THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-04-02

`Solemn`: Solaris Emulation Mode for Sparc Sulima

Bill Clarke

February 2004

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

Solemn: Solaris Emulation Mode for Sparc Sulima[*]

Bill Clarke
Bill.Clarke@anu.edu.au
CC-NUMA Project, Department of Computer Science
Australian National University
Acton ACT 0200, Australia

February 10, 2004

Abstract

In this paper we present Solemn, a new user-level simulation mode for Sparc Sulima, a SPARC V9 complete machine simulator. Solemn extends Sparc Sulima allowing it to simulate at user-level an unmodified Solaris executable: 32 or 64-bit, and statically or dynamically linked. This yields some advantages over both complete machine simulators and traditional system call emulation. To do this, Solemn manages the virtual address space and files that the simulated program requires, and intercepts and emulates system call traps. Another novel feature is the emulation of memory mapped files. We describe some of the implementation details of Solemn, including its memory management architecture and portability.

1 Introduction

Execution-driven simulation is an increasingly important tool in architecture performance analysis. These can be split into two main types: user-level simulators and complete machine simulators.

Complete machine simulators model the entire computer system, including the operating system kernel and devices.

User-level simulators model the execution of a single user process, and emulate any system calls made on the *target* (simulated) computer, possibly using equivalent system calls available on the *host* computer.

As a result of their detail, complete machine simulators can be quite accurate, but are extremely difficult to implement correctly. User-level simulators are excellent for investigating the behaviour of compute-bound algorithms, but are not as good at modelling IO-dominant programs, nor can they model the effects of cache and TLB pollution caused by operating system activity and other processes.

This paper presents a user-level simulator built on top of a complete machine simulator. This approach yields advantages over both traditional user-level simulation and complete machine simulation.

Sparc Sulima[2] is a *complete machine simulator* for the SPARC V9 architecture, in particular for the UltraSPARC[9] family of processors from Sun Microsystems. Solemn is a new user-level simulation mode with Sparc Sulima, allowing it to simulate unmodified Solaris executables (including *dynamically linked executables*).

Solemn emulates some of the work a real kernel has to do, such as file-related system calls. In particular it has a memory management subsystem, allowing the simulated program to use mmap, and in the future, threads.

The novel part of Solemn is that it emulates true virtual memory: different virtual addresses at different stages of the simulated program's execution could be mapped to the same physical address, and in some stages of the execution a virtual address may not be mapped at all.

All user-level instructions and most exceptions are simulated by Sparc Sulima. Solemn intercepts the trap instructions a program uses to communicate with the operating system kernel. The effect of the trap instruction (usually a system call) is emulated by Solemn before returning control back to Sparc Sulima.

Some of the reasons for extending Sparc Sulima to support the emulation of the Solaris ABI are:

- Sparc Sulima currently does not boot a full operating system, due to some device support issues. Booting a full OS in a complete machine simulator without vendor support is extremely difficult; only SimOS [7] and SimICS [4] have published success in this. Another emulation mode will help in testing more thoroughly the

[*]This paper is a greatly expanded version of [1], which is © 2004 IEEE.

components of the Sparc Sulima system without requiring full OS boot.

- We can support a broader range of programs than tools like RSIM [6], since we can support more system calls (such as `mmap`) and we can simulate *unmodified* and *dynamically linked* programs.

- We can observe more interesting effects than those obtained in a traditional user-level simulator like RSIM such as paging, since the basis is a complete machine simulator. This means, for example, we can examine the effect of running a program in a system with limited RAM.

- We can simulate the effects of small changes to the architecture quite easily. Within a complete machine simulator, such changes would require changes to the operating system, which would be difficult (if the source is available) if not impossible. With an almost entirely user-level simulator like `Solemn` such changes would be simple. Such changes, if required, would occur in a small assembler nucleus.

This paper is organised as follows: §2 provides background, including related work. The overall structure of `Solemn` is presented in §3, with detailed descriptions of significant parts therein. §4 details the system call handlers while §5 discusses the development of `Solemn` and its current status. Finally, conclusions and future work is in §6.

2 Background

This section discusses some related work, and provides background about the Sparc Sulima complete machine simulator and the Solaris operating system and ABI.

