter alia*, the range of sizes its LABs may take on, how those LABs may be resized, the range of objects' sizes that may be allocated in the LAB, and how threads allocating—or *refilling*—LABs do so from the generation. These per-LAB descriptions are allocated as generations are initialized. To simplify their association with thread-specific information about LABs, these descriptions and the associated per-thread data structures are represented by arrays and share a common index.

With this association between descriptions and particular LAB structures, the ResearchVM further specializes LAB-based object allocation for each class' instances. Since the instances of a non-array class are uniform in size and we know this size when that class has been loaded and resolved, we associate a set of allocation functions with the class for each generation. These sets support precompiled allocation functions specialized for runtime constants such as LAB description index, object size, and even object characteristics. This generalization allows efficient choice of the allocation function to call from dynamically compiled code. We will see that this same framework makes it straightforward to support both per-thread and per-processor LABs in the same virtual machine.

To the above operations we must add some collector-specific actions. When an application is suspended for a garbage collection of part or all of the heap, special steps must be taken. Many collection techniques such as the mark-compact technique used for the old generation for this study rely on the ability to iterate over the objects in the heap. To support this iteration, unused portions of local-allocation buffers must be made to look like valid objects for the duration of the collection. In the event that a given generation is collected, we also discard all the outstanding local-allocation buffers in the portion of the generation that is reclaimed.

Per-thread local-allocation buffers work well for many applications. For example, those that have relatively few threads or whose threads are compute-bound, are able to make efficient use of LABs. This efficiency results from the fact that such threads typically allocate most of the memory reserved for local-allocation buffers between any two collections. This behavior remains true for most buffer-sizing policies so long as the maximum buffer size remains below a suitable fraction of the generation from which the buffers are allocated. However, when the number of threads greatly exceeds the number of processors, the ability of threads to make use of LABs is diminished as most threads will be suspended for long periods of time with no chance to run. In such cases, as the efficiency of LAB usage decreases, the rate of collection increases forcing the application to spend more time suspended while garbage collections are performed.

To combat this effect, memory managers employing local-allocation buffers have evolved adaptive sizing policies. The ResearchVM makes use of a carefully tuned sizing policy due to Ole Agesen in which local-allocation buffers for a given thread start at a small size (24 words), and as the thread requests additional LABs, the size is increased by a fixed multiplicative factor (of 1.5). At each garbage collection, the sizes of each thread's LABs are then decayed by another factor (of 2). Finally, the young generation's size is adjusted based on the number of observed allocating threads. We will see that this sizing policy is remarkably resilient at providing good performance across a wide range of applications. As good as such sizing policies are, however, they require that the garbage collector efficently support direct allocation from the heap over a range of sizes. This rules out some otherwise useful garbage collection implementations. We would like to gain the benefits of local-allocation buffers without need for sensitive LAB sizing policies.

In this study, we will show that associating local-allocation buffers with processors allows us to gain this independence in sizing policy. What makes this study challenging is that augmentation of the local-allocation buffer mechanism so that LABs may be associated with processors not only does not remove synchronization from the common operations, but comes at a small cost in that object-allocation paths are now dependent on dynamic information for a thread, namely, the processor on which it is executing. What we show is that despite this potential cost, per-processor local-allocation buffers have the same latency as per-thread local-allocation buffers and that together with per-thread local-allocation buffers, per-processor LABs remove the need to carefully tune LAB-sizing policies. Before we examine their implementation, however, we must turn to the underlying notification mechanisms and how these are exposed to the application and garbage collector.

3 Multi-processor restartable critical sections

Multi-processor restartable critical sections (MP-RCS) provide a facility that provides two complimentary services: a mechanism to notify threads when they may have been preempted, and access to knowledge of where the thread is currently executing. These services may be combined to implement non-blocking [Her91a, Her91b, Gre99] algorithms that manipulate processor-specific data in a consistent manner and without interference from other threads. In particular, MP-RCS allows a thread executing a critical section of code to either commit an operation if the thread ran without interference, or detect the interference and restart.

Among our design goals for MP-RCS are that it impose a minimal burden on the underlying operating system, and that it not presume a particular underlying thread model. In the first case, we rely solely on notification of potential interference when a thread begins running on a processor. This ON-PROC notification means that threads' MP-RCS transactions may only react to potential interference after the fact, but it also means that user-level threads may be preempted at any time and that the kernel need do no additional work as it does deschedule them. In the second case, we desire that our mechanism work well whether the underlying threading model is preemptively or cooperatively scheduled and whether it maps user-level threads directly to scheduler-level threads or whether it multiplexes user-level threads over a smaller number of scheduler-level threads. On the Solaris™ operating environment, these scheduler-level threads are referred to as lightweight processes or LWPs.

At a high level, MP-RCS provides a way to accurately identify the processor running the restartable critical section (or more generally, identify the correct processor-specific data to be modified), and a way to detect interference in the critical section before writing to any processor-specific data. An attempt to write to processor-specific data is the commit point that ends the critical section. Since restartable critical sections only guard the modification of processor-specific data, the only sources of interference are preemption, and if signal handlers are allowed to use MP-RCS, signal handling.

In the Solaris operating environment [MM01], MP-RCS detects preemption by hooking into the kernel's thread-scheduling mechanism. This mechanism supports the loading of modules that may examine and alter the state of threads as those threads are descheduled from processors or begin running on processors after being blocked. Support for these modules, accessed via `saveCtx` and `restoreCtx`, was first added to support performance-monitoring tools that needed to multiplex hardware performance counters among the threads running on a processor. We make use of these same hooks to notify threads that they have been preempted and that interference may have taken place while they were blocks. Because we notify threads at scheduling points, MP-RCS' services can be implemented without locking, memory barriers, or atomic instructions. The performance and scalability improvements of this scheme can be dramatic. An MP-RCS-based implementation of malloc, using processor-specific allocation arenas is discussed in our previous paper [DG02].

To install the kernel-level MP-RCS code that will handle ON-PROC notification, we currently require that applications employing the MP-RCS service register the appropriate loadable driver through the call:

```
int rcsInitialize();
```

In addition, we require that individual threads register with the service so that we can rely on the availability of thread-specific state. When a given thread is started, it must call

```
int rcsInitializeThread();
```

to ensure that its state is properly initialized for MP-RCS. These registrations may be performed explicitly by the application or may be performed implicitly by a shared library loaded at the start of the application and interposing on the various thread-creation entry points. Removing the need for this kind of registration would require modifications in the operating system and are beyond the immediate scope of this study.

This work extends the MP-RCS service in two ways. First, it makes use of a new mechanism for communicating interference. Second, this mechanism is used to implement a *critical-section* library, or *CS interface*, of primitives for constructing MP-RCS transactions in high-level languages. These transactions are somewhat different from what we have explored previously in [DG02]. Before we turn to these extensions, let us first review our original approach: upcall-based MP-RCS.

3.1 Upcall-based MP-RCS

We first implemented MP-RCS using an upcall mechanism. 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 basic scheme is for the upcall kernel driver to arrange for a user-level notification routine to run when a thread has been preempted, before resuming the thread. The user-level routine then determines if the preemption occurred while the thread was in a critical section, and if so arranges to restart the critical section.

The operating system passes control of the thread to the kernel driver whenever a thread is about to resume execution. The kernel driver posts an upcall by saving the interrupted user-level instruction program counter (PC), and substituting the address of the user-level upcall routine. When the thread returns to user-level, it resumes in the upcall routine instead of at the interrupted instruction. Because a thread may be preempted while attempting to perform an upcall's actions, the notification mechanism also walks the thread's stack, adjusting PC's as appropriate. To assist the upcall routine, the kernel driver passes the current processor ID and the original interrupted instruction PC as arguments into the notification routine as well as providing a downcall to recover the original values of the registers holding these two passed-in values.

By convention, we do not permit blocking system calls or stores to memory shared between processes within a critical section. These restrictions allow the kernel driver to filter out benign cases of preemption, such as voluntarily context switching and preemption by threads from another process. To test for cases of noninterference, the kernel driver maintains a mapping of processors-to-threads; by comparing a thread's notion of the processor it is on with this table, the driver is able to verify both that the thread has not migrated elsewhere and that no other thread was on the processor since the thread was last ON-PROC.

