Supporting Per-Processor Local-Allocation Buffers Using Lightweight User-Level Preemption Notification

Alex Garthwaite
Sun Microsystems
1 Network Drive
Burlington, MA 01803
alex.garthwaite@sun.com

David Dice
Sun Microsystems
1 Network Drive
Burlington, MA 01803
dave.dice@sun.com

Derek White
Sun Microsystems
1 Network Drive
Burlington, MA 01803
derek.white@sun.com

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.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection)*

General Terms

languages, performance

Keywords

restartable critical sections, locality, memory allocation

VEE'05, June 11-12, 2005, Chicago, Illinois, USA.
ACM 1-59593-047-7/05/0006.

1. INTRODUCTION

A key aspect of supporting scalable multi-threaded programs is the efficient management of shared resources like memory. One common approach 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. Such an approach works well if the threads are each able to make good use of the allotted resources. However, as the number of threads exceeds the number of processors and as the rate of context switching rises, runtime systems must rely on sophisticated allotment policies to continue to provide good resource utilization.

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 using multi-processor restartable critical sections (MP-RCS). Compared to thread-local management, this approach 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 on notification only 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.

In using MP-RCS, we build on the work of Dice *et al.* [6] which employed an *upcall* mechanism. With the upcall mechanism, overhead is only incurred for those threads in transactions at the time they come back on-processor, and the mechanism is general enough that threads may use it to directly maintain thread-specific references to processor-local data. However, this mechanism also requires the maintenance of a mapping between processors and threads in the kernel, and 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. We 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.

Dice *et al.* used this upcall-based mechanism to implement a highly scalable malloc library. Much of the performance improvement arose from the elimination of memory-barriers, atomic instructions, and mutexes from most of the commonly executed paths in the malloc interface. In this study, we apply the MP-RCS service to a different problem: the implementation of local-allocation buffers (LABs) in a garbage-collected system.

LABs are more challenging in two ways. 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 use of LABs remains a challenge 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 and that the behavior of the Java platform is more consistent under a wider range of conditions. 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 LABs.

1.1 Roadmap

In section 2, we describe our new implementation of MP-RCS using a register to control the behavior of memory operations on SPARC® processors. In section 3, we present a library of MP-RCS primitive operations, the critical-section interface, that allow MP-RCS transactions to be written in high-level languages. In section 4, we introduce the problem of sizing local-allocation buffers in a garbage-collected Java virtual machine, and in section 5, we examine how the critical-section interface may be used to implement per-processor LABs. In section 6, we offer performance results demonstrating the efficiency of per-processor LABs. Finally, we consider related work and conclude.

2. MP-RCS

Multi-processor restartable critical sections (MP-RCS) provide a facility that provides two complementary services: a mechanism to notify threads when they are preempted, and access to knowledge of where the thread is currently executing. These services are combined to implement non-blocking [7] algorithms that manipulate processor-specific data in a consistent manner and without interference from other threads. MP-RCS allows a thread executing a critical section 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 deschedules 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 [12], 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 in some cases, signal handling.

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 a thread as that threads is descheduled or begins running on some processor. Support for these modules, accessed via `saveCtx` and `restoreCtx`, was first added to support performance-monitoring tools that needed to multiplex hardware 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 occurred while they were blocked. 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.

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 has been explored previously by Dice *et al.* [6].

2.1 `%asi`-based MP-RCS

We now describe a new 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. Unlike the upcall-

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

Figure 1: %asi-based kernel driver

based mechanism, our approach does not require registering critical sections or ending a transaction with a committing store. While our new approach depends on a SPARC® - specific mechanism, the same effect can be achieved on IA32 platforms through the use of segment registers or on AMD64 processors by using the `r15` register.

The SPARC® architecture provides a global `%asi` register that affects the behavior of memory operations. The `%asi` register informs the memory system how to interpret an address while executing certain instructions, such as `sta` (store into address space). 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. 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 set the `%asi` register to ASI_RCS using a single ALU instruction. We then 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 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 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.