2.1 Related Work

RSIM [6] is a user-level execution-driven SPARC V8 simulator. It has detailed CPU modelling (including pipelines and branch prediction) and detailed modelling of the memory system. SMP simulation can be used on programs using a restricted threads library, providing its source code is available.

RSIM can only emulate a quite restrictive set of system calls, and the program to be simulated must be statically linked with RSIM's own C library.

Shade [3] is an address trace generator / user-level simulator for unmodified SPARC binaries (32 or 64-bit, statically or dynamically linked). There is nothing in the literature explaining how it supports dynamically linked executables. It uses advanced techniques including binary translation to speed up its simulation.

Shade relies on the same host platform as the program it is simulating to simplify its system call emulation. `Solemn` uses some of the same techniques used by Shade, such as file descriptor wrapping. A feature of `Solemn` is that it does not require the same host platform for system call emulation.

SimICS [4] is a commercial complete machine simulator for various architectures, including SPARC V9. SimICS had a Solaris emulation mode for 32-bit binaries, but it is no longer maintained.

RSIM is open source, while Shade and SimICS are not open source.

2.2 Sparc Sulima

Sparc Sulima models the UltraSPARC CPU and memory system "as is", using an object-oriented design implemented in C++. This modular approach, with modules corresponding closely to the components of the real system, aids in the readability and understandability of the simulator source.

Sparc Sulima explicitly models the UltraSPARC CPU, along with its MMU and caches, as well as a shared bus, with attached devices including RAM and ROM.

When simulating a multiple CPU system, Sparc Sulima gives each simulated CPU a time-slice of execution (usually something like 50 simulated cycles).

Each CPU interprets each instruction to be executed in a standard fetch-decode-execute cycle. After fetching and decoding the next instruction, the simulator evaluates the instruction.

For more details about the implementation of Sparc Sulima, see [2].

2.3 The Solaris operating system architecture

Sun's Solaris operating environment is an implementation of UNIX. We assume some knowledge of UNIX, but knowledge of Solaris is not required.

Solaris is available on other platforms such as Intel x86 and Itanium. We are not interested in those, so where we refer to Solaris it is generally presumed to be SPARC Solaris. We are also only interested in more recent versions of Solaris (as far back as Solaris 2.6) and only on SPARC V9 based machines (e.g., UltraSPARC I, II and III).

For more detail on the SPARC ABI and Solaris internals see [8, 5, 11].

2.3.1 A process

A process in Solaris (and all UNIXes) is a running program including the current state of its execution. This state includes such things as the values of various registers, the virtual address space and files.

A process's virtual address space is divided up into *segments*. Each segment is divided into *pages*, which may or may not be allocated to physical addresses (physical pages) at any particular point in time. In SPARC Solaris a page is typically fixed at 8KB, although recent versions of Solaris are allowing variable page sizes.

Every segment in a process's virtual address space is a *memory map*: a virtual address space that maps to part of a file. That file may be part of a program file (in the case of the text segment) or an *anonymous* file created as required by the kernel (in the case of the heap or stack segments). A memory map has some associated permissions: readable, writable, and executable. Memory maps can be created by the programmer using the system call `mmap`.

The typical virtual address spaces of 32 and 64-bit processes in Solaris are shown in Figure 1. The stack segment is automatically grown as required. The heap segment is explicitly grown using the `sbrk` or `brk` system calls.

The initial process stack for both 32 and 64-bit processes is shown in Figure 2.

2.3.2 System calls

A UNIX program communicates with the kernel via *system calls*. There are various levels of viewing system calls. At the top is the function level which is the level viewed by the programmer when he/she wants a particular service. The function is implemented in a library which translates the system call into an implementation-defined message to the kernel for the service. To save confusion, we will use the term *system call* only when referring to the implementation-defined message to and from the kernel; we will use the term *C library function* to refer to the system call wrapper.