The upcall-based approach has several good performance characteristics. For example, the need to make upcalls is infrequent since it depends on the frequency with which threads block or are rescheduled, on the order of 10's or 100's of milliseconds. Further, most of the cost to transactions making use of the upcall

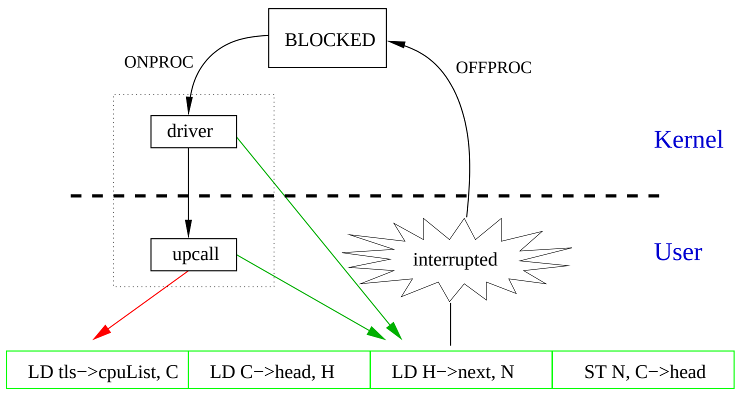


Figure 3: Upcall-based MP-RCS notification of preemption.

version of MP-RCS is the extra overhead on resuming a thread in a transaction in the upcall handler. On a 400 MHz UltraSPARC® II system the latency of an upcall is 13 μsecs. Although more expensive than we would like, the kernel driver is able to filter out many preemption events to keep upcalls even less frequent. Combining these behaviors with the fact that in the absence of preemption there is no cost to enter and execute an upcall-enabled MP-RCS transaction makes the overall performance efficient.

Figure 3 shows an example of an MP-RCS transaction making use of the upcall mechanism. In this example, we are popping an item off the head of a linked list. Starting with the left-most instruction, we load the current processor's `cpuList` into a register. Using this processor-specific value, we load the address of the head item in the list. We then extract the address of the next item. Finally, we store this address into the head of the processor's list claiming the popped item in the process. Suppose that we are preempted at the third instruction. When we begin running again, we resume in the MP-RCS kernel driver. If we can detect that it is a case of benign preemption, then control returns to our previously interrupted instruction and we continue as before. If, however, there is the possibility of interference, then the upcall handler redirects our execution to the start of transaction.

Having touted its many benefits, what are the weaknesses of the upcall-based approach? One immediate cost is that the kernel driver must maintain state for the process as a whole in the form of the thread-to-processor mapping structures as well as per-thread save-areas where saved-off register values may be stored.

This cost means that long-lived applications with many threads must be careful to appropriately deregister threads so that kernel resources may be reclaimed.

Another limitation has to do with the recognition of the bounds of MP-RCS transactions. To determine how to redirect or recover an aborted transaction, the MP-RCS service must be able to identify the extent of a given transaction as well as to determine where the transaction should restart. There are many mechanisms for performing this registration—see [MDP94, MDP96], for example. We have chosen to register each transaction's starting, ending, and restart program counters. Such an approach is simple to implement but does not scale well. First, this approach doesn't easily extend to transactions structured as an extended basic block with multiple committing stores at each of the exiting leaves. Second, to determine whether a transaction is on-going and how to recover requires traversing the registered mappings. Finally, this approach also makes the registration of dynamically compiled MP-RCS transactions more complex, as additional metadata must be generated to support each newly generated transaction's registration.

Aside from these issues of registration, there are more subtle challenges. For example, we noted that the upcall mechanism can be used for two purposes: maintenance of thread-locally cached processor-specific state and the support of transactions for managing processor-specific state. If multiple application services wish to use MP-RCS transactions for different purposes, then each may attempt to register a different upcall handler. How these handlers are chained together is an open question and one we did not answer. Another difficulty is that because we rely on notification of preemption to determine if we have successfully operated on processor-local resources in isolation, it is not possible to use standard debugging techniques to single-step through, or set break- or watch-points within, an MP-RCS transaction.

Finally, we found that the fact that we could not rely on how compilers transform high-level languages into object code limited our transactions to ones written in assembler code. This restriction allowed us to ensure certain programming disciplines—for example, avoiding the need to save and restore input values to support the restarting of transactions; our transactions typically left the registers containing these values unchanged throughout a transaction. Additionally, our preemption and redirection at any point in a given transaction means that in the upcall handler, we have difficulty in tracking how side-effects in the transaction on other resources have occurred. This last point will become an issue when we look at how per-processor local-allocation buffers are allocated from the garbage-collected heap.

All of these issues—registration and identification of transactions, different uses of MP-RCS transactions, debuggability, restrictions on coding style—led us to look for simpler mechanisms for implementing the notification mechanism and for expressing MP-RCS transactions.

3.2 `%asi`-based MP-RCS

Before we consider how to express MP-RCS transactions in high-level languages, let us first turn our attention to a simpler mechanism for notification of preemption. Recall that the existing upcall-based mechanism requires both that we identify the bounds of each transaction and that each transaction ends with a committing store operation. In this section, we describe a SPARC® -specific mechanism that requires no maintenance of per-thread or per-processor state in the MP-RCS kernel driver, that needs no upcall handler, and with which we guarantee that we only receive notification of preemption at precise points within a transaction.

The SPARC architecture provides a global Address Space Identifier `%asi` register that affects the behavior of memory operations. In effect, the `%asi` register informs the memory system how to interpret an address while executing certain instructions, such as `sta` (store into alternate). The SPARC architecture defines `%asi` values that aid in refilling the TLB, in performing block and non-caching moves, and even ones that interact with the floating-point subsystem. We have chosen two values not ordinarily in use in user-level code that we identify as ASI_RCS and ASI_INVALID.[2] These values have been chosen so that memory operations through the `%asi` register when it is set to ASI_RCS behave as if they were ordinary load, store, or atomic operations, and so that the same operations through the `%asi` register when it is set to ASI_INVALID trap into a signal-handler. Using these two values, we can now create more flexible forms of MP-RCS transactions.

To begin an MP-RCS transaction, we need only set the `%asi` register to be ASI_RCS using a single ALU instruction. Having done this, we may proceed to compute the new value we wish to store. Should this new value depend on the processor on which we are executing, we may load the `cpu_id` of the most recent processor on which we have executed from our thread's `schedctl` block. By loading this value within the context of an MP-RCS transaction, we are guaranteed that it reflects our current location of execution. To make obtaining this `cpu_id` as

[2]Note that to use this approach to MP-RCS notification, we should work to reserve to values in the value space of the `%asi` register. For purposes of this study, however, the two values we have selected, ASI_SECONDARY and ASI_NUCLEUS, allow us to prototype the approach and evaluate its efficacy.

```
(1)    asi = ucontext(%asi);
(2)    if (asi == ASI_RCS)
(3)        ucontext(%asi) = ASI_INVALID;
```

Figure 4: `%asi`-based kernel driver

inexpensive as possible, we reserve a slot in each thread's thread structure which holds a reference to that thread's `schedctl` block. Finally, we can detect that the transaction has failed by testing the value of the `%asi` register and we can attempt to commit the new value by using a store or atomic operation that makes use of the `%asi` register.

As we noted above, this approach has many benefits: simplicity, fewer kernel resources, the ability to control precisely where transactions abort or retry. To appreciate these qualities, consider figure 4 which shows the actions that the `%asi`-based MP-RCS kernel driver must perform as a thread transitions back onto a processor: it need only flip the state of `%asi` register in that thread's user-level context, should its old value indicate that the thread has an open MP-RCS transaction. Further, it allows us to form more flexible forms of transactions. Since each memory operation through the `%asi` register does not, itself, side-effect that register, we may chain together a sequence of such stores. In addition, if we register program counters with the signal-handler that handles the trapping of failed committing memory operations, we need at most to register the particular program counters of those operations themselves.

However, here we see an important advantage of this approach to implementing the MP-RCS service: we need not explicitly register MP-RCS critical sections. Instead, we may use a register, say `%o5`, to communicate success or failure of any memory operation through the `%asi` register. Consider the following SPARC instruction sequence:

```
mov %g0, %o5
sta %o1, [%o0]
subcc %o5, %g0, %g0
```

The first instruction clears the value of the `%o5` register, setting it to 0. The "sta" instruction is a store-through-`%asi` instruction. If, when this instruction executes, it succeeds, then the final comparison will find that `%o5` is still 0. If it fails and traps, however, we can employ a handler like that shown in figure 5. This handler will test to see if we are in an MP-RCS transaction and toggle the `%asi` register

```
( 1)  pc = ucontext(%pc);
( 2)  asi = ucontext(%asi);
( 3)  if (asi == ASI_RCS)
( 4)      asi = ucontext(%asi) = ASI_INVALID;
( 5)  if (asi == ASI_INVALID && isTrapping(*pc)) {
( 6)      if (ucontext(%o5) == 0)
( 7)          ucontext(%o5) = 1;
( 8)      ucontext(%pc) += 4;
( 9)      ucontext(%npc) += 8;
(10)      return 1;
(11)  }
(12)  return 0;
```

Figure 5: `%asi`-based trap/signal handler `fixUContext()`

if necessary (lines 3-4). Further, it will test to see if our `%asi` register is set to ASI_INVALID and if we are on a memory operation of interest (line 5). If so, it changes the value of the register we have chosen to use to reflect success or failure—here, `%o5`—to 1, and it adjusts the program counter to resume execution at the instruction after the failing memory operation (lines 6-9). By not requiring the explicit registration of MP-RCS transactions, it becomes much easier to support MP-RCS transactions in the presence of dynamically generated code such as we find in the Java platform.

