PROTECTION STRUCTURES IN THE SOMBRERO OPERATING SYSTEM

by

Shahjehan A. Khatri

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 1997

PROTECTION STRUCTURES IN THE SOMBRERO OPERATING SYSTEM

by

Shahjehan A. Khatri

has been approved

December 1997

APPROVED:

_____, Chair

_____

_____

Supervisory Committee

ACCEPTED:

_____

Department Chair

_____

Dean, Graduate College

ABSTRACT

A single address space operating system (SASOS) facilitates communication and sharing among different programs. However, since the barriers imposed by the traditional multiple address spaces are removed, protection amongst these programs becomes an issue.

Sombrero is a SASOS being implemented at ASU. The developers of Sombrero have identified a sophisticated protection model that requires special hardware. This thesis concerns itself with the Sombrero protection structures, in particular, implementation of a prototype of the software structures and algorithms, simulation of the hardware protection buffer (RPLB), and their evaluation.

The performance of the Sombrero protection model is evaluated by simulating the average cost of memory access in Sombrero and in a typical, many address space system, namely, Alpha-NT. To simulate the average memory access cost in an Alpha-NT, its address translation buffer (TLB) and virtual memory page tables are simulated. Cache access traces obtained from the Internet are used to perform accesses against the simulated hardware caches, namely, the Sombrero RPLB and the Alpha-NT TLB. Performance statistics such as cache miss rates and the attendant penalties are collected. A test and simulation environment was implemented to facilitate various experiments.

By necessity, the numbers obtained are approximate, but are useful to observe and characterize behavior differences between the two systems as various parameters are changed. Results to date show that while the RPLB miss penalty is likely to be twice as high as the TLB miss penalty, the RPLB miss rate is likely to be lower than the TLB miss rate under many common scenarios, for instance, when the memory objects accessed are likely to be larger than a regular page. In these situations performance, as measured by the product of miss rate and miss penalty, appears to be roughly comparable.

To my parents and siblings

Table of Contents

Page

List of Figures viii

1. Introduction 1

2. Background: The Sombrero Protection Model 3

2.1 The RPLB Access Matrix 3
2.2 Protection and Resource Access List (PRAL) 4
2.3 Range Mask 9
2.4 Software Protection Data Structures 11
2.5 Range Protection Lookaside Buffer (RPLB) 14

3. Methodology 16

3.1 Evaluation of the Sombrero Protection Model 16
3.2 Assumptions 18

4. Implementation 19

4.1 PRAL Prototype 19
4.1.1 Implementation Details 20
4.1.2 Binary Search of the PRAL 21
4.1.3 Hash Search of the PRAL 21
4.2 RPLB Simulation 23
4.3 TLB Simulation 23
4.4 Cache Access Traces 25
4.5 Registry Simulation 29
4.6 Testing Environment 29
4.7 Switches 30
4.8 Multi-Threading Concerns 30

5. Results 32

5.1 Performance of the RPLB Eviction Algorithms 32

5.2 Performance of PRAL Search Algorithms....34
5.3 Performance of RPLB/PRAL versus TLB/Page Tables....36
6. Conclusion and Future Work....45
6.1 Summary of Value of Research....45
6.2 Future Work....46
6.2.1 Memory Access Traces....46
6.2.2 Additional Experiments....46
6.2.3 Multi-Thread, Multi-Processor and Multi-Node Issues....46
6.2.4 RPLB Miss Handling....46
6.2.5 Universal Trusted Domain....46
6.2.6 System Calls....47
6.2.7 Unaligned Data....47
6.2.8 Principals....47
6.2.9 Code Clean-Up....47
7. Bibliography....48
8. Additional References....50
A. User's Manual....53
A.1 Execution....53
A.2 Test Environment....53
A.3 Valid Commands....54
b_stats_category:....55
b_stats_column_groups:....55
b_stats_file_open:....55
b_stats_file_close:....56
b_stats_print_miss_rate:....56
b_stats_print_miss_penalty:....57
b_stats_print_rate_times_penalty:....57
drplb_size: (see irplb_size:)....58
gpd_add:....58
gpd_del:....58
gpd_print:....59
hash_structure_build:....59
hash_structure_print:....59

irplb_size:/drplb_size:/srplb_size: 60
mo_add: 60
mo_add_specific: 61
mo_del: 62
mo_print: 62
pral_search: 62
quantum: 63
reference_set: 63
reference_del: 64
registry_print: 64
rplb_accesses:/tlb_accesses:/rplb_and_tlb_accesses: 65
rplb_clear/tlb_clear: 65
rplb_evictions: 66
rplb_stats_print:/tlb_stats_print: 66
srplb_size: (see irplb_size:) 67
switch_set: 67
switch_del: 67
thread_add: 67
thread_del: 68
thread_print: 68
tlb_accesses: (see rplb_accesses:) 69
tlb_clear: (see rplb_clear:) 69
tlb_stats_print: (see rplb_stats_print:) 69
windows_thread_add: 69

A.4 Sample Input and Output Files 70
MAIN.IN 70
MAIN.OUT 71

B. Source Code 76

C. Test Inputs and Outputs 94

MAIN.IN 94
GCC_2E20.3 95
VARY_THREADS_B.IN 95
RPLB_VS_TLB.IN 95
MAIN.OUT 96
RPLB_VS_TLB.OUT 98

D. Aligned Memory Objects 101

E. Hashing Algorithm and Implementation 103

List of Figures

Page

Figure 1: RPLB Access Matrix for a 64 bit virtual address system. 3
Figure 2: Standard access matrix. 4
Figure 3: The Protection and Resource Access List (PRAL) 5
Figure 4: Protection Domain Access Lists (PALs). 6
Figure 5: Resource Access Control Lists (RACLs). 7
Figure 6: Access Descriptor 8
Figure 7: Range Function example. 9
Figure 8: Detailed view of MOCB 1 showing the associated Region Control Blocks. 10
Figure 9: Sombrero Software Protection Data Structures. 13
Figure 10: The Operation of the DRPLB and IRPLB 14
Figure 11: The Operation of the SRPLB. 15
Figure 12: RPLB eviction algorithms: NLU vs. LRU. 33
Figure 13: PRAL search algorithms: Binary vs. hash structure vs. linear. 35
Figure 14: RPLB/PRAL vs. TLB/page tables, using 3 (1 MB) aligned memory objects. 37
Figure 15: RPLB/PRAL vs. TLB/page tables, using 15 (64 KB) aligned memory objects. 38
Figure 16: RPLB/PRAL vs. TLB/page tables, using 28 (32 KB) aligned memory objects. 39
Figure 17: RPLB/PRAL vs. TLB/page tables, using 53 (16 KB) aligned memory objects. 40
Figure 18: RPLB/PRAL vs. TLB/page tables, using 101 (8 KB) aligned memory objects. 41
Figure 19: RPLB/PRAL vs. TLB/page tables, using 177 (4 KB) aligned memory objects. 42
Figure 20: RPLB/PRAL vs. TLB/page tables, using 2,461 (128 byte) aligned memory objects. 43

1. Introduction

Sombrero [Sk*96a] is a single address space operating system (SASOS) designed to take advantage of the wide-addressing capability of modern processors (those offering 64 or more addressing bits). Traditionally, addressing bits have been in short supply, forcing operating system designers to make do by giving each process the same (overlapping) virtual address space. This practice has been successful and it is so ingrained that going back to a single address space for all processes constitutes a non-trivial paradigm shift. The old, many address space operating systems (MASOSs) like UNIX and NT are popular and offer applications adequate protection by means of address space firewalls. However, there are problems with the many address space "patch," the main one being that the multiple address spaces are an obstacle to seamless sharing and cooperation between processes. It is significant that there has been a lot of time, money, and effort expended researching how to make communication between processes more efficient.

Sombrero is the result of a systemic analysis and design of a cluster of cooperating workstations connected by a network. Based on the SASOS paradigm, Sombrero supports efficient "networking" amongst the different processes within a machine and across independent machines, providing efficient object-level protection all the while. That is made possible, to a great extent, by a special-purpose hardware device used to cache protection information. Traditionally, a Translation Lookaside Buffer (TLB) has been used as part of the virtual memory system. This buffer caches translations of virtual to physical memory addresses and also includes protection information. However, address translation and protection are orthogonal concepts. Thus, Sombrero uses a memory Range or Region Protection Lookaside Buffer (RPLB) for protection (per processor) and a translation cache for address translation (per memory bus).

Sombrero provides a distributed, seamless store that can be used as distributed shared memory (DSM). However, the scope of this thesis is limited to a single node with a single processor, albeit the data structures and algorithms implemented can be easily extended to a distributed environment.

A SASOS removes artificial, inhibitive boundaries between processes. The concept of a SASOS is not new, of course, but the way a hardware-supported SASOS is designed is. Resource access and

protection in Sombrero, including the protection model and structures, are documented in [Sk*96b]. Based on that design, a prototype of Sombrero's software protection data structures and algorithms was implemented. To estimate how the Sombrero protection model performs in comparison with a typical MASOS protection model, hardware and software structures for Sombrero and an Alpha-NT system were simulated. Furthermore, a test driver that parses a set of exogenous inputs was written. It can be used to simulate access granting and revoking, and access-right checking.

2. Background: The Sombrero Protection Model

The Sombrero protection model is carefully explained in [Sk*96b]. A brief coverage of its highlights follows.

2.1 The RPLB Access Matrix

The Sombrero protection model is embodied in the RPLB Access Matrix (RPLB Matrix for short). Figure 1 ([Sk*96b]) illustrates.

Protection Domains	All 2^64 Virtual Addresses/ Access Rights for each address														
	$VA_{0000000000000000}$			$VA_{0000000000000001}$			. . . . .			$VA_{FFFFFFFFFFFFFFFE}$			$VA_{FFFFFFFFFFFFFFFF}$		
	R	W	X	R	W	X	R	W	X	R	W	X	R	W	X
GPD-1	**A**											**A**			**A**
GPD-2	**S-3**									**A**	**A**		**A**	**A**	
GPD-3	**A**	**A**		**A**	**A**							**S-1**			
GPD-4															

Figure 1: RPLB Access Matrix for a 64 bit virtual address system.
A for *ACCEPT*; *REJECT* is indicated by a blank entry and S for *SWITCH* which returns a GPD name.

The RPLB Matrix is a three-dimensional matrix and, like the standard access matrix [Lam71] (see Figure 2 [Sk*96b]), it has protection domains as rows, but the columns are the set of all valid virtual addresses. The third axis is the set of all valid access rights (namely, read, write and execute). The RPLB Matrix, which, also like the standard matrix, is a combination of access and switch matrices, provides a Boolean "Accept"/"Reject" or, for a domain switch, a protection domain name. However, unlike the standard matrix, the RPLB Matrix is "CPU-friendly." The CPU can only see the world as addresses (not objects/resources) and permissions, and in the RPLB Matrix all objects, including domains, are represented by a unique address. In other words, the CPU emits requests to access a memory object by providing its virtual address and the type of access. The RPLB contains all the necessary information to determine whether the access is permitted.

Protection Domains \ Resources	MO1	MO2	MO3	MO4	Ptr1	Plt2	PD1	PD2	PD3
PD1	read	R/W						Enter	
PD2			read	RWX	write				
PD3					write	write			

Figure 2: Standard access matrix.
PD - Protection Domain, MO - Memory Object, Ptr - Printer, Plt - Plotter.

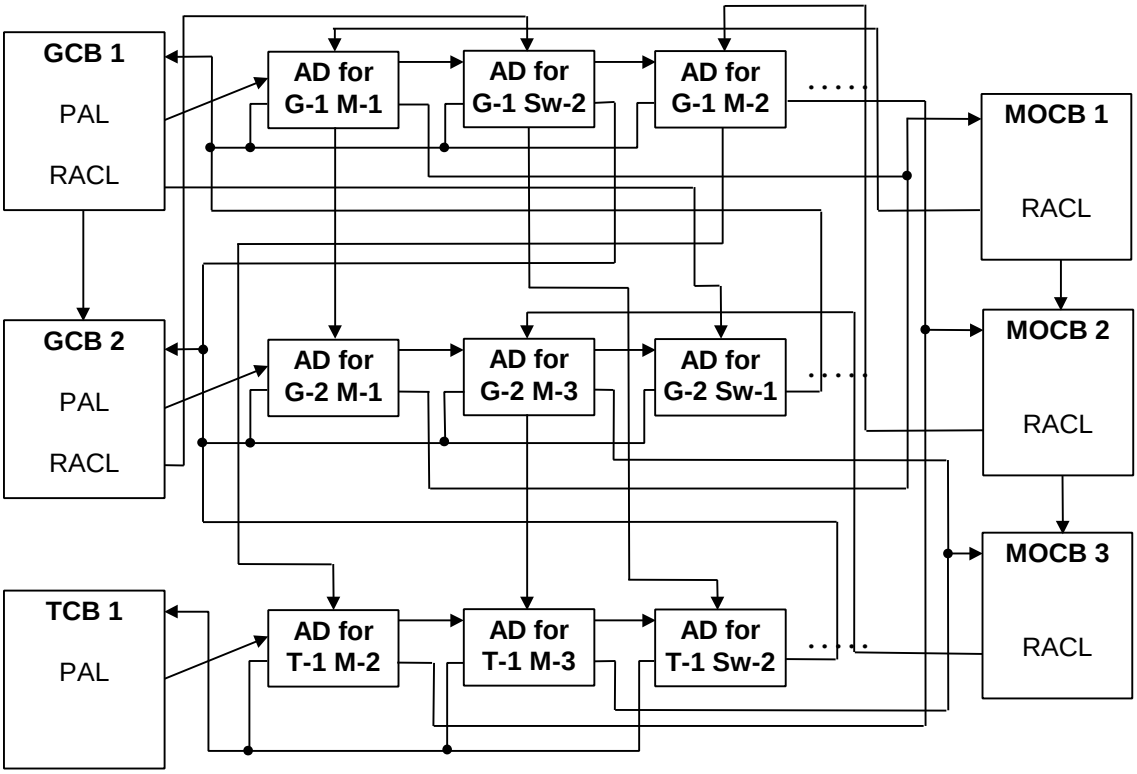
The contents of the RPLB Matrix cannot be stored directly; it's just too large. Software protection data structures (i.e., the PRAL) must organize and store the currently required subset of this data. In a process-oriented system, this data is stored in capability lists or access control lists.

The PRAL represents an integration of the standard matrix and the RPLB Matrix. The PRAL provides data that is directly usable by a CPU augmented with some additional memory protection hardware, namely, the RPLB. The RPLB is intended to work in a manner analogous to the protection operations of a modern TLB. It stores the most recent hits in an associative cache and on a miss initiates a search of the PRAL for a valid entry.

2.2 Protection and Resource Access List (PRAL)

The PRAL is a kernel-based software structure of Resource Access Control Lists (RACLs) and Protection domain Access Lists (PALs), combined together to provide efficient software protection checks and resource access revocation. The RACLs and PALs consist of Memory Object Control Blocks (MOCBs), General protection domain Control Blocks (GCBs), and Thread/carrier protection domain Control Blocks (TCBs) linked together by Access Descriptor (AD) structures. Figure 3 ([Sk*96b]), illustrates.

The General Protection Domain (GPD) and the Thread/carrier Protection Domain (TPD), when combined, form the current protection domain.



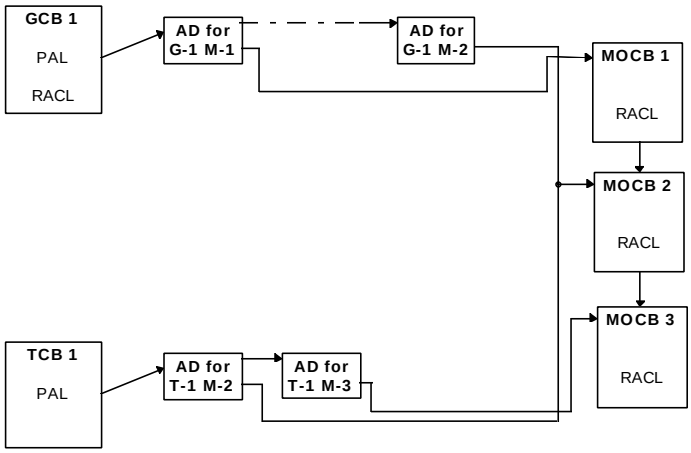
LEGEND:
AD for G-I (T-I) M-j ≡ Access Descriptor containing:
1. Access rights of GPD_i (CPD_i) to MO_j
2. Entry in MO_j's ACL for GPD_i (CPD_i).

AD for G-I (T-I) Sw-j ≡ Access Descriptor containing:
1. Entry right of GPD_i (CPD_i) to GPD_j and GPD_j entry address
2. Entry in GPD_j's ACL for GPD_i (CPD_i).

Figure 3: The Protection and Resource Access List (PRAL).
Protection Domain Control Blocks and Memory Object Control Blocks interlinked by Access Descriptors.
The ADs for switches and MOCBs are shown here mixed together. In fact, they are in separate lists.
Switches also require a switch address in the AD that is not shown here.
The region control blocks attached to the ADs and MOCBs are not shown either.

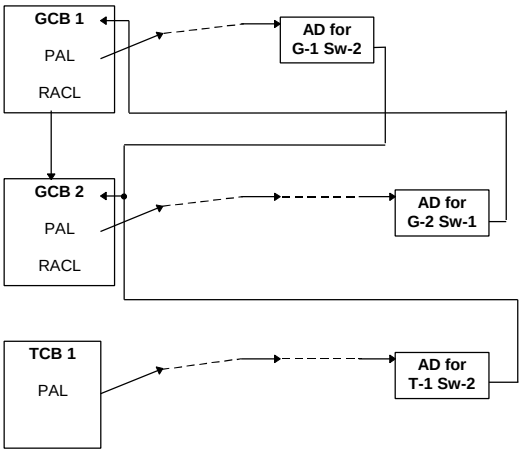
The memory objects accessible by a thread consist of the memory objects accessible by the general protection domain the thread is running in, and the memory objects accessible by the thread/carrier protection domain. The memory objects accessible by a thread may include additional objects if a thread can switch to other general protection domains.

Figure 3 is somewhat hard to follow. The figures below ([Sk*96b]) clarify.



A view of the PRAL drawing extracting an example of a GCB PAL and a TCB PAL for the MOCBs. The Switch ADs are excluded for clarity.

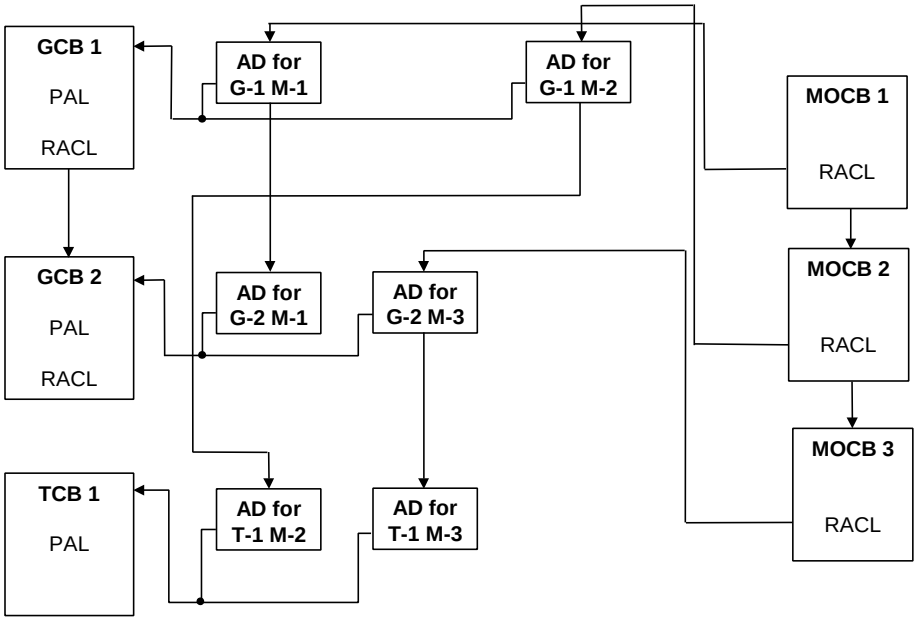
(a)



A view of the PRAL drawing extracting the GCB PALs and TCB PAL used in Switching.

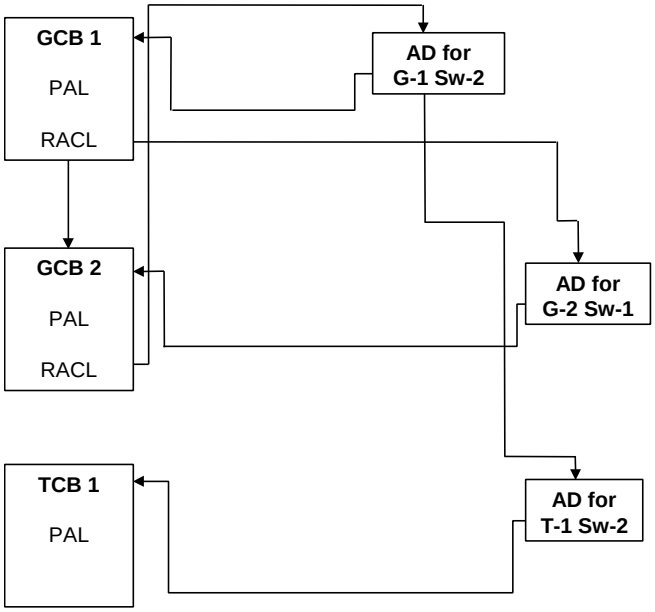
(b)

Figure 4: Protection Domain Access Lists (PALs).
(a) Memory Object Access Lists headed by GCB 1 and TCB 1.
(b) Protection Domain Access Lists for GPD 1, GPD 2 and CPD 1.



A view of the PRAL drawing extracting the MOCB RACLs.

(a)



A view of the PRAL drawing extracting the GCB RACLs used in switching.

(b)

Figure 5: Resource Access Control Lists (RACLs).
(a) Memory Object ACLs headed by MOCBs 1, 2 and 3.
(b) Protection Domain ACLs indicating which domains can enter GPD 1 and GPD 2.

Figure 4 depicts two sets of PALs contained in the PRAL of Figure 3. Figure 4a shows two PALs which contain the access permissions for GPD 1 to Memory Objects 1 and 2 and the access rights of CPD 1 to Memory Objects 1 and 3. Figure 4b shows the PALs which contain access permissions for GPD 1 to switch to GPD 2, GPD 2 to switch to GPD 1 and threads in CPD 1 to enter GPD 2. Note that these figures assume that a single linked list contains both the MOCBs and the GPDs accessible to a protection domain. In fact, for each protection domain, there are separate PALs for accessible memory objects and for accessible domains.

Figure 5 shows two sets of RACLs contained in the PRAL of Figure 3. Figure 5a shows three RACLs which contain the ACL entries for Memory Objects 1, 2 and 3. Figure 5b shows the RACLs which contain switch entries allowing entry into GPD 1 from GPD 2 and entry into GPD 2 from GPD 1 and CPD 1. The latter would typically be for the return to GPD 2 of a "traveling thread" in CPD 1 from a called domain if the caller, in GPD 2, trusted the called one (whose domain is not shown.)

Figure 6 is a blow-up of a Sombrero Access Descriptor showing only the fields relevant to the PRAL in Figure 3. ADs are also used to store access redefinitions (access "overrides").

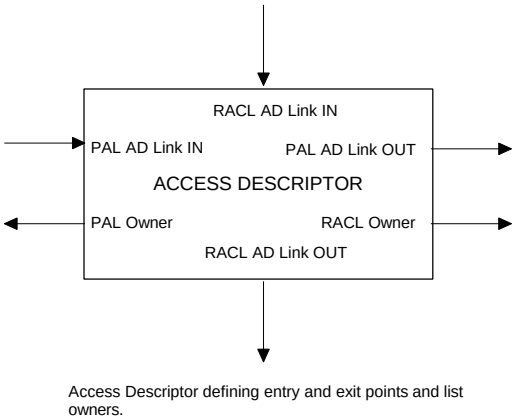


Figure 6: Access Descriptor.

