USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The following paper was originally published in the

Proceedings of the USENIX Annual Technical Conference

Monterey, California, USA, June 6-11, 1999

The Region Trap Library: Handling Traps on Application-Defined Regions of Memory

Tim Brecht
University of Waterloo
Harjinder Sandhu
York University

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

The Region Trap Library: Handling Traps on Application-Defined Regions of Memory

Tim Brecht
Department of Computer Science
University of Waterloo, Waterloo, ON Canada
brecht@cs.uwaterloo.ca

Harjinder Sandhu
Department of Computer Science
York University, Toronto, ON Canada
hsandhu@cs.yorku.ca

Abstract

User-level virtual memory (VM) primitives are used in many different application domains including distributed shared memory, persistent objects, garbage collection, and checkpointing. Unfortunately, VM primitives only allow traps to be handled at the granularity of fixed-sized pages defined by the operating system and architecture. In many cases, this results in a size mismatch between pages and application-defined objects that can lead to a significant loss in performance. In this paper we describe the design and implementation of a library that provides, at the granularity of application-defined regions, the same set of services that are commonly available at a page-granularity using VM primitives. Applications that employ the interface of this library, called the Region Trap Library (RTL), can create and use multiple objects with different levels of protection (i.e., invalid, read-only, or read-write) that reside on the same virtual memory page and trap only on read/write references to objects in an invalid state or write references to objects in a read-only state. All other references to these objects proceed at hardware speeds.

Benchmarks of an implementation on five different OS/architecture combinations are presented along with a case study using region trapping within a distributed shared memory (DSM) system, to implement a region-based version of the lazy release consistency (LRC) coherence protocol. Together, the benchmark results and the DSM case study suggest that region trapping mechanisms provide a feasible region-granularity alternative for application domains that commonly rely on page-based virtual memory primitives.

1 Introduction

Modern operating systems typically export to the user-level the ability to manipulate the protection levels of virtual memory pages and to handle traps to those pages from within an application. Although originally intended for user-level virtual memory management, these mechanisms have been used in many application domains beyond those for which they were originally designed, including distributed shared memory, persistent stores, garbage collection and checkpointing. Unfortunately, pages are often the wrong unit for data management, since their size is fixed by the operating system and architecture and this size usually has little in common with the size of variable-length data objects defined within an application. When multiple data objects with different access patterns occupy the same virtual memory page, a trap to one object on the page may adversely affect the state of any other object on the same page. Conversely, for data objects that cross page-boundaries, traps must typically be handled one page at a time and may incur greater overhead than necessary.

A variety of mechanisms have been proposed for managing data at finer granularity or at a granularity defined by the application. There has been much research, for example, on the use of software checks, inserted by the compiler prior to memory references, to determine the status of persistent objects in a persistent store (e.g., White [26] and Moss [18]). Some systems have also used software checks to implement fine-grained sharing or write collection in a distributed shared memory (DSM) system e.g., Shasta [22], Blizzard-S [23], and Midway [6]). Although the trade-offs between incurring a small software check overhead on common memory references versus a large overhead on traps (which are much less common) have been studied from an efficiency point of view (Hosking and Moss [10], Zekauskas *et al.* [28], Thekkath and Levy [25]), one of the advantages of VM trap handling mechanisms seems to be their availability on most modern operating systems and architectures. Other object-based DSM systems have explored the use of program-level constructs to explicitly inform the system when shared objects are referenced (e.g., Orca [5], Amber [9], Midway [6] and CRL [13]). Although

these systems avoid the need for both software checks and traps, they also impose a more restrictive programming model on the user.

In this paper, we present the design of mechanisms for providing, at the granularity of application-defined regions of memory, the same set of services that are commonly available at a page-granularity using virtual memory primitives. These mechanisms, implemented as a C library called the Region Trap Library (RTL), use a combination of pointer swizzling and virtual memory protection to provide a portable set of primitives for manipulating protection levels to regions and for handling traps to regions from within an application domain (and they do not depend on features of a particular programming language). Thus, for example, if three regions A, B, and C use the RTL interface and occupy the same virtual memory page, and A is in an invalid state, B is in a read-only state, and C is in a read-write state, the RTL mechanisms will generate a trap on a read or write reference to A or a write reference to B, but allow read/write references to C and read references to B to proceed at hardware speeds. The RTL mechanisms also allow an application domain to map the same region at different protection levels for different threads within the same address space, and to determine both the set of regions that have been modified since a previous check and the set of modified addresses within those regions. Together, these services form a superset of those commonly offered by the operating system through VM primitives at a page-granularity and listed by Appel and Li in their paper discussing the requirements of application domains that make use of VM primitives [4].

One of the main contributions of the mechanisms described in this paper is that they are general purpose (i.e., they can be used in different application domains and languages) and they can be ported to many different modern architectures and operating systems. While earlier papers have discussed the use of pointer swizzling for object faulting [10][25], to our knowledge pointer and register swizzling have been implemented previously only within an interpreted Smalltalk environment and specifically for persistent storage [10]. Earlier systems have also not considered the problem of providing more than one level of protection using these techniques, or the problem of providing a solution that can work across different architectures or in application domains other than persistent storage and garbage collection. We have currently implemented the RTL mechanisms on several operating system/architecture combinations: Solaris/MicroSparc, Solaris/UltraSparc, IRIX/R4400, AIX/PowerPC, and LINUX/Pentium, and explored their use within a distributed shared memory system. We present some of the issues involved in the design and implementation of these mechanisms on different systems and the overhead incurred by these mechanisms on each of these different systems. We also describe the implementation and use of these mechanisms for a region-based version of the *lazy release consistency* protocol within the TreadMarks DSM system. In the DSM case study, we find that the overhead estimates for region trapping account for less than 1% of the parallel execution time in five of the six applications examined, and 6% in the other application. The use of regions rather than pages for sharing data also leads to significant improvements (up to 41%) in performance for the applications used in this study.

An overview of this paper follows. Section 2 presents some of the background and related work for the ideas presented in this paper. Section 3 describes the design and implementation of the Region Trap Library. Section 4 presents the results of some micro-benchmarks that compare the overhead of region trapping mechanisms to page-based VM primitives. Section 5 presents our case study using region trapping within a DSM environment. In Section 6 we discuss the potential and limitations of our approach and present our conclusions in Section 7.

2 Background and Related Work

Our goal in this paper is to explore the design and implementation of mechanisms for modern architectures and modern operating systems that offer the same set of services typically provided through virtual memory primitives, but at the granularity of user-defined regions. Appel and Li, in their paper on virtual memory (VM) primitives for user applications, list the set of services commonly required by applications that make use of these VM primitives [4]. These services, generalized to include region-based primitives in addition to VM primitives, are shown in Table 1. These services are available in some form or another on virtually all modern architectures but only at a page-granularity. Some very early architectures (e.g., the Burroughs 6000 series of computers [16]) once provided non-page granularity support for handling traps and managing data from user applications, but such support is not available on any modern architectures.