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 1 which shows the actions that the `%asi`-based MP-RCS kernel driver must perform as a thread transitions back onto a processor: it flips the state of `%asi` register in that thread's user-level context, if its old value indicates 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 side-effect that register, we may chain together a sequence of such stores.

An important advantage of this approach is that 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. If, when the `sta` instruction executes, it succeeds, then the

```
( 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 2: %asi-based trap handler `fixUContext()`

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 2. This handler will test to see if we are in an MP-RCS transaction and toggle the `%asi` register 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 processor-local resources. 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 simplicity, this approach has some drawbacks. For example, 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 reflect an advantage in flexibility and control as well as a cost in additional steps that the application must take on each transaction.

We must also 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, although the kernel-driver has adjusted the value of `%asi`to `ASI_INVALID`, on exiting the signal handler, all registers including `%asi`are restored to their previous values. The signal handler must ensure that `%asi`is reset to `ASI_INVALID` in the underlying frame as shown in figure 2. We enforce this requirement on exit from any handler by interposing on `setcontext`.

```
int csBegin();       /* Start trans; return cpu */
int csValid();       /* Still valid? */
void csEnd();        /* End trans normally */
void csInvalidate(); /* Invalidate trans */

/* Attempt store: return 0 iff success. */
int csST32C(void *addr, int32_t val);
int csCAS32C(int32_t new, int32_t old, void *addr);
int csST64C(void *addr, int64_t val);
int csCAS64C(int64_t new, int64_t old, void *addr);
```

Figure 3: Partial interface to critical-section library

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 abort or reopen the transaction in such cases.

3. 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. A subset of the interface is shown in figure 3. 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 to what one finds with load-linked (LL)/store-conditional (SC). Like LL/SC, MP-RCS transactions benefit from the fact that their use is immune to A-B-A issues [13]. The CS interface allows applications to craft MP-RCS transactions with complete control over how and where notification of preemption is handled as well as how particular resources are organized.

Because the CS interface relies simply on notification of potential interference, the `%asi`-based implementation of MP-RCS is ideally suited to support it. For example, figure 4 shows how `csBegin()` and `csST32C()` are implemented using the `%asi` register. Figure 5 shows an example of a MP-RCS transaction making use of the CS interface. In the example, we use a `cpu_id` to index a table mapping processor id's to lists. Should we be preempted before we reach a committing operation, control is always returned once 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.

4. LOCAL-ALLOCATION BUFFERS

One of the services of the garbage collector is the allocation of properly initialized memory from the heap [10]. Because the memory available for allocation is a shared resource, care must be taken to allow multiple, independent threads to perform allocations concurrently.

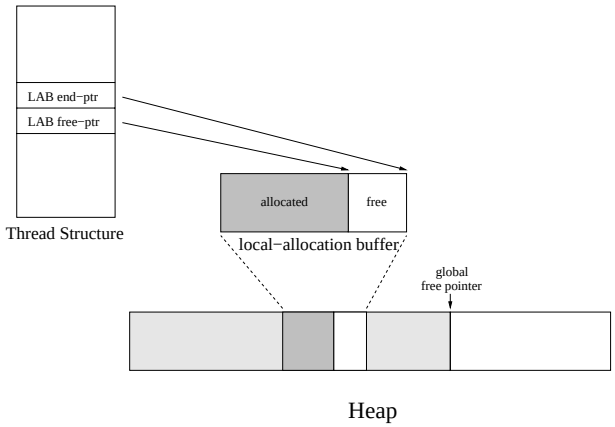


Figure 6: Per-thread local-allocation buffer

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 6 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 LABs to threads on an exclusive basis, 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 LABs—for example, as large as 64KB to 256KB.

LABs alleviate contention for allocating memory directly from the heap by reducing the frequency of such allocations. Clearly, the larger each LAB, the better 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 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]. For this study, we employed 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 [10]. Each of the two generations maintains a free-pointer which is atomically incremented to allocate memory 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.

Allocation of memory directly from the young generation is performed for one of three reasons:

[1]The ResearchVM was formerly known as the ExactVM and is available in the Java 2 SDK 1.2.2 [21].