Region Control Blocks (RCBs) can be associated with MOCBs and ADs, to describe protection regions. These are discussed in section 2.3.

The PRAL is accessed via Sombrero System Interface (SSI) [Car96] calls from higher, user-level programs. It is also accessed by the lower-level Sombrero kernel as a result of protection misses which occur in the CPU-resident protection hardware (the RPLB) assumed in the Sombrero design.

To summarize, the PRAL must be designed in such a way that it ([Sk*96b]):

- Implements the translation from the standard access matrix information to the RPLB matrix contents
- Supports higher-level protection policy, to conduct both rights granting and revocation
- Provides hardware-based protection constraints for higher-level object-oriented design strategies
- Facilitates rapid searching by the Sombrero kernel on an RPLB protection miss
- Upon SSI requests, allows easy modification for addition and deletion of protection domains, memory objects, and access privileges
- Is kernel-resident

2.3 Range Mask

The unit of protected resource access in Sombrero is a memory region. A region is a possibly non-contiguous collection of bytes with fixed access privileges from the set {rwx}. The virtual address range or region to be protected can be efficiently represented using a "match mask" and a "don't-care mask." A Range Function, applied to these masks, produces the "range mask." Figure 7 ([Sk*96b]) exemplifies. The range mask is a set of tri-state bits.

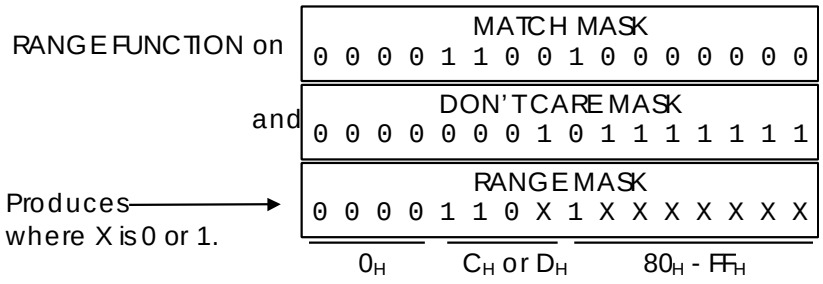
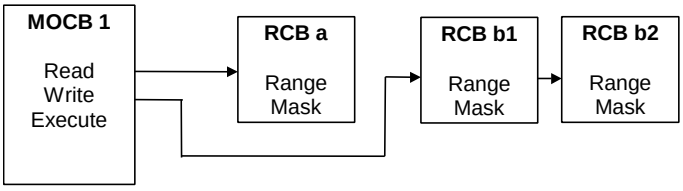


Figure 7: Range Function example.

The Range Function is that all bit positions in the range mask are enabled and have the value of the corresponding bits in the match mask except when a bit position is disabled by a corresponding 1 value in the don't-care mask.

In the example above, the match mask (0C80h) and don't-care mask (017Fh) given define two regions: range 0C80h to 0CFFh and 0D80h to 0DFFh.

Protection regions can be stored in the PRAL using Region Control Blocks (RCBs). Figure 8 shows an example of protection regions that could be associated with the MOCB 1 from Figure 3. Match masks and don't-care masks are stored in the RCBs, but, for simplicity, range masks are shown in the RCB blocks.



MOCB 1 is a data memory object with read and write regions. There is no execute region. The read region is described by one range mask in RCB a. The write region is described by two Range Masks in RCB b1 and RCB b2.

Figure 8: Detailed view of MOCB 1 showing the associated Region Control Blocks.

Protection regions, in the form of match masks and don't-care masks, are cached in the Range/Region Protection Lookaside Buffer (RPLB). If there is no RCB defining a protection region for a memory object, the match mask and don't-care mask are generated from the memory object range.

A CPU-emitted virtual address is compared with a range mask (and some other data) from the RPLB for access validation.

Generally speaking, entries can be stored more efficiently in an RPLB than in a TLB because the latter stores page-sized entries. The former stores regions instead, which can be as large as the address space will allow and as small as a byte.

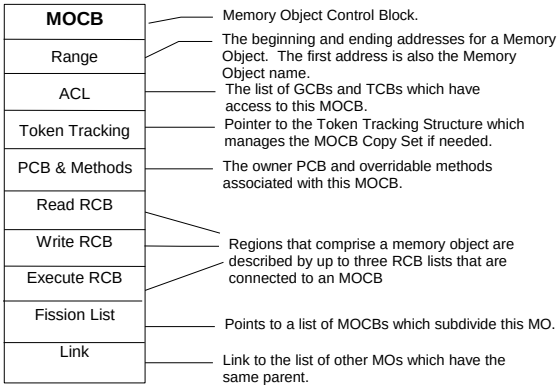
A match mask and don't-care mask can be generated using the RangeToRangeMask() function ([Sk094]), which takes three input parameters: the starting and ending address (that is, the range) of the

memory object, and a virtual address. It returns a match mask and a don't-care mask. Depending on the layout of the memory object (and the virtual address parameter), multiple masks can be generated. However, if the memory object is "aligned" (see Appendix D) for a discussion of what an aligned memory object is), only one range mask is generated, regardless of the virtual address parameter. Thence, aligned memory objects require less RPLB entries than non-aligned memory objects.

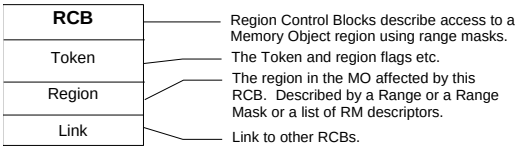
2.4 Software Protection Data Structures

Figure 9 contains Sombrero's principal software protection data structures and indicates the contents of the main fields of each. The Memory Object Control Block (MOCB), Region Control Block (RCB) and Access Descriptor (AD) provide protection-related information about the resources contained in memory objects. The General Protection Domain Control Block (GCB), Carrier Protection Domain Control Block or Thread Control Block (TCB), Access Descriptor (AD) and Principal Control Block (PCB) provide information on what can be accessed from a given protection domain and the user or principal the protection domain is associated with. These are discussed in detail in [Sk*96a]. The Root MOCB serves as a list head for the MOCBs, GCBs, TCBs and PCBs. The Token Tracking structure contains information about copy set locations and read and write access permissions relevant to distributed systems in order to manage the replication and consistency issues caused by network latencies. These are discussed in [Sk*96c].

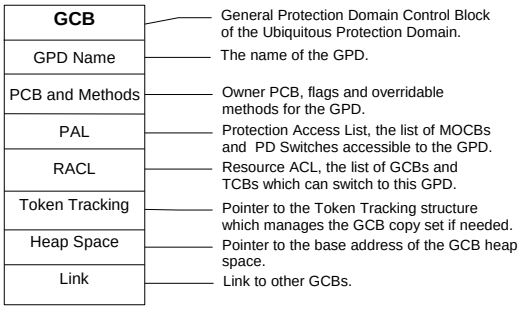
The figure below describes the following elements: (a) Memory Object Control Block; (b) Region Control Block; (c) General Protection Domain Control Block; (d) Carrier or Thread Protection Domain Control Block; (e) Principal Control Block; (f) Access Descriptor; (g) Root Memory Object Control Block; (h) Token Tracking Structure.


MOCB
Range
ACL
Token Tracking
PCB & Methods
Read RCB
Write RCB
Execute RCB
Fission List
Link
Memory Object Control Block.
The beginning and ending addresses for a Memory Object. The first address is also the Memory Object name.
The list of GCBs and TCBs which have access to this MOCB.
Pointer to the Token Tracking Structure which manages the MOCB Copy Set if needed.
The owner PCB and overridable methods associated with this MOCB.
Regions that comprise a memory object are described by up to three RCB lists that are connected to an MOCB
Points to a list of MOCBs which subdivide this MO.
Link to the list of other MOs which have the same parent.

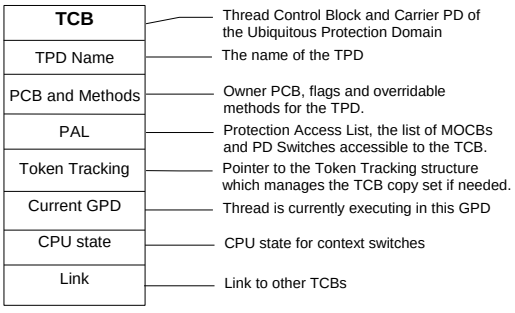
(a)


RCB
Token
Region
Link
Region Control Blocks describe access to a Memory Object region using range masks.
The Token and region flags etc.
The region in the MO affected by this RCB. Described by a Range or a Range Mask or a list of RM descriptors.
Link to other RCBs.

(b)


GCB
GPD Name
PCB and Methods
PAL
RACL
Token Tracking
Heap Space
Link
General Protection Domain Control Block of the Ubiquitous Protection Domain.
The name of the GPD.
Owner PCB, flags and overridable methods for the GPD.
Protection Access List, the list of MOCBs and PD Switches accessible to the GPD.
Resource ACL, the list of GCBs and TCBs which can switch to this GPD.
Pointer to the Token Tracking structure which manages the GCB copy set if needed.
Pointer to the base address of the GCB heap space.
Link to other GCBs.

(c)


TCB
TPD Name
PCB and Methods
PAL
Token Tracking
Current GPD
CPU state
Link
Thread Control Block and Carrier PD of the Ubiquitous Protection Domain
The name of the TPD
Owner PCB, flags and overridable methods for the TPD.
Protection Access List, the list of MOCBs and PD Switches accessible to the TCB.
Pointer to the Token Tracking structure which manages the TCB copy set if needed.
Thread is currently executing in this GPD
CPU state for context switches
Link to other TCBs

(d)

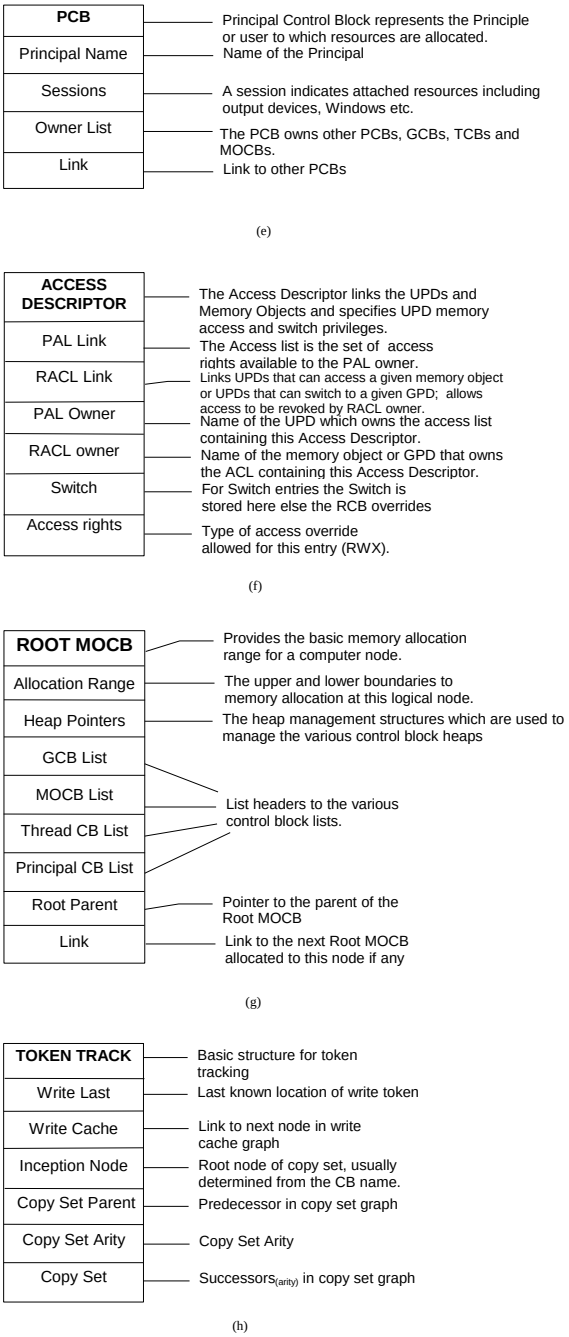


Figure 9: Sombrero Software Protection Data Structures.

2.5 Range Protection Lookaside Buffer (RPLB)

The RPLB is a CPU-resident hardware cache that stores protection information. Actually, it consists of a Instruction RPLB (IRPLB), a Data RPLB (DRPLB) and a Switch RPLB (SRPLB). The former two are used for "intra-domain" operations and the latter is used for "inter-domain" operations.

Sombrero associates a Thread/Carrier Protection Domain (TPD) with the thread being executed and a General Protection Domain (GPD) the thread is executing in. The current values of these, actually the virtual addresses of the control blocks that store data about them, are stored in two high speed base registers, TBR and GPDBR as shown at the bottom of
Figure 10.

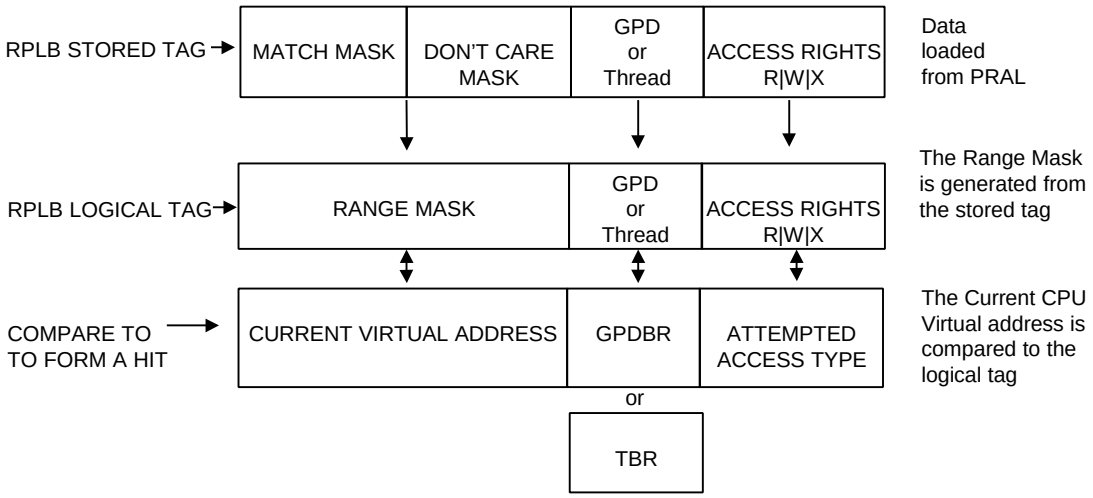


Figure 10: The Operation of the DRPLB and IRPLB.

An RPLB entry for part or all of a region is shown at the top of Figure 10. The top set of fields is loaded into the RPLB on a miss. Combinational logic generates a Range Mask that defines the set of valid addresses stored in the RPLB. CPU emitted data is compared against the RPLB logical information on every memory access to determine whether the access is allowed.

Figure 11 gives an overview of SRPLB. The SRPLB stores switch (virtual) addresses, GPDs or TPDs, new GPDs, and access rights. If the current virtual address matches the switch address, the access

mode conforms to the access rights, and the current protection domain matches the GPD or TPD stored in the SRPLB, then the current GPD can be "switched" to the new GPD stored in the SRPLB.

The top set of fields is loaded into the SRPLB on a miss. The CPU emitted data is compared against the RPLB stored information to determine whether the domain crossing is allowed. The RPLB associatively compares each entry against the CPU emitted data. For allowed accesses the new GPD name is stored in the GPDBR and this is followed by an intra-domain access attempt. More information on RPLB operation can be found in [Sko94], [Sk*96b] and [Sk*97a].

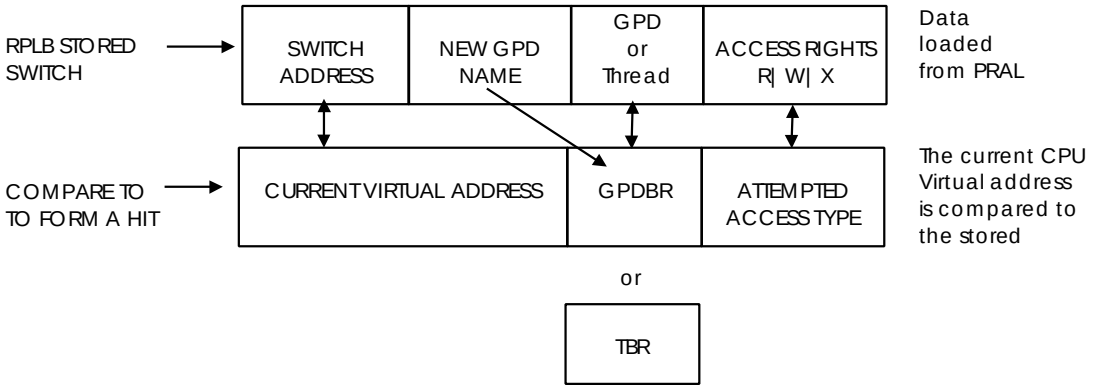


Figure 11: The Operation of the SRPLB.

3. Methodology

A prototype of the Sombrero software protection structures (the PRAL) and algorithms is implemented, based on the Sombrero protection model. Also, the implementation, and, more broadly, the Sombrero protection model is evaluated. Obviously the performance of, for instance, the PRAL search upon an RPLB miss significantly impacts the performance of the Sombrero protection model in particular and the Sombrero OS in general.

As for evaluating the Sombrero protection model, it is a daunting task, at least at this time. To mention a few difficulties, there is no Sombrero OS, there is no RPLB, and there are no real-world SASOS applications. The best that can be done is to simulate these, so the results obtained are approximate at best. However, the simulation allows one to make rough comparisons and observe and characterize behavior differences as various parameters are changed.

The "C" programming language was used for most of the development, but eventually some code (e.g., the RPLB miss handler) will need to be converted to Alpha "PALcode" (Privileged Architecture Library subroutines) for the sake of efficiency.

3.1 Evaluation of the Sombrero Protection Model

The Sombrero protection model is evaluated by:

1. Simulating the Sombrero Instruction RPLB (IRPLB), Data RPLB (DRPLB) and Switch RPLB (SRPLB).
2. Simulating the Instruction TLB (ITLB), Data TLB (DTLB) and page tables of a typical MASOS (the Alpha-NT was chosen).
3. Simulating memory-access validation using the RPLB and TLB. Cache-access traces were obtained from the Internet and used to simulate memory accesses (CPU-emitted pairs of virtual addresses and access modes [read, write or execute]). These pairs are validated via the protection structures implemented. (More information about the cache-access traces used is provided in section 4.4.)

4. And, finally, the Sombrero protection model is evaluated by recording performance statistics for the two simulated environments (Sombrero and Alpha-NT).

When all is said and done, the performance of the (simulated) Sombrero RPLB/PRAL mechanism is compared against the (simulated) Alpha-NT TLB/page tables mechanism. At first glance, it appears that apples and oranges are being compared, but the approach taken is justified as follows.

Evaluating the Sombrero protection model per se is too difficult at this time. But the Sombrero protection model will affect the average cost of memory access (and, ultimately, the average cost of instruction execution in Sombrero). So it was decided to compare the average cost of memory access in Sombrero versus the average cost in an Alpha-NT, focusing on the protection aspects as much as possible.

To make a first approximation, factoring out costs that are going to be common (and probably of the same order of magnitude) between the Sombrero and Alpha-NT sides, e.g., the buffer hit-related costs, it was assumed the average memory-access cost on the Alpha-NT side is:

```
(Alpha-NT TLB Miss Rate) multiplied by (Alpha-NT TLB Miss Penalty)
```

The average memory access cost in Sombrero will depend on *both* the RPLB and the TLB. Gloom! However, in a Sombrero environment, the TLB can be removed from the CPU and moved to the memory bus. Furthermore, because Sombrero is a single address space system and the RPLB contains all protection information, it is possible to have an inverted page table (with as many entries as the number of page frames in the RAM). And, because of its small size, this inverted page table can be stored in a hardware content-addressable memory (CAM) buffer. In fact, in may be possible to do away with a TLB altogether in Sombrero if the said CAM with the inverted page table can be used to provide its functionality. For simplicity, it was assumed that the cost to search this inverted page table in Sombrero was negligible.

Thus, as a first approximation, the average cost of memory access in Sombrero is compared to the the average cost in a traditional system by comparing, under different scenarios:

(Sombrero RPLB Miss Rate) multiplied by (Sombrero RPLB Miss Penalty)

versus

(Alpha-NT TLB Miss Rate) multiplied by (Alpha-NT TLB Miss Penalty)

By necessity, the results obtained will be approximate, especially because real cache misses, real TLB misses, real page faults, native NT operating system tasks, and more could affect the results of the test runs.

While this simulation is not rigorous, there are some weaknesses in a traditional TLB-based MASOS system also which are ignored in this simulation too:

- Unnecessary TLB and cache misses and/or redundant entries of data shared amongst different processes, caused by, for example, multiple address spaces and address space switches
- Increased overhead resulting from the absence of virtually indexed caches

3.2 Assumptions

While Sombrero is meant to be a distributed operating system, the following assumptions were made to limit the scope of this research:

- Single-CPU workstation
- Single node

4. Implementation

The Microsoft Development Studio and the Microsoft Visual C++ V5.0 compiler were used to write most of the code. Some Windows functions were used, but C++ classes were not used. An effort was made to make the 32-bit implementation easily portable to a 64-bit environment, as shown in the source code "header" (definition) files in the Appendix B.

4.1 PRAL Prototype

The PRAL should facilitate efficient access validation (on an RPLB miss) and efficient access revocation.

The following algorithm is used to search and retrieve an entry for the RPLB from the PRAL:

1. Check MOCBs attached to current GCB
2. If not found, check MOCBs attached to current TCB
3. If not found, check switches from the current GCB
4. If not found, check switches from the current TCB
5. If not found, handle PRAL fault

When a switch entry is found, it is loaded in the SRPLB and the current GCB is "switched" to the one specified in the switch entry. Then the whole access-validation process is repeated, starting with the RPLB checks.

The following algorithm is used to implement items 1 and 2 above, i.e., to find an entry from the memory object access list attached to a domain:

i. Look for the Access Descriptor (AD) and MOCB containing the address accessed by the CPU. Check for access redefinition ("access override" in the AD structure). Make sure that the access allowed (read, write and/or execute) matches the access desired. Next, check for "region redefinition."

ii. If there was no access redefinition, check for access permissions in the MOCB itself. Next, check for "regions."

iii. If a region is found, return the "match mask" and the "don't-care mask" stored in the Region Control Block (RCB).

iv. Otherwise, generate the match mask and the don't-care mask using the memory object range. Return the match mask and the don't-care mask. Return the access permissions specified in the AD, if any; otherwise, if there are no regions, return the access permissions specified in the MOCB.

The following algorithm is used to implement items 3 and 4 above, i.e., to find an entry from the switch access list attached to a domain:

i. Look for the AD matching the address accessed by the CPU.

ii. Check for access permissions (in the AD structure). Make sure that the access allowed (read, write and/or execute) matches the access desired.

iii. Return the new GCB and the access permissions specified in the AD.

4.1.1 Implementation Details

A separate heap is used to allocate each type of Sombrero structure. That is, there is a "AD heap" from which AD structures are allocated. Also, there are heaps for MOCBs, GCBs, TCBs and RCBs. These are linked together using pointers. New memory objects and switches are added to the front of the PRAL, which can affect the performance of the PRAL search.

Upon a RPLB miss, one can search the PRAL directly, by traversing the linked lists sequentially.

A linear search of the PRAL is inefficient on two accounts. First, it takes too long if the access lists have a significant number of elements. Second, the various objects referred to could be spread amongst different pages of memory, which could result in extra page faults.

To speed-up PRAL searches, the ranges of (physically allocated) memory are stored in the AD structures themselves (where applicable). However, this means that, whenever a memory object is changed, it may be necessary to update all the appropriate ADs. Wherever there was a tradeoff between PRAL updates and PRAL searches, faster PRAL searches were favored.