The mechanisms we propose use pointer swizzling for region trap handling. Pointer swizzling has been used previously in persistent object and garbage collection systems. In a persistent object system, pointers to objects may have different representations when they reside on disk than when they reside in memory, and pointer swizzling is used to update the values of pointers when objects are brought into memory or written out to disk. Most implementations use either software checks to detect references to invalid objects (see White [26] for a review) or VM page trapping mechanisms to fault ob-

Primitive	Description
TRAP	handle traps to {page,region} in user handler
PROT1	decrease accessibility of a {page,region}
PROTN	decrease accessibility of N {pages,regions}
UNPROT	increase accessibility of a {page,region}
DIRTY	return the list of dirty {pages,regions}
MAP2	map physical {page,region} at two different addresses at different protection levels, in the same address space

Table 1: Description of services typically required of applications using page-based VM primitives, and analogous region-based RTL services.

jects into memory at a page-granularity (e.g., Texas [24] and Objectstore [15]).

Hosking and Moss [10] implemented a technique called *object faulting* within an interpreted Smalltalk environment. In their strategy, pointers to a persistent object are swizzled to refer to a fault block that lies on a protected page. This fault block acts as a stand in for the object when it is not in memory. A reference to the object generates a trap, and the object is faulted in to memory. In this scheme, references to the object once it is brought into memory are indirect, unless, as suggested by Hosking and Moss, a garbage collection system is used that can recognize these indirect references and convert them to direct references. Although this strategy provides some advantages over persistent object systems that use page-granularity trapping mechanisms or software checks prior to each memory reference, its implementation is language dependent and it makes extensive use of virtual method invocation and built-in indirect references to objects within an interpreted Smalltalk environment.

Thekkath *et al.* describe the use of *unaligned access traps* for object faulting [25]. Unaligned access traps are generated by some architectures on memory references to data that should to be word aligned but is not. This mechanism was used for fast synchronization in the APRIL processor [2]. Using this approach for object faulting, pointers to an object would be swizzled so that they are unaligned, and a subsequent dereference to the pointer would generate an unaligned access trap that could be handled by the application. However, this strategy, while providing a language independent solution to object granularity trap handling, cannot be used on architectures that do not support unaligned access traps (e.g., the PowerPC architecture), and will only work in limited cases on architectures that support unaligned traps for some memory reference instructions but byte level accesses for others (e.g., the SPARC architecture).

The mechanisms we describe in this paper are similar in some respects to these latter two strategies, but are designed to provide greater functionality using mechanisms that do not depend on features of a particular programming language, and in an architecturally portable fashion. Many applications that use page-based virtual memory primitives require the ability to trap on read/write references to inaccessible pages *or* on write references to read-only pages. However, the use of unaligned access traps, as well the strategy proposed by Hosking and Moss, provide only the ability to trap on inaccessible objects. One level of protection may be sufficient in the context in which these earlier strategies have been proposed (to fault objects into memory), but they do not suffice in many other contexts. This includes checkpointing applications, where the system has to be able to detect write operations to objects that are either in a read-only or invalid state, or in the implementation of coherence protocols within a distributed shared memory system, where the system typically needs to be able to obtain updates to an object on read references in an invalid state and to mark changes to the object on write references in the invalid or read-only state.

3 The Region Trap Library

In this section, we describe the design and implementation of the Region Trap Library (RTL). We begin by describing the interface to the RTL.

3.1 RTL Interface

Applications that use the RTL must identify the areas of memory that are to be managed as regions, the set of pointers that are used to reference regions, and a handler function that will be invoked when a region trap occurs. A variety of primitives are provided for specifying each. To identify regions, an application may use a memory allocator called `region_alloc(size)` to both allocate and define a region, or define a previously allocated range of memory as a region using `region_define(addr, size)`. An additional parameter may be provided to these functions that identifies a pointer that will be used to reference that region. A region pointer identified in this way is referred to as a *bounded* region pointer, since it can only be used to refer to the specified region. Additional bounded pointers to the same region can be specified using a primitive called `region_bptr(&ptr1, &ptr2)`, which indicates that `ptr2` will be used to refer to the same region as `ptr1`. *Unbounded* region pointers, those that may refer to any region, can be specified using a primitive called `region_ptr(&ptr)`. Region pointers can be destroyed using a call to `region_ptr_free(&ptr)`. The `region_ptr` and `region_ptr_free` calls are used to maintain a list of pointers associated with each

Page-based trapping using VM primitives	RTL region trapping with a bounded region pointer	RTL region trapping with an unbounded region pointer	RTL region trapping with C++ pointer declaration

char *x;
struct sigaction s1,s2;
...
x = valloc(N);
...
s1.sa_handler = page_handler;
sigaction(SIGSEGV,&s1,&s2);
mprotect(x,N,PROT_READ);
...
a = x[0]; /* no trap */
x[1] = 1; /* trap */
x[2] = 2; /* no trap */
	char *x;
	
	...
	x = region_alloc(N,&x);
	...
	
	region_handler(x,reg_handler);
	region_protect(x,PROT_READ);
	...
	a = x[0]; /* no trap */
	x[1] = 1; /* trap */
	x[2] = 2; /* no trap */
		char *x;
		
		...
		x = region_alloc(N);
		...
		region_ptr(&x);
		region_handler(x,reg_handler);
		region_protect(x,PROT_READ);
		...
		a = x[0]; /* no trap */
		x[1] = 1; /* trap */
		x[2] = 2; /* no trap */
			region_ptr<char *> x;
			
			...
			x = region_alloc(N);
			...
			
			region_handler(x,reg_handler);
			region_protect(x,PROT_READ);
			...
			a = x[0]; /* no trap */
			x[1] = 1; /* trap */
			x[2] = 2; /* no trap */

Table 2: Example of how regions are defined in the RTL using both the C and C++-style declaration of region pointers, and compared to the analogous code using page-based virtual memory mechanisms.

Trap handler for virtual memory primitives	Trap handler for region trapping library

page_handler(signal-context)
{
char *addr = faulting_address();
...
 	
if (is_a_write_fault)
mprotect(addr,page_size,PROT_READ | PROT_WRITE);
else
mprotect(addr,page_size,PROT_READ);
}
	reg_handler(region-trap-context)
	{
	char *addr = faulting_address();
	...
	
	if (is_a_write_fault)
	region_protect(addr,PROT_READ | PROT_WRITE);
	else
	region_protect(addr,PROT_READ);
	}

Table 3: Skeletons of trap handler functions contrasting the way traps are handled using virtual memory primitives and the region trapping mechanisms.

region. Later, when protection levels are changed for a region each of these pointers will be swizzled by the RTL (see Section 3.2 for further details).

All calls to allocate or define regions, or to modify protection levels, update the region pointers that are declared by the user to reference that particular region. Region pointers can be used in the same way as other C pointers including the ability to use offsets within a region and to manipulate pointers using pointer arithmetic. The only restriction is that pointers not explicitly identified to the RTL should not be used to reference regions allocated or defined by the RTL.

The region handler function, potentially a different one for each region, can be provided as an additional parameter when the region is defined, or it may be specified separately using a primitive called `region_handler`. Region protection levels are set using a function called `region_protect`. Protection levels are specified as they are for the VM primitive `mprotect`, as `PROT_NONE` (to trap on any subsequent reference to this region), `PROT_READ` (to trap only on subsequent write references to this region), and `PROT_READ | PROT_WRITE` (to enable all read or write references to this region).