```
!  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 4: SPARC® assembler for starting and committing actions using %asi-based MP-RCS.

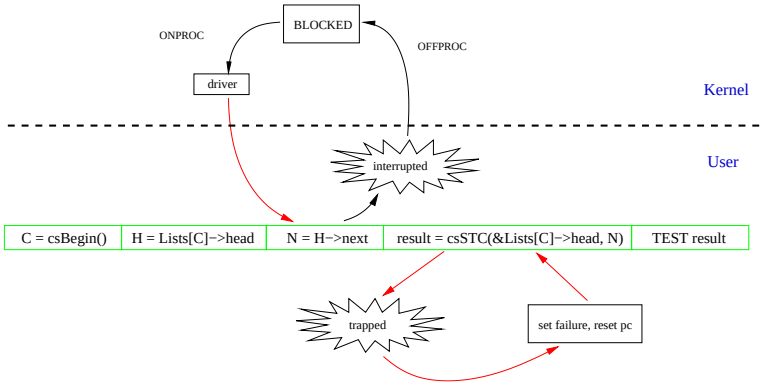


Figure 5: Manipulation of a processor-specific linked-list using the CS interface.

- to reserve memory for a thread's LAB,
- to allocate an object too large to fit in a LAB, or
- to allocate a small object when not enough memory remains to allocate a suitably sized LAB.

By properly sizing the LABs 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 category impact the efficiency of LABs in an unexpected manner. When the heap is almost exhausted, forcing the remaining allocations to be performed directly in the heap gives time to threads with unused LABs to allocate their memory. As LABs 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 LABs 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 shortly.

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 LAB 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 on a first-come basis as generations are initialized. To simplify their association with, for example, thread-specific information about LABs, these descriptions and the per-thread data structures are represented by arrays and a given description together with per-thread LAB structures associated with that description share a common index.

With this association between descriptions and particular LAB structures, the ResearchVM further specializes the second operation for each class's instances. Since instances of a non-array classs are uniform in size and this size is fixed when that class is loaded, we associate a set of allocation functions with each class for each generation. These sets allow support of precompiled allocation functions specialized for runtime constants such as LAB description index and object size, enabling the efficient choice of allocation function to be emitted in dynamically compiled code. This same framework makes it simple to support both per-thread and per-processor LABs in the same virtual machine.

When an application is suspended for a garbage collection, 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 LABs must be made to look like valid objects for the duration of the collection. In the event that a given generation is collected, we discard all outstanding LABs in the portion of the generation that is reclaimed.

Per-thread LABs work well for many applications. For example, those that have relatively few threads or whose threads are relatively 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 LABs 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 LABs have evolved adaptive sizing policies. The ResearchVM makes use of a carefully tuned sizing policy due to Ole Agesen in which LABs 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

```
( 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].
(10)                 allocateWords(Gen, thr, class, numWords);
(11)     } else {
(12)        tlabAvail(thr, labNum) = newLabAvail;
(13)        res = (word32 *)(tlab(thr, labNum) - labAvail);
(14)     }
(15)     return res; }
```

Figure 7: Code for allocating memory from a TLAB

```
( 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].
(12)                    allocateWords(Gen, thr, class, numWords);
(13)        } else {
(14)           res = (word32 *)(plab(cpuid, labNum) - labAvail);
(15)           if (csST32C(&plabAvail(cpuid, labNum), newLabAvail))
(16)              res = PLAB_RETRY;
(17)        }
(18)     } while (res == PLAB_RETRY);
(19)     return res; }
```

Figure 8: Code for allocating memory from a PLAB

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, they constrain the kinds of allocation policies supported by collection techniques. We would like to gain the benefits of LABs without unduly constraining their sizes.