It is estimated that, normally, a typical domain will have less than twenty memory objects, which makes the impact of optimizing the domain-access list search algorithm less significant. Nonetheless, to speed up PRAL searches, two alternative mechanisms were explored, as explained below. Both of these PRAL-search aids apply to lists of memory objects, not switches.

The RPLB miss handling code could have been improved by reducing the number of subroutine calls. However, the way it is coded, it should be easier to port it to PALcode.

4.1.2 Binary Search of the PRAL

For each domain-access list, an array consisting of starting addresses of memory object ranges and pointers to the corresponding AD structures is allocated. This array is allocated in the default heap and is updated whenever there is a pertinent change to the domain.

Additions to this array are followed by a call to the Windows QuickSort() function. In other words, the array is always kept sorted.

The ending addresses of memory object ranges are not stored in the array to minimize its size. The more compact the array, the less likely the potential for page faults.

Given a virtual address, a binary search of the sorted array is performed by comparing it to the starting addresses. If the virtual address falls between two starting addresses (or at the end of the array), it is compared against the ending address of the range in the corresponding AD to determine whether there is a match.

4.1.3 Hash Search of the PRAL

Because the domain-access lists point to variable-sized memory objects, it is problematic to take a virtual address and directly hash to the containing memory object. For instance, if an access list had two memory objects, the first ranging from 12380 to 12387 (hexadecimal) and the second ranging from ACFF through AD00, we'd need each CPU-emitted virtual address in the first range to hash to the first object, and

each virtual address in the second range to hash to the second one. The problem is that there is no way to do that just by looking at a virtual address, without additional information.

Fortunately, [Sko97] proposed a separate hash structure for each domain access-list and a "hint" in the hash structure header. Hashing is done after removing the appropriate number of trailing bits, according to the hint.

In the example above, while the largest memory object is 8 bytes long, it is necessary to remove the trailing *9* bits to insure that all virtual addresses contained in the same memory object become exactly the same. Once they are "normalized," these virtual addresses will all hash to the same memory object (bucket). In the example above, after 9 bits are right-shifted, all the virtual addresses in the first range would be turned into 91 (hexadecimal), and all the addresses in the second range would become 56 (hexadecimal).

The division (modulo) hashing function is used, so, in the case above, if the size of the hash table were 3, the first set of virtual addresses would hash to bucket 1, whereas the second set of addresses would hash to bucket 2.

Of course, hashing conflicts may result, but if some buckets get too full due to too many conflicts, the "chains" (lists of ranges in the buckets) themselves are hashed, which results in a recursive structure.

Unfortunately, sometimes one simply can't convert a long list to a hash table. Consider the following list of memory object ranges (in hexadecimal notation):

```
AB00 -> AB88
ABC0 -> ABCD
ABD0 -> ABD0
ABD1 -> ABDF
ABE0 -> ABF0
```

Since it is desired that all virtual addresses in a range fall into the same bucket (to resolve to the same memory object), it becomes necessary to throw away the trailing 8 bits. Unfortunately, because of the way these particular memory objects are laid out, there is nothing distinctive left after the trailing 8 bits are ignored. So there is nothing more to hash on. This leads to long chains in the hash structure. Fortunately, the chains themselves are binary search arrays.

In fact, the input parameter to the hash structure-building routine is the sorted array (with starting addresses and pointers to AD structures) generated for the binary search of the PRAL (as described in the previous section). The algorithm implemented to build the hash structure in this research is documented in the Appendix E.

4.2 RPLB Simulation

The hardware Data RPLB (DRPLB), Instruction RPLB (IRPLB) and Switch RPLB (SRPLB) were simulated using software arrays. Their sizes can be specified in an input file. Their maximum size is 1024.

These buffers are assumed to be fully-associative caches. Their entries have a "valid" bit that is cleared when the corresponding entry is invalidated from the PRAL. The algorithm to be used to evict old entries to make room for new ones (Not Last Used or the Least Recently Used) can be specified in an input file too.

To calculate the miss handling time, the Windows function QueryPerformanceCounter() is called before and after the RPLB miss handler is invoked. This function returns the number of clock ticks that have occurred since the machine was started. The Windows function QueryPerformanceFrequency(), which returns the number of clock ticks per second, is also used for printing the statistics in terms of microseconds or nanoseconds.

The RPLB miss handling time consists of the time it takes to search for and retrieve an entry from the PRAL.

4.3 TLB Simulation

So comparisons can be made with the Sombrero side, TLB accesses, TLB hits, TLB misses, and TLB miss handling (in particular, page table searching) are simulated.

In the simulation, a TLB miss penalty consists solely of searching a page table. For simplicity, page faults, which will occur in Sombrero too, are ignored. In a traditional OS, each process has its own page table. The construction of a page table was implemented, but, for simplicity and to conserve resources, all "processes" use the same page table, which ranges from 0 to 7FFFFFFF (hexadecimal). This is justified by the desire to measure only the time it takes to search a typical page table for and retrieve an entry from it.

The Alpha-NT page tables can be viewed as both (i) a single-level virtually contiguous table and (ii) a two-level physical table, as documented in the "Windows NT AXP Software" section of the second edition of the *Alpha AXP Architecture Reference Manual* [Si*95].

Two algorithms can be used to traverse the page tables: Either single-level virtual traversal or two-level physical traversal.

In the virtual traversal, the TLB itself is used to obtain the page table entry (PTE) in question. That is, the page table entry desired is loaded virtually (i.e., using its virtual address), which means that the TLB comes into play again. If there is another TLB miss, it becomes necessary to use "physical traversal" to load the appropriate "page directory page entry" into the TLB. As stated in [Si*95], "The two-level physical method can be used to find the corresponding PTE for a virtual address when the virtual access method cannot be used (for example, if the PTE address is not valid). The key to physically traversing the page tables is the PDR internal processor register. The PDR is maintained on a per-process basis whenever process context is swapped. The PDR is the physical address of the page directory page that forms the first level of the page tables. The first level of the page tables easily fits within a single page. Each entry in the page directory page is called a PDE (page directory entry). One PDE maps one page of PTEs."

While the virtual traversal of the page tables requires more information to be cached, e.g., there is additional demand for precious TLB slots (which can lead to more misses), Digital Equipment Corporation recommends that it be used anyway, for the sake of performance.

In the simulation of the 32-bit Alpha-NT implementation of the TLB, the virtual traversal could not be implemented efficiently as documented. Thence, a single array was defined to simulate all page

tables, with 256 K (page) entries. On a TLB miss, the "virtual page number" is obtained from the virtual address being accessed (by discarding the rightmost 13 bits corresponding to the page offset). The virtual page number is then used to index into the said page entry array. So, the virtual page table traversal implemented is more efficient than the actual Alpha-NT virtual traversal, as, in the simulation, the virtual traversal never fails. That is, a physical traversal is never required. Also, in the simulation, the overhead of building the page tables is not accounted for. The Alpha-NT virtual page table traversal also requires "page directory page entries" to be stored in the Data TLB, which are ignored here too.

Therefore, in reality, the TLB miss penalty would be higher (relative to RPLB miss penalty) than suggested by the simulation. And, the DTLB miss rate would be higher (relative to the DRPLB miss rate) than suggested by the simulation.

In the global definitions file (see Appendix B), the size of the DTLB is set to 64 and the size of the Instruction TLB (ITLB) is set to 48 entries. These are assumed to be fully-associative caches. The NLU algorithm is used to evict old entries to make room for new ones when the buffer is full. The page size is fixed at 8 KB. (This matches the memory management implementation in the Alpha-NT machines in the ASU OS Lab.)

As in the RPLB simulation, the Windows functions QueryPerformanceCounter() and QueryPerformanceFrequency() are used to measure TLB miss handling times.

4.4 Cache Access Traces

To generate RPLB and TLB performance statistics, cache access traces were obtained from the Internet. These accesses were used to simulate CPU-emitted pairs of virtual addresses and access modes (read, write or execute) and access the RPLB and TLB.

The trace files were copied from ftp://max.stanford.edu/pub/hennessy-patterson.software. There is a "README" file there with details about the software available there. This URL was reached via the Morgan Kaufmann Publishers home page (http://www.mkp.com), where there are additional resources for

the “Computer Architecture: A Quantitative Approach, Second Edition” book [He*96]. These traces are mentioned in this book.

As described in the README file above, there are “… cache traces of the three benchmarks (Spice,gcc,TeX)” and other “… cache traces taken on the VAX” there. The traces for the benchmark programs were downloaded. Unfortunately, it is not clear exactly what platform the benchmark traces were taken on.

The benchmark trace files, shown below, are installed in the SOMBRERO1 (Alpha) machine in the ASU OS Lab (in GWC-321).

```
TEX      RWX     8,795,094  09-14-97  4:14p
SPICE    RWX    10,434,484  03-02-97  4:42p
GCC      RWX    10,485,342  03-01-97  1:16p
         3 file(s)     29,714,920 bytes
```

These files have about 2,800,000 read, write and execute accesses, consisting of “0,” “1” or “2” (read, write or execute access mode, respectively) and a 32-bit virtual address. For instance, here are the first few lines of gcc.rwx:

```
2 408ed4
0 10019d94
2 408ed8
1 10019d88
2 408edc
0 10013220
```

After the above access traces were obtained, it was necessary to determine memory maps. That is, it was necessary to generate memory objects that would contain the virtual addresses above. The approach followed is described next.

First the hexadecimal virtual addresses are converted to decimal notation so that the UNIX sort utility can be used to order the cache accesses by virtual address. For instance:

```
4196816 2
4196816 2
4196816 2
4196824 2
4196824 2
4196824 2
4196824 2
```

```
4196824 2
4196828 2
4196828 2
```

Then two programs were written to identify memory objects using the sorted addresses, as described below.

The first program identifies unaligned memory objects. (See Appendix D for the difference between unaligned and aligned memory objects.)

Basically this program "creates" a memory object whenever it sees a group of virtual address that are "spatially" close together. In particular, this program takes the minimum "distance" or "separation" between memory objects as an input parameter. For example, in the case above, if the desired minimum number of bytes separating memory objects were 6, 2 memory objects would be created. The first one ranging from 4196816 through 4196816 and the second one from 4196824 through 4196828. (So the first and second

memory objects are separated by at least 6 bytes.)

The output of these programs is in a format that can be fed into the test environment (see section 4.6) directly. A "#echo_off" directive is added to the top of the memory object files generated to limit the amount of output during the simulation. Here's a sample output file, using memory objects at least 8 KB apart:

```
#echo_off

mo_add_specific:     4001e0     439104 0 X owner_gpd_for_gcc gcc_8192;1
mo_add_specific:     43c3a0     44164c 0 X owner_gpd_for_gcc gcc_8192;2
mo_add_specific:     444380     44d350 0 X owner_gpd_for_gcc gcc_8192;3
mo_add_specific:     44ff90     47b8b8 0 X owner_gpd_for_gcc gcc_8192;4
mo_add_specific:     47e100     48b690 0 X owner_gpd_for_gcc gcc_8192;5
mo_add_specific:   10000008   1001c334 0 RW owner_gpd_for_gcc gcc_8192;6
mo_add_specific:   1001f014   10029490 0 RW owner_gpd_for_gcc gcc_8192;7
mo_add_specific:   1002b658   1002fd16 0 RW owner_gpd_for_gcc gcc_8192;8
mo_add_specific:   10032008   10032213 0 W owner_gpd_for_gcc gcc_8192;9
mo_add_specific:   1003507c   10036804 0 RW owner_gpd_for_gcc gcc_8192;10
mo_add_specific:   1003c4dc   100458e6 0 RW owner_gpd_for_gcc gcc_8192;11
mo_add_specific:   7ffd830c   7ffda47c 0 RW owner_gpd_for_gcc gcc_8192;12

! Total memory objects: 12
! Average size: 62759
! Interval between objects: 8192
! Total memory accesses: 1000002
```

The second program written to identify memory objects takes the desired (constant) size of memory objects as an input parameter. The first program produces memory objects that are likely to be of varying sizes. Unfortunately, it also produces objects that are likely to be unaligned. Sombrero works best with aligned memory objects, hence this program. Here's the output of this program, using 8 KB as the memory object size:

```
#echo_off

mo_add_specific:     400000     401fff 0 X owner_gpd_for_gcc gcc_2e13.1
mo_add_specific:     402000     403fff 0 X owner_gpd_for_gcc gcc_2e13.2
mo_add_specific:     404000     405fff 0 X owner_gpd_for_gcc gcc_2e13.3

[...]

mo_add_specific:     486000     487fff 0 X owner_gpd_for_gcc gcc_2e13.65
mo_add_specific:     488000     489fff 0 X owner_gpd_for_gcc gcc_2e13.66
mo_add_specific:     48a000     48bfff 0 X owner_gpd_for_gcc gcc_2e13.67
mo_add_specific:   10000000   10001fff 0 R owner_gpd_for_gcc gcc_2e13.68
mo_add_specific:   10002000   10003fff 0 R owner_gpd_for_gcc gcc_2e13.69
mo_add_specific:   10004000   10005fff 0 R owner_gpd_for_gcc gcc_2e13.70

[...]

mo_add_specific:   10040000   10041fff 0 RW owner_gpd_for_gcc gcc_2e13.97
mo_add_specific:   10042000   10043fff 0 RW owner_gpd_for_gcc gcc_2e13.98
mo_add_specific:   10044000   10045fff 0 RW owner_gpd_for_gcc gcc_2e13.99
mo_add_specific:   7ffd8000   7ffd9fff 0 RW owner_gpd_for_gcc gcc_2e13.100
mo_add_specific:   7ffda000   7ffdbfff 0 RW owner_gpd_for_gcc gcc_2e13.101

! Total memory objects: 101
! Memory object size (2 to the power of): 13
! Total memory accesses: 1000002
```

The algorithm being used is this: Given a virtual address, the bits corresponding to the size of the memory object are "zeroed." For instance, for an 8 KB object, the rightmost 13 bits would be cleared. Then the result (of zeroing the current virtual address) is compared with the previous result (of zeroing the previous virtual address). If it is greater, then it can be concluded that this (the current) virtual address belongs to a new memory object. So, a memory object is created by setting the starting address to the zeroed virtual address and the ending address to the "oned" virtual address (that is, the same virtual address, but with the rightmost bits corresponding to the size of the memory object [e.g., 13] set to 1). As an example, consider this set of virtual addresses, in hexadecimal notation:

```
96816 1
96824 1
96828 1
97901 1
979AA 1
```

If the desired size memory object size were 256, the 8 trailing bits would be zeroed to generate the starting address (96800) of the first object and they would be "oned" to generate its ending address (968FF). The next object would start at 97900 and end at 979FF.

4.5 Registry Simulation

A rudimentary "name server" was implemented to allow the user to refer to objects in command parameters (see chapter A for the list of valid commands). For instance, the user can issue the command "gpd_add: 3 Floppy*" to create 3 general protection domains and assign the first the name Floppy;1, the second Floppy;2, and the third Floppy;3. Then the user can, say, issue the command "gpd_print: Floppy*" to print information about each of the general protection domains created earlier with names starting with the string "Floppy."

No duplicate names are allowed in the registry. And, of course, not all objects need to be named, so not all objects are in the registry.

4.6 Testing Environment

A test/simulation facility was developed to functionally evaluate and test the code written. Furthermore, it was developed it to run experiments and conduct performance evaluation of the various components developed, using exogenous inputs. For instance, it can be used to generate performance statistics for the RPLB/PRAL and the Alpha-NT TLB/page tables simulations, under different scenarios. And, it can be used to compare the various RPLB/PRAL configurations possible.

A command language and an input parser for it were developed. Both can be easily augmented. The command language allows one to name objects so that one can reference them. And, it allows one to refer to multiple objects by using wildcards. There are, for example, commands to (i) add, delete and link

objects, (ii) print various data and format output, (iii) specify RPLB- and PRAL-related parameters, and (iv) perform RPLB and TLB accesses. See Appendix A for details.

4.7 Switches

The Sombrero protection model also includes transparent switching from one general protection domain to another. It is expected that the SRPLB cache of allowed domain crossings will provide considerable performance improvement over the hardware and software operations required in process-oriented operating systems and SASOS without CPU protection hardware ([Sk*97a], [Sk*97b] and [Sk*97c]).

So no experiments were run related to domain switches. However, my test environment does allow one to set and delete switches.

4.8 Multi-Threading Concerns

The Sombrero design calls for a hardware RPLB and two additional registers to store the current TPD and GPD, i.e., the current thread and the current general protection domain it is executing in. In the simulation, the Windows "thread local storage" functions TlsSetValue() and TlsGetValue() are used to set and retrieve the current TPD and GPD.

The Windows EnterCriticalSection() and LeaveCriticalSection() functions are used too. These do not work across processes, but the initial prototype of Sombrero will run in a single NT process.

While the testing/simulation environment allows the user to create true Windows threads, the effects of multi-threading were not investigated. To limit the scope of this research, a single-processor workstation was assumed all along. In such an environment, it can be assumed that a thread in the Sombrero kernel will not be preempted by another thread. In other words, since it is the kernel that updates

the PRAL, it will finish updating it before anyone else can use or update it. Thence, the PRAL won't be left in an inconsistent state.

5. Results

All tests were performed on the SOMBRERO1 and SOMBRERO2 workstations in the ASU OS Lab (in GWC-321), over Windows NT V4.0. Both of these machines have 500 MHz DEC 21164 Alpha processors and 128 MB RAM; except for disk space, they have identical hardware configurations.

To reduce the effects of any real cache misses, real TLB misses, real page table faults, and other operating system tasks, e.g., interrupts, each RPLB and TLB miss penalty is measured 3 times and the minimum value is used.

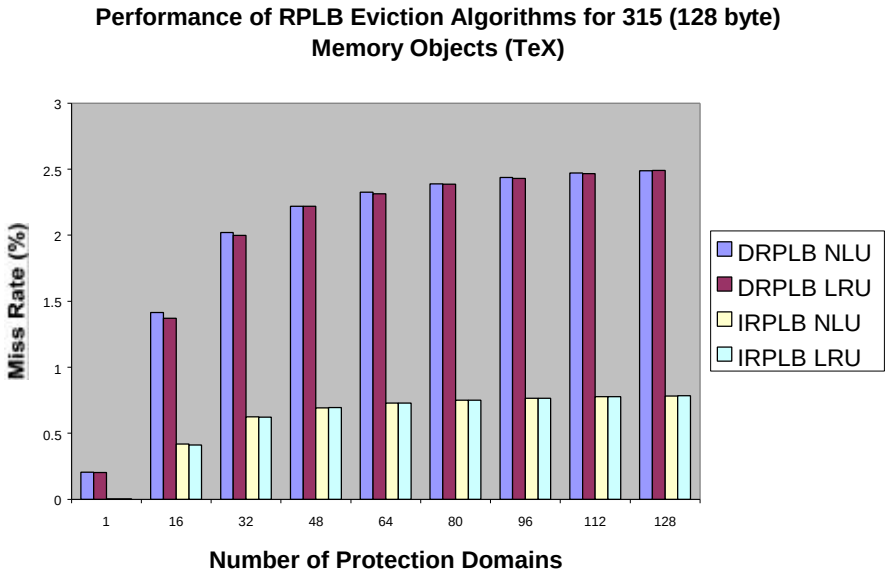
5.1 Performance of the RPLB Eviction Algorithms

The number of entries in a feasible RPLB is limited. Once it is full, it is necessary to remove and entry to make room for a new one. Two RPLB eviction algorithms were tested: Not Last Used (NLU) and Least Recently Used (LRU).

Figure 12 compares the Data RPLB (DRPLB) and Instruction RPLB (IRPLB) miss rates using NLU versus LRU. The number of (simulated) threads performing "simultaneous" cache accesses was varied from 1 to 128, in increments of 16. The threads ran in separate protection domains, so, in the graphs, the "category" axis is labeled "Number of Protection Domains," rather than "Number of Threads." Each thread was given a quantum of 1,000 cache accesses, and threads were scheduled randomly.

Cache access traces from the TeX and Spice benchmark programs were used. All threads read from the same input file. For instance, when performing tests using the traces from the TeX benchmark program, all threads read from the tex.rwx input file.

The DRPLB and IRPLB sizes used were 64 and 48, respectively.



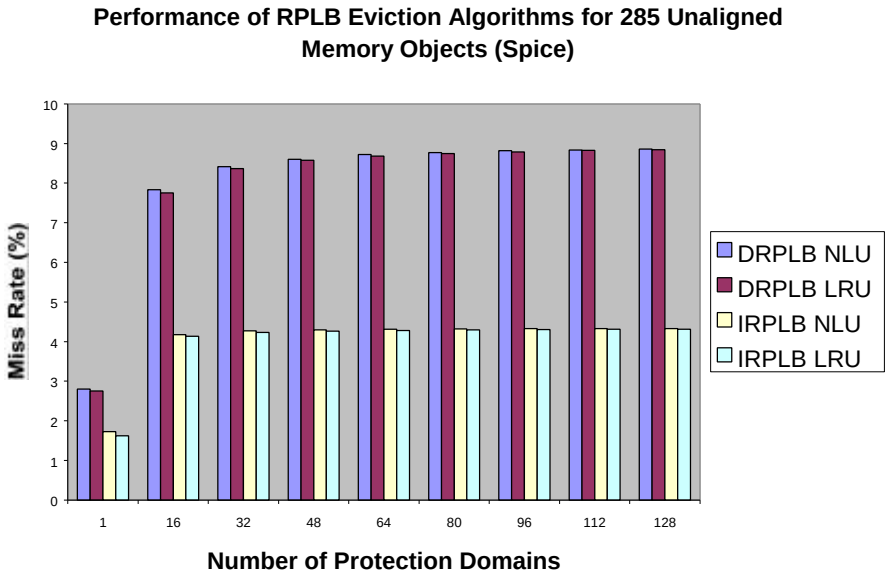


Figure 12: RPLB eviction algorithms: NLU vs. LRU.

The initial observations that can be made from these tests are:

1. Varying the number of threads past a certain point has insignificant effect.
2. NLU performs almost as well as LRU, and has the advantage of requiring less overhead than LRU. NLU is the TLB eviction algorithm used on the Alpha-NT.

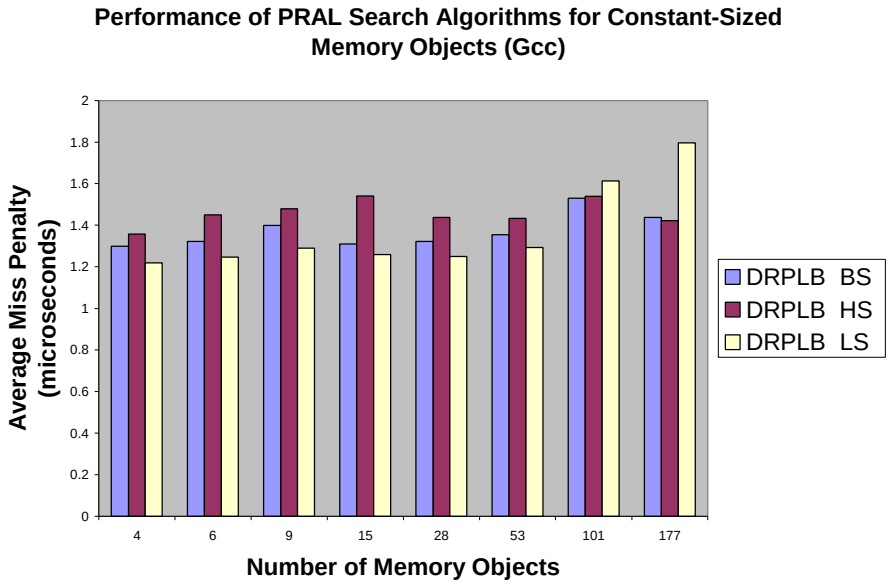
5.2 Performance of PRAL Search Algorithms

Upon a RPLB miss, it is necessary to search the PRAL to validate the attempted access. The process of searching the PRAL and retrieving an entry from it for the RPLB must be optimized, to optimize RPLB miss handling. This is similar to the process of searching the page tables and retrieving and entry from them for the TLB, which must be optimized too.