These C functions present a low-level interface that is intended to mimic corresponding page-based VM primitives whenever possible. Table 2 shows how VM primitives (column 1) and RTL primitives using this C interface (columns 2 and 3) are used for handling traps, to virtual memory pages in the former case and for regions in the latter case. Table 3 shows the skeleton of analogous application trap handler functions for both the VM and region trapping cases. While the use of C as an interface permits this library to be used in virtually any language environment, the semantics of C are such that a programmer using this interface directly would have to exercise some discipline in the way regions and region pointers are defined and used. For instance, there is no way in C to indicate to the RTL when a pointer is declared that it will be used to refer to regions, and to automatically inform the RTL library that a pointer is no longer being used to refer to regions when the scope within which it was declared has ended. This necessitates the use of the routines `region_ptr` (and `region_ptr_free` for pointers that are not declared globally) when using the C interface directly.

Within specific languages and application domains, higher-level interfaces can be built that provide more el-

egant ways to define regions, region pointers, and trap handlers. In C++, for example, the template facility provides a means to simplify the way in which region pointers are allocated and deallocated from the RTL, through a region pointer wrapper class as shown in column 4 of Table 2. Using C++ templates, the declaration of a region pointer as `region_ptr<type *>` is sufficient both as a declaration of the pointer and an indication to the RTL that this pointer will be used to refer to regions. Calls to the RTL functions `region_ptr` and `region_ptr_free` are made automatically from within the constructor and destructor for this pointer class. The access operators for these wrapper classes are overloaded so that references to a pointer declared in this way are statically replaced by the compiler with direct pointer references.

3.2 Implementing Protection Levels

Within the RTL, three separate memory areas are maintained, an *invalid* area, a *read-only* area, and a *read-write* area. The invalid area does not occupy any physical memory and may in fact be outside of the address space of the process (e.g., in the kernel's address space). The other two areas are each composed of a set of virtual memory pages that are mapped into the user address space. In the read-only area, all pages are mapped read-only, and in the read-write area, all pages are mapped read-write. Space is allocated to a specific region in each of these areas on demand and according to the protection levels that are used for that region. When region protection for a particular region is set to x (where x is invalid, read-only, or read-write), pointers to that region are swizzled so that they refer to the copy of the region occupying pages in the area mapped to protection level x. This strategy is illustrated in Figure 1.

If a region is currently in an invalid state, pointers to that region point to an area within the invalid space reserved for that region (Figure 1(a)), and a reference to that region through any of these pointers will generate a trap. If a region is currently in a read-only state, pointers to that region refer to the copy of the region in the read-only area (Figure 1(b)), and only write references to that region will generate a trap. Finally, if a region is in a read/write state (Figure 1(c)), pointers to that region are swizzled to point to a space reserved for that region in the read/write area, so that all references to that region can proceed without a trap. In this way, a lower protection level for one region will not result in unnecessary traps to another region occupying the same set of virtual memory pages, since the pointers to each respective region will simply point to different areas. Mapping the same region at two different protection levels within the same address space can be done by declaring two different sets of pointers to the same region but with different protection levels.

3.3 Region Trap Handling

When a region trap occurs, the RTL goes through the following sequence of steps:

- Determine the region to which the faulting address belongs, the region pointer(s) that refer to this region, and the application's trap handler for that region.
- If the protection level is being changed within the region trap handler, decode the faulting instruction and then swizzle both the register containing the faulting address and pointer(s) in memory referring to the faulting region. Recall that the pointers referring to each region have been identified using `region_ptr` or `region_bptr` calls.

Our prototype implementation of the RTL uses a binary search to locate region addresses stored within an AVL tree. Although faster implementations of region lookup are possible, this approach provides adequate lookup times for the applications on which we have conducted experiments. Once the region is located, the trap handler specified by the application for that region is invoked. Region pointers bound to this region and the address of the application's handler function for this region are stored within the nodes of this tree and thus require no additional effort to locate. Unbounded region pointers, if any have been declared, must be searched separately to see if any refer to this region. It is the responsibility of the application's trap handler to determine how to handle a region trap and to indicate to the RTL what state the region should be mapped to prior to returning from the handler (as in Table 3).

When `region_protect` is invoked from the application's handler, the register and all pointers known to be referring to the region (as specified as parameters to `region_alloc`, `region_define`, `region_ptr`, or `region_bptr`) are swizzled. Swizzling the register containing the faulting address requires first decoding the instruction that generated the fault in order to determine which register requires swizzling, and then modifying that register. Region pointers may point to any address within a region. Consequently, when pointers or registers are swizzled from one memory area to another, their offset from the start of the region in those respective areas must be preserved (this is what permits offsets and pointer arithmetic to be used).

A key concern with the efficiency of this strategy arises from the fact that some transitions to the read-only state for a region require *updating* the read-only copy of

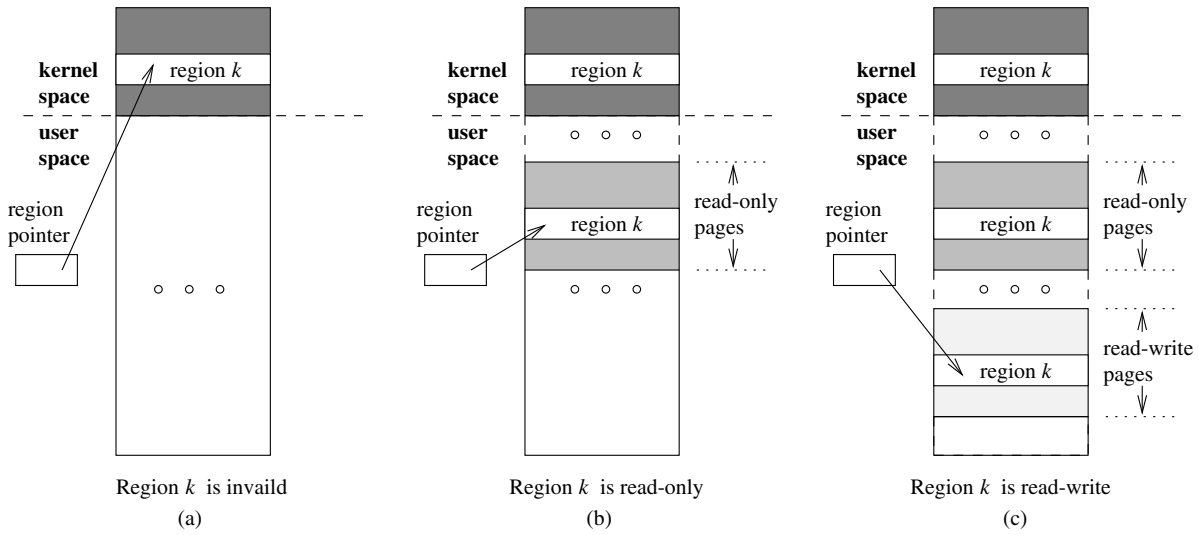

Figure 1: Setting region protection levels by swizzling region pointers between three areas: invalid, read-only, and read-write. On architectures with a large kernel address space (e.g., MIPS), the invalid area is mapped to kernel space. On other architectures, the invalid area is mapped to user space but does not occupy physical memory.

the region, in order to maintain consistency. In particular, on transitions directly from the read-write state to the read-only state, or transitions from the read-write state to the invalid state and then to the read-only state, this update must be performed. Further, the pages occupied by the read-only copy of the region are themselves in a read-only state, and they must first be unprotected (using `mprotect`) prior to beginning the update, and then re-protected once the update is complete. The performance implications of this read-only copy update are considered in the benchmarks of Section 4. Regions that transition only between read/write and invalid states or read-only and invalid states do not incur this overhead.