We will see in the next section that the `%asi`-based approach also fits well with how we expose the MP-RCS in high-level languages. Basically, it allows the application to clear an interference flag, perform some set of operations, and then atomically commit the result of those operations so long as the interference flag remains cleared. This simplicity allows the application to manage how, for example, processor-local resources are managed. The approach also avoids the costs and complexities of supporting upcalls from the MP-RCS kernel-driver. Further, so long as MP-RCS transactions are not nested, it allows multiple application services to make use of MP-RCS transactions for different purposes without the need to arbitrate their uses of preemption notification.

Despite its overall simplicity, this approach does have some drawbacks. The first is that it is oblivious to why a thread has been blocked or preempted and will simply report such events without any attempt to detect benign cases of preemption. Instead, the application itself must decide of a failed transaction whether the transaction may be reopened because the relevant state on which that transaction depends has not changed. Just as this detection and recovery has been pushed onto the application, so too must the application now access processor-specific resources through a dynamic index, namely the `cpu_id`. These shifts in responsibility both reflect an advantage in flexibility and control as well as a cost in additional steps that the application must take on each transaction.

More subtly, we must be careful of how signal-handling interacts with this mechanism. Signal delivery and handling presents two challenges for the `%asi`-based approach to MP-RCS. First, it may be that a thread transitions back onto a processor directly into a signal-handler. In this case, the context the kernel-driver adjusts is for that signal handler's frame. However, the underlying context may, itself, be in an MP-RCS transaction, and without additional steps, its `%asi` register will be restored to whatever value that register had prior to the delivery of the signal when the signal handler returns. To handle such cases, we must be sure to invalidate the underlying transaction by modifying the context for that frame's `%asi` register much as the kernel-driver does for the current context. This modification can be achieved in one of several ways:

- underhooking signal delivery,
- underhooking signal exit by interposing on `setcontext`, or
- wrapping all signal handlers in use.

We have chosen the second approach together with selective application of the third approach for those signal handlers we do control. In both cases we call the MP-RCS function `fixUContext`, listed in figure 5.

The second challenge posed by the handling of signals is that in some contexts, the signal handler itself needs to make use of MP-RCS transactions. In such cases, it is simplest if such handlers invalidate any outstanding transactions in the underlying stack frames of the thread handling the signal to avoid interference with itself.

Finally, the `%asi` register is used for other kinds of operations. For example, `memset`, `memcpy`, and related functions make use of the block-data properties of the `%asi` register. In constructing an MP-RCS transaction, we must be careful not to invoke such operations, or if we do, we must have some mechanism in place to test whether the `%asi` register is still set to ASI_RCS after those portions of the transaction that are questionable so that we may either abort the transaction or attempt to reopen it.

4 The Critical Section Interface

To enable the expression of MP-RCS transactions in high-level languages, we have introduced a library of primitives, the *Critical Section* (CS) interface. These primitive operations allow a transaction to be constructed. Within a transaction, they allow the application to query the status of the transaction and to conditionally perform memory operations that only succeed if the transaction is still valid.

The basic interface is shown in figure 6. An MP-RCS transaction is initiated by a call to `csBegin()` which returns the id of the current processor. Conditional stores and atomic operations are provided through a number of entry points, each specifying the types of the input arguments. The status of the current transaction may be queried by calling `csValid()` and the transaction may be ended or terminated by invoking either `csEnd()` or `csInvalidate()`. These operations are combined to form transactions that are similar in flavor to what one finds with load-linked (LL)/store-conditional (SC). And like LL/SC, MP-RCS transactions benefit from the fact that their use of committing operations is immune to A-B-A issues [MS98]. The CS interface minimizes the amount of low-level code that must be written and hides this code behind a simple interface. This modularization allows applications to craft MP-RCS transactions to suit their needs with complete control over how, for example, and where notification of preemption is handled. Further, it leaves the organization and management of different sets of processor-local resources entirely up to the application.

We have implemented the CS interface using the upcall-based MP-RCS service. Doing so helped ameliorate some of the drawbacks of the upcall-based approach such as fixing the set of registered PC ranges to those functions in the CS interface's library. However, because the CS interface relies simply on notification of potential interference, the `%asi`-based implementation of MP-RCS is ideally suited to supporting it. For example, figure 7 shows how `csBegin()` and `csST32C()` are implemented using the `%asi` register.

Figure 8 shows an example of an MP-RCS transaction making use of the CS interface. It illustrates both the similarities with, and differences from, the kind of MP-RCS transaction we saw in figure 3. Most notably, in this example, we do not load a processor-specific pointer to a list; instead, we use a `cpu_id` to index a table mapping processor id's to lists. Further, when we happen to be preempted before we reach a committing operation, control is always returned once our interference flag—for example, the value in the `%asi` register—is properly set. Finally, we are always notified of success or failure at each committing operation and we must check the status of the operation to determine what to do next.

```
/* Start a transaction returning the current cpuid */
int csBegin();

/* Commit a transaction by performing a store.
   Returns "0" for success; "1" for failure, */
/* 32-bit stores */
int csST32C(void *addr, int32_t val);
int csCAS32C(int32_t new, int32_t old, void *addr);
int csSTWC(void *addr, int val);
int csCASWC(int new, int old, void *addr);
int csSTC(void *addr, int val);
int csCASC(int new, int old, void *addr);

/* 64-bit stores */
int csST64C(void *addr, int64_t val);
int csCAS64C(int64_t new, int64_t old, void *addr);
int csSTXC(void *addr, int64_t val);
int csCASXC(int64_t new, int64_t old, void *addr);

/* pointer-wide stores */
int csSTPC(void *addr, void *val);
int csCASPC(void *new, void *old, void *addr);

/* Checks if the current transaction is still valid.
   Returns "0" for failure; "1" for success */
int csValid();
/* Ends the outstanding critical section. */
void csEnd();
/* Marks the outstanding critical section as invalid. */
void csInvalidate();
```

Figure 6: Interface to the critical-section library

```
!  Returns cpu_id as result in %o0
ld [%g7 + sc_off], %o5
mov ASI_RCS, %asi
retl
ld [%o5 + cpu_off], %o0
```

(a) `int csBegin()`

```
!  Returns 0 iff store succeeded
mov %g0, %o5
sta %o1, [%o0]
retl
mov %o5, %o0
```

(b) `int csST32C(addr, value)`

Figure 7: SPARC code for starting and committing actions using `%asi`-based MP-RCS

A trivial example using a restartable critical section is a lock-free, memory barrier-free, atomic instruction-free counter:

```
int count[MAXprocessorS] = {0};

void countIt() {
     restart:
          int cpuid = csBegin();
          int newVal = count[cpuid] + 1;
          if (csST32C(&count[cpuid], newVal) != 0)
               goto restart; // preempted
          csEnd();
}
```

Simple extensions to this code could add a contention counter, or to do some action predicated on a consistent snapshot of several cpu-specific variables. In the next section, we describe a more realistic use of MP-RCS to manage per-processor local-allocation buffers.

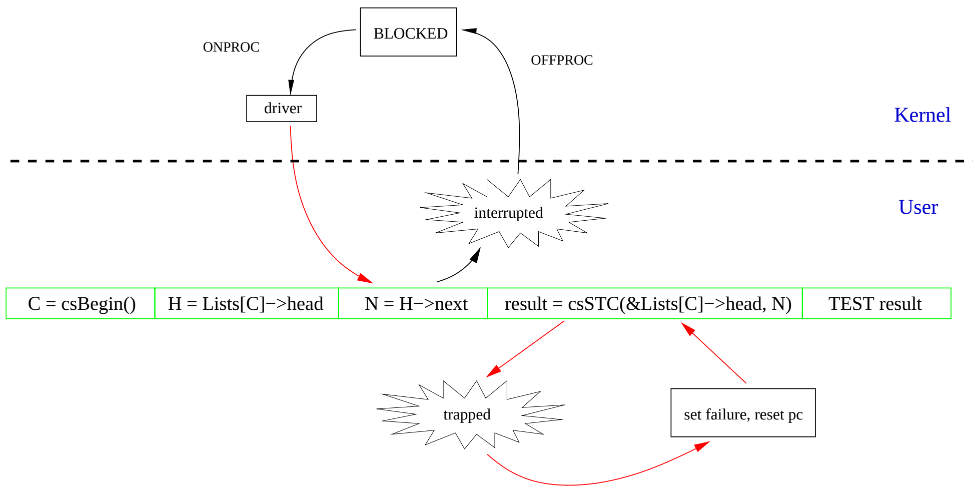


Figure 8: Manipulation of a processor-specific linked-list using the CS interface

5 Per-processor local-allocation buffers

Having presented the CS interface, let us now show that it is effective in managing processor-local resources. Recall that our chosen area of application is the scalable support of local allocation buffers. Our platform, the ResearchVM, already has support for associating LABs with threads. However, as the number of threads exceeds the number of processors, our concern is that many suspended threads will have partially unused allocation buffers and that this memory will not be used before the LABs are discarded at the next collection. In this section, we describe an alternative association, one that ties allocation buffers to processors and so makes the LABs available to any threads that happen to be running.