The PRAL search mechanisms tested were:

i. Binary search of sorted array (BS)
ii. "Hash structure" search (combination of hash tables and sorted arrays) search (HS)
iii. Linear search of linked lists (LS)

Figure 13 compares the RPLB miss penalties using the 3 search algorithms. Cache access traces from the Gcc benchmark program were used. The DRPLB and IRPLB sizes were set to 64 and 48, respectively.



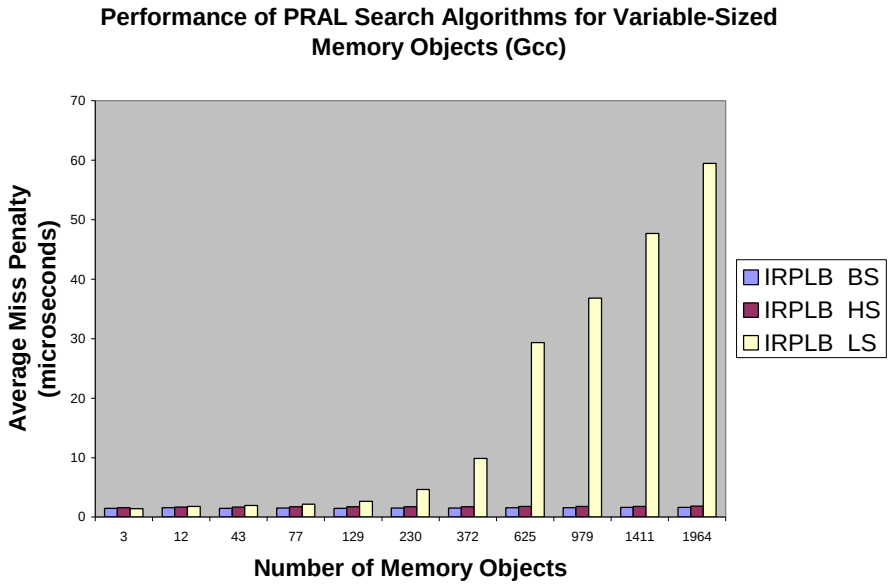


Figure 13: PRAL search algorithms: Binary vs. hash structure vs. linear.

The initial observations can be made from these tests are:

1. For 50 or so memory objects, linear search performs comparably well, but for large numbers, it fares poorly.
2. Binary search works better than hash structure search, regardless of the composition of the memory objects. That indicates that there is room for improvement of the hash structure.
3. It is expected that a typical domain will have less than 20 memory objects, which will make PRAL search optimizations less significant.

5.3 Performance of RPLB/PRAL versus TLB/Page Tables

The next few figures compare the miss rates and "memory access costs" for Sombrero and Alpha-NT. The number of (simulated) threads performing "simultaneous" cache accesses was varied from 1 to 80, in increments of 16. The threads ran in separate protection domains, so, in the graphs, the "category" axis is labeled "Number of Protection Domains," rather than "Number of Threads." The various pairs of charts differ in the number and size of memory objects.

All threads read from the same input file, containing traces from the Gcc benchmark program (gcc.rwx).

Parameters used:

- DRPLB and DTLB sizes: 64
- IRPLB and ITLB sizes: 48
- Page size: 8 KB
- Buffer entry eviction algorithm: NLU
- PRAL search algorithm: binary search
- Quantum per thread: 10,000 cache accesses (threads were "scheduled" randomly)

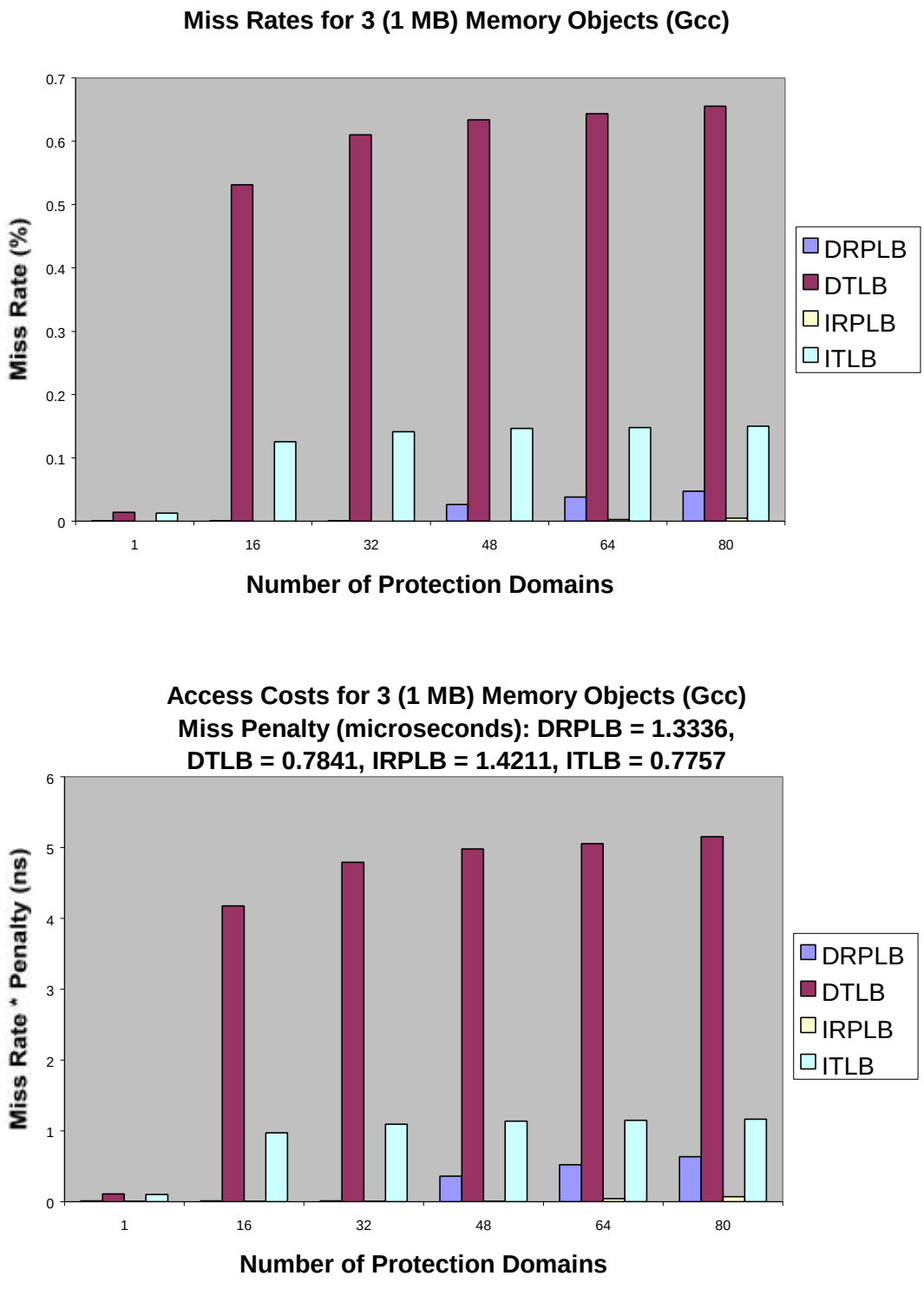


Figure 14: RPLB/PRAL vs. TLB/page tables, using 3 (1 MB) aligned memory objects.

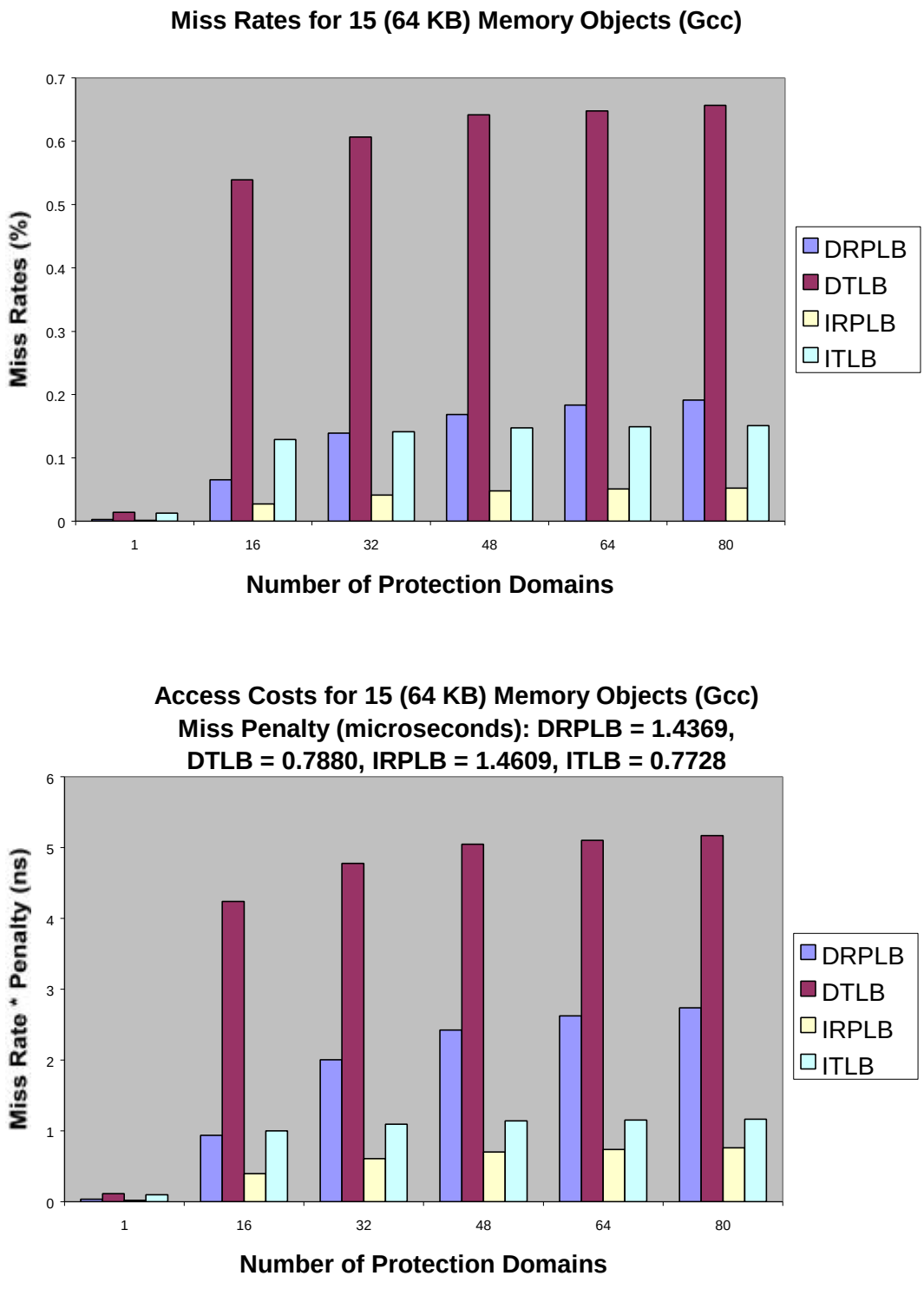


Figure 15: RPLB/PRAL vs. TLB/page tables, using 15 (64 KB) aligned memory objects.

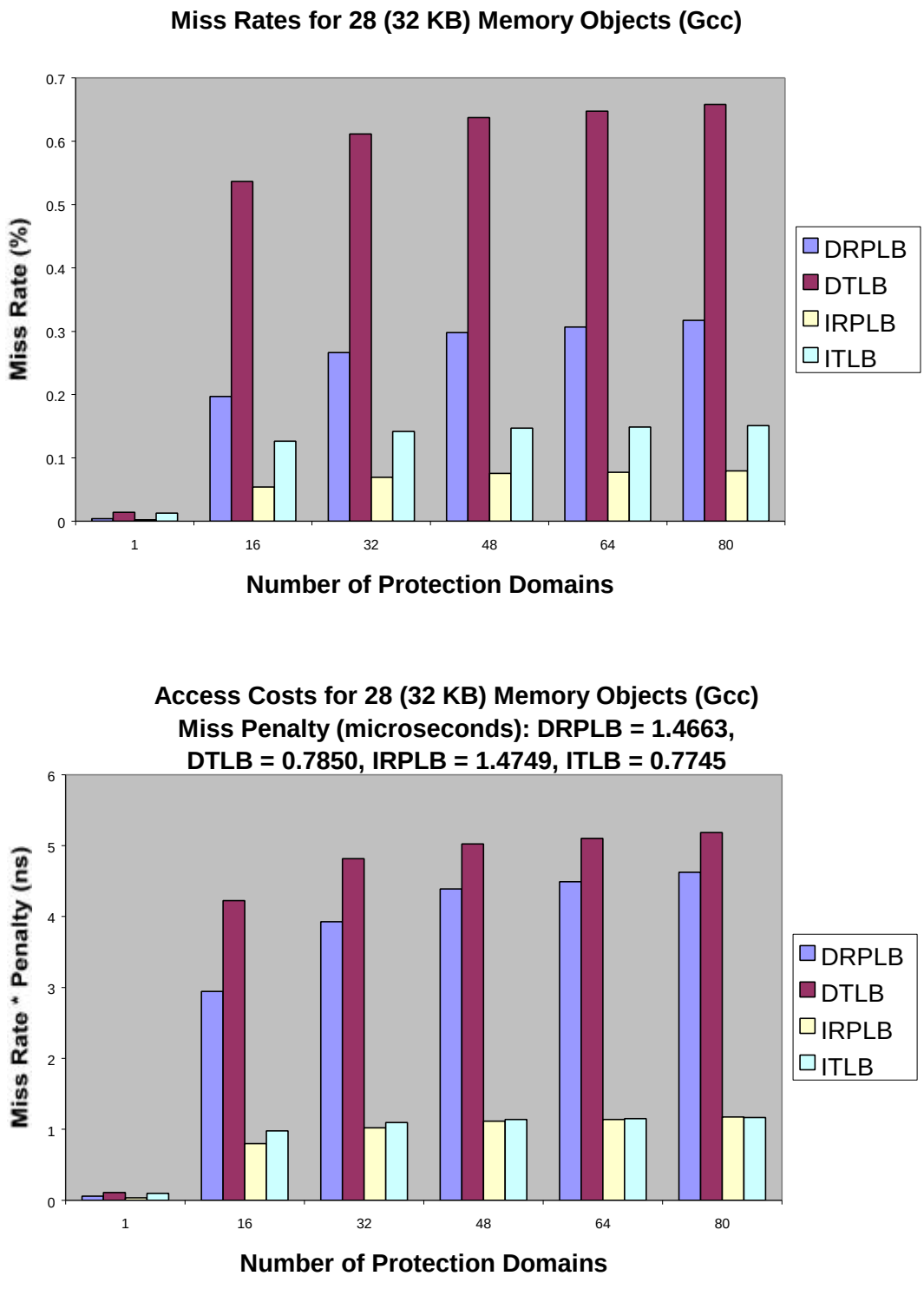


Figure 16: RPLB/PRAL vs. TLB/page tables, using 28 (32 KB) aligned memory objects.

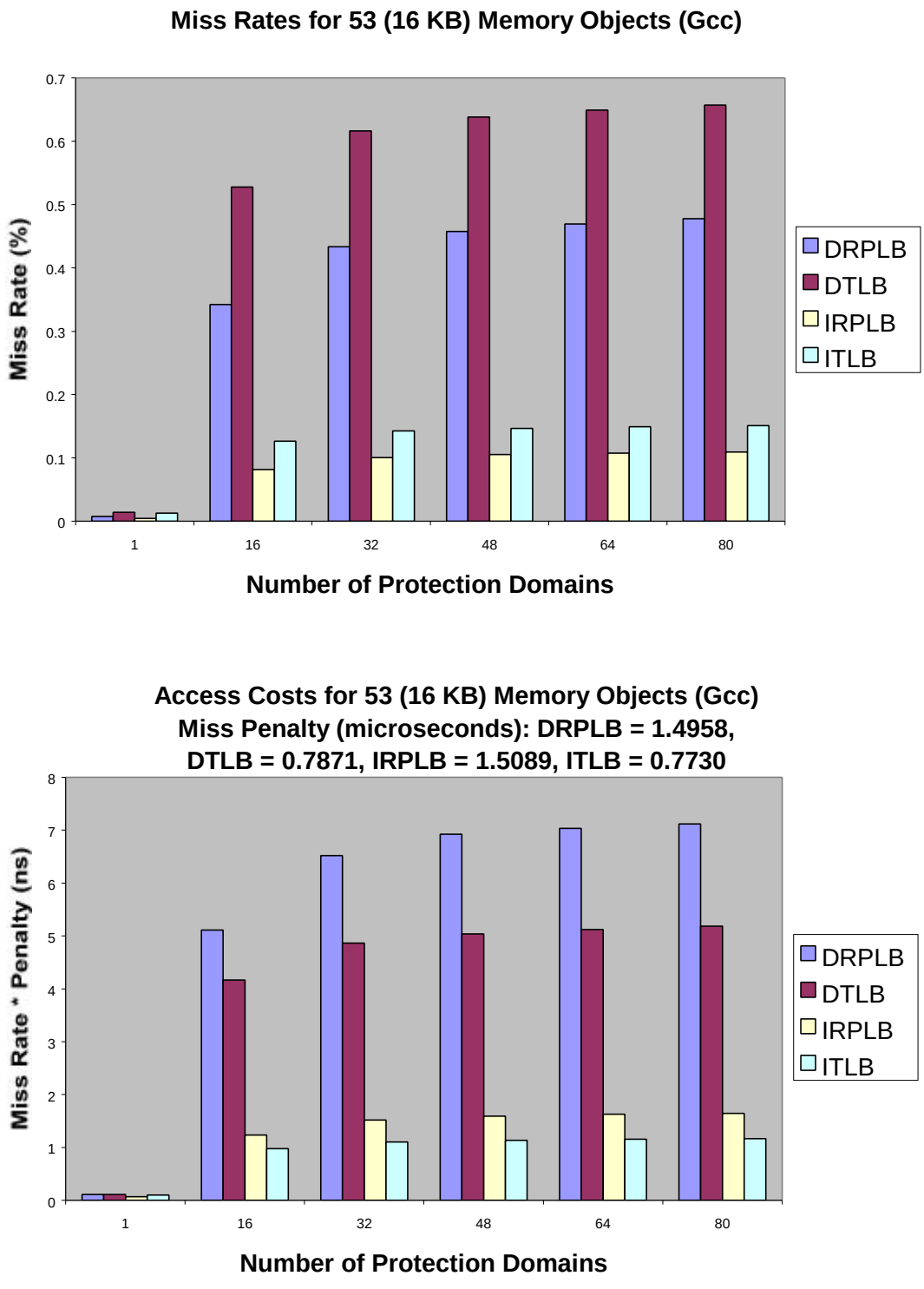


Figure 17: RPLB/PRAL vs. TLB/page tables, using 53 (16 KB) aligned memory objects.

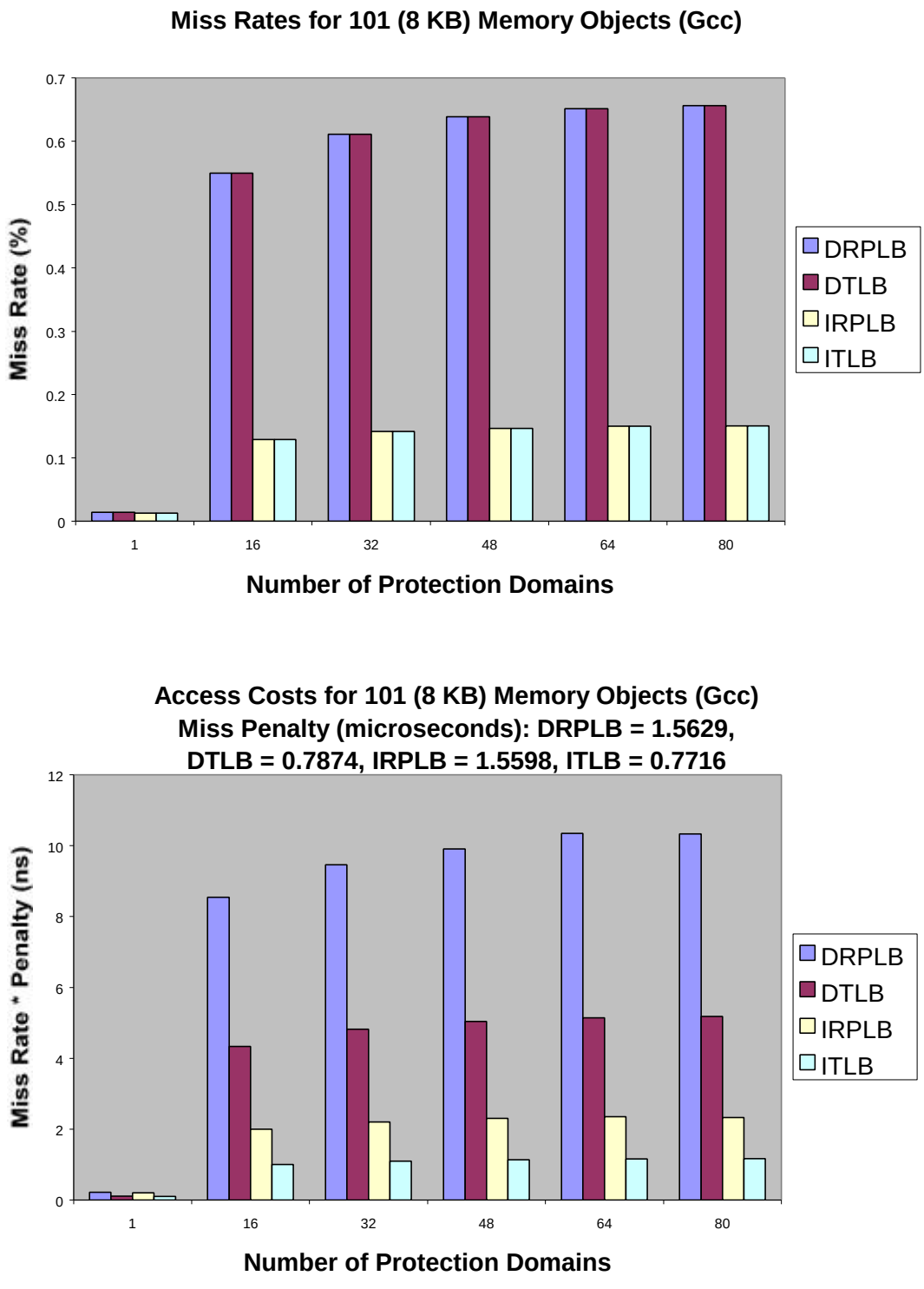


Figure 18: RPLB/PRAL vs. TLB/page tables, using 101 (8 KB) aligned memory objects.