The allocation of the region in two areas of memory, a read-only area and a read-write area, facilitates an additional service that many applications can make use of which is often referred to as a *diff*. Using this service, an application may, at any point during execution, query the RTL to determine the set of memory locations in a region that have been modified since a previous write trap or checkpoint operation. Many distributed shared memory systems and checkpointing applications, for example, implement such a service at a page-granularity. The region-based distributed shared memory implementation within Treadmarks, described in Section 5 also exploits this service, replacing Treadmarks' existing page-based diff mechanisms with region-based diff mechanisms.

3.4 Implementation Issues

The RTL has currently been implemented on the MicroSparc and UltraSparc architectures, both running Solaris, the SGI MIPS R4400 architecture running IRIX, the IBM RS/6000 architecture running AIX, and the Pentium architecture running LINUX. Some of the issues involved in implementing these strategies on various architectures, and in particular, some of the requirements from the operating system, architecture, and compiler for an implementation of region trap handling on these and other platforms are discussed in this subsection.

Architectural requirements

Setting region protection levels to invalid requires the ability to swizzle pointers to an area that is guaranteed to generate a trap. The MIPS architecture, one of the four architectures on which we have currently implemented the RTL, makes this particularly easy. On the MIPS, addresses with the high order bit set refer to kernel space addresses, and the region trapping implementation can take advantage of this by swizzling this high order bit on pointers to regions in an invalid state. On the other architectures, our implementation creates an additional area within a processes' address space that is mapped as inaccessible and never occupies physical memory. Invalid regions are mapped to this area. These two approaches for mapping the invalid area behave the same and use the same amount of physical memory, but the latter solution potentially occupies three areas in the virtual memory address space of a process for each region.

The use of precise interrupts on protection violations by the architecture is an important prerequisite for the implementation of region trap handling. In the absence of precise interrupts, the trap handler would

have a more difficult time determining which instruction/register caused the fault, and what machine state may have been altered. Fortunately, most modern architectures (including all of the architectures discussed in this paper) use precise interrupts on protection violations.

Operating system requirements

The requirements from the operating system for an implementation of region trapping are relatively few. When a trap occurs and the RTL's trap handler is called, the operating system must provide enough processor context to the RTL trap handler to allow it to determine the faulting address, the faulting instruction, and the register that contains the faulting address. Additionally, trap recovery in the RTL requires the ability to modify and restore execution context. The versions of UNIX on which we have currently implemented the RTL (IRIX, AIX, LINUX, and Solaris) all provide this level of support.

Compiler requirements

There is some concern that the compiler might cache region pointers in registers and that these registers, which are not known to the RTL will not be swizzled upon subsequent changes in protection levels. As a result later references to the region using the cached register value will not generate the proper behaviour (either generating unwanted traps or not generating desired traps). Since protection levels are changed by calling the function `region_protect` and since the compiler should not cache values that can be changed inside of the function call across such calls, the compiler should not create these potentially dangerous register caches. The problem is exacerbated because protection levels might be changed by a function that is called asynchronously (e.g., as a result of a trap). In the benchmarks that we have run on the five different platforms, as well as in the distributed shared memory experiments we have conducted using the RTL in the SGI MIPS environment, we have not encountered any instances of this type of pointer aliasing. It is conceivable that some compilers may perform such aliasing within registers. Fortunately, if such a problem were to arise, by declaring region pointers as `volatile`, the compiler is forced to generate code that reloads a region pointer (albeit probably from the cache) each time it is dereferenced, and the compiler is then not able to create register aliases to regions. This solution is clearly more restrictive than necessary, and will result in some performance degradation. Ideally, compilers should also support a flag that prohibits the aliasing of pointers in registers without requiring these pointers to be declared as volatile.

4 Micro-benchmarks

In this section, we present the results of benchmarks that measure the overhead of region trapping mechanisms relative to standard VM page trapping primitives. These benchmarks are not designed to provide insight into overall application performance using page or region-based trapping mechanisms, since the two techniques will likely result in a different number of traps being generated. Instead, these benchmarks provide some indication of how well different architectures and operating systems support region trapping mechanisms and whether the overheads incurred by the region trapping mechanisms would be prohibitive for application domains that commonly make use of VM primitives. Section 4.1 describes the benchmarks that we use and Section 4.2 discusses the results of these benchmarks.

4.1 Benchmark Descriptions

Three benchmarks are used. The first, referred to as `trap`, measures the overhead of using trap mechanisms. For VM primitives, this is simply the cost of entering and exiting a signal handler. For RT primitives, the `trap` time also includes the additional cost (within the signal handler) of decoding the instruction, locating the region, calling a null application trap handler, and swizzling the region pointer and a register.

Appel and Li observe in their paper that applications that use virtual memory primitives typically perform one of the following two sequence of operations:

1. `Prot1`: protect one page and, on a subsequent trap to that page, unprotect the page from inside the trap handler, or

2. `ProtN`: protect a set of N pages and, on a trap to any of these protected pages, unprotect the page that caused the trap from inside a trap handler.

A comparison of these two sequences provides a better understanding of the relative overheads involved in using trapping mechanisms than simply measuring trap costs alone. Consequently, we use two benchmarks that are patterned after these two sequences. These two benchmarks, referred to as `Prot1` and `ProtN`, are constructed in the same way as described in the Appel and Li paper. In the `Prot1` test, a protected page or region is referenced and, inside the trap handler, the page or region is unprotected and another one is protected. In the `ProtN` test, 100 pages or regions are protected, and each one is referenced and unprotected one at a time within a trap handler.

A large number of repetitions of these sequences are conducted in order to obtain an average cost per sequence

OS	Arch	page	VM			RT-basic			RT-update		
		size	trap	prot1	protN	trap	prot1	protN	trap	prot1	protN
IRIX	MIPS R4400	4 KB	65	275	128	96	130	103	477	664	580
6.2	175 MHz					1.5	0.5	0.8	7.3	2.4	4.5
Solaris	MicroSparc	8 KB	286	778	591	352	365	366	2258	2539	2518
2.5.1	70 MHz					1.2	0.5	0.6	7.9	3.3	4.3
Solaris	UltraSparc	8 KB	94	200	167	112	119	127	854	910	830
2.5.1	168 MHz					1.2	0.6	0.8	9.1	4.5	5.0
LINUX	Pentium Pro	4 KB	12	39	30	23	32	28	115	198	174
2.0.0	200 MHz					1.9	0.8	0.9	9.6	5.1	5.8
AIX	PowerPC	4 KB	61	169	146	92	107	104	190	774	688
4.1	133 MHz					1.5	0.6	0.7	3.1	4.6	4.7

Table 4: Times (in microseconds) comparing VM primitives, RT-basic and RT-update. RT-basic refers to region traps on which the read-only copy of the region does not need to be updated, while RT-update refers to region traps on which the read-only copy of the region must be updated. RT region size is equal to the system page size. Numbers on the 2nd line for each system are the ratio of RT test costs relative to the corresponding VM test. Times reported are the average of multiple iterations of each test on different pages or regions.