5.1 Allocating from LABs

Replacing per-thread local-allocation buffers (TLABs) with per-processor local-allocation buffers (PLABs) presents a significant challenge. First, because the memory in a given TLAB is allocated by a single owning thread, there is no further synchronization that is being removed. Second, the CS interface, itself, and the accessing of processor-local data add cost to the operation of allocating memory from a given PLAB. These additional costs result from:

- the starting of the transaction and accessing the current `cpu_id`,

```
( 1)  allocateWordsInGenWithTLAB(Generation *Gen, Thread *thr,
( 2)                                          Class *class, int numWords) {
( 3)          labNum = Gen->labDescriptor->number;
( 4)          labAvail = tlabAvail(thr, labNum);
( 5)          newLabAvail = labAvail - numWords;
( 6)          if (newLabAvail < 0) {
( 7)                  res = refillTLAB(thr, labAvail, class, numWords, labNum);
( 8)                  if (!res)
( 9)                    res = class->gen[Gen->number].allocateWords(Gen, thr, class, numWords);
(10)          } else {
(11)                  tlabAvail(thr, labNum) = newLabAvail;
(12)                  res = (word32 *)(tlab(thr, labNum) - labAvail);
(13)          }
(14)          return res;
(15)  }
```

Figure 9: Pseudocode for allocating memory from a TLAB

- the accessing of that processor's resources via the `cpu_id`,
- the testing for success or failure on the committing operation, and
- the need to include mechanisms to retry the allocation request in the presence of preemption.

To understand these costs, consider figures 9 and 10 that illustrate how objects are allocated from TLABs and PLABs, respectively.

Beginning with figure 9, we identify the TLAB associated with the current generation (line 3). Using this value, we measure whether the current allocation request, `numWords`, may be satisfied from the existing LAB (lines 4-6). If there is sufficient memory, we update the state of the LAB (lines 11 and 12). Otherwise, we allocate memory for the requested amount together with memory for a new LAB, discarding or merging the old one into the new LAB, if necessary (lines 7-9).

In contrast, allocation from a PLAB is somewhat more complicated. In figure 10, we begin as before by identifying the index of the allocation buffer assigned to the generation in which we wish to allocate an object (line 3). Now, however, we must start an MP-RCS transaction and look up the processor on which we are running (line 5). Further, if we find that we can allocate memory

```
( 1)  allocateWordsInGenWithPLAB(Generation *Gen, Thread *thr,
( 2)                                  Class *class, int numWords) {
( 3)      labNum = Gen->labDescriptor->number;
( 4)      do {
( 5)          cpuid = csBegin();
( 6)          labAvail = plabAvail(cpuid, labNum);
( 7)          newLabAvail = labAvail - numWords;
( 8)          if (newLabAvail < 0) {
( 9)              res = refillPLAB(thr, labAvail, class, numWords, labNum);
(10)              if (!res)
(11)                res = class->gen[Gen->number].allocateWords(Gen, thr, class, numWords);
(12)          } else {
(13)              res = (word32 *)(plab(cpuid, labNum) - labAvail);
(14)              if (csST32C(&plabAvail(cpuid, labNum), newLabAvail))
(15)                res = PLAB_RETRY;
(16)          }
(17)      } while (res == PLAB_RETRY);
(18)      return res;
(19)  }
```

Figure 10: Pseudocode for allocating memory from a PLAB

from the current PLAB, we must use a committing store (line 14) to ensure that our manipulation of the PLAB is properly isolated from interference from other allocation requests. Finally, if either the request to refill the existing PLAB (line 9) or the attempt to commit our allocation request (lines 14-15) result in `res` having a distinguished `PLAB_RETRY` value indicating the possibility of interference, then we must retry the allocation request. All of these extra steps roughly double the number of instructions required to satisfy an allocation request.

Recall that the ResearchVM's allocation framework allows each generation to register a set of allocation functions with each class as the class is loaded and resolved. This flexibility may be seen in line 9 of figure 9 and in line 11 of figure 10 where allocation requests too large to be satisfied from an allocation buffer are allocated directly from the heap. Because the instances of a given non-array class are all uniform in size and share the same allocation functions, and because LAB assignments to generations are runtime constants, the ResearchVM provides specialized versions of the allocation functions for object sizes up to 40 words and with the LAB indices compiled in as constants. Using these specialized allocation functions, the typical fast-path SPARC instruction sequence to allocate a small object from a TLAB requires 2 loads, 1 branch, 3 ALU instructions, and 1 store for a total of 7 instructions. In contrast, the facts that we must obtain the current processor's id from the thread's `schedctl` block and that we must check whether the committing store succeeds explain why the typical fast-path instruction sequence to allocate a small object from a PLAB requires 4 loads, 2 branches, 10 ALU instructions, and 1 committing store for a total of 17 instructions. Note that, here, we inline the calls into the CS interface and rely on the property of `%asi`-base MP-RCS to implicitly recognize when we are in an MP-RCS transaction to minimize the additional instruction count.

So, even as PLABs may aid the throughput of highly threaded applications by making better use of the memory dedicated to local allocation buffers, it must do so well enough that it offsets a cost incurred in every allocation request.

5.2 Managing PLABs

Another challenge to the support of PLABs concerns how these buffers are, themselves, allocated or refilled. The refilling of a given PLAB presents a number of issues. When a thread attempts to refill a given per-processor buffer, it only makes sense for it to do so so long as it remains on that processor. So, the MP-RCS transaction becomes important not only for ensuring that the thread has exclusive access to update the state for the appropriate PLAB but also that the thread is notified if it has been preempted so that it can check if it is now executing on some other processor. The fact that in the midst of the transaction, the thread may allocate memory from the heap—a relatively long and unrestricted operation that may employ `memset`, for example, to initialize the newly allocated memory—increases the likelihood that the transaction will not be valid after the memory for the LAB is returned. If the transaction has failed but the thread has successfully allocated memory for a LAB, we are left with the problem of what to do with that LAB. One option would be to abandon it—turn it into a dead object in the heap and retry our allocation request.

We have chosen an alternative strategy. Instead of abandoning the LAB, we attempt to reopen the transaction. To do this, we start a new MP-RCS transaction with `csBegin()`, check that the `cpu_id` is the same as it was, and check that the current allocation buffer's bounds are unchanged. If so, we consider the intervening interference to be benign and continue as before. Otherwise, we simply terminate the transaction by calling `csInvalidate()`. We do not, however, leave the context of the transaction as we are not finished with the newly allocated

LAB. Subsequently, if at the end of the transaction, we have not been able to install the LAB for the current processor, we place it on a free-list of LABs available for use for the current generation. Other threads attempting to refill allocation buffers from the same generation will preferentially take LABs previously sidelined before allocating memory from the heap. These two strategies—reopening transactions and sidelining unused LABs for later reuse—have proven effective in reducing wasted memory. Key to their success is the fact that the CS interface allows applications to control how MP-RCS transactions are structured.

Per-processor local-allocation buffers present other challenges. For example, to update the bounds of the current LAB for a processor, we must atomically update a pair of values—the limit address of the LAB and its size. Strategies we considered employing included embedding this metadata in the LAB's memory and employing a multiword-commit protocol similar to that used by Johnson and Harathi [JH94]. For expedience, however, we employed the 64-bit `csST64C()` method to write the pair with a single atomic store. This approach was possible because our Java virtual machine assumes a 32-bit address space. On a different platform, we would likely have to revisit this decision. In addition to these two fields, there are a number of other fields governing LAB resizing and statistics. As these are advisory, we exploit the fact that the committing store for the bounds of the newly installed LAB leave the transaction open. Once this first store succeeds, we then optimistically update the related statistics using committing stores until we have either updated them all or one of them fails. Finally, to abandon an existing PLAB that contains unused memory, care must be taken within the context of the MP-RCS transaction to force the number of words available in the LAB to be zero with a committing store. This protocol ensures that if we then turn the unused portion into a dead object or if we attempt to merge that portion with the newly allocated LAB, no other thread that begins executing on the same processor will attempt to allocate that memory for an object.

Despite all of these additional complexities, the refilling of a PLAB is only slightly more expensive than refilling a TLAB. Further, since the sizes of buffers are typically large enough to allow the allocation of between a few hundred and a few thousand objects per buffer, this extra cost is easily amortized across those individual allocation requests.

5.3 Flipping LAB association

In the next section, we evaluate the relative performance of per-processor and per-thread local-allocation buffers for several sizing policies. In order to compare their behavior fairly, we benefit from the allocation framework in the ResearchVM that allows us to easily customize object allocation and control LAB sizing policies. This framework allows us to specify at the command line the LAB association policy and the LAB sizing policy we wish to use. In each experiment, we are able to control whether observed effects are due to differences in sizing or differences in association, knowing that all other aspects of the platform are the same.