In SPARC Solaris, the system call message is transmitted via the `ta` (trap-always) instruction. The `ta` instruction generates an immediate exception which causes a transfer of control to a trap-table

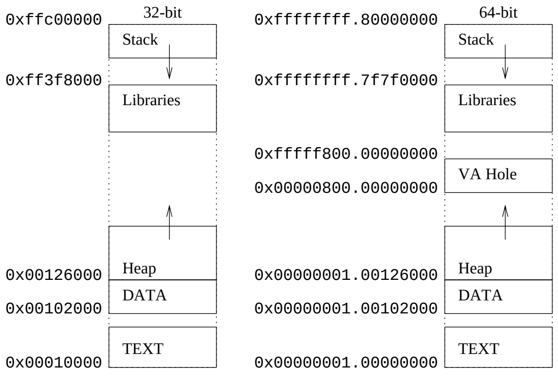

Figure 1: Typical 32 and 64-bit process address spaces in SPARC Solaris. The TEXT and DATA are mapped from the executable file. The Libraries are mapped from dynamically linked libraries. The Heap and Stack are anonymously mapped segments. The VA Hole is because UltraSPARC I and II have only 44-bit virtual addresses.

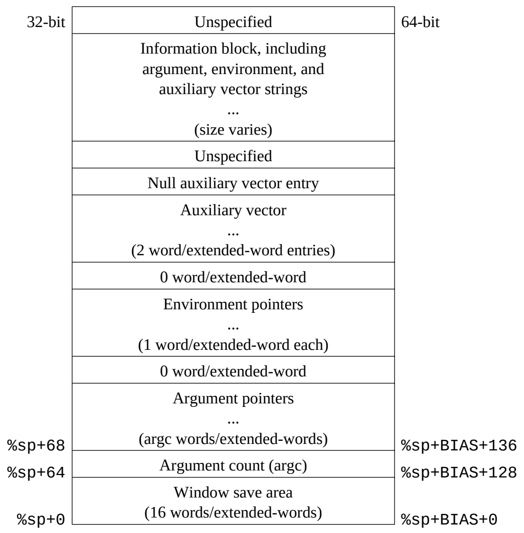
32-bit
64-bit
Unspecified
Information block, including argument, environment, and auxiliary vector strings ... (size varies)
Unspecified
Null auxiliary vector entry
Auxiliary vector ... (2 word/extended-word entries)
0 word/extended-word
Environment pointers ... (1 word/extended-word each)
0 word/extended-word
Argument pointers ... (argc words/extended-words)
Argument count (argc)
Window save area (16 words/extended-words)
%sp+68
%sp+64
%sp+0
%sp+BIAS+136
%sp+BIAS+128
%sp+BIAS+0

Figure 2: The initial process stack for both 32 and 64-bit processes. `%sp` is the stack pointer register. The auxiliary vector is generally used to communicate information from the kernel to the dynamic linker about the process to be linked. In 64-bit processes, the stack pointer is offset from the stack frame by a `BIAS` of 2047.

in the kernel address space. The kernel can then perform the system call request. It eventually transfers control back to the program via the `done` instruction (which signifies a return from an exception and skips the trapping instruction[1]).

The `ta` instruction includes a *trap number*, which is a 7-bit number (0 to 127). About 30 trap numbers are recognised by the Solaris kernel.

Some trap numbers are used for special or *fast* requests. Non system-call traps include such things as: user breakpoints, flush/clean window requests, 32-bit get/set CCR, and fast pseudo-system calls such as `gethrtime`, `gettimeofday` and `getcpuid`.

Trap numbers 0, 8 and 64 are system call trap numbers (respectively used in SunOS 4.x, Solaris 32-bit and Solaris 64-bit executables). The value of the `%g1` register is used to determine which system call is being called.

The registers `%o0` to `%o5` contain the parameters to the system call, like a normal function call in the SPARC ABI[8]. After the system call returns `%o0` and possibly `%o1` contain the return value(s).

The kernel communicates errors by setting the carry bit in the condition code register and returns the error number in `%o0`.

Some example assembler code implementing the `open` C library function for 32-bit Solaris is shown in Figure 3.

Some system calls are multiplexed in that they return multiple values (in `%o0` and `%o1`). An example is `getuid`/`geteuid`. In addition, some 32-bit system calls return a 64-bit number (e.g., `llseek`), in which case the 64-bit number is split up into `%o0` (high) and `%o1` (low).