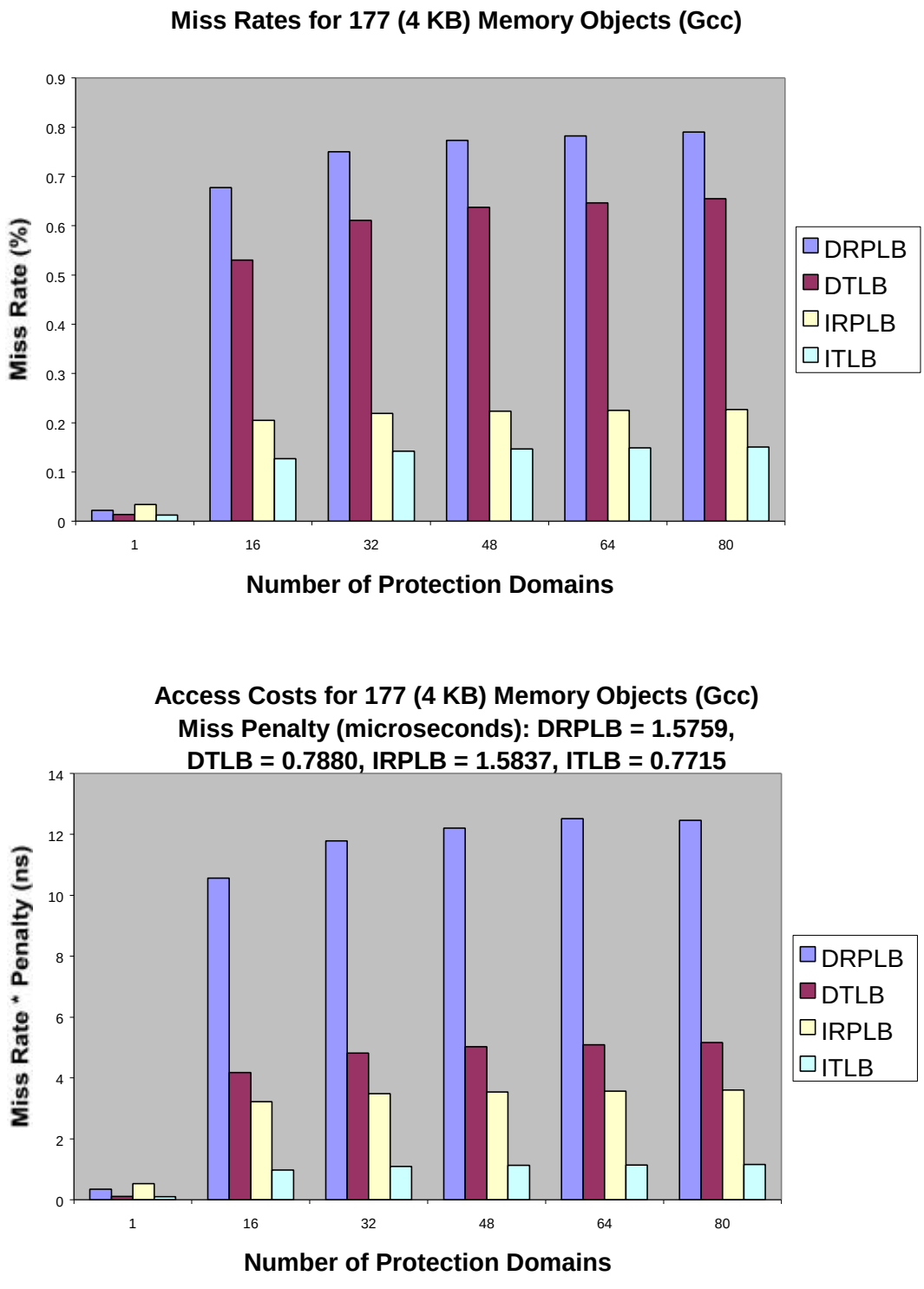


Figure 19: RPLB/PRAL vs. TLB/page tables, using 177 (4 KB) aligned memory objects.

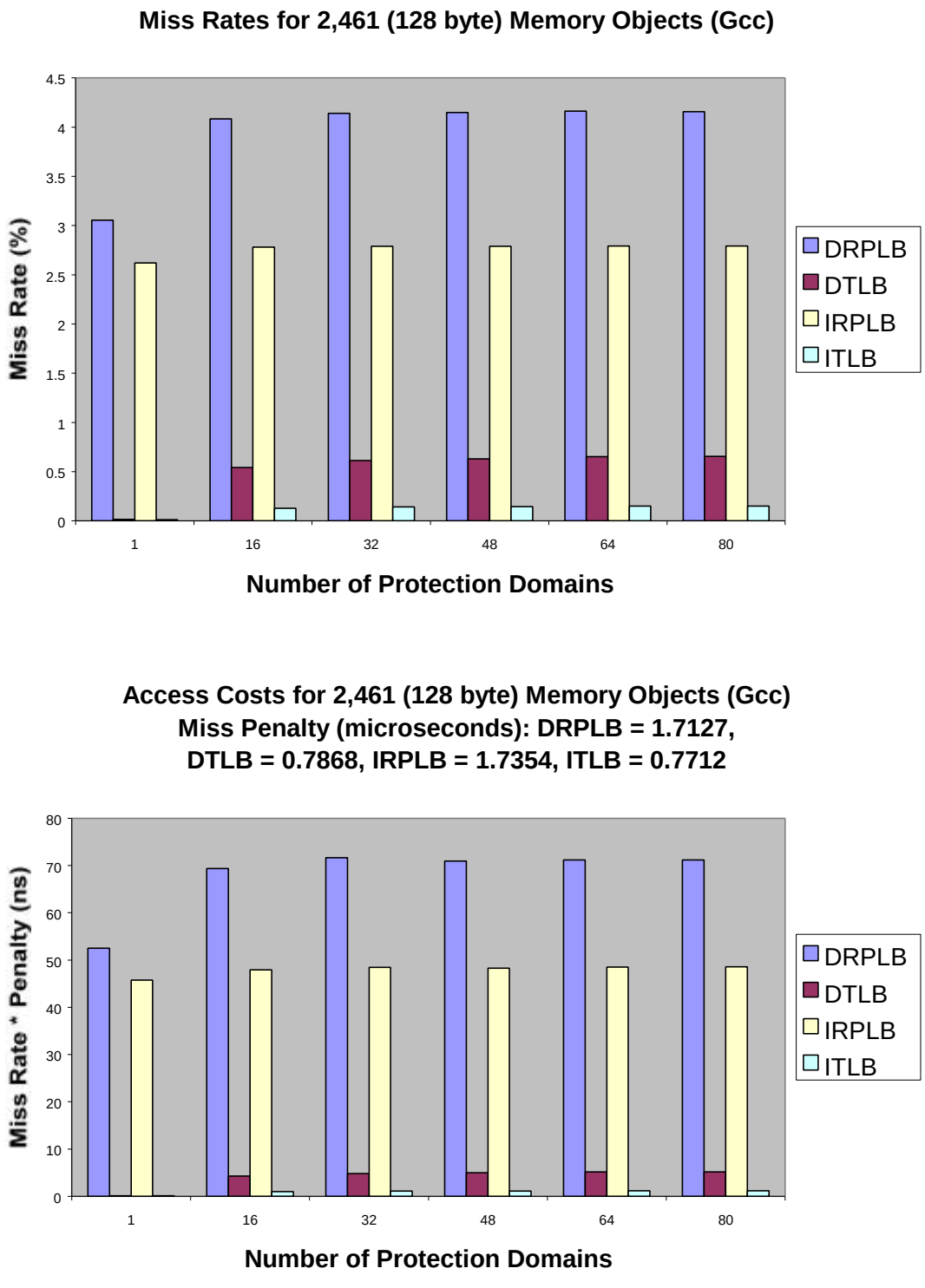


Figure 20: RPLB/PRAL vs. TLB/page tables, using 2,461 (128 byte) aligned memory objects.

Observations:

1. Varying the number of protection domains past a certain point has insignificant effect.
2. Sombrero performs significantly better than Alpha-NT for memory object-sizes significantly larger than a page.
3. The average RPLB miss penalty is likely to be twice as high as the average TLB miss penalty.
4. The higher RPLB miss penalty can be offset by the lower RPLB miss rate.
5. The RPLB miss rate is likely to be lower than the TLB miss rate under many common scenarios, for instance, when the memory objects accessed are likely to be twice as large as a regular page. In these situations performance, as measured by the product of miss rate and miss penalty, appears to be roughly comparable.
6. Fewer memory objects per domain lead to a faster PRAL search.
7. The PRAL search speed can be improved by careful programming, perhaps at assembly level.
8. Aligned memory objects lead to better Sombrero performance.
9. In a typical MASOS, if multiple small memory objects were stuffed in a page, they would be protected via software. That overhead is unaccounted for in the TLB/page tables simulation and comparison charts. Thence, in reality, Sombrero's performance in scenarios with small memory objects may be more competitive than suggested by the results shown here.
10. In Sombrero, small-grained protection can be done via the RPLB or via software.

6. Conclusion and Future Work

This research and implementation represent a small step, but progress was made in that now there is a prototype of the Sombrero protection structures and algorithms. In particular, now there is a prototype of the Sombrero PRAL and a simulation of the RPLB. And there is a "first-approximation" evaluation of the Sombrero protection model. This work has facilitated better understanding of the scope, strengths and weaknesses of the Sombrero protection model.

And, significantly, now there is a flexible test environment that allows one to create, modify and delete the PRAL and PRAL elements, set relationships amongst various objects, print control blocks, perform "memory accesses" via the RPLB and/or the TLB, vary the size of the RPLB, etc. There is the ability of checking how the Sombrero RPLB/PRAL behaves differently than the Alpha-NT TLB/page tables as various parameters are changed, and it is easier to tell which situations are likely to produce better results on the Sombrero side.

This work brought to light some issues, difficulties and techniques involved, and resulted in functioning software for further experimentation and development.

6.1 Summary of Value of Research

This research contributed to the development of a novel operating system that provides a single address space and requires some special-purpose protection caching and management hardware. For example, this research:

1. Produced the first synthesis of the PRAL and other Sombrero protection-related structures and algorithms.
2. Implemented a prototype that is a step towards demonstration of the feasibility of the Sombrero protection model and is "proof-of-concept" for certain aspects of Sombrero protection.
3. Generated the first profile of behavior differences between the RPLB and a regular TLB.
4. Experimented with PRAL search optimization. Explored the implementation of a hashing mechanism.

6.2 Future Work

6.2.1 Memory Access Traces

Memory access traces of applications run on a SASOS would provide more representative inputs for the simulation.

6.2.2 Additional Experiments

The test/simulation environment could be used to perform experiments involving domain switches. Additionally, it could be used to identify the "optimal" RPLB size, quantum per thread, etc.

6.2.3 Multi-Thread, Multi-Processor and Multi-Node Issues

The control blocks and algorithms will need to be modified to work in a multi-thread, multi-processor and multi-node environment. In a multi-processor system, PRAL locking will need to be implemented. Probably a "read lock" of the PRAL will be required during RPLB miss handling.

6.2.4 RPLB Miss Handling

The PRAL data structures and search algorithms can be improved. For instance, it may be possible to implement a faster PRAL search by using a carefully constructed combination of hash tables and binary search arrays.

For efficiency, the RPLB miss handling code needs to be converted to Alpha PALcode. (The Alpha-NT TLB miss handling code is in PALcode.)

6.2.5 Universal Trusted Domain

Everyone should be allowed to issue system calls, but it would probably be inefficient to establish switches for all domains. So, "universally" accessible code and data such as system calls might be attached to a domain called the Universal Trusted Domain (UTD, domain "0"). When checking whether a thread had access to a certain object, first the GPD, then the CPD, and finally the UTD would be checked. The

tradeoff is more overhead. Further exploration of the UTD concept is required. Should the RPLB be used for UTD code and data? How should the UTD be implemented? It could, for example, be implemented as a passive server with executable code and read-only data.

6.2.6 System Calls

The system calls need to be implemented.

6.2.7 Unaligned Data

The difficulty with "unaligned" data needs to be investigated. For instance, depending on the architecture of a system, a word of memory might be accessed directly even though the RPLB authorizes access to the first byte only (not the second one).

6.2.8 Principals

The Sombrero papers refer to an additional fundamental object type: principal ("user"). All objects in this prototype are "owned" by the same principal: "ROOT." The test environment doesn't allow for the creation of new principals. The implementation of principals, i.e., unique protected identities, is beyond the scope of this thesis. However, as indicated in [Car96], authorization by principals can be done entirely at user level. It also appears that establishment and authentication of users in a SASOS with CPU hardware protection can also be done at user level by passive services.

6.2.9 Code Clean-Up

Some memory allocated (e.g., heaps, page table) is not explicitly released; the assumption is made that Windows will free-up all the resources used by the prototype after it exits.

7. Bibliography

- [Car96] Carnes, M., “Sombrero System Interface,” Computer Science and Engineering Department, Arizona State University, *MCS Project*, December 1996.
- [He*96] Hennessy, John L., and Patterson, David A., *Computer Architecture A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., 1996.
- [Lam71] Lampson, B.W., “Protection,” in *Proceedings. Fifth Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971, pp. 437-443, reprinted in *Operating Systems Review*, 8, 1, January 1974, pp. 18-24.
- [Si*95] Sites, Richard L., and Witek, Richard T., *Alpha AXP Architecture Reference Manual*, Second Edition, Digital Press, 1995.
- [Sko94] Skousen, A. C., “Sombrero: A Very Large Single Address Space Distributed Operating System", Department of Computer Science and Engineering, Arizona State University, *MS Thesis*, December 1994.
- [Sko97] Skousen, A. Private consultation, September 1997.
- [Sk*96a] Skousen, A.C., Miller, D.S., and Feigen, R.G., “The Sombrero Operating System An Operating System For A Distributed Single Very Large Address Space General Introduction,” *ASU 64-bit OS Group White Paper 0*, Computer Science and Engineering Department, Technical Report TR-96-005, April 1996. (Available at http://www.eas.asu.edu/~sasos.)
- [Sk*96b] Skousen, A. C. and Miller, D. S., "Resource Access and Protection in Sombrero: Protection Model, Software Protection Data Structures and Hardware Range Protection Lookaside Buffer", *ASU 64-bit OS Group WP 2*, Computer Science and Engineering Department, Arizona State University, TR-95-013, May 1996.
- [Sk*96c] Skousen, A. C. and Miller, D. S., "Implementing a Single Very Large Address Space across Multiple Nodes: Memory Partitioning, Protection Domain Migration, Kernel Replication, Consistency

and Fault Tolerance", *ASU 64-bit OS Group WP 10*, Computer Science and Engineering Department, Arizona State University, TR-95-021, May 1996.

- [Sk*97a] Skousen, A., and Miller, D., "Hardware/Software Design in a Distributed Single Address Space," Computer Science and Engineering Department, Arizona State University, TR-97-018, July 1997.
- [Sk*97b] Skousen, A., and Miller, D., "Using a Single Address Space Operating System to Support Client/Server Distributed Computing", Computer Science and Engineering Department, Arizona State University, TR-97-019, July 1997.
- [Sk*97c] Skousen, A., and Miller, D., "A Hardware Supported Single Address Space Operating System Design," Computer Science and Engineering Department, Arizona State University, December 1997, submitted to the USENIX Annual Technical Conference, June 1998.

8. Additional References

- Chase, Jeffrey S., "An Operating System Structure for Wide-Address Architectures," Ph.D. thesis, Department of Computer Science and Engineering, University of Washington, August 1995
- Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L., *Introduction to Algorithms*, The Massachusetts Institute of Technology Press, 1990.
- Custer, Helen, *Inside Windows NT*, Microsoft Press, 1993.
- Engler, Dawson R., Kaashoek, M. Frans, and O'Toole Jr., James, "Exokernel: An Operating System Architecture for Application-Level Resource Management," M.I.T. Laboratory for Computer Science, copyright by the Association for Computing Machinery, Inc. (ACM), 1995.
- Heiser, Gernot, Elphinstone, Kevin, Russell, Stephen, and Hellestrand, Graham R., "A Distributed Single Address-Space Operating System Supporting Persistence," School of Computer Science and Engineering, The University of New South Wales, SCS&E Report 9302, March 1993.
- Koldinger, Eric J., Chase, Jeffrey S., and Eggers, Susan J., "Architectural Support for Single Address Space Operating Systems." In *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems,* pages 175-186, October 1992.
- Liedtke, Jochen, "On µ-Kernel Construction," GMD – German National Research Center for Information Technology, copyright by the Association for Computing Machinery, Inc. (ACM), 1995.
- Liedtke, Jochen, Elphinstone, Kevin, Schönberg, Sebastian, Härtig, Hermann, Heiser, Gernot, Islan, Nayeem, Jaeger, Trent, "Achieved IPC Performance," 6th Workshop on Hot Topics in Operating Systems (HotOS), Chatham (Cape Cod), Massachusetts, May 5-6, 1997.
- Tanenbaum, A. S., *Distributed Operating Systems*, Prentice-Hall, Inc., 1995.
- Talluri, Madhusudhan, Hil, Mark D., and Khalidi, Yousef A., "A New Page Table for 64-bit Address Spaces," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles,* 1995.
- Wilkinson, Timothy James, "Implementing Fault Tolerance in a 64-bit Distributed Operating System," Ph.D. thesis, Systems Architecture Research Centre, City University, July 1993.

- Wilkinson, Tim, Murray, Kevin, Russell, Stephen, Heiser, Gernot, Liedtke, Jochen, “Single Address Space Operating Systems,” School of Computer Science and Engineering, The University of New South Wales, UNSW-CSE-TR-9504, November 13, 1995.

Appendix A

User's Manual

A. User's Manual

A.1 Execution

To run the "PRAL Prototype program," log on to the SOMBRERO1 (Alpha) machine in the ASU OS Lab (in GWC-321). Click on the "Start" button, select "Run …," and enter `D:\SHAH\PRAL\PRAL.EXE`. Make sure the input file `MAIN.IN` and all the other files accessed are in the same directory and reflect what you intend to do. A sample input file is provided below.

A.2 Test Environment

The test and simulation facility consists of a main input file called MAIN.IN and a corresponding output file called MAIN.OUT. Additional input files (which generate separate output files) can be specified recursively. The input and output files are in plain text.

In the input file:

- An exclamation mark (!) in the first line denotes a comment line. Comment lines and blank lines are skipped.
- A period (.) denotes "current object," e.g., "current thread," or default.
- An asterisk (*) denotes a wildcard, and can be used to specify multiple objects. For instance, Michael* could mean Michael;1, Michael;2, etc.
- A * by itself means all instances in the Sombrero registry.
- For instance, if thread Michael;2 were registered before thread Michael;1, Michael* would resolve to Michael;1 first and then Michael;2. (Names are retrieved from the registry in the order that they were added.)
- A percent sign (%) by itself means all instances of a particular object type, even those without "user-friendly" names.

- There is a limit of 2,500 instances per parameter. That is, the maximum "batch" size is 2,500. For instance, if Michael* is specified as a parameter, there can be at most 2,500 objects named starting with "Michael."
- Objects of different types should be named with unique root strings so that there is no "mix up" when batch operations are performed. For instance, if GPDs were named starting with "Floppy" and threads were named starting with "Flop," if "Flop*" threads were referenced by the user, some GPDs could be included in the batch of objects retrieved from the registry.
- All lines are converted to upper-case and "trimmed" before processing.
- The #INCLUDE directive allows the inclusion of a file. If there is a file with, say, hundreds of commands to create memory objects, rather than copying those commands into the input file at hand, #INCLUDE can be used instead to execute those commands. For instance, #INCLUDE SPICE.IN would initiate the parsing of the commands in the file SPICE.IN. It can also be viewed as a "macro." The #INCLUDE directive can be nested.
- The #ECHO_OFF keyword can be used to turn off the echoing of the input commands. If, for example, an input file has hundreds of memory objects generated programmatically, it makes sense to have #ECHO_OFF at the top of the file.
- The #ECHO_ON keyword turns the echoing of the input commands back on. It is not necessary to append this command at the end of "#INCLUDE" files. That's because #ECHO_OFF and #ECHO_ON do not apply to the including (parent) file.
- The #ECHO keyword can be used to echo a string to the output file. For instance, if a header is required for a table comparing the RPLB and TLB, this directive can be specified:

  #ECHO DRPLB Miss Rate DTLB Miss Rate
- The #EXIT keyword means abort (it is useful to stop parsing in the middle of an input file).

A.3 Valid Commands

The commands that can be specified in an input file are documented below, in alphabetical order.

b_stats_category:

Parameters:

Category name (or nothing). Valid options: NUMBER_OF_THREADS, NUMBER_OF_MEMORY_OBJECTS, RPLB_SIZE (or nothing)

Remarks:

When generating statistics in table format, the rows can be preceded by one of the above categories.

See “b_stats_file_open” for more information.

Example:

b_stats_category:

b_stats_column_groups:

Parameters:

Number of “column groups” or series

Remarks:

The command “b_stats_column_groups: 3” should be used if, for instance, statistics regarding RPLB and/or TLB accesses are collected for the Gcc, Spice and TeX benchmarks and it is desired to print these statistics in columns groups, each corresponding to Gcc, Spice and TeX, respectively.

“b_stats_column_groups: 0” signals the end of column-formatted output.

See “b_stats_file_open:” for more information.

Example: b_stats_column_groups: 1

b_stats_file_open:

Parameters:

Output file name

Remarks:

This command allows “buffer statistics” to be redirected to a separate file. Using other commands documented herein, it is possible to collect the statistics in a table format that can be easily imported into the MS Excel program, wherefrom graphs can be generated.

There can only be one active stats file at a time. If this command is reissued, the current stats file, if any, is automatically closed.

Example:

b_stats_file_open: stats.out

b_stats_file_close:

Parameters:

(None)

Remarks:

The stats file is automatically closed upon thread exit, so this command would not be used normally. However, this command is useful to redirect output from the “stats file” to the thread output file.

Example: b_stats_file_close: stats.out

b_stats_print_miss_rate:

Parameters:

Valid options: Yes or No

Remarks:

This command turns on or off the inclusion of “miss rate” in the “buffer stats” table.

If miss rate, miss penalty, and “cost” (rate times penalty) are all selected to be printed for both the RPLB and TLB, they are printed in this order:

DRPLB miss rate, DRPLB miss penalty, DRPLB cost, DTLB miss rate, DTLB miss penalty, DTLB cost, IRPLB miss rate, IRPLB miss penalty, IRPLB cost, ITLB miss rate, ITLB miss penalty, ITLB cost.

Example:

b_stats_print_miss_rate: yes

b_stats_print_miss_penalty:

Parameters:

Valid options: Yes or No

Remarks:

This command turns on or off the inclusion of “average miss penalty” in the “buffer stats” table. See b_stats_print_miss_rate: for more information.

Example:

b_stats_print_miss_penalty: yes

b_stats_print_rate_times_penalty:

Parameters:

Valid options: Yes or No

Remarks:

This command turns on or off the inclusion of “ miss rate * average miss penalty” in the “buffer stats” table. See b_stats_print_miss_rate: for more information.

Example:

b_stats_print_rate_times_penalty: yes

drplb_size: (see irplb_size:)

gpd_add:

Parameters:

Number of general protection domains to add

User-friendly name to assign (optional)

Example:

gpd_add: 3 Floppy*

gpd_del:

Parameters:

General protection domain name

Remarks:

You might want to delete a thread associated with a domain before you delete a GPD. Also, be careful not to delete an Authorized Protection Domain without deleting all the objects that refer to it.

If you delete a GPD, you might unwittingly delete a memory object too, as a memory object is automatically deleted when there are no references to it left.

Example:

gpd_del: Floppy*

gpd_print:

Parameters:

General protection domain name

Remarks:

If no name is specified, all GPDs are printed. Otherwise, the regular naming convention applies.

Example:

gpd_print: Floppy*

hash_structure_build:

Parameters:

Domain name (GPD [default] or thread name)

Remarks:

Before the PRAL search algorithm "HASH_DIVISION" (documented below) can be used with a domain, it is necessary to build a "hash table" with all the memory objects the said domain has access to. That's because presently there is no code to automatically update the hash structure when the PRAL is updated.

Note that, if there is already a hash structure attached to a domain, it is automatically deleted before a new one is built.

Example:

hash_structure_build: Floppy*

hash_structure_print:

Parameters:

Domain name (GPD [default] or thread name)

Remarks:

This command is useful to analyze the “hash table” associated with a domain, e.g., for debugging or fine-tuning.

Example:

hash_structure_print: Floppy*

irplb_size:/drplb_size:/srplb_size:

Parameters:

Integer less than or equal to MAX_I/D/SRPLB_SIZE (which is set at compile time; a typical value is 1024)

Remarks:

The default I/D/SRPLB size is 32.

Example:

irplb_size: 64

mo_add:

Parameters:

Number of memory objects to add

Size wanted

Size needed (0 means the same as size wanted)

Domain name (GPD [default] or thread name)

User-friendly name to assign (optional)

Remarks:

Unlike with the mo_add_specific: command, if more domains than memory objects are specified, the remaining domains are ignored.

Example:

mo_add: 13 10 0 Floppy* Kira*

mo_add_specific:

Parameters:

Start address (in hexadecimal)

End address (in hexadecimal)

"Needed" size (or 0 to make the physical size the same as the virtual size) (in decimal)

Access desired {RWX}

Domain name (GPD [default] or thread name)

User-friendly name to assign (optional)

Remarks:

Unlike a couple of the other parameters, the size is in decimal notation.