(typically 10,000 to 100,000 repetitions were used, depending on the time taken to execute each sequence). However, because of the caching effects that occur as a result of doing multiple repetitions, all results should be considered optimistic. Additionally, the results do not consider overheads the RTL would incur as a result of swizzling multiple pointers (since the number of such pointers will typically be small and the overhead required to simply modify a pointer will be negligible) and the design of the `trap` benchmark does not consider overheads incurred to search for the faulting region. However, since the `Prot1` and `ProtN` benchmarks use 100 regions, the costs to find the appropriate region are included in those benchmarks. Although the time required to find the appropriate region in the RTL depends on the number of regions, similar overheads would be incurred using VM primitives if the action taken on a trap depends on the object generating the trap.

For all three benchmarks (`trap`, `Prot1`, and `ProtN`), region trap overheads are classified as RT-basic and RT-update. RT-basic measures the cost of region traps that *do not* require an update of the read-only copy of the region, while RT-update measures the cost of region traps that *do* require such an update. Region size plays a significant role in the cost of RT-update traps, but no role in the cost of RT-basic traps. The first set of tests use a region size that is equivalent to the size of a page on each system (8 KB on Solaris and 4 KB on the others). Subsequent tests show the effect of varying the region size on the `Prot1` benchmark.

4.2 Benchmark Results

Table 4 shows the results of the three tests on each of the systems on which our RTL prototype has been implemented. For each OS/architecture shown in this table, the second line shows the ratio of the cost of the RT benchmark relative to the equivalent VM benchmark. Thus, for example, RT-basic *trap* time under IRIX is 96 microseconds, while the VM *trap* time is 65 microseconds, resulting in a relative cost ratio of 1.5. A number less than 1 implies that the RT benchmark is faster than the equivalent VM benchmark.

Entering and exiting a signal handler requires crossing OS protection boundaries. This is fairly expensive on all architectures that we study, though the fast exception handling technique described by Thekkath and Levy [25] would significantly reduce this cost for both VM and RT primitives. Protection trap times on the LINUX/Pentium configuration are particularly low when compared with the other OS/architecture configurations, with trap times five times lower (12 μs) than the next best time (61 μs on the AIX system).

RT-basic trap costs are between 1.2 and 1.5 times more expensive than VM trap costs on all architectures except LINUX (due to its fast protection traps and more complex instruction set), where RT trap costs are 1.9 times more expensive. RT-update trap costs, as would be expected, are significantly higher, ranging from 3.1 times more expensive than VM trap costs on the AIX system to 9.6 times more expensive on LINUX. However, in our measurements of the components of this overhead, we found that for regions equivalent to or smaller than the size of a page, the cost of the `memcpy` function used to copy regions from the read-write area to the read-only area makes up only a small portion of this cost. Most of the additional cost for RT-update comes from the fact that the pages occupied by the read-only copy must be unprotected using `mprotect` before the update can begin, and then reprotected (again using `mprotect`) after

OS	Arch	region = 1 page		64 KB Regions			256 KB Regions		
		(VM)	RT-update	K pages	(VM)$*K$	RT-update	K pages	(VM)$*K$	RT-update
IRIX	MIPS R4400	275	664	16	4400	1548	64	17600	5650
6.2	175 MHz		2.4			0.4			0.3
Solaris	MicroSparc	778	2539	8	6224	8990	32	24896	23042
2.5.1	70 MHz		3.3			1.4			0.9
Solaris	UltraSparc	200	910	8	1600	1448	32	6400	3552
2.5.1	168 MHz		4.5			0.9			0.6
LINUX	Pentium Pro	39	198	16	624	464	64	2496	2606
2.0.0	200 MHz		5.1			0.7			1.0
AIX	PowerPC	169	774	16	2704	1915	64	10816	10151
4.1	133 MHz		4.6			0.7			0.9

Table 5: `Prot1` benchmarks for varying region size. K is the number of pages occupied by a region. All times shown are in microseconds. Times reported for RT-update are the average of multiple iterations of the `Prot1` test on the *same* region.

the update has completed. `mprotect`, a system call that requires crossing OS protection boundaries and shooting down TLB entries, is relatively expensive on all of the platforms we have used in conducting our experiments.

The `Prot1` and `ProtN` benchmarks reveal a very different picture than trap overheads alone. For the VM case, protection levels are set using `mprotect`. A `region_protect` in RT-basic is very cheap by comparison (typically less than 1 us), since it does not require any system calls. As a result, RT-basic `Prot1` times are smaller than VM `Prot1` times across all architectures, with differences ranging from a factor of 0.5 to 0.8. `Prot1` and `ProtN` times are almost identical for RT-basic, due to the small overhead of a `region_protect` call. For the VM `ProtN` test, protecting multiple pages in one call to `mprotect` is cheaper than protecting each page one at a time, so VM `ProtN` times are less than `Prot1` times. However, RT-basic `ProtN` times are still lower than VM `ProtN` times, by factors of 0.6 to 0.9.

RT-update `Prot1` times are significantly higher than RT-basic `Prot1` times, as would be expected. In RT-update however, since a significant proportion of the cost is due to using `mprotect` twice to unprotect and reprotect the page occupied by the region, and the VM `Prot1` test performs the same number of `mprotect` calls, RT-update `Prot1` times range from being just 2.4 times higher than VM `Prot1` on IRIX to 5.1 times higher on LINUX. RT-update `ProtN` times range from being 4.3 to 5.8 times higher than VM `ProtN` times.

4.3 Large Regions

In this subsection, we consider the effects of defining large regions that span multiple pages. Since some region traps require copying the region from one area to another, it stands to reason that region trapping overheads in these cases will increase significantly as the region size is increased, although the cost of using `mprotect` within a `region_protect` call in such cases will be amortized over the larger regions. The cost of handling traps to large regions is considered from two perspectives: (1) relative to the cost of handling a trap to a region for which the size is equivalent to one page, and (2) relative to the cost of handling a trap to a page-based strategy that would handle traps to the same number of pages as spanned by the region. For page-based strategies, the premise for the Appel and Li benchmarks is that typical applications incur a fault on each page separately regardless of how large the object is, since such strategies do not usually take application characteristics into account. Consequently, if a region spans K virtual memory pages in a region-based approach, the analogous page-based applications using VM mechanisms will likely incur K traps for every one trap incurred by the region-based approach.

Table 5 shows the results of the `Prot1` benchmark for region sizes equivalent to one 1 page, 64 KB and 256 KB. RT-basic times, which are not affected by region size, are not shown. Although experiments in the previous subsection were performed using multiple iterations of the same test across one hundred regions, this was not possible for the experiments here because of the large region sizes involved (declaring one hundred such regions results in paging on some of these machines). These tests are conducted using multiple iterations on the same region, and are thus more prone to caching effects than the experiments in the previous subsection. For comparison to the region trapping version, VM times show the cost of K `Prot1` sequences when the region spans K pages.