Solaris 9 defines 231 system calls. Solaris system calls are numbered and named in the file `/etc/name_to_sysnum`. For easy reference this is shown in Table 1.

3 The structure of `Solemn`

Figure 4 shows the structure of `Solemn`. The left-half of the figure is an example standard Sparc Sulima SMP simulation, with 3 CPU's. The right-half of the figure is all `Solemn`-specific modules; the only interaction the standard simulation has with `Solemn` is via the `ExternalHandler` interface, which `Solemn` inherits (not shown in the figure). This allows `Solemn` to intercept `Tcc` calls (see §4).

The ROM attached to the Bus in the figure is a special `Solemn`-specific nucleus. This is described in §3.1.

Files are managed by the `Files` object; more detail on this is in §3.2.

The `MemoryManager` maintains all the information necessary for handling mapped memory. More detail on the memory manager is in §3.3, including details on the `FreeSpace`, `VAMappings` and `PageReplacement` classes.

The `ElfLoader` is responsible for loading a particular file into simulated virtual memory. `Solemn` uses up to two instances of an `ElfLoader` to load an executable, and optionally the dynamic linker. This process is described in more detail in §4.5.

The `SolConvert` namespace and related classes are used for type and structure conversions; this is described in more detail in §4.1.

The `ReentryBuffer` is used by `Solemn` to maintain state when an exception occurs during system call emulation. This process is described in §4.3.

3.1 The Nucleus

The `Solemn` nucleus is a special ROM attached to the bus. It is located at virtual address = physical address = RSTVaddr, an UltraSPARC-specific address for the location of the RED-state trap table. The `Solemn` nucleus contains two trap tables: the RED-state trap table (at RSTVaddr) and a normal trap table located somewhere above RSTVaddr.

3.1.1 Boot

Upon simulation boot, a power-on reset trap (POR) is triggered. The `Solemn` nucleus sets up the initial state:

1. It assumes that the simulator puts various things such as window control registers, caches and MMUs in a sane state: all window control registers are set up sanely, and data in caches and MMUs is invalid. This is stronger than the actual requirements of POR, but saves wasting time.

2. The trap base address (TBA) is initialised to the location of the normal trap table.

3. A 64-KB TLB entry containing both trap tables (VA = PA) is locked into the ITLB. No

[1]The alternative to `done` is `retry` which returns from an exception but retries the trapping instruction. This is generally used by exception handlers that change state that the trapping instruction depended on, so it will no longer trap; e.g., on an MMU miss.

```
open:
        mov     5, %g1                  ! 5 is the open system call number
        ta      %icc, 8                 ! 8 is Solaris 32-bit system call
        bcs     _cerror                 ! branch if condition code carry is set
                                        ! (carry code is set by kernel on error)
        nop                             !  (delay slot)
        retl                            ! otherwise return to caller
        nop                             !  (delay slot)
...
_cerror:
        sethi   %hi(errno), %o1
        or      %o1, %lo(errno), %o1    ! o1 <- address of errno
        st      %o0,[%o1]               ! store errno
        retl                            ! return to caller
        mov     -1, %o0                 !  with -1 as return value
```

Figure 3: Sample assembler of the `open` C library function and system call. `_cerror` is a helper routine to set the `errno` global variable.