Note that, if multiple domains are specified, the memory object is attached to each of them. That is not the case with the mo_add: command!

Since the memory access traces obtained from the Internet are in the same range, and since no actual memory is reserved or committed, the Start address and End address of different memory objects can overlap. This facilitates testing. But, if there are overlapping memory objects, they should be in separate domains.

Example:

mo_add_specific: 400000 500000 0 X Michael;1 Kira;1

mo_del:

Parameters:

Memory object name

Example:

mo_del: Kira*

mo_print:

Parameters:

Memory object name

Remarks:

If no name is specified, all memory objects are printed. Otherwise, the regular naming convention applies.

Example:

mo_print: Kira*

pral_search:

Parameters:

Search algorithm name. Valid options: BINARY_SEARCH_ARRAY (default), LINEAR_SEARCH_LINKED_LIST and HASH_DIVISION

Remarks:

On a RPLB miss, the RPLB miss handler must search the PRAL to validate access (and, if appropriate, load the RPLB). This allows one to select the search algorithm to be used. The default (if this command is not issued at all) is BINARY_SEARCH_ARRAY.

The "HASH_DIVISION" parameter specifies that the division hashing algorithm should be used. Note that, before "HASH_DIVISION" can be used to search a particular domain, a "hash structure" must be explicitly built for it using the "hash_structure_build:" command (as documented above).

Example:

pral_search: hash_division

quantum:

Parameters:

Number of cache accesses a thread can perform before being preempted

Remarks:

The default (if this command is not issued at all) is 10,000.

Example:

quantum: 1000

reference_set:

Parameters:

Memory object name

Domain name (GPD [default] or thread name)

Access desired {RWX.} ("." means typical RWX access [the same as the access set in the MOCB])

Remarks:

This command can be used to attach memory objects to general or thread/carrier protection domains.

A "." cannot be used to specify the memory object name.

Example:

reference_set: Kira* Floppy;1 .

reference_del:

Parameters:

Memory object name

Domain name (GPD [default] or thread name)

Remarks:

This command can be used to detach memory objects from general or thread/carrier protection domains. Note that a memory object is automatically deleted when there are no references to it left.

A "." cannot be used to specify the memory object name.

Example:

reference_del: Kira;1 Floppy;1

registry_print:

Parameters:

(None)

Remarks:

Prints all "user-friendly" names and the corresponding addresses. Note that when objects are deleted their names are automatically deleted from the registry.

Example:

registry_print:

rplb_accesses:/tlb_accesses:/rplb_and_tlb_accesses:

Parameters:

Thread name

File name

Remarks:

If there are multiple threads, each thread in the batch performs “quantum” accesses. (See the description of the “quantum:” command.) Threads are scheduled randomly.

If there are multiple files that match the “File name” parameter, they are sorted by name first. Then threads are matched with files sequentially. For instance, if there were 2 threads in the “Michael” batch, “tlb_accesses Michael* Kichi*” would result in thread Michael;1 being matched with file Kichi1 and thread Michael;2 being matched with file Kichi2.

If there are more threads than files specified, some threads end up reading from the same file. In the case above, for example, if there were no Kichi2 file, both threads Michael;1 and Michael;2 would read from Kichi1 as if each opened Kichi1 separately.

The file(s) should contain access-mode (0 [read], 1 [write] or 2 [execute]) and virtual address pairs. For instance:

```
2 408ed4
0 10019d94
2 408ed8
1 10019d88
```

Example:

rplb_and_tlb_accesses: Michael* gcc.rwx

rplb_clear/tlb_clear:

Parameters:

(None)

Remarks:

Clears/flushes/resets the RPLB/TLB (buffer) and the corresponding “stats” counters.

Example:

rplb_clear:

rplb_evictions:

Parameters:

RPLB entry eviction algorithm name. Valid options: NLU (“Not Last Used”) or LRU (“Least Recently Used”)

Remarks:

The RPLB cache can hold only so many entries. When it is full and a new entry must be inserted, an algorithm is needed to remove an existing entry. The NLU algorithm throws out any entry, as long as it was not the last one accessed. The LRU algorithm is more sophisticated (but more costly), as it throws out the entry that was least recently accessed.

Since the NLU algorithm is always used in the TLB simulation, the default RPLB eviction algorithm (assumed if the “rplb_evictions:” command is not issued at all) is NLU.

Example:

rplb_evictions: LRU

rplb_stats_print:/tlb_stats_print:

Parameters:

(None)

Remarks:

These print RPLB/TLB access statistics in detail.

Example:

rplb_stats_print:

srplb_size: (see irplb_size:)

switch_set:

Parameters:

“From” domain name (GPD [default] or thread name)

“To” general protection domain

Switch address (in hexadecimal)

Access desired {RWX}

Example:

switch_set: Michael* Floppy;1 400000 X

switch_del:

Parameters:

“From” domain name (GPD [default] or thread name)

“To” general protection domain

Switch address (“.” means the first switch address found)

Example:

switch_del: Michael* Floppy;1 .

thread_add:

Parameters:

Number of threads to add

General protection domain name

User-friendly name to assign (optional)

Example:

thread_add: 3 Floppy* Michael*

thread_del:

Parameters:

Thread name

Remarks:

If you delete a thread, you might unwittingly delete a memory object too, as a memory object is automatically deleted when there are no references to it left.

Example:

thread_del: Michael*

thread_print:

Parameters:

Thread name

Remarks:

If no name is specified, all threads are printed. Otherwise, the regular naming convention applies.

Example:

thread_print: Michael*

tlb_accesses: (see rplb_accesses:)

tlb_clear: (see rplb_clear:)

tlb_stats_print: (see rplb_stats_print:)

windows_thread_add:

Parameters:

Number of threads to add

General protection domain name

Sombrero thread name (can't be ".")

Input file name

Remarks:

This command can be used to generate multiple Windows threads to simulate a multithreading environment.

The output file name is generated from the input file name, using "OUT" as the extension. For example, if the input file name is "Kichi1.IN," the output file name will be "Kichi1.OUT."

Caveat: This command has not been thoroughly tested, as, for the sake of performance, all experiments were performed with only one Windows thread.

Example:

windows_thread_add: 3 Floppy* Michael* Kichi*.IN

A.4 Sample Input and Output Files

MAIN.IN

```
pral_search: binary_search_array
irplb_size: 48
drplb_size: 64
rplb_evictions: NLU

gpd_add: 1 owner_gpd_for_gcc
gpd_add: 1 owner_gpd_for_spice
gpd_add: 1 owner_gpd_for_tex

#include gcc_4096.43
#include spice_2048.34
#include tex_8192.8

gpd_add: 1 gpd_for_gcc*
thread_add: 1 gpd_for_gcc* thread_for_gcc*
gpd_add: 1 gpd_for_spice*
thread_add: 1 gpd_for_spice* thread_for_spice*
gpd_add: 1 gpd_for_tex*
thread_add: 1 gpd_for_tex* thread_for_tex*

reference_set: gcc* gpd_for_gcc* .
reference_set: spice* gpd_for_spice* .
reference_set: tex* gpd_for_tex* .

rplb_clear:
tlb_clear:
rplb_and_tlb_accesses: thread_for_gcc* gcc.rwx
rplb_stats_print:
tlb_stats_print:
rplb_clear:
tlb_clear:
rplb_and_tlb_accesses: thread_for_spice* spice.rwx
rplb_stats_print:
tlb_stats_print:
rplb_clear:
tlb_clear:
rplb_and_tlb_accesses: thread_for_tex* tex.rwx
rplb_stats_print:
tlb_stats_print:

thread_del: thread_for_gcc*
gpd_del: gpd_for_gcc*
thread_del: thread_for_spice*
gpd_del: gpd_for_spice*
thread_del: thread_for_tex*
gpd_del: gpd_for_tex*

mo_del: gcc*
mo_del: spice*
mo_del: tex*
```

MAIN.OUT

```
Current time: Wed Dec 03 10:01:36 1997

> PRAL_SEARCH: BINARY_SEARCH_ARRAY

> IRPLB_SIZE: 48

> DRPLB_SIZE: 64

> RPLB_EVICTIONS: NLU

> GPD_ADD: 1 OWNER_GPD_FOR_GCC

> GPD_ADD: 1 OWNER_GPD_FOR_SPICE

> GPD_ADD: 1 OWNER_GPD_FOR_TEX

> #INCLUDE GCC_4096.43

> #ECHO_OFF

> #INCLUDE SPICE_2048.34

> #ECHO_OFF

> #INCLUDE TEX_8192.8

> #ECHO_OFF

> GPD_ADD: 1 GPD_FOR_GCC*

> THREAD_ADD: 1 GPD_FOR_GCC* THREAD_FOR_GCC*

> GPD_ADD: 1 GPD_FOR_SPICE*

> THREAD_ADD: 1 GPD_FOR_SPICE* THREAD_FOR_SPICE*

> GPD_ADD: 1 GPD_FOR_TEX*

> THREAD_ADD: 1 GPD_FOR_TEX* THREAD_FOR_TEX*

> REFERENCE_SET: GCC* GPD_FOR_GCC* .

> REFERENCE_SET: SPICE* GPD_FOR_SPICE* .

> REFERENCE_SET: TEX* GPD_FOR_TEX* .

> RPLB_CLEAR:

> TLB_CLEAR:

> RPLB_AND_TLB_ACCESSES: THREAD_FOR_GCC* GCC.RWX

> RPLB_STATS_PRINT:
Number of threads:                                        1
RPLB miss handling:                    Binary search (array)
RPLB evictions:                          Not Last Used entry
Data RPLB (times are in microseconds, unless otherwise noted)
  Size:                                                  64
  Hits:                                              242187
  Misses:                                               474
  Miss rate:                                            0.1953%
  Min miss handling time:                               1.3222
  Max miss handling time:                               2.0323
  Average miss handling time (miss penalty):            1.6057
  Total miss handling time:                           761.0844
  Miss rate * miss penalty (in nanoseconds):            3.1364
```

```
Instruction RPLB (times are in microseconds, unless otherwise noted)
  Size:                                                  48
  Hits:                                              756748
  Misses:                                               593
  Miss rate:                                            0.0783%
  Min miss handling time:                               1.3362
  Max miss handling time:                               1.9723
  Average miss handling time (miss penalty):            1.5996
  Total miss handling time:                           948.5356
  Miss rate * miss penalty (in nanoseconds):            1.2525

> TLB_STATS_PRINT:
Number of threads:                                      1
TLB evictions:                          Not Last Used entry
Data TLB (times are in microseconds, unless otherwise noted)
  Size:                                                 64
  Hits:                                              242627
  Misses:                                                34
  Miss rate:                                            0.0140%
  Min miss handling time:                               0.7821
  Max miss handling time:                               1.0041
  Average miss handling time (miss penalty):            0.8188
  Total miss handling time:                            27.8380
  Miss rate * miss penalty (in nanoseconds):            0.1147
Instruction TLB (times are in microseconds, unless otherwise noted)
  Size:                                                 48
  Hits:                                              757246
  Misses:                                                95
  Miss rate:                                            0.0125%
  Min miss handling time:                               0.7821
  Max miss handling time:                               0.9301
  Average miss handling time (miss penalty):            0.7877
  Total miss handling time:                            74.8269
  Miss rate * miss penalty (in nanoseconds):            0.0988

> RPLB_CLEAR:

> TLB_CLEAR:

> RPLB_AND_TLB_ACCESSES: THREAD_FOR_SPICE* SPICE.RWX

> RPLB_STATS_PRINT:
Number of threads:                                      1
RPLB miss handling:                   Binary search (array)
RPLB evictions:                         Not Last Used entry
Data RPLB (times are in microseconds, unless otherwise noted)
  Size:                                                 64
  Hits:                                              217015
  Misses:                                               222
  Miss rate:                                            0.1022%
  Min miss handling time:                               1.2802
  Max miss handling time:                               2.5444
  Average miss handling time (miss penalty):            1.6344
  Total miss handling time:                           362.8346
  Miss rate * miss penalty (in nanoseconds):            1.6702
Instruction RPLB (times are in microseconds, unless otherwise noted)
  Size:                                                 48
  Hits:                                              782541
  Misses:                                               223
  Miss rate:                                            0.0285%
  Min miss handling time:                               1.3882
  Max miss handling time:                               2.1883
  Average miss handling time (miss penalty):            1.5894
  Total miss handling time:                           354.4374
  Miss rate * miss penalty (in nanoseconds):            0.4528

> TLB_STATS_PRINT:
Number of threads:                                      1
TLB evictions:                          Not Last Used entry
Data TLB (times are in microseconds, unless otherwise noted)
  Size:                                                 64
```

```
  Hits:                                                 217221
  Misses:                                                   16
  Miss rate:                                                 0.0074%
  Min miss handling time:                                    0.7821
  Max miss handling time:                                    1.0001
  Average miss handling time (miss penalty):                 0.8026
  Total miss handling time:                                 12.8419
  Miss rate * miss penalty (in nanoseconds):                 0.0591
Instruction TLB (times are in microseconds, unless otherwise noted)
  Size:                                                     48
  Hits:                                                 782739
  Misses:                                                   25
  Miss rate:                                                 0.0032%
  Min miss handling time:                                    0.7821
  Max miss handling time:                                    0.7841
  Average miss handling time (miss penalty):                 0.7826
  Total miss handling time:                                 19.5648
  Miss rate * miss penalty (in nanoseconds):                 0.0250

> RPLB_CLEAR:

> TLB_CLEAR:

> RPLB_AND_TLB_ACCESSES: THREAD_FOR_TEX* TEX.RWX

> RPLB_STATS_PRINT:
Number of threads:                                           1
RPLB miss handling:                       Binary search (array)
RPLB evictions:                             Not Last Used entry
Data RPLB (times are in microseconds, unless otherwise noted)
  Size:                                                     64
  Hits:                                                 235127
  Misses:                                                   41
  Miss rate:                                                 0.0174%
  Min miss handling time:                                    1.2902
  Max miss handling time:                                    1.9303
  Average miss handling time (miss penalty):                 1.5411
  Total miss handling time:                                 63.1852
  Miss rate * miss penalty (in nanoseconds):                 0.2687
Instruction RPLB (times are in microseconds, unless otherwise noted)
  Size:                                                     48
  Hits:                                                 597292
  Misses:                                                   16
  Miss rate:                                                 0.0027%
  Min miss handling time:                                    1.3642

  Max miss handling time:                                    1.5322
  Average miss handling time (miss penalty):                 1.4640
  Total miss handling time:                                 23.4234
  Miss rate * miss penalty (in nanoseconds):                 0.0392

> TLB_STATS_PRINT:
Number of threads:                                           1
TLB evictions:                              Not Last Used entry
Data TLB (times are in microseconds, unless otherwise noted)
  Size:                                                     64
  Hits:                                                 235157
  Misses:                                                   11
  Miss rate:                                                 0.0047%
  Min miss handling time:                                    0.7821
  Max miss handling time:                                    0.8301
  Average miss handling time (miss penalty):                 0.7878
  Total miss handling time:                                  8.6653
  Miss rate * miss penalty (in nanoseconds):                 0.0368
Instruction TLB (times are in microseconds, unless otherwise noted)
  Size:                                                     48
  Hits:                                                 597304
  Misses:                                                    4
  Miss rate:                                                 0.0007%
  Min miss handling time:                                    0.7821
  Max miss handling time:                                    0.7821
```

```
  Average miss handling time (miss penalty):              0.7821
  Total miss handling time:                               3.1285
  Miss rate * miss penalty (in nanoseconds):              0.0052

> THREAD_DEL: THREAD_FOR_GCC*

> GPD_DEL: GPD_FOR_GCC*

> THREAD_DEL: THREAD_FOR_SPICE*

> GPD_DEL: GPD_FOR_SPICE*

> THREAD_DEL: THREAD_FOR_TEX*

> GPD_DEL: GPD_FOR_TEX*

> MO_DEL: GCC*

> MO_DEL: SPICE*

> MO_DEL: TEX*

> #EXIT

Current time: Wed Dec 03 10:01:57 1997
```

Appendix B

Source Code

B. Source Code

The source code written consists of the files listed below. Only the contents of the "header" (definition) files are shown.

GLOBAL.H

Throughout the code, types like "VA_Size_Int" (instead of "unsigned long int") were used to facilitate the portability to a 64-bit environment.

A large number of global variables were used, to expedite the implementation and testing. However, some of the global variables cannot be used in true multi-threaded situations, as one thread might override another's settings. But some of these global variables were added to obtain performance statistics only, when there was only one Windows thread running.

```
/* global.h
 * SAK 11/20/97
 */

#ifndef _GLOBAL_H
#define _GLOBAL_H

#define WIN32_LEAN_AND_MEAN

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define NO_CHANGE 15
#define END_OF_FILE 20
#define ILLEGAL_ACCESS 100
#define ILLEGAL_EXECUTE 101
#define ILLEGAL_WRITE 102
#define ILLEGAL_READ 104
#define NO_ACCESS_DESCRIPTOR 110
#define LOCKED_AD 130
#define LOCKED_MOCB 140
#define NO_TOKEN 200
#define OUT_OF_MEMORY 1000
#define TROUBLE_PIPING 1005
#define IO_ERROR 1007
#define MEMORY_MANAGEMENT_PROBLEM 1010
#define THREAD_CREATION_PROBLEM 1111
#define THREAD_SYNCHRONIZATION_PROBLEM 1121
#define THREAD_LOCAL_STORAGE_PROBLEM 1131
#define DATA_INTEGRITY_ERROR 2000
#define DUPLICATE_ENTRY 10000
#define INPUT_ERROR 10100
#define OUTPUT_ERROR 10150
#define INVALID_PARAMETER 10200
```

```
#define UNKNOWN_FUNCTION_NAME 10300
#define NOT_FOUND 20000
#define DOMAIN_NOT_FOUND 20010
#define THREAD_NOT_FOUND 20020
#define MO_NOT_FOUND 20040
#define CANNOT_ATTACH_TCB_TO_EXECUTABLE_CODE 50000
#define CANNOT_SWITCH_TO_THREAD 50500
#define SWITCH_SET_TO_DIFFERENT_DOMAIN 50510
#define SWITCH_SET_TO_DIFFERENT_ADDRESS 50515
#define SWITCH_ALREADY_SET 50520
#define SWITCH_DOES_NOT_EXIST 50530
#define REFERENCE_ALREADY_SET 50620
#define REFERENCE_DOES_NOT_EXIST 50630
#define MULTI_THREAD_CONTENTION 60000
#define OUT_OF_ROOM_FOR_MORE_NAMES 100000
#define DIFFICULTY_HASHING 70000
#define BATCH_SIZE_EXCEEDED 100010
#define BATCH_NAME_LENGTH_EXCEEDED 100012
#define AUTHORIZED_AGENT_PROBLEM 500000

#define MAX_HASH_TABLE_SIZE 4097

#define MO_Space 100000

#define SWITCH_ADDRESS 020
#define GCB_IT_IS 0200

#define ROOT "root"
#define SOE_DOMAIN "Sombrero Operating Environment"
#define UTD "Universal Trusted Domain (UTD)"

#define MAX_NAME_LENGTH 20
#define MAX_NAMES 10000
#define MAX_BATCH_SIZE 2500
#define MAX_ITLB_SIZE 48
#define MAX_DTLB_SIZE 64
#define MAX_IRPLB_SIZE 1024
#define MAX_DRPLB_SIZE 1024
#define MAX_SRPLB_SIZE 1024
#define MAX_FILE_NAME 60
#define MAX_INPUT_LINE 1000
#define THREAD1_INPUT "MAIN.IN"
#define THREAD1_OUTPUT "MAIN.OUT"
#define HI_VALUE 0X7FFFFFFF

#define PAGE_SIZE_BITS 13 /* Page size: 8K */
#define BELL 07

#define EXECUTE_ACCESS 01 /* 1: execute access */
#define WRITE_ACCESS 02 /* 1: write access */
#define READ_ACCESS 04 /* 1: read access */
#define EXECUTE_REGION 010 /* 1: execute rcb */
#define WRITE_REGION 020 /* 1: write rcb */
#define READ_REGION 040 /* 1: read rcb */
#define ACCESS_OVERRIDE 0200 /* 1: AD determines access privileges */
#define VALID 0200 /* 1: valid -- Used for RPLB and TLB purposes */
#define ODD_ADDRESS 01 /* Used in the PRAL search */

/* The definition of VA_Size_Int should be changed when porting
   to a 64-bit architecture */
typedef unsigned long int VA_Size_Int; /* unsigned long is 4 bytes */

typedef long int VA_Size_Signed_Int;

typedef VA_Size_Int sys_error;

/* ?Will the pointer size be automatically 4 bytes or 8 bytes,
   as appropriate? */
typedef void * VA_Size_Pointer;

typedef struct range_tag {
```

```
  VA_Size_Pointer start_address;
  VA_Size_Pointer end_address;
} Range;

typedef struct region_tag {
  VA_Size_Int match_mask;
  VA_Size_Int dont_care_mask; /* The don't care bits should be stored
                                 as 0, the others as 1 */
} Region;

typedef struct name_tag {
  VA_Size_Pointer address;
  char name[MAX_NAME_LENGTH + 1];
} Name;

typedef struct drplb_tag {
  VA_Size_Pointer domain_pointer;
  Region region;
  char flags;
} DRPLB;

typedef struct irplb_tag {
  VA_Size_Pointer domain_pointer;
  Region region;
  char flags;
} IRPLB;

typedef struct srplb_tag {
  VA_Size_Pointer old_domain_pointer;
  VA_Size_Pointer new_domain_pointer;
  VA_Size_Pointer switch_address;
  char flags;
} SRPLB;

typedef struct page_entry_tag {
  VA_Size_Pointer table_or_frame_pointer;
  char flags;
} Page_Entry;

typedef struct tlb_tag {
  VA_Size_Pointer domain_pointer;
  char flags;
  VA_Size_Int tag;
  VA_Size_Pointer physical_address;
} TLB;

typedef enum Boolean {
  NO,
  YES
};

extern Name registry[]; /* Global variable */
extern DRPLB drplb[]; /* Global variable */
extern IRPLB irplb[]; /* Global variable */
extern SRPLB srplb[]; /* Global variable */
extern TLB dtlb[]; /* Global variable */
extern TLB itlb[]; /* Global variable */
extern int drplb_insertion_index; /* Global variable */
extern int drplb_hits; /* Global variable */
extern int drplb_misses; /* Global variable */
extern int irplb_insertion_index; /* Global variable */
extern int irplb_hits; /* Global variable */
extern int irplb_misses; /* Global variable */
extern int srplb_insertion_index; /* Global variable */
extern int srplb_hits; /* Global variable */
extern int srplb_misses; /* Global variable */
extern int dtlb_insertion_index; /* Global variable */
extern int dtlb_hits; /* Global variable */
extern int dtlb_misses; /* Global variable */
extern int itlb_insertion_index; /* Global variable */
extern int itlb_hits; /* Global variable */
```

```
extern int itlb_misses; /* Global variable */
extern int drplb_accesses; /* Global variable */
extern int irplb_accesses; /* Global variable */
extern int dtlb_accesses; /* Global variable */
extern int itlb_accesses; /* Global variable */
extern int irplb_size; /* Global varibale */
extern int drplb_size; /* Global variable */
extern int srplb_size; /* Global variable */
extern CRITICAL_SECTION registry_cs; /* Global variable */
extern CRITICAL_SECTION drplb_cs; /* Global variable */
extern CRITICAL_SECTION irplb_cs; /* Global variable */
extern CRITICAL_SECTION srplb_cs; /* Global variable */
extern CRITICAL_SECTION dtlb_cs; /* Global variable */
extern CRITICAL_SECTION itlb_cs; /* Global variable */
extern VA_Size_Int tls_current_gcb; /* Global variable */
extern VA_Size_Int tls_current_tcb; /* Global variable */
extern Page_Entry *page_directory_pointer; /* Global variable */
extern LARGE_INTEGER counter_frequency; /* Global variable */
extern LARGE_INTEGER dtlb_miss_min_time; /* Global variable */
extern LARGE_INTEGER dtlb_miss_max_time; /* Global variable */
extern LARGE_INTEGER dtlb_miss_total_time; /* Global variable */
extern LARGE_INTEGER itlb_miss_min_time; /* Global variable */
extern LARGE_INTEGER itlb_miss_max_time; /* Global variable */
extern LARGE_INTEGER itlb_miss_total_time; /* Global variable */
extern LARGE_INTEGER drplb_miss_min_time; /* Global variable */
extern LARGE_INTEGER drplb_miss_max_time; /* Global variable */
extern LARGE_INTEGER drplb_miss_total_time; /* Global variable */
extern LARGE_INTEGER irplb_miss_min_time; /* Global variable */
extern LARGE_INTEGER irplb_miss_max_time; /* Global variable */
extern LARGE_INTEGER irplb_miss_total_time; /* Global variable */
extern int total_number_of_windows_threads; /* Global variable */
extern VA_Size_Pointer windows_thread_pointer[MAX_BATCH_SIZE + 1]; /*
                                                 Global variable */
extern VA_Size_Pointer windows_thread_pointers[MAX_BATCH_SIZE + 1];
                                                 /* Global variable */
extern VA_Size_Pointer windows_thread_input[MAX_BATCH_SIZE + 1][4];
                                                 /* Global variable */
extern char windows_thread_input_file_names[MAX_BATCH_SIZE + 1]
                                          [MAX_NAME_LENGTH + 1];
                                                 /* Global variable */
extern FILE *windows_thread_output_file_pointers[MAX_BATCH_SIZE + 1];
                                                 /* Global variable */
extern char search_algorithm; /* Global variable */

extern VA_Size_Int normal_chain_limit; /* Global variable */

extern Boolean not_last_used_algorithm; /* Global variable */

extern VA_Size_Pointer chain_heap_pointer; /* Global variable */

extern FILE *stats_file; /* Global variable */
extern int column_groups; /* Global variable */
extern int current_column_group; /* Global variable */
extern Boolean print_number_of_threads_flag; /* Global variable */
extern Boolean print_number_of_memory_objects_flag; /* Global variable */
extern Boolean print_rplb_size_flag; /* Global variable */
extern Boolean print_miss_rate_flag; /* Global variable */
extern Boolean print_miss_penalty_flag; /* Global variable */
extern Boolean print_miss_rate_times_miss_penalty_flag; /* Global variable */

extern int quantum; /* Global variable */

#endif
```