In this study, we show that associating LABs with processors allows us to gain this independence in sizing policy. What makes this study challenging is that augmentation of the LAB mechanism so that LABs may be associated with processors not only does not remove synchronization from the common operations, but comes at a small potential cost in that object-allocation paths are now dependent on dynamic information for a thread, namely, the processor on which it is executing. We show that despite this cost, per-processor LABs have the same latency as per-thread LABs and that using both policies together removes the need to carefully tune LAB-sizing policies.

5. PER-PROCESSOR LABS

Among the costs incurred when using per-processor LABs (PLABs) in place of per-thread LABs (TLABs) are:

- starting a transaction and accessing the current `cpu_id`,
- accessing of that processor's resources via the `cpu_id`,
- testing the result of the committing operation, and
- including mechanisms to retry the allocation request.

To understand these costs, consider figures 7 and 8 that illustrate how objects are allocated from TLABs and PLABs. Beginning with figure 7, we identify the TLAB associated with the current generation (line 3). Using this value, we measure whether the allocation request, `numWords`, may be satisfied from the existing LAB (lines 4-6). If there is enough memory, we update the state of the LAB (lines 12 and 13). Otherwise, we allocate memory for the requested amount together with memory for a new LAB, discarding or merging the old LAB into the new one, if necessary (lines 7-10).

In contrast, allocation from a per-processor buffer is somewhat more complicated. In figure 8, 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 from the current PLAB, we must use a committing store (line 15) 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 15-16) 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 for an allocation request.

The ResearchVM's allocation framework allows each generation to register a set of allocation functions with each class as the class is loaded. This flexibility may be seen in lines 9-10 of figure 7 and in lines 11-12 of figure 8 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 for each LAB index. Using these specialized allocation functions, the typical fast-path SPARC® instruction sequence to allocate a small object from a thread-local allocation buffer requires 2 loads, 1 branch, 3 ALU instructions, and 1 store for a total of 7 instructions. In contrast, 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. So, even as PLABs may aid the throughput of highly threaded applications, it must do so well enough that it offsets the extra cost incurred on every allocation request.

5.1 Managing PLABs

Another challenge when using PLABs concerns how these buffers are allocated or refilled. When a thread attempts to refill a given PLAB, it should do so only 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. 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 is to abandon the LAB—turn it into a dead object in the heap and retry our allocation request. Instead of abandoning it, we attempt to reopen the transaction. 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. If at the end of the transaction, we have not been able to install the LAB for the current processor, we place—or *sideline*—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.

PLABs 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 [9]. For expedience, 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 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 leaves 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 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.

Despite all of these added complexities, the refilling of a PLAB is only slightly more expensive than refilling a TLAB. 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.

5.2 Flipping LAB association

The use of per-processor resources can result in poor memory utilization. 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 generation. Using the statistics for the youngest generation, we implemented a simple policy that allows us to dynamically switch association from threads to processors or back again. Initially, the application begins by using TLABs. 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 thread-local allocation buffers. Here, the ease with which we are able to implement this switching policy results from 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 lazily 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** [5]—a multi-threaded benchmark simulating a set of warehouses processing orders, and
- **VolanoMark 2.1.2** [17]—a highly multi-threaded benchmark measuring the Volano chat server.

We chose `_213_javac` from the SPECjvm98 [5] benchmark suite because it is single-threaded and because its performance is the most sensitive to the speed of allocation and to the effects of collection. We chose SPECjbb2000 so that we could study a compute- and allocation-bound application with a moderate 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

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.

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 conclude both 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.

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 4 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 in figure 9 show 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 "flips" and begins to behave like the per-processor policy. With up to 8 allocating threads and the default LAB sizing policy, PLABs do worse than either of the other two modes 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 higher preemption rate, PLABs perform significantly worse than TLABs.