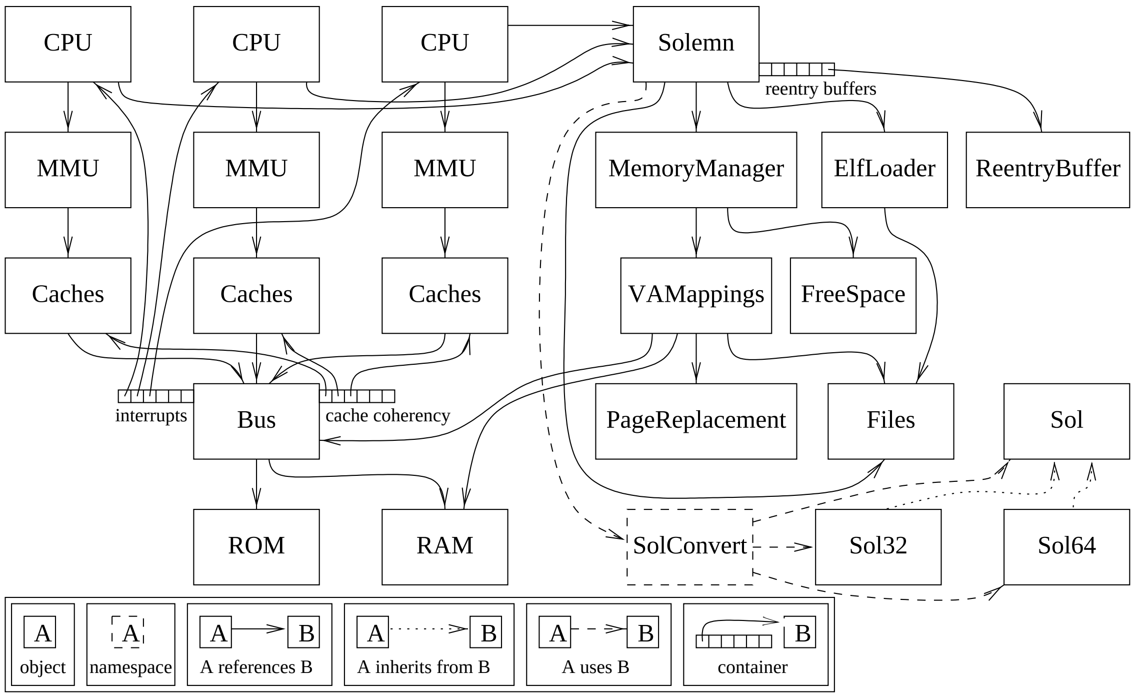

Figure 4: The structure of `Solemn` and its interactions with other Sparc Sulima modules.

Num	Name	Num	Name	Num	Name	Num	Name
0	nosys	62	fcntl	134	rename	196	timer_settime
1	rexit	63	ulimit	135	uname	197	timer_gettime
2	fork	70	tasksys	136	setegid	198	timer_getoverrun
3	read	71	acctctl	137	sysconfig	199	nanosleep
4	write	72	exacctsys	138	adjtime	200	facl
5	open	73	getpagesizes	139	systeminfo	201	doorfs
6	close	74	rctlsys	141	seteuid	202	setreuid
7	wait	75	issetugid	142	vtrace	203	setregid
8	creat	76	fsat	143	fork1	204	install_utrap
9	link	77	lwp_park	144	sigwait	205	signotify
10	unlink	78	sendfilev	145	lwp_info	206	schedctl
11	exec	79	rmdir	146	yield	207	pset
12	chdir	80	mkdir	147	lwp_sema_wait	208	sparc_utrap_install
13	gtime	81	getdents	148	lwp_sema_post	209	resolvepath
14	mknod	84	sysfs	149	lwp_sema_trywait	210	signotifywait
15	chmod	85	getmsg	150	lwp_detach	211	lwp_sigredirect
16	chown	86	putmsg	151	corectl	212	lwp_alarm
17	brk	87	poll	152	modctl	213	getdents64
18	stat	88	lstat	153	fchroot	214	mmap64
19	lseek	89	symlink	154	utimes	215	stat64
20	getpid	90	readlink	155	vhangup	216	lstat64
21	mount	91	setgroups	156	gettimeofday	217	fstat64
22	umount	92	getgroups	157	getitimer	218	statvfs64
23	setuid	93	fchmod	158	setitimer	219	fstatvfs64
24	getuid	94	fchown	159	lwp_create	220	setrlimit64
25	stime	95	sigprocmask	160	lwp_exit	221	getrlimit64
27	alarm	96	sigsuspend	161	lwp_suspend	222	pread64
28	fstat	97	sigaltstack	162	lwp_continue	223	pwrite64
29	pause	98	sigaction	163	lwp_kill	224	creat64
30	utime	99	sigpending	164	lwp_self	225	open64
31	stty	100	setcontext	165	lwp_setprivate	226	rpcmod
32	gtty	103	statvfs	166	lwp_getprivate	230	so_socket
33	access	104	fstatvfs	167	lwp_wait	231	so_socketpair
34	nice	105	getloadavg	168	lwp_mutex_wakeup	232	bind
35	statfs	106	nfs	169	lwp_mutex_enter	233	listen
36	syssync	107	waitsys	170	lwp_cv_wait	234	accept
37	kill	108	sigsendsys	171	lwp_cv_signal	235	connect
38	fstatfs	112	priocntlsys	172	lwp_cv_broadcast	236	shutdown
39	setpgrp	113	pathconf	173	pread	237	recv
41	dup	114	mincore	174	pwrite	238	recvfrom
42	pipe	115	mmap	175	llseek	239	recvmsg
43	times	116	mprotect	176	inst_sync	240	send
44	profil	117	munmap	177	srmlimitsys	241	sendmsg
46	setgid	118	fpathconf	178	kaio	242	sendto
47	getgid	119	vfork	179	cpc	243	getpeername
48	ssig	120	fchdir	180	meminfosys	244	getsockname
49	msgsys	121	readv	184	tsolsys	245	getsockopt
51	sysacct	122	writev	185	acl	246	setsockopt
52	shmsys	123	xstat	186	c2audit	247	sockconfig
53	semsys	124	lxstat	187	processor_bind	248	ntp_gettime
54	ioctl	125	fxstat	188	processor_info	249	ntp_adjtime
55	uadmin	126	xmknod	189	p_online	250	lwp_mutex_unlock
56	uexch	128	setrlimit	190	sigqueue	251	lwp_mutex_trylock
57	utssys	129	getrlimit	191	clock_gettime	252	lwp_mutex_init
58	fdsync	130	lchown	192	clock_settime	253	cladm
59	exece	131	memcntl	193	clock_getres	254	lwp_sigtimedwait
60	umask	132	getpmsg	194	timer_create	255	umount2
61	chroot	133	putpmsg	195	timer_delete		