CB.H

This file consists of 2 sections: type definitions (e.g., control blocks) specific to Sombrero, and the signatures (“prototypes”) of the functions implemented.

```
/* cb.h
 * SAK 11/20/97
 */

#ifndef _CB_H
#define _CB_H

#include "global.h"

typedef struct ad_tag *AD_Pointer;
typedef struct gcb_tag *GCB_Pointer;
typedef struct mocb_tag *MOCB_Pointer;
typedef struct pcb_tag *PCB_Pointer;
typedef struct rcb_tag *RCB_Pointer;
typedef struct session_tag *Session_Pointer;
typedef struct tcb_tag *TCB_Pointer;
typedef struct tt_tag *TT_Pointer;

typedef struct bsearch_element_tag *BSearch_Element_Pointer;
typedef struct hchain_info_tag *HChain_Info_Pointer;
typedef struct hbucket_tag *HBucket_Pointer;

typedef struct rcb_pointers_tag {
  RCB_Pointer execute_rcb_list;
  RCB_Pointer write_rcb_list;
  RCB_Pointer read_rcb_list;
} RCB_Pointers;

/* "Override" function definition follow */

typedef sys_error (*Gateway_Function) (char function_name[],...);

/* ?Should there be 2 versions of AD?  One for MOCB lists, another
   for GPD (switch) lists? */
typedef struct ad_tag {
/* The Access Descriptor links the UPD's and MO's and
   specifies UPD memory access and switch privileges */

  AD_Pointer pal_link; /* The protection access list is the set
                          of access rights available to the
                          pal_owner */
  AD_Pointer racl_link; /* Links UPD's that can access a given
                           MO or UPD's that can switch to a given
                           GPD; allows access to be revoked by
                           racl_owner */

  VA_Size_Pointer pal_owner; /* Name of the GPD or TPD which owns
                                the access list containing this
                                access descriptor */

  VA_Size_Pointer racl_owner; /* Name of the MO or GPD which owns the
                                 resource access control list
                                 containing this access
                                 descriptor */

  union switch_or_rcb_overrides_tag {
    VA_Size_Pointer switch_address; /* For switch entries the switch is
                                       stored here */
    RCB_Pointers rcb_overrides;
  } switch_or_rcb_overrides;
```

```
  char flags; /* The following masks are used:
    EXECUTE_ACCESS -- 1: execute access
    WRITE_ACCESS -- 1: write access
    READ_ACCESS -- 1: read access
    EXECUTE_REGION -- 1: at least one execute rcb override
    WRITE_REGION -- 1: at least one write rcb override
    READ_REGION -- 1: at least one read rcb override
This is no longer used! RCB_OVERRIDES 0100 -- 1: rcb overrides,
                 was meant to select a union member
    ACCESS_OVERRIDE 0200 -- 1: AD determines access privileges
              */

/* The "range" from the MOCB is copied here to help improve the
   search performance. */
  Range range;

} AD; /* End of AD definition */

typedef struct bsearch_element_tag {
  VA_Size_Pointer start_address;
  AD_Pointer ad_pointer;
} BSearch_Element;

typedef struct hchain_info_tag {
  BSearch_Element_Pointer sorted_array;
  VA_Size_Int array_size;
} HChain_Info;

typedef struct hbucket_tag {
  VA_Size_Pointer bucket;
} HBucket;

typedef struct gcb_tag {
/* General Protection Domain Control Block of the Ubiquitous
   Protection Domain */

/* Note!  Leave "link" as the first field, as it is in other classes.
   Then a pointer to some other class may be used for traversal */

  GCB_Pointer link; /* Pointer to the next GCB */
  VA_Size_Int name; /* The name of the GPD */
  GCB_Pointer authorized_pd; /* The protection domain that can
                                update this GCB */
  PCB_Pointer owner_pcb; /* Owner PCB */
  char flags; /* Flags.  ?Which? */

/* "Authorized agent": */
  Gateway_Function gateway_function;

  AD_Pointer mocb_pal; /* Protection access list, the list of MOCB's
                          accessible to the GPD */
  AD_Pointer switch_pal; /* Protection access list, the list of
                            PD Switches accessible to the GPD */
  AD_Pointer racl; /* Resourse access control list, the list of
                      GCB's and TCB's which can switch to this
                      GPD */

  BSearch_Element_Pointer mocb_pal_bsearch_array; /* Used to quickly
                             locate the AD desired via binary search */

  VA_Size_Int mocb_pal_total_elements; /* Number of elements in
                                          "mocb_pal_bsearch_array" */

  VA_Size_Pointer mocb_pal_hash_structure; /* Used to quickly locate the
                                      the AD desired via hashing */

/* Should there be search structures for switches too? */

  TT_Pointer token_tracking; /* Pointer to the Token Traking
                                structure which manages the GCB
```

```
                                copy set if needed */

/* ?Is the following right? */
  VA_Size_Pointer heap_space; /* Pointer to the base address of the
                                 GCB heap space */
  CRITICAL_SECTION cs;

  VA_Size_Pointer memento;

} GCB, sys_domain; /* End of GCB definition */

typedef struct mocb_tag {
/* Memory Object Control Block */

/* Note!  Leave "link" as the first field, as it is in other classes.
   Then a pointer to some other class may be used for traversal */

  MOCB_Pointer link; /* Pointer to the list of other MO's which
                        have the same parent */
  VA_Size_Int name; /* Name of the MOCB */
  GCB_Pointer authorized_pd; /* The protection domain that can
                                update this MOCB */
  Range range; /* The beginning and ending addresses for a MO */
/* ?Is the following adequate?  Note that a region mask could be
   stored in "physical_size_range" */
  Range physical_size_range;
  AD_Pointer racl; /* The list of GCB's and TCB's which have
                      access to this MOCB */
  TT_Pointer token_tracking; /* Pointer to the Token Traking
                                structure which manages the MOCB
                                copy set if needed */
  PCB_Pointer owner_pcb; /* Owner PCB */
  char flags; /* The following masks are used:
    EXECUTE_ACCESS -- 1: MOCB is executable
    WRITE_ACCESS -- 1: MOCB is writeable
    READ_ACCESS -- 1: MOCB is readable
    EXECUTE_REGION -- 1: MOCB has at least one execute rcb
    WRITE_REGION -- 1: MOCB has at least one write rcb
    READ_REGION -- 1: MOCB has at least one read rcb */
  char tokens; /* The following masks are used:
    EXECUTE_ACCESS -- 1: MOCB has execute token
    WRITE_ACCESS -- 1: MOCB has write token
    READ_ACCESS -- 1: MOCB has read token */

  RCB_Pointers rcb_pointers; /* Read RCB list, write RCB list,
                                execute RCB list */
  MOCB_Pointer fission_list; /* Pointer to a list of MOCB's which
                                subdivide this MO */

  VA_Size_Pointer memento;

  Boolean memory_allocated; /* This is here for testing only. */

} MOCB, sys_memory; /* End of MOCB definition */

typedef struct pcb_tag {
/* Principal Control Block.  Represents the principal or user to
   whom or to which resources are allocated. */

/* Note!  Leave "link" as the first field, as it is in other classes.
   Then a pointer to some other class may be used for traversal */

  PCB_Pointer link; /* Pointer to the next PCB */
  VA_Size_Int name; /* Name of the principal (PCB). */
  GCB_Pointer authorized_pd; /* The protection domain that can
                                update this PCB */
  Session_Pointer session_list; /* A "session" indicates attached
                                   resources including output
                                   devices, Windows, etc. */
/* !Note that an extra "owner_link" field in the corresponding CB's
   is required to implement the following 4 lists! */
```

```
  PCB_Pointer pcb_owner_list; /* The PCB owns other PCB's */
  GCB_Pointer gcb_owner_list; /* The PCB owns GCB's */
  TCB_Pointer tcb_owner_list; /* The PCB owns TCB's */
  MOCB_Pointer mocb_owner_list; /* The PCB owns MOCB's */

  VA_Size_Pointer memento;

} PCB, sys_principal; /* End of PCB definition */

typedef struct rcb_tag {
/* Region Control Block.  Describes access to a MO region using
   region masks */

/* Note!  Leave "link" as the first field, as it is in other classes.
   Then a pointer to some other class may be used for traversal */

  RCB_Pointer link; /* Pointer to the next RCB */
  Region region; /* The region in the MO affected by this RCB */

/* The following was moved to the MOCB */
/*  char token; */

} RCB; /* End of RCB definition */

typedef struct rmocb_tag {
/* Root MOCB.  Provides the basic memory allocation range for a
   computer node */

/* Note!  Leave "link" as the first field, as it is in other classes.
   Then a pointer to some other class may be used for traversal */
  MOCB_Pointer link; /* Pointer to the next Root MOCB allocated
                        to this node, if any */
  Range range; /* The upper and lower boundaries to memory allocation
                  at this logical node */
/* Heap Pointers -- The heap management structures which are used to
   manage the various control block heaps */
  VA_Size_Pointer ad_heap_pointer;
  VA_Size_Pointer gcb_heap_pointer;
  VA_Size_Pointer mocb_heap_pointer;
  VA_Size_Pointer pcb_heap_pointer;
  VA_Size_Pointer tcb_heap_pointer;
  VA_Size_Pointer rcb_heap_pointer; /* Added for testing by Shah on 9/1/97.
                                       Is this the best place for it?  Is there
                                       need for it at all? */
  VA_Size_Pointer mo_heap_pointer; /* This is here for testing purposes
                                      only */
  GCB_Pointer gcb_list; /* Pointer to the GCB list */
  MOCB_Pointer mocb_list; /* Pointer to the MOCB list */
  TCB_Pointer tcb_list; /* Pointer to the TCB list */
  PCB_Pointer pcb_list; /* Pointer to the PCB list */
  MOCB_Pointer parent; /* Pointer to the parent of the Root
                          MOCB */
  CRITICAL_SECTION gcb_cs;
  CRITICAL_SECTION mocb_cs;
  CRITICAL_SECTION pcb_cs;
  CRITICAL_SECTION tcb_cs;

} RMOCB; /* End of RMOCB definition */

typedef struct session_tag {
/* A session indicates attached resources including output devices,
   Windows, etc. */

/* Note!  Leave "link" as the first field, as it is in other classes.
   Then a pointer to some other class may be used for traversal */
  Session_Pointer link; /* Pointer to the next "session" */

  VA_Size_Pointer memento;

/* ?What else goes in here? */
```

```
} Session; /* End of Session definition */

typedef struct tcb_tag {
/* Thread Control Block and Carrier Protection Domain control
   block of the Ubiquitous Protection Domain */

/* Note!  Leave "link" as the first field, as it is in other classes.
   Then a pointer to some other class may be used for traversal */
  TCB_Pointer link; /* Pointer to the next TCB */
  VA_Size_Int name; /* The name of the Thread Protection Domain */
  GCB_Pointer authorized_pd; /* The protection domain that can
                                update this TCB */
  PCB_Pointer owner_pcb; /* Owner PCB */
  char flags; /* Flags.  ?Which? */

/* Overridable methods: */
/*
  User_Exception_Handler user_exception_handler;  The user may register
                                                  a function here
*/

  AD_Pointer mocb_pal; /* Protection access list, the list of MOCB's
                          accessible to the TPD */
  AD_Pointer switch_pal; /* Protection access list, the list of
                            PD Switches accessible to the TPD */

  BSearch_Element_Pointer mocb_pal_bsearch_array; /* Used to quickly
                                        locate the AD desired */

  VA_Size_Int mocb_pal_total_elements; /* Number of elements in
                                          "mocb_pal_bsearch_array" */

  VA_Size_Pointer mocb_pal_hash_structure; /* Used to quickly locate the
                                              the AD desired via hashing */

/* Should there be search structures for switches too? */

  TT_Pointer token_tracking; /* Pointer to the Token Traking
                                structure which manages the TCB
                                copy set if needed */

/* ?Is the following right? */
  GCB_Pointer current_gcb; /* Thread is currently executing in
                              this GPD */

  VA_Size_Pointer exec_address;
  VA_Size_Pointer stack_address;
  Boolean wait_flag;

/* Where is the "state" (e.g., "suspended") stored? */

  VA_Size_Pointer cpu_state; /* CPU state for context switches */

  CRITICAL_SECTION cs;

  VA_Size_Pointer memento;

} TCB, sys_thread; /* End of TCB definition */

typedef struct tt_tag {
/* For Token Tracking */

/* Write Last -- Last known location of write token */
/* Write Cache -- Link to next node in write cache graph */
/* Inception Node -- Root node of copy set, usually determined
                     from the CB name */
/* Copy Set Parent -- Predecessor in copy set graph */
/* Copy Set Arity -- Copy set arity */
/* Copy Set -- Successors (arity) in copy set graph */

} TT; /* End of TT */
```

```
extern RMOCB rmocb; /* Global variable */

/* Function signatures ("prototypes") */

sys_error analyze_hash_structure(VA_Size_Pointer f_hs_pointer,
                                 VA_Size_Int &total_elements,
                                 VA_Size_Int &total_hash_tables);

sys_error batch_build_hash_structure(char *f_input_line, FILE *f_out_file,
                                     GCB_Pointer f_current_gcb,
                                     TCB_Pointer f_current_tcb);

sys_error batch_gpd_add(char *f_input_line, FILE *f_out_file,
                        GCB_Pointer f_current_gcb,
                        TCB_Pointer f_current_tcb);

sys_error batch_gpd_del(char *f_input_line, FILE *f_out_file,
                        GCB_Pointer f_current_gcb,
                        TCB_Pointer f_current_tcb);

sys_error batch_gpd_print(char *input_line, FILE *f_out_file,
                          GCB_Pointer f_current_gcb,
                          TCB_Pointer f_current_tcb);

sys_error batch_hash_structure_print(char *input_line, FILE *f_out_file,
                                     GCB_Pointer f_current_gcb,
                                     TCB_Pointer f_current_tcb);

sys_error batch_memory_accesses(char *input_line,
                                FILE *f_out_file,
                                GCB_Pointer f_current_gcb,
                                TCB_Pointer f_current_tcb,
                                int &f_number_of_threads);

sys_error batch_mo_add(char *f_input_line, FILE *f_out_file,
                       GCB_Pointer f_current_gcb,
                       TCB_Pointer f_current_tcb);

sys_error batch_mo_add_specific(char *input_line, FILE *f_out_file,
                                GCB_Pointer f_current_gcb,
                                TCB_Pointer f_current_tcb);

sys_error batch_mo_del(char *f_input_line,  FILE *f_out_file,
                       GCB_Pointer f_current_gcb,
                       TCB_Pointer f_current_tcb);

sys_error batch_mo_print(char *input_line, FILE *f_out_file,
                         GCB_Pointer f_current_gcb,
                         TCB_Pointer f_current_tcb);

/* sys_error batch_principal_add(char *f_input_line, FILE *f_out_file,
                                 GCB_Pointer f_current_gcb,
                                 TCB_Pointer f_current_tcb); */

/* sys_error batch_principal_print(char *input_line, FILE *f_out_file,
                                   GCB_Pointer f_current_gcb,
                                   TCB_Pointer f_current_tcb); */

sys_error batch_reference(char *input_line, FILE *f_out_file,
                          GCB_Pointer f_current_gcb,
                          TCB_Pointer f_current_tcb);

sys_error batch_switch(char *input_line, FILE *f_out_file,
                       GCB_Pointer f_current_gcb,
                       TCB_Pointer f_current_tcb);

sys_error batch_thread_add(char *f_input_line, FILE *f_out_file,
                           GCB_Pointer f_current_gcb,
                           TCB_Pointer f_current_tcb);
```

```
sys_error batch_thread_del(char *f_input_line, FILE *f_out_file,
                           GCB_Pointer f_current_gcb,
                           TCB_Pointer f_current_tcb);

sys_error batch_thread_print(char *input_line, FILE *f_out_file,
                             GCB_Pointer f_current_gcb,
                             TCB_Pointer f_current_tcb);

sys_error batch_windows_thread_add(char *f_input_line,
                                   FILE *f_out_file,
                                   GCB_Pointer f_current_gcb,
                                   TCB_Pointer f_current_tcb);

sys_error build_page_table(VA_Size_Pointer f_start_address,
                           VA_Size_Pointer f_end_address);

sys_error change_current_thread(TCB_Pointer old_thread_pointer,
                                TCB_Pointer new_thread_pointer);

void change_insertion_index(int &f_index, int f_maximum);

sys_error check_execute_access(VA_Size_Pointer f_va_pointer);

sys_error check_execute_access_via_rplb(VA_Size_Pointer f_va_pointer);

sys_error check_execute_access_via_tlb(VA_Size_Pointer f_va_pointer,
                                       Page_Entry &f_entry);

sys_error check_read_or_write_access_via_rplb(VA_Size_Pointer f_va_pointer,

char f_desired_access);

sys_error check_read_or_write_access_via_tlb(VA_Size_Pointer f_va_pointer,
                                             char f_access_desired,
                                             Page_Entry &f_entry);

sys_error check_switch_access_via_rplb(VA_Size_Pointer f_va_pointer,
                                       char f_access_desired,
                                       GCB_Pointer f_gcb_pointer,
                                       TCB_Pointer f_tcb_pointer);

int compare_bsearch_elements(const void *elem1,
                             const void *elem2);

sys_error default_agent(char function_name[],...);

sys_error delete_from_registry(VA_Size_Pointer f_va_pointer,
                               char f_name[], Boolean f_is_name);

sys_error delete_from_bsearch_array(AD_Pointer f_ad_pointer,
                                    VA_Size_Pointer f_domain_pointer,
                                    Boolean f_domain_is_gcb);

sys_error delete_hash_structure(VA_Size_Pointer f_hs_pointer);

sys_error execute(char function_name[],...);

sys_error free_ad(AD_Pointer f_ad_pointer);

sys_error free_gcb(GCB_Pointer f_gcb_pointer);

sys_error free_mo(MOCB_Pointer f_mocb_pointer);

sys_error free_rcbs(RCB_Pointer f_rcb_pointer);

sys_error free_tcb(TCB_Pointer f_tcb_pointer);

GCB_Pointer gcb_declare(PCB_Pointer f_owner_pcb,
                        GCB_Pointer f_authorized_pd,
                        sys_error &f_err);
```

```
VA_Size_Pointer get_address_from_name(char f_name[], sys_error &f_err);

sys_error get_addresses_from_names(char f_name[],
                                   VA_Size_Pointer f_va_pointers[],
                                   int &f_number_of_pointers);

sys_error get_input_file_names(char f_file_name[],char f_object_file_name[],
                               char f_access_file_name[]);

sys_error get_file_names(char f_name[], int f_instances_requested,
                         char f_names[MAX_BATCH_SIZE + 1][MAX_NAME_LENGTH + 1],
                         int &f_number_of_files);

void get_name_from_address(VA_Size_Pointer f_va_pointer,
                           char f_name[],
                           sys_error &f_err);

sys_error get_names(char f_name[], int f_instances_requested,
                    int f_last_version_number,
                    char f_names[MAX_BATCH_SIZE + 1][MAX_NAME_LENGTH + 1]);

sys_error get_pointers(char f_class[], char f_name[],
                       int f_number_requested,
                       GCB_Pointer f_current_gcb,
                       TCB_Pointer f_current_tcb,
                       VA_Size_Pointer f_va_pointers[],
                       int &f_number_of_pointers);

sys_error initialize();

sys_error insert_into_bsearch_array(AD_Pointer f_ad_pointer,
                                    VA_Size_Pointer f_domain_pointer,
                                    Boolean f_domain_is_gcb);

int ltrim(char s[]);

MOCB_Pointer mocb_alloc(VA_Size_Int f_wanted_size,
                        VA_Size_Signed_Int f_needed_size,
                        VA_Size_Pointer f_domain_to_attach_to,
                        sys_error &f_err);

MOCB_Pointer mocb_alloc_internal(VA_Size_Pointer f_start_address,
                                 VA_Size_Int f_wanted_size,
                                 VA_Size_Signed_Int f_needed_size,
                                 char f_flags,
                                 VA_Size_Pointer f_domain_to_attach_to,
                                 sys_error &f_err,
                                 Boolean f_specific);

MOCB_Pointer mocb_alloc_specific(VA_Size_Pointer f_start_address,
                                 VA_Size_Pointer f_end_address,
                                 VA_Size_Signed_Int f_needed_size,
                                 char f_flags, VA_Size_Pointer f_domain_to_attach_to,
                                 sys_error &f_err);

AD_Pointer new_ad(sys_error &f_err);

sys_error parse_input(char f_input_file_name[],
                      FILE *f_out_file,
                      GCB_Pointer f_current_gcb,
                      TCB_Pointer f_current_tcb,
                      Boolean f_echo_input);

PCB_Pointer pcb_create(GCB_Pointer f_authorized_pd, sys_error &f_err);

sys_error perform_memory_accesses(FILE* out_file, FILE *f_access_file,
                                  int f_number_of_accesses, char f_buffer);

void print_ad_info(FILE *f_out_file, AD_Pointer f_ad_pointer,
                   Boolean f_switch);
```

```
void print_permission_flags(FILE* f_out_file, char f_flags);

sys_error print_gcb(FILE *f_out_file, GCB_Pointer f_gcb_pointer);

sys_error print_gcb_list(FILE *f_out_file);

sys_error print_hash_structure(FILE *f_out_file,
                               VA_Size_Pointer f_hs_pointer,
                               int f_level);

void print_miss_penalty(FILE *f_out_file, LARGE_INTEGER f_time,
                        int f_misses);

void print_miss_rate(FILE *f_out_file, int f_misses, int f_accesses);

void print_miss_rate_times_miss_penalty(FILE *f_out_file,
                                        int f_misses, int f_accesses,
                                        LARGE_INTEGER f_time);

sys_error print_mocb(FILE *f_out_file, MOCB_Pointer f_mocb_pointer);

sys_error print_mocb_list(FILE *f_out_file);

sys_error print_name(FILE *f_out_file, VA_Size_Pointer va_pointer);

sys_error print_pal(FILE *f_out_file, AD_Pointer f_ad_pointer,
                    Boolean f_switch);

/* sys_error print_pcb(PCB_Pointer f_pcb_pointer);

sys_error print_pcb_list(); */

sys_error print_racl(FILE *f_out_file, AD_Pointer f_ad_pointer,
                     Boolean f_switch);

void print_region(FILE *f_out_file, RCB_Pointer f_rcb_pointer,
                  char f_indentation[]);

void print_registry(FILE *f_out_file);

/* sys_error print_rplb_miss_parameters(
                     char *f_message,
                     GCB_Pointer f_current_gcb_pointer,
                     TCB_Pointer f_current_tcb_pointer,
                     VA_Size_Int f_attempted_va,
                     char f_attempted_access_type,
                     VA_Size_Pointer f_kernel_structure_pointer); */

void print_rplb_stats(FILE *f_out_file, int number_of_threads);

void print_time(FILE *f_out_file, LARGE_INTEGER f_time);

sys_error print_tcb(FILE *f_out_file, TCB_Pointer f_tcb_pointer);

sys_error print_tcb_list(FILE *f_out_file);

void print_tlb_stats(FILE *f_out_file, int number_of_threads);

void rangeToRangeMask(VA_Size_Int *matchMask, VA_Size_Int *dontCareMask,
                      VA_Size_Int rangeFrom, VA_Size_Int rangeTo,
                      VA_Size_Int address);

sys_error read(VA_Size_Pointer f_va_pointer, char f_message[]);

sys_error rebuild_hash_structure(VA_Size_Pointer f_domain_pointer,
                                 Boolean f_domain_is_gcb);

void register_name(VA_Size_Pointer f_va_pointer,
                   char f_name[], sys_error &f_err);

void rplb_clear();
```

```
sys_error rplb_miss_handler(GCB_Pointer f_gcb_pointer,
                            TCB_Pointer f_tcb_pointer,
                            VA_Size_Pointer f_va,
                            char f_access_desired,
                            Region &f_entry,
                            char &f_entry_type,
                            VA_Size_Pointer &f_cb_pointer);

void rtrim(char *f_input_string);

sys_domain *sd_declare(sys_error &f_err);

AD_Pointer search_ad_ranges_hash_structure(VA_Size_Pointer f_domain_pointer,
                                           Boolean f_domain_is_gcb,
                                           VA_Size_Pointer f_va);

AD_Pointer search_ad_ranges_linked_list(AD_Pointer f_mocb_pal,
                                        VA_Size_Pointer f_va);


AD_Pointer search_ad_ranges_sorted_array(VA_Size_Pointer f_domain_pointer,
                                         Boolean f_domain_is_gcb,
                                         VA_Size_Pointer f_va);

AD_Pointer search_ad_switches(AD_Pointer f_switch_pal,
                              VA_Size_Pointer f_va);

AD_Pointer search_hash_structure(VA_Size_Pointer f_hs_pointer,
                                 VA_Size_Pointer f_va);

sys_error search_mocb_ranges(MOCB_Pointer mocb_list,
                             VA_Size_Int attempted_va,
                             MOCB_Pointer &mocb_pointer);

sys_error search_mocb_pal(VA_Size_Pointer f_domain_pointer,
                          Boolean f_domain_is_gcb,
                          VA_Size_Pointer f_va,
                          char f_access_desired,
                          Region &f_entry,
                          char &f_entry_type,
                          VA_Size_Pointer &f_cb_pointer);

sys_error search_switch_pal(AD_Pointer f_switch_pal,
                            VA_Size_Pointer f_va,
                            char f_access_desired,
                            Region &f_entry,
                            char &f_entry_type,
                            VA_Size_Pointer &f_cb_pointer);

sys_error set_reference_internal(MOCB_Pointer f_mocb_pointer,
                                 VA_Size_Pointer f_domain_pointer,
                                 Boolean f_mode,char f_access_type,
                                 Boolean &mo_freed);

sys_error set_switch_internal(VA_Size_Pointer f_active_domain,
                              VA_Size_Pointer f_new_domain,
                              VA_Size_Pointer f_address,
                              Boolean f_mode,
                              char f_access_type,
                              Boolean f_any_switch);

VA_Size_Int size_bits(VA_Size_Pointer f_start_address,
                      VA_Size_Pointer f_end_address);

sys_memory *sm_alloc(VA_Size_Int f_wanted_size,
                     VA_Size_Signed_Int f_needed_size,
                     VA_Size_Pointer f_domain_to_attach_to,
                     sys_error &f_err);

VA_Size_Pointer sm_data(MOCB_Pointer f_mocb_pointer, sys_error &f_err);
```

```
void sort_drplb(int f_index);

void sort_irplb(int f_index);

void sort_srplb(int f_index);

void sort_tlb(TLB tlb[], int f_index);

sys_principal *sp_create(sys_error &f_err);

sys_thread *st_create(VA_Size_Pointer f_exec_address,
                      VA_Size_Pointer f_stack_address,
                      Boolean f_wait_flag,
                      sys_error &f_err);

void st_transfer(sys_thread *f_thread, sys_principal *f_new_owner,
                 sys_error &f_err);

TCB_Pointer tcb_create(VA_Size_Pointer f_exec_address,
                       VA_Size_Pointer f_stack_address,
                       Boolean f_wait_flag,
                       PCB_Pointer f_owner_pcb,
                       GCB_Pointer f_authorized_pd,
                       GCB_Pointer f_current_gcb,
                       sys_error &f_err);

sys_error tcb_transfer(TCB_Pointer f_thread, PCB_Pointer f_new_owner);


/* sys_error thread2(VA_Size_Pointer f_current_domains[]);

sys_error thread3(VA_Size_Pointer f_current_domains[]); */

void tlb_clear();

sys_error tlb_miss_handler(VA_Size_Pointer f_va_pointer,
                           char f_access_desired,
                           Page_Entry &f_entry);

sys_error traverse_hash_structure(VA_Size_Pointer f_hs_pointer);

sys_error update_mocb_pal_search_structures(AD_Pointer f_ad_pointer,
                                            VA_Size_Pointer f_domain_pointer,
                                            Boolean f_domain_is_gcb,
                                            Boolean f_insert);

sys_error windows_thread(VA_Size_Pointer f_input[]);

sys_error write(VA_Size_Pointer f_va_pointer, char f_message[]);

#endif
```