For 64 KB regions, although `memcpy` costs go up significantly, RT-update `Prot1` times increase by relatively small factors in the range of 1.6 (on the UltraSparc) to 3.5 (on the MicroSparc), when compared to the times for re-

gion sizes equivalent to the page size. RT-update times for the `Prot1` benchmark are lower than K `Prot1` sequences in the VM case for four of the systems studied (ranging from 0.4 to 0.9), and somewhat higher on the other (1.4 on the Solaris/MicroSparc system). This difference is reduced on some architectures and increased on others for 256 KB regions, so that the costs of RT-update relative to K VM `Prot1` tests range from 0.3 to 1.0. However, these latter comparisons assume for the VM case that every page spanned by the region would be referenced and that the VM strategy does not employ any strategy to increase the effective page size. A worst case comparison for the region trapping case would be in instances where only one of the K pages spanned by the region is actually referenced. In such cases, RT-basic costs, which are unaffected by region size, are the same relative to the VM case. However, RT-update costs would look significantly worse, by factors of 7 to 11 for 64 KB regions, when compared to a VM based strategy that handles a trap only to the page that was referenced.

4.4 Benchmark Summary

The benchmark results of this section provide some insight into the overheads involved in the use of the region trapping mechanisms described in this paper. The architectures on which region trapping mechanisms were implemented and studied vary significantly in speed and in the complexity of the instruction sets. Operating system overheads also play a significant role in these costs. Overall, despite the seemingly high overhead of keeping the read-only copy up-to-date with respect to the read-write copy, region trapping overheads are competitive with VM overheads. For instance, for regions equal to the size of a page, the `Prot1` benchmarks show region trapping to be faster by factors of 0.5 to 0.8 for transitions on which the read-only copy is not updated, and slower by factors of 2.4 to 5.1 when the read-only copy does need to be updated. For much larger regions, RTL costs in the `Prot1` benchmark are typically comparable to or lower than VM costs when K traps to a page in the VM case are considered equivalent to one trap in the RTL case for regions spanning K pages.

Since these benchmarks provide only a microscopic view of RTL and VM overheads, the question of how real applications will perform using these mechanisms cannot be answered without examining the applications themselves. In particular, the number of traps that are actually generated is entirely application dependent and is likely to be different within a page-based and region-based version of the same application. For instance, when multiple objects occupy a single page, a page-based strategy may generate more traps than a region-based strategy (as a result of false sharing for example), or fewer (if all of the objects on that page are accessed together). Conversely, if an object is much larger than a page, a page-based strategy may generate more traps than a region-based strategy if it faults on each page of the object separately, or an equivalent and perhaps fewer number of traps if not all of the pages occupied by an object are typically referenced at one time.

To obtain a clearer picture of how RTL mechanisms would behave within a real application domain, we implemented a region trapping based coherence protocol within a distributed shared memory system. This is described in the next section.

5 Case Study: Distributed Shared Memory

This section presents some results from a case study which uses region trapping within the TreadMarks distributed shared memory (DSM) system. TreadMarks uses page-based VM primitives to implement an efficient coherence protocol called *lazy release consistency* (LRC), described in detail by Keleher *et al.* [14]. We have modified TreadMarks to support a region-based version of LRC that uses region trapping to handle traps and manage data at the granularity of regions rather than pages.

For comparison, we have designed and implemented another coherence protocol, called Multiple-Writer Entry Consistency (or MEC) which is described in detail in an earlier publication [20]. This protocol is similar to *entry consistency* [6] from the programming perspective except that it uses program-level annotations that are non-synchronizing. In this paper versions of the applications that have been implemented using this protocol are referred to as the Annotated Regions (AN) versions. They behave like the region trapping versions except that they use program-level annotations rather than traps to indicate when regions are referenced for read or write.

While the annotated (AN) versions are significantly more difficult to program, a comparison between the region trapping and annotated regions versions of these applications highlights the overhead of the region trapping mechanisms. At the same time, a comparison between the page-based version and region-trapping versions highlights both the cost of using region trapping versus VM mechanisms as well as the trade-offs between using regions rather than pages for data management.

Six applications were used in this study: matrix multiplication (MM), red-black successive over-relaxation (SOR), blocked contiguous LU-decomposition (LU), a Floyd-Warshall algorithm for finding shortest paths in a directed graph (FLOYD), integer sort (IS), and the traveling salesperson problem (TSP). TSP, SOR, and IS are all from the suite of applications used in earlier TreadMarks studies [1], LU is from the Splash-2 benchmark

Treadmarks version (VM)	Region Trapping version (RT)	Annotated Regions version (AN)

float **red, **black
 	
. . .
for (i=0;i<M+1;i++) {
red[i] = Tmk_malloc(NS)
black[i] = Tmk_malloc(NS)
}
. . .
for (j=begin;j<=end;j++) {
for (k=0;k<N;k++) {
black[j][k] = (red[j-1][k] +
red[j+1][k] + red[j][k] +
red[j][k+1])/4.0;
}
}
	float **red, **black
	
	. . .
	for (i=0;i<M+1;i++) {
	red[i] = region_alloc(NS, &red[i])
	black[i] = region_alloc(NS, &black[i])
	}
	. . .
	for (j=begin;j<=end;j++) {
	for (k=0;k<N;k++) {
	black[j][k] = (red[j-1][k]+
	red[j+1][k] + red[j][k] +
	red[j][k+1])/4.0;
	}
	}
		float **red, **black
		int *redX, *blackX
		. . .
		for (i=0;i<M+1;i++) {
		red[i] = region_alloc(NS,&redX[i])
		black[i] = region_alloc(NS,&blackX[i])
		}
		. . .
		for (j=begin;j<=end;j++) {
		writeaccess(blackX[j])
		readaccess(redX[j-1])
		readaccess(redX[j+1])
		readaccess(readX[j])
		for (k=0;k<N;k++) {
		black[j][k] = (red[j-1][k] +
		red[j+1][k] + red[j][k] +
		red[j][k+1])/4.0;
		}
		}

Table 6: Snapshots of some code within SOR using VM, RT, and AN. M is the number of rows and NS is the size of each row.

suite [27], and MM and FLOYD were written locally.

5.1 Programming with Regions

Table 6 contrasts how one of the applications used in our study, SOR, is written to use each of the three protocols that we compare, page-based LRC (VM), region trapping LRC (RT), and annotated regions LRC (AN). Only a portion of SOR is shown but the example illustrates the program-level differences between these three approaches. In the original TreadMarks system, shared data must be allocated dynamically using the `Tmk_malloc` routine. In the original VM version of SOR (obtained with the TreadMarks distribution), each row of the two matrices (called `red` and `black`) is allocated separately using `Tmk_malloc`. In the RT version, each row is defined as a region by changing the `Tmk_malloc` call to `region_alloc` and providing a pointer to that region as a parameter. The rest of the code is identical for both page-based and region trapping versions of SOR. In the annotated regions version, each region is explicitly associated with a region identifier for that region (`redX` and `blackX` in the example), that is used in subsequent `readaccess` or `writeaccess` calls to identify a series of references to the region. Obviously a key motivation for implementing the RTL is to avoid having to annotate programs as shown in the AN example.

The other five applications required similar modifications to implement region trapping and annotated versions, although some of these applications use aliases to region pointers that also need to be declared as region pointers. In MM, all of the rows in a matrix used by a single processor are aggregated into a single region. In FLOYD, each row of the shared matrix is defined as a single shared region. In LU, each block is laid out contiguously and allocated separately in the original Splash-2 version. These blocks are allocated as regions in the RT version. In IS, there is a single shared data structure, a shared bucket, which is defined as a region. Finally, the original TreadMarks version of TSP allocates a single block of shared data using a single `Tmk_malloc` call. This block contains several different data structures. The RT version separates some of these data structures into separate regions, the largest of which is still about 700 KB in size.