Table 1: Solaris system calls, from `/etc/name_to_sysnum` on Solaris 9.

DTLB entry is required since the `Solemn` nucleus has no state; all state is accessed via calls to `Solemn`.

4. The Caches and MMUs are enabled.
5. `Solemn` is initialised, using the `solemn-init` system trap (see §4.5). This returns with: `%o0` = is64 (zero or one), `%o1` = entry point, `%sp` = stack pointer. The nucleus needs to know is64 in order to set up address-masking and register window spill/fill processing.
6. User-level state is prepared, by setting the `tstate`, `tpc` and `tnpc` registers.
7. User-level simulation is entered by calling the `retry` instruction.

3.1.2 Normal trap table

The normal trap table is really made up of two halves: the user trap table and the nucleus trap table.

The user trap table consists of trap entries for when exceptions occur when the trap-level is zero (i.e., user code is executing). Most of the exception handlers are filled with dummy code, simply halting the simulation. Some handlers are required:

- Clean-window: clears all locals and outs.
- Spill (and fill): do the usual stores to (loads from) the stack. These are specialised depending on whether the user-level program is 32 or 64-bit.
- IMMU and DMMU miss: this extracts the faulting address from the MMU and calls the `Solemn` page-miss trap with `%g1` = VA (§4.4.4). If it returns, `%g1` is the physical address. A new entry is added to the TLB, and the faulting instruction is retried.
- DMMU protection: a write was performed to a non-writable page. The old TLB entry is removed, and then a `Solemn` page-protection trap is invoked with `%g1` = VA (§4.4.5). If it returns, then the page is actually writable, so a new (writable) entry is added to the DTLB.

The nucleus trap table entries are called when an exception occurs during trap processing (either user-level or nucleus-level), unless the trap level gets too high, at which point the processor enters RED state and uses a trap base address of RSTVaddr.

The only nucleus traps that can occur in `Solemn` are DMMU miss and protection faults. These can occur during window spill or fill trap processing, since the part of the stack used may not have been accessed at user level yet (or recently, if a TLB entry was replaced). These handlers simply branch to the corresponding user-level handlers.

3.2 The `Files` Manager

The `Files` manager is a wrapper around the host standard IO mechanism. It includes its own set of file descriptors which is distinct from the host file descriptors. Internally each `Files` file descriptor is mapped to a host file descriptor.