Beyond 16 warehouses, PLABs perform consistently better than TLABs. This improvement is due to less wastage of memory in unused portions of LABs, and hence a corresponding reduction in the frequency of collections. The warehouse threads 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.

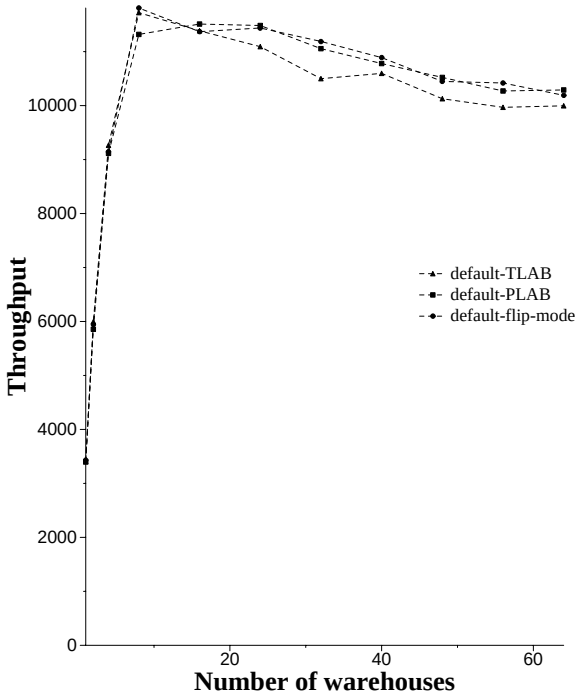


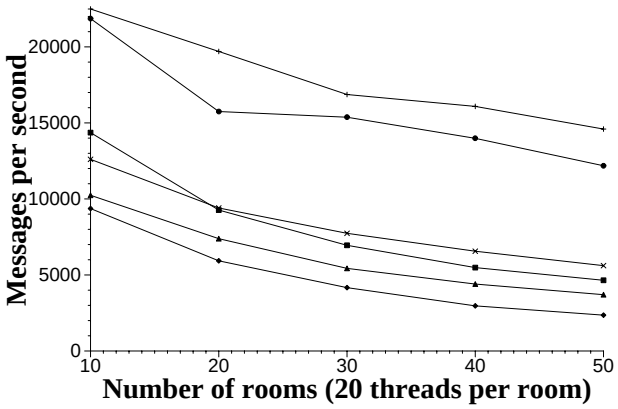
Figure 9: SPECjbb throughput scores for TLABs, PLABs, and flip-mode (default LAB sizing).

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.

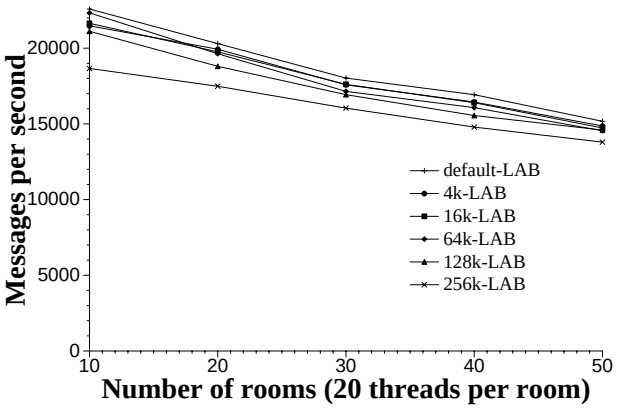
6.3 VolanoMark Results

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 figure 10.