5.2 Performance and RTL Overhead

We have conducted a series of experiments on a cluster of four 175 MHz R4400 SGI workstations connected by 155 Mbps links to a Fore Systems ASX-200 ATM switch. Table 7 shows the problem sizes used in these experiments, the size and number of regions defined in the region trap and annotated region-based versions (only the main regions are described), and the execution times of the applications on one processor and on four processors for each of the three models. Table 8 shows the number of traps that occur on a typical processor in the RT and VM versions of the applications, the RT overhead as a percentage of the runtime, and the number of messages and bytes transmitted between processors (relative to one processor) for both the RT and VM versions.

app	problem size and shared data structure	regions		execution times			
		num	size	1	VM	RT	AN
MM	640x640 matrices	9	$\approx$ 1 MB	81	33	22	22
SOR	200x4096 matrices, 100 iterations	4097	8 KB	46	27	16	15
LU	1024x1024 matrix, blocksize = 64	1024	32 KB	59	34	21	22
FLOYD	567 node graph	567	2.2 KB	137	64	48	48
TSP	19 city tour vector	1	1 MB	87	73	60	51
IS	2^{22} keys, bucketsize = 2^9, 10 iterations	1	2 KB	22	14	10	10

Table 7: Applications used in DSM study and the corresponding problem size descriptions and execution times (in seconds) on one processor and under VM, RT and AN on four processors.

app	Traps (#)		RT overhead	Messages		KBytes	
	RT	VM	% of runtime	RT	VM	RT	VM
MM	3	600	$< 0.1\%$	7	606	2458	2463
SOR	548	2545	1%	490	1863	3226	3256
LU	201	1327	0.5%	528	2068	11475	6495
FLOYD	709	1702	$< 0.1\%$	1721	2559	119	2054
TSP	309	2045	0.3%	940	3320	1077	711
IS	40	960	6%	93	1013	4021	3914

Table 8: Total number of traps generated, and messages and kilobytes transferred for RT and VM and the estimated RT overhead as a percentage of parallel execution time.

Region trapping overheads are estimated by multiplying the number of traps incurred of each type by the cost measured for that type of trap on the appropriate architecture (as shown in Table 4). These overheads account for less than 1% of the parallel execution time in five of the six applications, and 6% in the other application (TSP). Those overheads that do exist arise largely from the cost of updating the read-only copy of the region (when required) on transitions to the read-only state for a region. In TSP, this overhead is incurred within a critical section and has rippling effects on other processors waiting to enter that critical section, thereby causing a still larger difference in overall performance between the two region-based protocols (15%). The negligible difference in performance between the region trapping and annotated regions versions in all but one of these applications (TSP) suggests that region trapping costs play a minimal role in most cases. and that the significant difference in the performance between these region trapping applications and those based on VM primitives results from the differences in managing data at a region rather than page-granularity. These results demonstrate that the RTL can be used to eliminate the need for programmer annotations in such programs while maintaining efficient execution.

Compared to the VM version, the two region-based protocols improve performance in these applications by significant margins ranging from 18% in TSP to 41% in SOR on this platform. Using application-defined regions as the medium for sharing data reduces the number of traps that occur and the number of messages communicated in all six applications. Interestingly, however, the number of bytes communicated between processors is significantly higher in the region-based protocols in two of the applications, LU and TSP. While a better choice of regions might improve this situation, the increase in bytes using regions in these two applications is a result of defining large regions that span multiple virtual memory pages. These applications suffer from false sharing within regions, where modifications to the entire region are transmitted between processors on a trap even though much of region may not be used by the other processor. On the SGI platform, the reduction in the number of messages communicated between processors in these two applications compensates for the increase in the number of bytes transmitted.

It is worth noting that in our environment the RT and AN versions of MM execute significantly faster than the VM version. This is rather surprising, since other studies report near-linear speedup for page-based DSM imple-

mentations of MM. We do not obtain near-linear speedup for the VM version of MM because we use a matrix size that results in false sharing, our execution times include the time required to fault all data to remote machines, and IRIX 6.2 appears to delay the delivery of SIGIO signals to an executing process until it either blocks or its quantum expires. In some cases, this results in delays when requesting remote pages or regions. While such delays are not present in other environments we've used in previous studies we have found that the AN version of MM still outperforms the VM version (although in this case, the execution time is only improved by 12%) [20]. This earlier publication [20] also provides a more detailed discussion comparing the performance of page-based (VM) and region-based protocols (AN).

5.3 Case Study Summary

These results provide some evidence that the overhead of the region trapping mechanisms within a DSM environment are reasonable. Since trapping overheads account for a small proportion of the execution time for both RT and VM mechanisms, the key factor in determining whether region trapping is useful within the distributed shared memory context lies in the trade-offs between using regions rather than pages for managing shared data, both from a programming and performance perspective. Studies by Adve *et al.* [1] and by Buck and Keleher [7] compare the performance of page-based LRC to object-based EC. Each study suggests that page-based systems are competitive with and sometimes better than object-based systems, while a similar study by Neves *et al.* [19] has found object-based systems to be much better. However, unlike the page and region-based versions of LRC used in the study presented in this section, page-based LRC and object-based EC differ not only in the use of pages rather than objects (regions) as the granularity of data management, but also in terms of the synchronization model (release consistency versus entry consistency) and coherence protocols (lazy release consistency versus write-update) that are used. Consequently, the results of those studies are not directly comparable to those presented in this section.

The comparison between the page and region-based versions of LRC presented in this section suggest that many applications may benefit from using these region trapping mechanisms instead of the traditional page-based VM mechanisms. However, a detailed examination of these trade-offs between pages and regions within a distributed shared memory system are beyond the scope of the study presented in this section. Consequently, a number of factors that may also influence the choice of whether to use pages or regions for managing shared data have not been considered here. This includes the use of page aggregation techniques, which increase the effective size of a page [3] and would likely improve the performance of the VM case for some of the coarse-grained applications used in this study, and the use of other coherence protocols such as scope consistency [12], which captures some of the advantages of object-based protocols such as entry consistency.

6 Discussion

The benchmarks and case study of Sections 4 and 5 highlight both the potential and the limitations of the current RTL implementation. The primary limitations, both in terms of the programming interface and performance, can, as it turns out, be easily addressed. In this section, we briefly discuss these limitations and how they can be overcome in the RTL.

One of the key performance costs in the RTL, the need to update the read-only copy of the RTL, can be eliminated by mapping the read-only and read-write memory areas to the *same* physical memory area. Once this is done, all of the *RT-update* costs described in Section 3.3 and shown in the figures of Section 4 are eliminated since only a single physical copy of the region needs to be maintained (unless a copy is required in order to compute *diffs*). This also eliminates the additional physical memory overhead incurred by the current RTL implementation.

The other major concern is with the RTL interface itself, which requires all pointers to a region to be explicitly declared. In work conducted concurrently and independently of our own, Itzkovitz and Schuster [11] present an alternative approach that provides the functionality that the RTL implements but without having to manipulate pointers to those regions. Itzkovitz and Schuster's MultiView system allocates each region on a separate virtual memory page, but maps each of those regions to the same set of physical memory pages. This allows the virtual memory protection levels of each region to be manipulated independently while still allocating different regions on the same page.