There are conditions when the use of per-processor resources leads to poor memory utilization. For example, when the number of allocating threads is less than the number of processors, or when threads are entirely compute-bound and are not preempted between collections, threads allocating from PLABs may be preempted and migrate among the processors, leaving partially-used buffers tied to idle processors. While the amount of wasted memory with PLABs is bounded by the number of processors (instead of threads with TLABs), it is still a concern and indicates that the best strategy may be to use whichever form of association works best for the application.

At each collection, we gather statistics on the allocation behavior of each thread for each of the generations. Using the statistics for the youngest generation, we have implemented a simple policy that allows us to dynamically switch association from threads to processors or back again. Initially, the application begins with allocation buffers associated with threads. When the amount of memory left unused in TLABs exceeds the maximum LAB size times the number of processors, we flip modes and begin associating allocation buffers with processors. Should the amount of unused memory exceed the maximum LAB size times the number of allocating threads, we revert back to TLABs. An important reason for the ease with which we are able to implement this switching policy lies in the fact that the ResearchVM has a flexible allocation framework. When we choose to switch modes, much of the work in making the switch lies in updating per-class allocation-function tables. The one challenge in this context is the management of dynamically generated code that allocates objects. In the ResearchVM, all such allocations are done through direct calls to the appropriate allocation functions. These functions' addresses are retrieved from the appropriate class structures when the code was generated. To ensure that these direct calls for allocation are properly updated, we segregate the allocation functions by the kind of association of allocation buffers they employ and we patch the disabled set of such functions to patch the calling sites redirecting them to the correct corresponding functions in the other set. This approach allows those cases of dynamically-generated code calling into the wrong set of allocation functions to lazy adjust themselves to call the correct set.

6 Performance

To evaluate the efficacy of associating LABs with processors, we consider three applications:

- **_213_javac**—a single-threaded compiler for the Java programming language compiling a set of class files,
- **SPECjbb2000-1.03** [Con]—a multi-threaded benchmark simulating a set of warehouses processing orders, and
- **VolanoMark 2.1.2** [Nef98]—a highly multi-threaded benchmark measuring the Volano chat server.

We chose the `_213_javac` from the SPECjvm98 [Con] benchmark suite because it is single-threaded and because its performance is the most sensitive to the speed of allocations and to the effects of collection. We chose SPECjbb2000 so that we could study a relatively compute- and allocation-bound application with a moderate and tunable number of threads. Finally, we chose VolanoMark because this benchmark makes use of large numbers of bursty threads.

All of the experiments were run on an otherwise idle eight-processor Sun Fire™ 880 configured with 32GB of memory. Each of the eight processors is a 750MHz UltraSPARC® III with an 8MB external cache.

6.1 `_213_javac` Results

The `_213_javac` benchmark allows us to understand the costs of MP-RCS and PLABs for a single-threaded application. For this benchmark, we used a 2MB fixed-sized young generation and an old generation that ranged in size between 8MB and 15MB. Table 1 shows the mean execution time, instruction count, and cycle count across five sets of measurements taken running this benchmark on a single processor. Given that the only difference between the two sets of runs was whether LABs are associated with threads or with processors, the difference in cost is due primarily to the use of MP-RCS transactions. This benchmark allocates slightly more than 5.913 million objects in the course of each run. From this fact and the data in the table, we can conclude that the measured impact of MP-RCS enabled allocation adds 10.2 instructions and 13.9 cycles per allocation. These measurements agree well with our analysis of the added costs described for the MP-RCS enabled allocation fast-paths. Here, we both conclude that the cost of MP-RCS transactions on the refilling of allocation buffers is well amortized across the individual allocation requests and that the use of PLABs does come at a small cost.

Association	Elapsed (secs)	Instructions	Cycles
per-thread	21.205	4.163 billion	6.636 billion
per-processor	22.004	4.223 billion	6.718 billion

Table 1: `_213_javac` with the default sizing policy.

6.2 SPECjbb2000 Results

The SPECjbb2000 benchmark emulates a set of client threads, one per warehouse, processing orders through a middle tier where business logic is applied to the requests and objects in a back-end database are updated. The benchmark is interesting because there is little interaction between different warehouse threads and, like `_213_javac`, the threads simply allocate and manipulate memory in the heap, and so, is also sensitive allocation performance. During each run, there is a 30-second ramp-up period in which the warehouse threads warm up and then a two-minute window in which the rate of order-processing is measured. By varying the number of warehouses, we show how our LAB-association and LAB-sizing policies interact as the number of threads begins to exceed the number of processors.

In our experiments, we varied the number of warehouses from 1 to 64 (8 times more threads than processors). We also used the the default policy described at the end of section 2 in combination with three association policies: TLABs, PLABs, and the flip-mode policy. We sized the young generation at 4MB so that collections occur once every 200-300ms. This ensures that threads are rescheduled one or two times between collections. Each of the reported throughput rates represents the mean of four runs.

The results, shown in figure 11, demonstrate that the flip-mode policy performs well. So long as the number of warehouses—and so, threads—is close to the number of processors, there is little wastage in unused LABs and the system behaves similarly to the policy of TLABs. As the number of threads increases, it does “flip” and begins to behave like the per-processor policy. With up to 8 allocating threads and the default LAB sizing policy, we see that PLABs do significantly worse than either of the other two modes. This is largely due to the added cost of allocating relatively small buffers and the reduced ability to amortize this cost across individual allocation requests. Coupled with a slightly higher preemption rate, PLABs perform significantly worse than TLABs.

Beyond 16 warehouses, we see that PLABs begin to behave consistently better than TLABs. This improved performance is due to less wastage of memory in

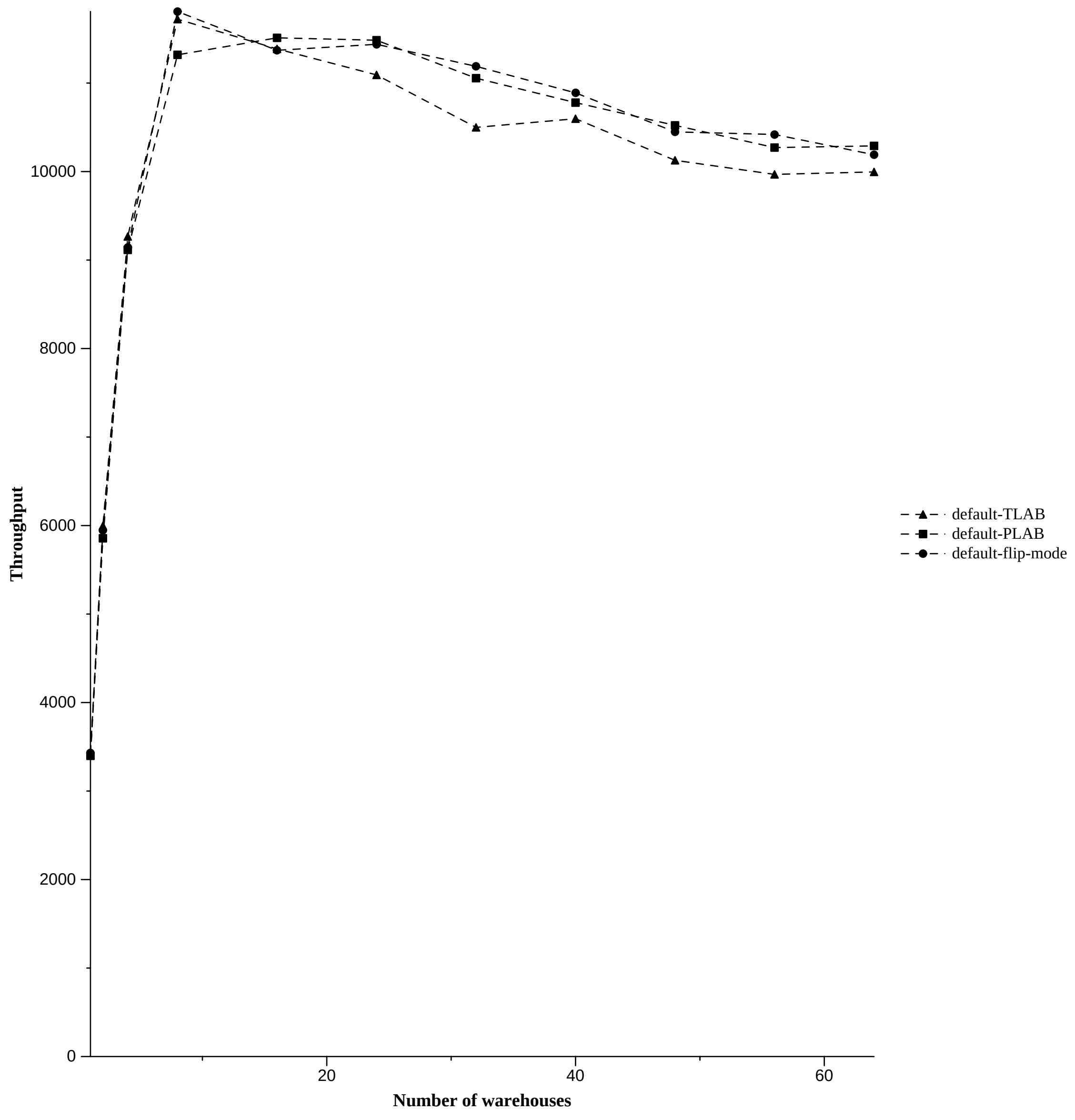


Figure 11: SPECjbb throughput scores for TLABs, PLABs, and flip-mode (default LAB sizing).

unused portions of LABs, and hence a corresponding reduction in the frequency of collections. The threads in this benchmark are compute-bound, and on a lightly loaded machine, they are rescheduled only when they exhaust their quanta. With collections occuring every 200 to 300 milliseconds, there is at most a fixed number of threads that are able to run, let alone be preempted, between each collection. Because the benchmark reported an aggregate throughput of work-units performed by all threads within the 2-minute timing window, we expect each of the lines in the graph to flatten out and we see this pattern for both sizing policies. The difference between PLABs and TLABs is due to the fact that the latter wastes from 2% to 5% of the heap.