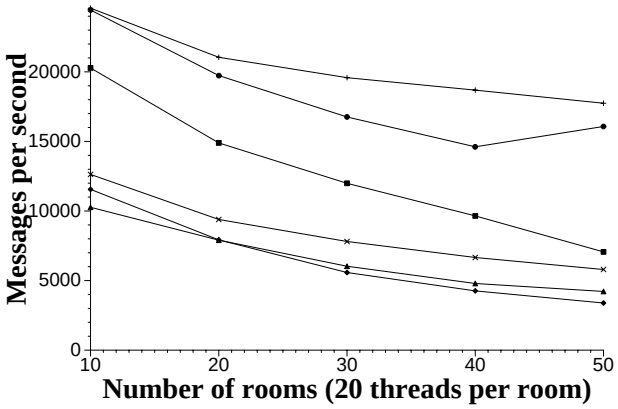
Each of the figures shows a pair of graphs, the first measuring the behavior of TLABs and the second, the behavior of PLABs. Within each graph, we graph the behavior of a particular LAB-sizing policy: the default adaptive sizing policy, and fixed-sized policies using 4KB, 16KB, 64KB, 128KB, and 512KB 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



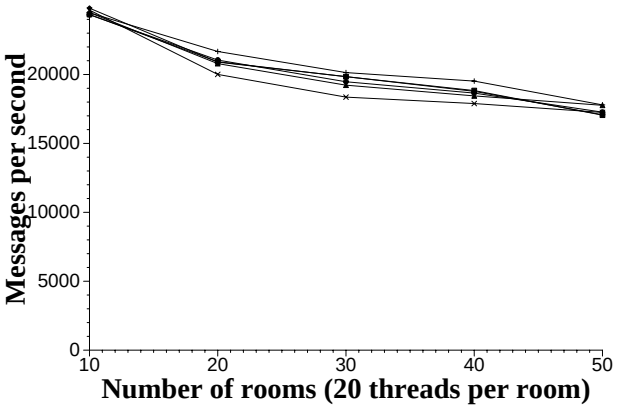
(a) 2MB semispaces with TLABs



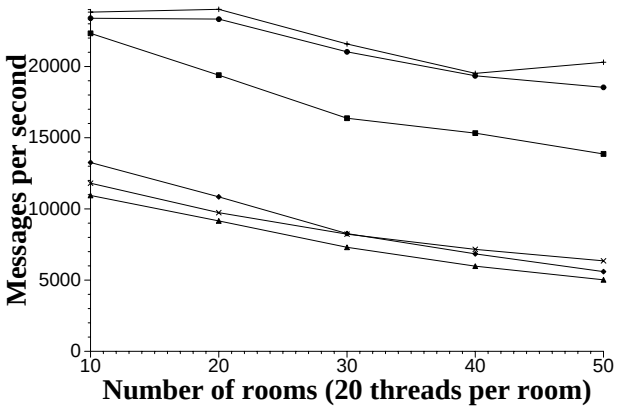
(b) 2MB semispaces with PLABs



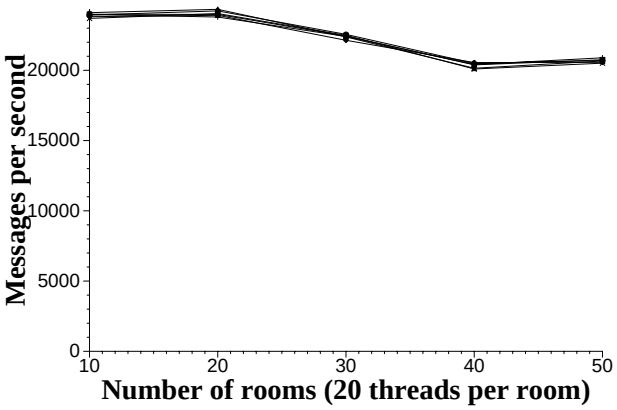
(c) 4MB semispaces with TLABs



(d) 4MB semispaces with PLABs



(e) 8MB semispaces with TLABs



(f) 8MB semispaces with PLABs

Figure 10: VolanoMark results.

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 some collection techniques and reduces the complexity of tuning JVMs for varying conditions, whether these are heap-sizing policies or the number of threads.