The MultiView approach presents a simpler programming paradigm than the RTL (regions must still be identified, but pointers to regions do not). However, MultiView may also consume a significant portion of the virtual address space because a single virtual memory page must be allocated for each region, even if the region is only a few bytes in size. This leads to a potentially more serious performance drawback in that each virtual memory page requires a single TLB entry. Since most current architectures have relatively limited TLB sizes, applications that reference many small regions may generate significantly more TLB misses using the MultiView approach. In contrast the RTL approach has a small fixed virtual memory

and TLB entry overhead that is dependent only on the total size of the virtual memory consumed by the application. Further, once the RT-update costs in the RTL have been eliminated using the technique noted above, changing RTL protection levels will be a fraction of the cost of manipulating virtual memory protection levels. These latter costs, using the expensive `mprotect` system call, are required both in traditional page-based approaches and by MultiView. Thus while the MultiView approach offers clear advantages as far as the programming interface is concerned these performance issues may make the RTL approach more suitable for very fine-grained sharing.

7 Conclusions

In this paper, we have described the design and implementation of the Region Trap Library, which provides the same functionality, at the granularity of user-defined regions, that application domains typically require from virtual memory primitives at a page-granularity. One of the main contributions of the mechanisms used in this library is that they do not depend on features of a particular programming language and they are portable across several architectures. Benchmarks on several operating systems and architectures suggests that the overhead of these mechanisms is typically competitive with their page-based counterparts. Our implementation of a region-based version of the *lazy release consistency* coherence protocol within the TreadMarks page-based DSM system demonstrates the applicability of region trapping mechanisms within some of the domains that make use of virtual memory primitives. In the DSM context, we found region trapping overheads to be typically less than 1%, with the exception of one application that incurred an overhead of 6%.

Together, the benchmark results and the DSM case study suggest that the region trapping mechanisms implemented in the Region Trap Library provide a feasible region-granularity data management alternative to VM primitives within some of the application domains that commonly rely on page-based VM primitives. Further study is needed to identify the overhead of the region trapping mechanisms within some of these other application domains, and to determine the value of using regions rather than pages as the unit of data management in these domains.

Acknowledgments

The authors gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC). We thank Graham Smith for porting the RTL to LINUX on the Pentium Pro and Diego Moscoso for discussions during the early stages of this work. We thank Ken Sevcik and the POW group at the University of Toronto for providing us with access to their IBM systems. Finally we wish to thank the anonymous referees for their careful reading of the paper and for their insightful comments.

References

[1] S.V. Adve, A.L. Cox, S. Dwarkadas, R. Rajamony and W. Zwaenepoel, "A Comparison of Entry Consistency and Lazy Release Consistency Implementations", Proceedings of the 2nd International Symposium on High-Performance Computer Architecture, February, 1996.

[2] A. Agarwal, B-H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A processor architecture for multiprocessing", Proceedings of the 17th International Symposium on Computer Architecture, pp. 104-114, May, 1990.

[3] C. Amza, A.L. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory", Proceedings of the Sixth Conference on Principles and Practice of Parallel Programming, pp. 90-99, June 1997.

[4] A.W. Appel and K. Li, "Virtual Memory Primitives for User Programs", Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 96-107, April, 1991.

[5] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, "Orca: A language for parallel programming of distributed systems", IEEE Transactions on Software Engineering pp. 190-205, March, 1992.

[6] B. Bershad, M. Zekauskas and W. Sawdon, "The Midway Distributed Shared Memory System", Proceedings of COMPCOM '93, pp. 528-537, February, 1993.

[7] B. Buck and P. Keleher, "Locality and Performance of Page- and Object-Based DSMs", Proceedings of the 12th International Parallel Processing Symposium, March, 1998.

[8] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Lettlefield, "The Amber System: Parallel programming on a network of multiprocessors", Proceedings of the 12th ACM Symposium on Operating Systems Principles, pp. 147-158, December, 1989.

[9] M. Feeley and H. Levy, "Distributed Shared Memory with Versioned Objects", Conference

on Object-Oriented Programming Systems Languages, and Applications, October, 1992.

[10] A.L. Hosking and J.E.B. Moss, "Protection Traps and Alternatives for Memory Management of an Object-Oriented Language", Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, pp. 106-119, December 1993.

[11] A. Itzkovitz and A. Schuster, "MultiView and Millipage – Fine-Grain Sharing in Page-Based DSMs", Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99), February, 1999.

[12] L. Iftode, J.P. Singh, and K. Li, "Scope Consistency: A Bridge between Release Consistency and Entry Consistency", Proceedings of the Symposium on Parallel Algorithms and Architectures, June 1996.

[13] K. Johnson, F. Kaashoek and D. Wallach, "CRL: High-Performance All Software Distributed Shared Memory", Proceedings of the 15th Symposium on Operating Systems Principles, pp. 213-228, December, 1995.

[14] P. Keleher, A. Cox, S. Dwarkadas and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", Proceedings of Winter 1995 USENIX Conference, pp. 115-131, 1994.

[15] C. Lamb, G. Landis, J. Orenstein and D. Weinreb, "The Objectstore Database System", Communications of the ACM, Vol. 34, No. 10, pp. 50-63, October, 1991.

[16] C. Lakos, "Implementing BCPL on the Burroughs B6700", Software Practices and Experience, Vol. 10, pp. 673-683, 1980.

[17] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", ACM Transactions on Computer Systems, Vol. 7, No. 4, pp. 321-359, November, 1989.

[18] J.E.B. Moss "Working with Persistent Objects: To Swizzle or not to Swizzle", IEEE Transactions on Software Engineering, Vol. 18, No. 8 pp. 657-673, August, 1992.

[19] N. Neves, M. Castro, and P. Guedes, "A Checkpoint Protocol for an Entry Consistent Shared Memory System", Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing August, 1994.

[20] H. Sandhu, T. Brecht, and D. Moscoso, "Multiple-Writer Entry Consistency", International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), pp. 355-362, July, 1998.

[21] D. J. Scales and M.S. Lam, "The Design and Evaluation of a Shared Object System for Distributed Memory Machines", Proceedings of the First Symposium on Operating System Design and Implementation, pp. 101-114, November, 1994.

[22] D.J. Scales, K. Gharachorloo, and C.A. Thekkath, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory", Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems, October, 1996.

[23] I. Schoinas, B. Falsafi, A.R. Lebek, S.K. Reinhardt, J.R. Larus, and D.A. Wood, "Fine-Grain Access Control for Distributed Shared Memory", Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 297-306, October, 1994.

[24] V. Singhal, S.V.Kakkad, and P.R.Wilson, "Texas: An Efficient, Portable Persistent Store", Proceedings of the Fifth Int'l. Workshop on Persistent Object Systems, September 1992.

[25] C.A. Thekkath and H. M. Levy, "Hardware and Software Support for Efficient Trap Handling", Proceedings of ASPLOS-IV, October 1994.

[26] S. T. White, "Pointer Swizzling Techniques for Object-Oriented Database Systems", Ph.D. Thesis, University of Wisconsin, 1994.

[27] S. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 24-36, 1995.

[28] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad, "Software Write Detection for Distributed Shared Memory", Proceedings of the First Symposium on Operating System Design and Implementation, pp. 87-100, November, 1994.