MAIN.CPP

INIT.CPP

BATCH.CPP

SYSCALLS.CPP

NAMES.CPP

TLB.CPP

RPLB.CPP

REGION.CPP

MISC.CPP

All of the above source code was developed in the Microsoft Visual C++ V5.0 environment. Additionally, the following auxiliary programs were developed, as alluded to in section 4.4:

CONVERT.C

The memory access traces obtained from the Internet were in hexadecimal format. It was necessary to identify memory objects from them. To do that, the memory accesses needed to be sorted. This program converts hexadecimal numbers to decimal format, and it moves the leading "0," "1," or "2," which denote "read," "write" or "execute" access, to the end. The output of this command can then be fed into the UNIX "sort" utility easily.

MO.C

This program takes the desired "minimum distance" between memory objects and a file generated by the aforementioned CONVERT program as input parameters. It creates a file with unaligned memory objects of different sizes.

ALIGNED_MO.C

This program takes the desired memory object size and a file generated by the aforementioned CONVERT program as input parameters. It creates a file with aligned memory objects, all of the same size.

All of the above source files are installed in the SOMBRERO1 (Alpha) machine in the ASU OS Lab (in GWC-321). Also, the files containing the cache access traces (gcc.rwx, spice.rwx and tex.rwx) were copied to this machine. These files, and all the others mentioned in this thesis, are in the D:\SHAH\PRAL directory.

Furthermore, several files with memory objects corresponding to the trace files above were copied to this machine. For instance:

```
gcc_1024.129
gcc_2e20.3
spice_8192.14
tex_2e13.14
tex_2e20.4
```

There is a method to the naming of the memory object files produced by the programs mentioned above. The alphabetical characters in the file name denote the benchmark program to which the memory objects belong. (For instance, cache accesses in gcc.rwx should only be used against gcc_* memory objects.) If there is a “2e” in the name following the underscore, it means that the file contains constant-sized memory objects. Furthermore, 2 to the power of the number following “2e” indicates the size of the memory objects in that file. If there is no “2e” in the file name, the number following the underscore denotes the minimum “gap” between memory objects. Finally, the number following the period indicates the number of objects in the file.

Note that memory objects generated from the different benchmark programs (Gcc, Spice and TeX) can overlap. That would not occur in a real Sombrero system, where each byte is assigned a unique address.

Appendix C

Test Inputs and Outputs

C. Test Inputs and Outputs

Some of the command files that were used to generate the RPLB/PRAL versus TLB/page tables chart data are shown below.

MAIN.IN

```
pral_search: binary_search_array
irplb_size: 48
drplb_size: 64
rplb_evictions: NLU
b_stats_file_open: rplb_vs_tlb.out
b_stats_column_groups: 1
#ECHO   DRPLB MISS RATE        DRPLB MISS PENALTY     DRPLB COST     DTLB MISS RATE DTLB
MISS PENALTY   DTLB COST      IRPLB MISS RATE        IRPLB MISS PENALTY     IRPLB COST
        ITLB MISS RATE ITLB MISS PENALTY      ITLB COST

b_stats_category: NUMBER_OF_THREADS
b_stats_print_miss_rate: yes
b_stats_print_miss_penalty: yes
b_stats_print_rate_times_penalty: yes

gpd_add: 1 owner_gpd_for_gcc
gpd_add: 1 owner_gpd_for_spice
gpd_add: 1 owner_gpd_for_tex

#ECHO ---------------- gcc_2e20.3
#include gcc_2e20.3
#include vary_threads_b.in
mo_del: gcc*

#ECHO ---------------- gcc_2e16.15
#include gcc_2e16.15
#include vary_threads_b.in
mo_del: gcc*

#ECHO ---------------- gcc_2e15.28
#include gcc_2e15.28
#include vary_threads_b.in
mo_del: gcc*

#ECHO ---------------- gcc_2e14.53
#include gcc_2e14.53
#include vary_threads_b.in
mo_del: gcc*

#ECHO ---------------- gcc_2e13.101
#include gcc_2e13.101
#include vary_threads_b.in
mo_del: gcc*

#ECHO ---------------- gcc_2e12.177
#include gcc_2e12.177
#include vary_threads_b.in
mo_del: gcc*

#ECHO ---------------- gcc_2e7.2461
#include gcc_2e7.2461
#include vary_threads_b.in
mo_del: gcc*

#exit
```

GCC_2E20.3

```
#echo_off

mo_add_specific:     400000     4fffff 0 X owner_gpd_for_gcc gcc_2e20.1
mo_add_specific:   10000000   100fffff 0 RW owner_gpd_for_gcc gcc_2e20.2
mo_add_specific:   7ff00000   7fffffff 0 RW owner_gpd_for_gcc gcc_2e20.3

! Total memory objects: 3
! Memory object size (2 to the power of): 20
! Total memory accesses: 1000002
```

VARY_THREADS_B.IN

```
! (Vary the number of threads from 1 to 80.)

gpd_add: 1 gpd_for_gcc*
thread_add: 1 gpd_for_gcc* thread_for_gcc*
#include rplb_vs_tlb.in

gpd_add: 16 gpd_for_gcc*
thread_add: 16 gpd_for_gcc* thread_for_gcc*
#include rplb_vs_tlb.in

gpd_add: 32 gpd_for_gcc*
thread_add: 32 gpd_for_gcc* thread_for_gcc*
#include rplb_vs_tlb.in

gpd_add: 48 gpd_for_gcc*
thread_add: 48 gpd_for_gcc* thread_for_gcc*
#include rplb_vs_tlb.in

gpd_add: 64 gpd_for_gcc*
thread_add: 64 gpd_for_gcc* thread_for_gcc*
#include rplb_vs_tlb.in

gpd_add: 80 gpd_for_gcc*
thread_add: 80 gpd_for_gcc* thread_for_gcc*
#include rplb_vs_tlb.in

#exit
```

RPLB_VS_TLB.IN

```
reference_set: gcc* gpd_for_gcc* .

rplb_clear:
tlb_clear:
rplb_and_tlb_accesses: thread_for_gcc* gcc.rwx

thread_del: thread_for_gcc*
gpd_del: gpd_for_gcc*
```

The output files produced by the above command files follow.

MAIN.OUT

```
Current time: Tue Dec 02 08:39:19 1997

> PRAL_SEARCH: BINARY_SEARCH_ARRAY

> IRPLB_SIZE: 48

> DRPLB_SIZE: 64

> RPLB_EVICTIONS: NLU

> B_STATS_FILE_OPEN: RPLB_VS_TLB.OUT

> B_STATS_COLUMN_GROUPS: 1

> #ECHO DRPLB MISS RATE        DRPLB MISS PENALTY     DRPLB COST     DTLB MISS RATE DTLB
MISS PENALTY   DTLB COST      IRPLB MISS RATE        IRPLB MISS PENALTY     IRPLB COST
        ITLB MISS RATE ITLB MISS PENALTY      ITLB COST

> B_STATS_CATEGORY: NUMBER_OF_THREADS

> B_STATS_PRINT_MISS_RATE: YES

> B_STATS_PRINT_MISS_PENALTY: YES

> B_STATS_PRINT_RATE_TIMES_PENALTY: YES

> GPD_ADD: 1 OWNER_GPD_FOR_GCC

> GPD_ADD: 1 OWNER_GPD_FOR_SPICE

> GPD_ADD: 1 OWNER_GPD_FOR_TEX

> #ECHO ---------------- GCC_2E20.3

> #INCLUDE GCC_2E20.3

> #ECHO_OFF

> #INCLUDE VARY_THREADS_B.IN

> GPD_ADD: 1 GPD_FOR_GCC*

> THREAD_ADD: 1 GPD_FOR_GCC* THREAD_FOR_GCC*

> #INCLUDE RPLB_VS_TLB.IN

> REFERENCE_SET: GCC* GPD_FOR_GCC* .

> RPLB_CLEAR:

> TLB_CLEAR:

> RPLB_AND_TLB_ACCESSES: THREAD_FOR_GCC* GCC.RWX

> THREAD_DEL: THREAD_FOR_GCC*

> GPD_DEL: GPD_FOR_GCC*

> GPD_ADD: 16 GPD_FOR_GCC*

> THREAD_ADD: 16 GPD_FOR_GCC* THREAD_FOR_GCC*

> #INCLUDE RPLB_VS_TLB.IN

> REFERENCE_SET: GCC* GPD_FOR_GCC* .

> RPLB_CLEAR:
```

```
> TLB_CLEAR:
> RPLB_AND_TLB_ACCESSES: THREAD_FOR_GCC* GCC.RWX
> THREAD_DEL: THREAD_FOR_GCC*
> GPD_DEL: GPD_FOR_GCC*
> GPD_ADD: 32 GPD_FOR_GCC*
> THREAD_ADD: 32 GPD_FOR_GCC* THREAD_FOR_GCC*
> #INCLUDE RPLB_VS_TLB.IN
> REFERENCE_SET: GCC* GPD_FOR_GCC* .
> RPLB_CLEAR:
> TLB_CLEAR:
> RPLB_AND_TLB_ACCESSES: THREAD_FOR_GCC* GCC.RWX
> THREAD_DEL: THREAD_FOR_GCC*
> GPD_DEL: GPD_FOR_GCC*
> GPD_ADD: 48 GPD_FOR_GCC*
> THREAD_ADD: 48 GPD_FOR_GCC* THREAD_FOR_GCC*
> #INCLUDE RPLB_VS_TLB.IN
> REFERENCE_SET: GCC* GPD_FOR_GCC* .
> RPLB_CLEAR:
> TLB_CLEAR:
> RPLB_AND_TLB_ACCESSES: THREAD_FOR_GCC* GCC.RWX
> THREAD_DEL: THREAD_FOR_GCC*
> GPD_DEL: GPD_FOR_GCC*
> GPD_ADD: 64 GPD_FOR_GCC*
> THREAD_ADD: 64 GPD_FOR_GCC* THREAD_FOR_GCC*
> #INCLUDE RPLB_VS_TLB.IN
> REFERENCE_SET: GCC* GPD_FOR_GCC* .
> RPLB_CLEAR:
> TLB_CLEAR:
> RPLB_AND_TLB_ACCESSES: THREAD_FOR_GCC* GCC.RWX
> THREAD_DEL: THREAD_FOR_GCC*
> GPD_DEL: GPD_FOR_GCC*
> GPD_ADD: 80 GPD_FOR_GCC*
> THREAD_ADD: 80 GPD_FOR_GCC* THREAD_FOR_GCC*
> #INCLUDE RPLB_VS_TLB.IN
> REFERENCE_SET: GCC* GPD_FOR_GCC* .
> RPLB_CLEAR:
```

```
> TLB_CLEAR:

> RPLB_AND_TLB_ACCESSES: THREAD_FOR_GCC* GCC.RWX

> THREAD_DEL: THREAD_FOR_GCC*

> GPD_DEL: GPD_FOR_GCC*

> #EXIT

> MO_DEL: GCC*

> #ECHO ---------------- GCC_2E16.15

[…]

> #ECHO ---------------- GCC_2E15.28

[…]

> #ECHO ---------------- GCC_2E14.53

[…]

> #ECHO ---------------- GCC_2E13.101

[…]

> #ECHO ---------------- GCC_2E12.177

[…]

> #ECHO ---------------- GCC_2E7.2461

[…]

> #EXIT

Current time: Tue Dec 02 12:01:02 1997
```

RPLB_VS_TLB.OUT

(The lines below were wrapped because they were too long for the page width)

```
        DRPLB MISS RATE        DRPLB MISS PENALTY     DRPLB COST     DTLB MISS RATE DTLB
MISS PENALTY   DTLB COST      IRPLB MISS RATE        IRPLB MISS PENALTY     IRPLB COST
        ITLB MISS RATE ITLB MISS PENALTY      ITLB COST
 ---------------- GCC_2E20.3
1               0.0008                1.2972                0.0107                0.0140
                0.7746                0.1085                0.0001                1.3382
                0.0018                0.0125                0.7827                0.0982
16              0.0008                1.3150                0.0108                0.5311
                0.7867                4.1783                0.0001                1.4227
                0.0019                0.1254                0.7739                0.9703
32              0.0008                1.3242                0.0109                0.6103
                0.7852                4.7921                0.0001                1.4685
                0.0019                0.1412                0.7746                1.0937
48              0.0263                1.3692                0.3604                0.6341
                0.7860                4.9843                0.0001                1.4808
                0.0020                0.1464                0.7744                1.1335
64              0.0384                1.3521                0.5198                0.6434
                0.7859                5.0570                0.0030                1.4117
                0.0425                0.1482                0.7742                1.1470
80              0.0473                1.3438                0.6356                0.6553
                0.7864                5.1531                0.0050                1.4046
                0.0709                0.1503                0.7742                1.1633
```

```
 ---------------- GCC_2E16.15
[…]
 ---------------- GCC_2E15.28
[…]
 ---------------- GCC_2E14.53
[…]
 ---------------- GCC_2E13.101
[…]
 ---------------- GCC_2E12.177
[…]
 ---------------- GCC_2E7.2461
[…]
```

Appendix D

Aligned Memory Objects

D. Aligned Memory Objects

Aligned memory objects are those that can be represented with only one range mask. For example, a memory object that is 8 KB in size is aligned if it starts at a boundary with 13 trailing zeros. A memory object that were 8193 (8 K + 1) bytes in size would not be aligned, as it would require 2 range masks. However, 16 KB of virtual memory could be assigned to such a memory object (and it could be placed on a virtual address with 14 trailing zeros). Then only one range mask would be required to protect the object. Since aligned memory objects require less RPLB entries than non-aligned memory objects do, in Sombrero, virtual memory would be assigned to objects judiciously, that is, so as to make them aligned. With a 64-bit address space, there should be plenty of virtual address space available to make that possible.

In other words, an aligned object is one whose size is a power of 2 and whose starting address has a number of trailing binary zeros that is equal to the base-2 log of the object's size. Examples:

- A 4-byte memory object may have a starting address of 40904C, 409040, 0r 400000, but not 40904E. The base-2 log of 4, the object's size, is 2 so there must be at least two trailing binary zeros in its starting address.
- A 4-kilobyte memory object can begin at F39000, but not at any address where the last three hexadecimal digits are non-zero because the base-2 log of 4,096 is 12.

Note that an unaligned memory object may require up to one RPLB entry *per address bit*.

Appendix E

Hashing Algorithm and Implementation

E. Hashing Algorithm and Implementation

After the number of trailing bits that need to be removed is calculated, a temporary array is allocated and the starting addresses from the sorted array are “normalized” and copied to it. Next, a temporary array is traversed to determine the number of unique entries in it. This number is used as the hash table size and an array of this size is allocated for the hash table. Then, the next group of identical entries from the temporary array is taken and the hashing function is used to determine which bucket the group should go into. Next, the hash structure-building routine is called recursively, passing it the entries from the original sorted array corresponding to the said group. Finally, the output of this recursive call is assigned to the bucket computed above.

Hashing is attempted only if the number of entries in the sorted array is more than 4 (i.e., the normal “chain” size is 4 or less). Otherwise, the sorted array is attached to a bucket (or a domain) without further ado.

Whenever there is a hashing conflict, the size of the hash table is increased and the hashing process is restarted. And, if the number of entries in the temporary array (obtained after “normalizing” addresses) is equal to the number of entries in the input array, the number of bits to be removed is increased and the hashing process is restarted. That may be adequate up to a certain point, but can’t continue forever. Presently, after a certain number of tries, “chaining” is resorted to. In other words, hashing of the whole input array is terminated and the entire array is attached to a bucket (or a domain) as is, without further ado. This aspect of the hashing implementation needs improvement.

Whether a bucket is pointing to a binary search array or a hash table can be determined by looking at the last bit of the pointer (address). (The last bit is normally 0, i.e., unused, as normally *double words* are addressed). If it is non-zero, it is another hash table.

According to [Sk097], the hashing algorithm could be refined so that it would handle some of the anomalies associated with variable-sized memory objects better. In particular, some of the entries of a

chain could be made to point to a hash structure. The trailing *2* bits of a pointer could be used to have more types of sub-structures in the overall hash structure.

Of course, this algorithm must not be carried to an extreme. Too many levels can lead to worse performance than the one attainable via a plain binary search of the entire access list.

Presently, a recursive algorithm is used to search the hash structure. This adds unacceptable overhead. Besides, it is unnecessary. Furthermore, the RPLB miss handling code will need to be converted to Alpha "PALcode," which may not support "C-style" recursion.

It may be possible to further improve the hash structure by using a different chain size (other than 4), or using a different hashing function. One could also experiment with having separate, more conveniently located chain arrays, instead of having pointers to the array generated for the binary search of the PRAL (as mentioned earlier). However, more arrays would require more memory, which could lead to more page faults.

Another improvement could be made by attempting hashing only if the number of *unique* entries in the sorted array in more than 4.

The current implementation does not include logic to insert/delete an entry into/from the hash structure, so the whole hash structure must be deleted and rebuilt when there is a pertinent change in the access list of the domain in question. The test environment developed (see section 4.6) allows the user to request that the hash structure for a domain be rebuilt. Furthermore, the user can try the different PRAL search algorithms and compare how they perform. For development purposes, the test environment also allows the user to print details about a hash structure.

So, with the recursion, the exceptions and other overhead attendant to the hashing algorithm and implementation, the plain binary search of the PRAL is giving better results than hashing. And, since the number of memory objects attached to a domain should be low, a binary search of the PRAL may be adequate.