Table 2 shows details of the VolanoMark client and server applications when run with 64KB LABs, a 4MB young generation, and either 10 or 50 rooms. The numbers reported are the means of 10 runs. For each configuration, we report the number of young and old collections, the time spent performing collection, the number of collections where TLABs or PLABs were used, and the number of LABs and the amount of words allocated in LABs. We also report the number of successful MP-RCS transactions, the number that failed while refilling a LAB, and the number of these failures that could not be revalidated and so were invalid. Finally, we report statistics about the LABs that were sidelined, about the sidelined LABs that were discarded at a collection, and about all of the LABs that were discarded at a collection. As the table shows, with larger individual LABs, TLABs waste more memory—by up to 3 orders of magnitude—and so incur many more collections.

Second, the default, adaptively-sized policy does 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. But even with this policy, as the number of connections increases, 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 LAB size.

Third, if we look closely at the graph of TLAB results in figure 10(a), 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 [3] and [4], Bershad introduces restartable atomic sequences. Mosberger et al. [15, 16] and Moran et al. [14] describe a number of mechanisms for implementing such restartable critical sections. Johnson and Harathi describe interruptible critical sections in [9]. Takada and Sakamura describe abortable critical sections in [20].

Our implementation makes use of the `restoreCtx` hook to install a device driver to handle notification of preemption. This ON-PROC notification hook can be viewed as a simple—and less expensive—form of scheduler activation. Scheduler activations [1] make kernel scheduling decision visible to user programs. Many operating systems provide a similar mechanisms to support system tracing tools. Modern SMP versions of Microsoft Windows [11] provide KeSetSwapContextNotifyRoutine, for example. The functional recovery routines [18] and related mechanisms of the OS/390 MVS operating system also allow transactions of this type.

Other processor architectures provide similar mechanisms to the `%asi` register. The Intel IA32 architecture has segment registers that could be used in a similar manner. Similarly, Hudson et al. [8] propose a number of allocation sequences that use the predicate registers of the IA64 to annul the effects of interrupted critical sections.

One influence on our development of MP-RCS and its application to PLABs is the work of Shivers *et al.* [19] on atomic heap transactions. There, 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, PCLSRing [2], allowing any thread to examine safely another thread's state.

Our approach differs from these other efforts in several important ways. First, our mechanism works on multiprocessors. Second, unlike the work of Bershad [3, 4], for example, we react to, rather than constrain, preemption. As such, we avoid limiting the underlying scheduling mechanisms. Lastly, 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. CONCLUSION

We have presented a new mechanism for MP-RCS 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 processor-local allocation buffers. In so doing, we have demonstrated that it is competitive with 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 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.

9. REFERENCES

[1] 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.

[2] A. Bawden. PCLSRing: Keeping Process State Modular. `ftp.ai.mit.edu:/pub/alan/ pclsr.memo`, 1993.

	10 Rooms			50 Rooms		
	TLABs	PLABs	Flip-mode	TLABs	PLABs	Flip-mode
# of young-gen collections	176.2	11.1	13.1	958	55.3	58
# of old-gen collections	2	2.1	2.1	2	1.8	1.8
GC time (msecs)	143.5	12.1	13.6	3,195.7	278.1	283
Using TLABs	176.2	0	3.1	958	0	3
Using PLABs	0	11.1	10	0	55.3	55
# of LABs allocated	10,486.5	700.3	663.3	57,990.7	3,044.3	3005.8
MBs allocated in LABs	654.7	33.7	40.8	3,623.7	180.2	187.2
Successful transactions	0	701.2	528.3	0	3,044.5	2,881.1
Failed transactions	0	1.7	1.3	0	1.3	1.1
Invalid transactions	0	1.4	1	0	0.7	0.3
# of sidelined LABs	0	2.6	1.5	0	1.5	2
MBs in sidelined LABs	0	0.153	0.094	0	0.094	0.125
# of unused sidelined LABs	0	0	0.1	0	0	0.00625
MBs of unused sidelined LABs	0	0	0	0	0	0
# of partially used LABs	10,687.3	62.4	182.3	59,342.8	298.6	423.2
MBs wasted in partially used LABs	641.1	0.937	7.818	3,576.6	2.792	9.262