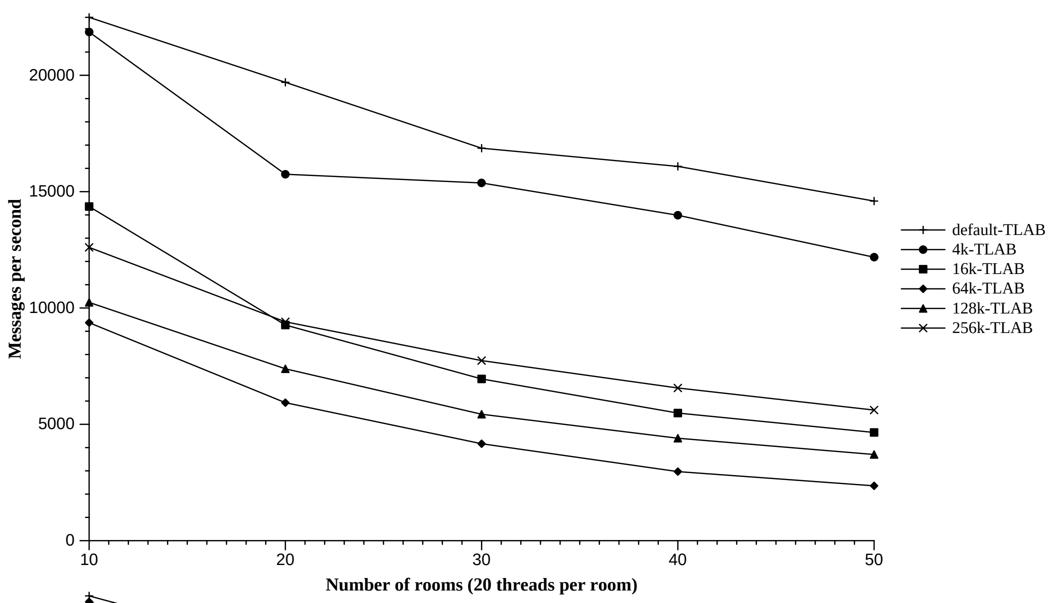
Note that having run these experiments on a lightly loaded machine biases the effects towards TLABs. With more system activity, we would expect the rate of preemption per thread to increase. In such environments, PLABs should begin to do even better than TLABs.

6.3 VolanoMark Results

For applications like the the Volano Chat Server, there may be hundreds or thousands of threads, most of which are suspended most of the time. We ran the VolanoMark 2.1.2 benchmark varying the number of threads, the size of the young generation, and LAB associativity and sizing policies. In all cases, we kept the older generation fixed in size at 64MB. The results are shown in figures 12–14.

Each of the figures shows a pair of graphs, the first measuring the behavior of TLABs and the second, the behavior of PLABs. Within each graph, we graph the behavior of a particular LAB-sizing policy: the default adaptive sizing policy, and fixed-sized policies using 4KB, 16KB, 64KB, 128KB, and 512KB LABs. We varied the number of threads employed in the benchmark from 200 (for 10 simulated chat rooms) to 1,000 (for 50 chat rooms). For each combination of parameters, we ran the benchmark 10 times and report the harmonic mean of the rates of messages per second.

First, as the number of threads exceeds the number of processors, the performance of PLABs is relatively insensitive to particular choices of LAB-sizing policies. This resilience is in stark contrast to that of TLABs. Further, this indifference to sizing policy increases as the size of the young generation increases. Being able to support larger LABs without penalty aids certain kinds of collection techniques and reduces the complexity of tuning JVMs for varying conditions, whether these are heap-sizing policies or the number of threads. As table 2 shows, with larger individual LABs, TLABs waste more memory—by up to 3 orders of



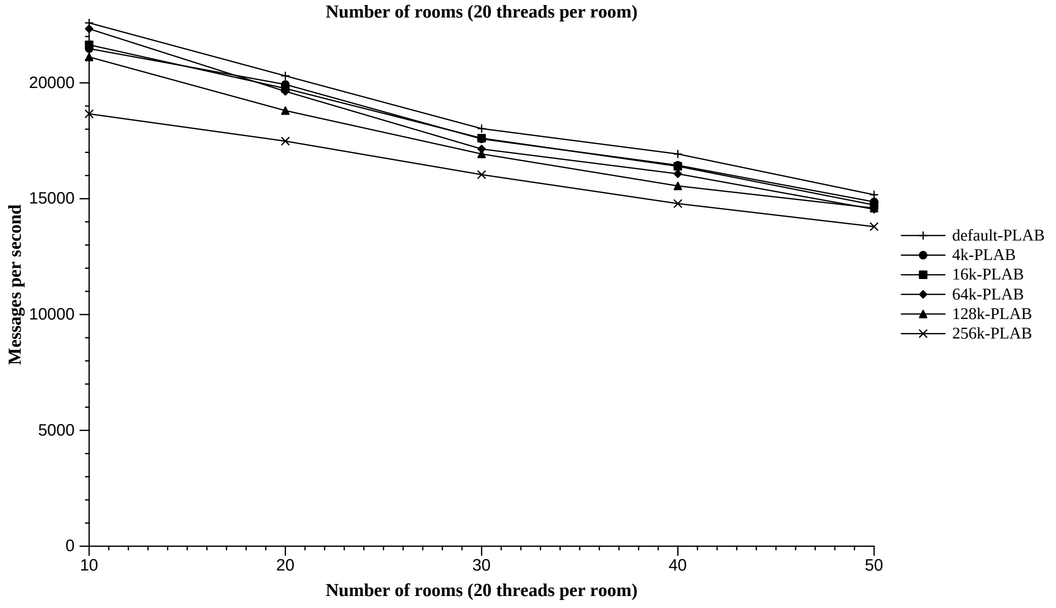
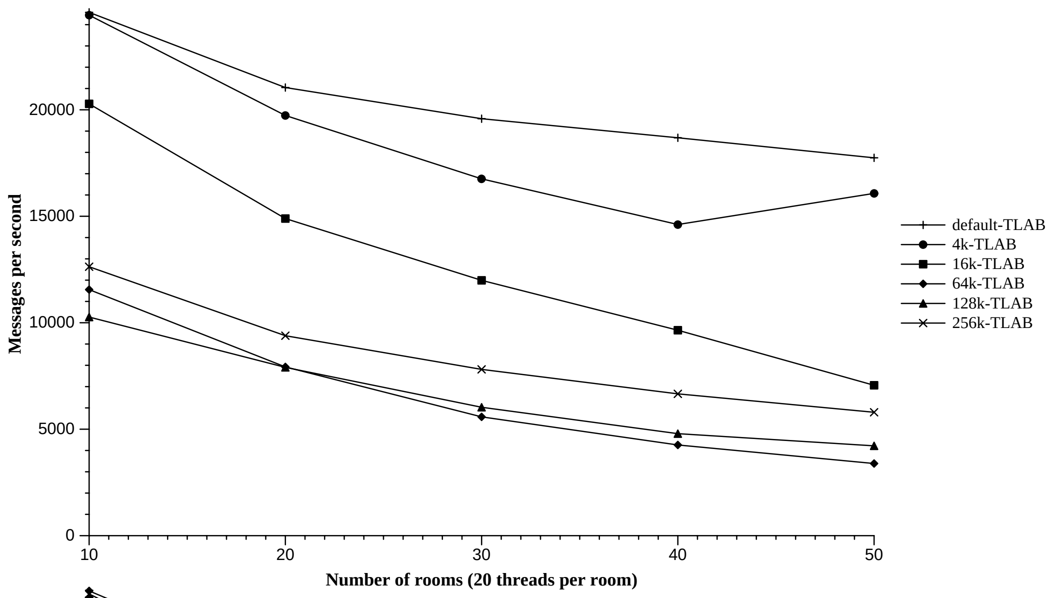


Figure 12: Results of TLABs and PLABs with 2MB semispaces



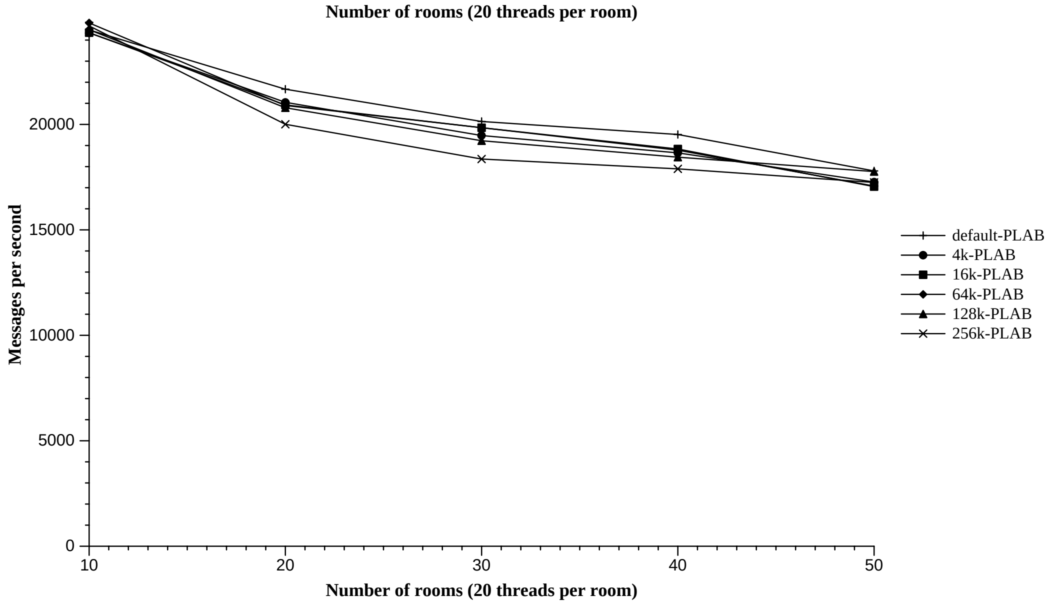
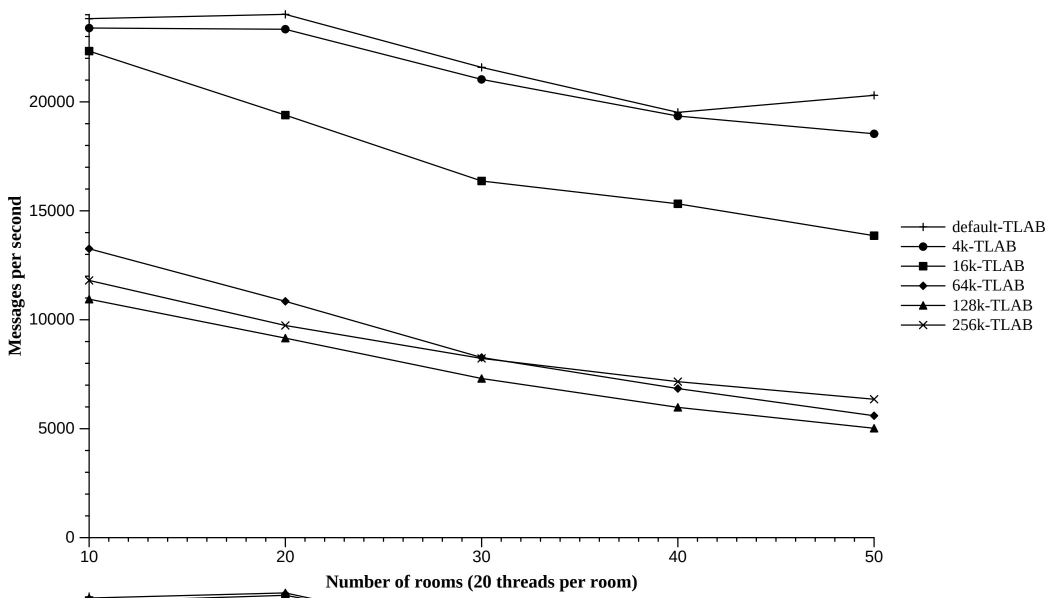


Figure 13: Results of TLABs and PLABs with 4MB semispaces



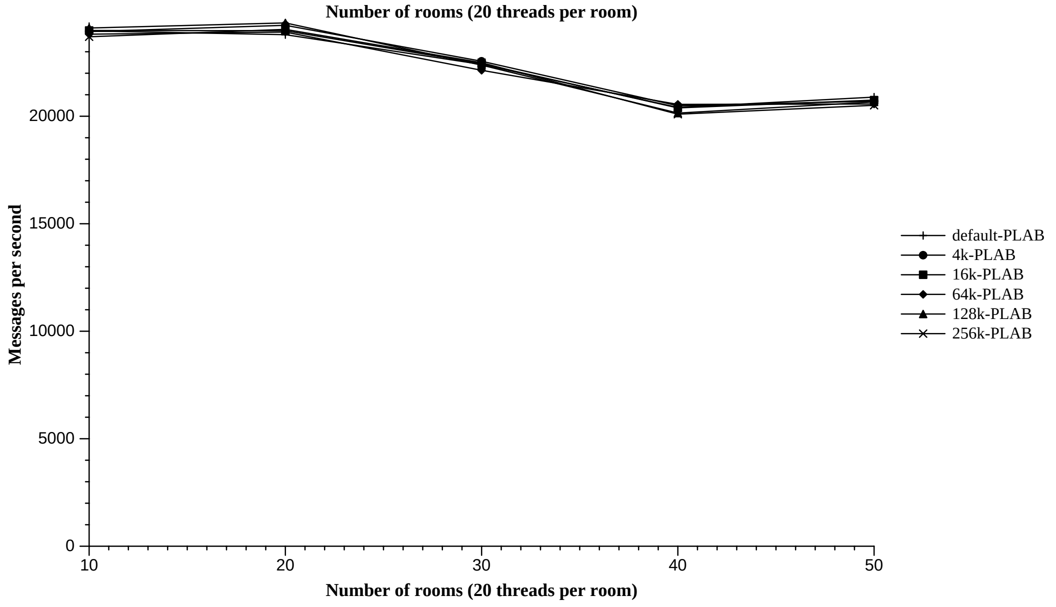


Figure 14: Results of TLABs and PLABs with 8MB semispaces

	10 Rooms			50 Rooms		
	TLABs	PLABs	Flip-mode	TLABs	PLABs	Flip-mode
# of young-gen collections	354	19	21	2256	98	103
# of old-gen collections	2	2	2	2	2	2
GC time (secs)	2.639	0.202	0.235	63.825	3.563	3.791
Using TLABs	354	0	4	2,256	0	4
Using PLABs	0	19	17	0	98	99
Words allocated in LABs	344,648,917	15,568,064	18,191,116	2,252,364,194	84,974,011	87,249,718
Invalid transactions	0	2	2	0	2	1
Number of parked LABs	0	4	4	0	3	3
Words in parked LABs	0	63,078	63,898	0	45,875	54,067
Number of unused parked LABs	0	0	0	0	98	0
Words of unused parked LABs	0	0	1,638	0	0	0
Number of partially used LABs	21,453	110	278	140,197	556	733
Words wasted in partially used LABs	339,230,649	451,588	2,961,602	2,233,391,754	1,922,212	4,278,650

Table 2: Per-GC LAB statistics for VolanoMark with 64KB LABs and a 4MB young generation.

magnitude—and so incur many more collections.

Second, the default, adaptively-sized policy does remarkably well for both TLABs and PLABs. So long as the collector used can efficiently support small LABs, this policy makes TLABs competitive even in highly multi-threaded applications. Nonetheless, even with this policy, as we move to the right in the graphs, we can see that PLAB performance improves by up to 15% over TLABs. The disparity is larger when considering the various fixed-sized sizing policies, with the improvement of PLABs over TLABs increasing with the size of the LABs.

Third, if we look closely at the graph of TLAB results in figure 12, we see that the 256KB sizing policy does better than all but two other sizing policies. Further, we see that the worst policy is for 64KB LABs. The reason for this unexpected behavior is that as the youngest generation is exhausted, threads that can no longer allocate LABs in that generation switch to allocating objects directly from whatever memory remains. The 256KB policy does better because more of the threads spend time in the final phase, unable to allocate LABs, and this provides a buffer of time in which extant LABs may be filled with newly allocated objects. Likewise, the 64KB policy does the worst because the threads efficiently claim most of the memory for LABs and the remainder does not provide a sufficient buffer of time for those still-allocated LABs to be filled. This behavior calls for a study to determine when to cease allocating LABs.

7 Related Work

Much research has been done in the area of kernel assisted non-blocking synchronization in the context of uniprocessors. In [Ber93] and [BRE92], Bershad introduces restartable atomic sequences. Mosberger et al. [MDP94, MDP96] and

Moran et al. [MJ92] describe a number of mechanisms for implementing such restartable critical sections. Johnson and Harathi describe interruptible critical sections in [JH94]. Takada and Sakamura describe abortable critical sections in [TS94].