(a) Statistics for the VolanoMark client

	10 Rooms			50 Rooms		
	TLABs	PLABs	Flip-mode	TLABs	PLABs	Flip-mode
# of young-gen collections	177.7	8	8.3	1297.8	43.2	44.7
# of old-gen collections	0	0	0	0	0	0
GC time (msecs)	155.6	9.6	10.8	4,142.8	155.2	187.7
Using TLABs	177.7	0	1	1,297.8	0	1
Using PLABs	0	8	7.3	0	43.2	43.7
# of LABs allocated	10,565.7	540.9	463.5	79,499.2	2,430.7	2,336.9
MBs allocated in LABs	660.0	25.7	28.6	4,968.4	143.8	145.7
Successful transactions	0	541.4	409.4	0	2,430.9	2,282.1
Failed transactions	0	0.9	1.5	0	1.3	1.1
Invalid transactions	0	0.5	0.7	0	0.8	0.2
# of sidelined LABs	0	1.4	2.4	0	1.5	1.3
MBs in sidelined LABs	0	0.0875	0.1500	0	0.09375	0.8125
# of unused sidelined LABs	0	0	0	0	0	0
MBs of unused sidelined LABs	0	0	0	0	0	0
# of partially used LABs	10,765.8	47.5	95.3	80,854.6	257.6	309
MBs wasted in partially used LABs	652.9	0.7855	3.4797	4,943.16	4.5266	7.1105

(b) Statistics for the VolanoMark server

Table 2: LAB statistics for the VolanoMark benchmark with 64KB LABs and a 4MB young generation.

[3] B. N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings 13th IEEE Distributed Computing Systems*, May 1993.

[4] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In Proceedings, ASPLOS'92, Boston, MA, 1992.

[5] S. P. E. Consortium. SPECjvm98 and SPECjbb1.03 Benchmarks. Available at: `http://www.spec.org`.

[6] D. Dice and A. Garthwaite. Mostly lock-free malloc. In D. Detlefs, editor, *ISMM'02*, Berlin, June 2002.

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

[8] R. L. Hudson, J. E. B. Moss, S. Subramoney, and W. Washburn. Cycles to recycle: Garbage collection on the IA-64. In T. Hosking, editor, *ISMM'00*, Minneapolis, MN, Oct. 2000. ACM Press.

[9] T. Johnson and K. Harathi. Interruptible critical sections. Technical report, Department of Computer Science, University of Florida, 1994. TR94-007.

[10] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[11] J. Lorch and A. J. Smith. The VTrace Tool: Building a System Tracer for Windows NT and Windows 2000. *MSDN Magazine*, Oct. 1999.

[12] J. Maura and R. McDougall. *Solaris™ internals: core kernel architecture*. Sun Microsystems Press, Prentice-Hall, Englewood Cliffs, NJ, 2001.

[13] 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.

[14] W. Moran and F. Jahanian. Cheap mutual exclusion. In *Proceedings, USENIX Technical Conference*, 1992.

[15] D. Mosberger, P. Druschel, and L. Peterson. A fast and general software solution to mutual exclusion on uniprocessors, 1994. TR 94-07.

[16] D. Mosberger, P. Druschel, and L. L. Peterson. Implementing atomic sequences on uniprocessors using rollforward. *Software: Prac. & Exp.*, 26(1), Jan. 1996.

[17] J. Neffinger. Which Java VM Scales Best? *JavaWorld*, Aug. 1998. See also: `http://www.volano.com`.

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

[19] O. Shivers, J. Clark, and R. McGrath. Atomic heap transactions and fine-grain interrupts. In *ICFP'99*, Paris, Sept. 1999.

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

[21] D. White and A. Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, Dec. 1998.