Our implementation makes use of the `restoreCtx` hook to install a device driver to handle notification of preemption. This ON-PROC notification mechanism can be viewed as an impoverished—and less expensive—form of scheduler activation. Scheduler activations, explored in [ABLL92, SS95], make kernel scheduling decision visible to user programs. Many operating systems provide a similar way to hook into this thread transition point in order to support system tracing tools. Modern SMP versions of Microsoft Windows [LS99] provide KeSetSwapContextNotifyRoutine, for example. Likewise, the functional recovery routines [OS/98] and related mechanisms of the OS/390 MVS operating system allow sophisticated transactions of this type.

We have made use of the SPARC `%asi` register to force failing committing stores to trap. Other processor architectures provide similar mechanisms that could be employed to obtain similar results. The Intel IA32 architecture has segment registers that could be used in a similar manner. Similarly, Hudson et al. [HMSW00] propose a number of clever allocation sequences that use the predicate registers of the IA64 to annul the effects of interrupted critical sections.

A primary influence on our development of MP-RCS and its application to the implementation of PLABs is the paper by Shivers et al. [SCM99] 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 PCLSRing [Baw93], which ensured that when any thread examined another thread's state, the observed state was consistent. The basic mechanism was a set of protocols to either roll a thread's state forward or backward to a consistent point at which the thread could be suspended.

Our approach differs from these other efforts in several important ways. First, our mechanism, MP-RCS, works on multiprocessors. Second, an important aspect of how we differ from the work of Bershad [Ber93, BRE92], for example, is that we react to, rather than constrain, preemption. This difference allows us to avoid limiting the decisions of the underlying scheduling mechanisms. Further, we have demonstrated that the ability to reopen a transaction and check that the relevant state of the per-processor resources have not changed is effective in overcoming the occurence of noninterfering cases of preemption. Finally, we have exposed the MP-RCS service through the critical-section interface so that processor-centric transactions may be expressed in high-level languages exposing to the application full control over how the per-processor resources are organized and how contention is handled.

8 Future Work

Directions for future exploration of MP-RCS fall into two categories: implementation and extension of the service, and new areas of application.

8.1 Implementation

One benefit lost with `%asi`-based MP-RCS is the ability to maintain thread-locally cached information about per-processor resources and to exploit this information to reduce the cost of locating these resources in MP-RCS transactions. We are considering adding support for thread-local state that is cleared during ON-PROC notification but that is otherwise available for this purpose. This support would enable MP-RCS transactions to make use of cached information about the locations of per-processor resources without the need to recalculate those locations unless the thread is preempted.

We are also considering other approaches to implementing the basic service. One thought is to replace use of the %asi register in favor of state in each thread's `schedctl` block. Doing so would enable us to avoid some current limitations—inability to make calls to `memset`, for example—but would come at the cost of using memory operations to initiate MP-RCS transactions.

8.2 Areas of application

In terms of new areas of application, there are many opportunities. One is to revisit our work on lf-malloc [DG02] in order to remove the remaining mutexes, to improve the affinity of superblocks to processors, and to make the allocator truly lock-free.

Dave Dice has developed a facility to support multiword transactions on per-processor data using MP-RCS. We would like to investigate this facility and see how it may be used to simplify the kinds of transactions we saw, for example, when refilling PLABs. Among the challenges are how different uses of such multiword transactions by different services compose and what the overhead of the facility is.

When waiting to perform some action on behalf of a particular processor, we envision using ON-PROC notification to implement smarter spin-loops and hand-offs between threads. For example, if a thread is spinning to gain exclusive access to a global resource so that part of that resource may be allocated to the thread's current processor, and that thread is preempted before it gains that access, then we do not want that thread to continue spinning once it resumes execution.

We are also interested in investigating how MP-RCS preemption notification may be used to augment scheduling decisions, for example, within a JVM. Such events seem opportune moments to sample or profile the state of the thread, to decide to remain suspended if we are at a safe point and the system wishes to suspend all threads, or to choose to yield control to a different thread if we wish to override the scheduler's choice.

9 Conclusion

In this report, we have continued our study of Multiprocessor Restartable Critical Sections (MP-RCS). We have presented a new mechanism based on the use of the SPARC `%asi` register that simplifies the implementation of this kind of service. We have presented a library, the Critical Section interface, that allows MP-RCS transactions for managing processor-specific resources to be expressed in high-level languages. Finally, we have applied the MP-RCS service to the implementation of per-processor local-allocation buffers. In so doing, we have demonstrated that it is competitive with per-thread local-allocation buffers despite the fact that its use does not remove any synchronization, and that as the number of allocating threads exceeds the number of available threads, support for per-processor local-allocation buffers both aids the throughput of the application by reducing the frequency of garbage collections and reduces the need to carefully tune the sizing policies used to manage the allocation of local allocation buffers.

References

[ABLL92] 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 (TOCS)*, 10(1), 1992.

[Baw93] Alan Bawden. PCLSRing: Keeping Process State Modular. Available at `ftp://ftp.ai.mit.edu/pub/alan/pclsr.memo`, 1993.

[Ber93] B. N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings 13th IEEE International Conference on Distributed Computing Systems*, pages 264–273. IEEE Computer Society Press, May 25–28 1993. Los Alamitos CA.

[BRE92] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 223–233, Boston, MA, 1992.

[Con] Standard Performance Evaluation Consortium. SPECjvm98 and SPECjbb1.03 Benchmarks. Available at: `http://www.spec.org`.

[DG02] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 163–174, Berlin, June 2002. ACM Press.

[Gre99] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, August 1999.

[Her91a] M. P. Herlihy. A methodology for implementing highly concurrent data objects. Technical Report CRL 91/10, Digital Equipment Corporation, Cambridge Research Lab, Cambridge, MA, 1991.

[Her91b] M.P. Herlihy. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems*, 13(1):123–149, January 1991.

[HMSW00] Richard L. Hudson, J. Eliot B. Moss, Sreenivas Subramoney, and Weldon Washburn. Cycles to recycle: Garbage collection on the IA-64. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume

36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[JH94] Theodore Johnson and Krishna Harathi. Interruptible critical sections. Technical report, Department of Computer Science, University of Florida, 1994. Technical Report TR94-007.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[LS99] Jacob Lorch and Alan Jay Smith. The VTrace Tool: Building a System Tracer for Windows NT and Windows 2000. *MSDN Magazine*, October 1999.

[MDP94] David Mosberger, Peter Druschel, and Larry L. Peterson. A fast and general software solution to mutual exclusion on uniprocessors. Technical report, Department of Computer Science, University of Arizona, June 1994. Technical Report 94-07.

[MDP96] David Mosberger, Peter Druschel, and Larry L. Peterson. Implementing atomic sequences on uniprocessors using rollforward. *Software—Practice and Experience*, 26(1):1–23, January 1996.

[MJ92] William Moran and Farnham Jahanian. Cheap mutual exclusion. In *Proceedings of the USENIX Technical Conference 1992*, 1992.

[MM01] Jim Maura and Richard McDougall, editors. *Solaris™ internals: core kernel architecture*. Sun Microsystems Press, Prentice-Hall, Englewood Cliffs, NJ, 2001.

[MS98] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[Nef98] John Neffinger. Which Java VM Scales Best? *JavaWorld*, August 1998. See also: `http://www.volano.com`.

[OS/98] *OS/390 MVS Programming: Resource Recovery*, September 1998. GC28-1739-03.

[SCM99] O. Shivers, J.W. Clark, and R. McGrath. Atomic heap transactions and fine-grain interrupts. In *Proceedings of International Conference on Functional Programming*, Paris, September 1999.

[SS95] Christopher Small and Margo Seltzer. Scheduler activations on bsd: Sharing thread management state between kernel and application. Technical report, Department of Computer Science, Harvard University, 1995. Harvard Computer Systems Laboratory Technical Report TR-31-95.

[TS94] Hiroaki Takada and Ken Sakamura. Real-time synchronization protocols with abortable critical sections. In *Proceedings of the First Workshop on Real-Time Systems and Applications*, 1994.

[WG98] Derek White and Alex Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, December 1998.

About the Authors

David Dice currently works at Sun Microsystems on threading and synchronization in the Java Virtual Machine. Before joining Sun Microsystems, he was a co-founder of Praxsys Technologies.

Alex Garthwaite is a member of the Iceberg project at Sun Microsystems Laboratories in Burlington, Massachusetts. His research interests include investigating the boundary between Java virtual machines and operating systems as well as the implementation of runtime systems and garbage collectors.

Derek White is a Staff Engineer in the Iceberg project at Sun Microsystems Laboratories in Burlington, Massachusetts, working on integrating the Java platform and Solaris for better performance and scalability. Other interests include performance and heap analysis tools, JVM performance, and incremental development environments.

Previously he worked on garbage collection and JVM performance issues at Sun Labs, a JVM for an unnamed 64-bit OS at Novell, and the Dylan programming language runtime and development environment at Apple Computer, Inc.