It also provides an interface to all the standard IO calls, such as `open`, `close`, `read`, `write`, `lseek`, `mmap`, etc, where these calls operate on a `Files` file descriptor. Other structures (such as `struct stat` for `fstat`) and flags (such as `open` and `mmap` flags) are the host operating-system structures and flags.

The return value of the corresponding host call is checked, and if an error occurred the error number is recorded within the `Files` instance. This ensures easy access to the error number if it needs to be returned to the simulation[2].

The `Files` manager also can optionally behave as though the program being simulated has been `chroot`-ed (i.e., the root of the file system has been changed). To provide for this, `Files` implements its own symbolic link resolution.

3.3 The Memory Manager

A `Solemn` instance has a `MemoryManager` instance, which is created during the nucleus-init (§4.5) `Solemn` call. The memory manager is responsible for dealing with all memory-related traps and system calls, such as page misses (trap) and `mmap` (system call). It contains structures that maintain the unused virtual address space (a `FreeSpace` instance) and all current virtual address mappings (a `VAMappings` instance). It also contains other information needed, such as the location and size of the stack and heap, and where unfixed `mmap`s should start searching for space.

[2]Obviously this is not thread-safe, and if and when the simulator and `Solemn` is made threaded then the `Files` class will need to be thread aware. There is only one `Files` instance in a `Solemn`-powered simulation so judicious use of locks will be required.

3.3.1 The free space manager

The free space manager (`FreeSpace`) is essentially a set of virtual address (VA) intervals. This uses the Sparc Sulima container adapter `interval_adapter` around an associative array. `interval_adapter` ensures the set is minimal, so that any insertions that adjoin (or overlap) already existing interval(s) result in extending the range of that interval (and possibly removing other intervals).

The requirements of the free space manager are to maintain this set such that it contains any virtual address range that is neither mapped nor unavailable by some ABI (e.g., from zero to the base address of the executable is usually not available for mapping unless specifically asked for) or platform requirement (the 64-bit VA hole in the UltraSPARC I and II).

As an example, since fixed memory mappings that succeed may replace other mappings and may also (or only) remove "free space", the `erase` method does not require that any of the given interval is already "free". Similarly, since `munmap` of an unmapped address interval does not result in an error, the `insert` method does not require that the given interval is not "free".

The free space manager also provides for the searching for free space of a given size and alignment. This is for use with an unfixed `mmap` (the address given to `mmap` may be the alignment required). This search can start from any given point in the address space, and works down, returning the top-most interval available.

3.3.2 The virtual address mappings manager

The virtual address mappings manager (`VAMappings`) provides an interface to the current virtual address mappings, as well as the current physical address mapping (if any) each page has. It also manages physical page replacement.

For the purposes of `Solemn` the RAM and virtual address space is divided up into fixed-size pages (8KB) which can be individually accessed via a page-index (the address divided by 8KB = 8192).

Virtual address mappings The virtual address mappings is implemented by an associative array from virtual address interval to a structure that contains:

- the memory protections that this virtual address mapping is allowed; this is a bitmap of readable, writable and executable;
- a pointer to the host memory for this virtual address mapping, allocated via host `mmap`;
- a vector of page-indices into the simulated RAM: this is a vector since the mapping may span more than one page; the index is given a fixed invalid value if the virtual page is not available in a RAM physical page.

As an example, this is used for a page miss to look up whether a given virtual address has a physical address currently assigned. If it does then that address is returned, otherwise a new physical address is assigned (using the page-replacement policy).

Physical address mappings There is also a reverse lookup mechanism, a mapping from RAM physical pages back to the virtual mapping structure. This is implemented as a vector (indexed by physical page index) to a structure containing:

- a pointer to the simulated RAM that this page refers to; this is fixed throughout the life of the virtual address mappings manager;
- an iterator into the virtual address mappings structure that points to the virtual address mapping that has a virtual page that has been allocated to this physical page; this is set to a fixed value if the physical page is not being used by any virtual page;
- a page index into the virtual pages in the above iterator; this is undefined if the page is not being used;
- a boolean value indicating whether the page has been written to or not; this is used during physical page demapping to write back changed pages from simulated RAM to host mappings; this defaults to false, and gets changed to true on a page protection fault that is allowed to continue.

This is used for a page fault, to look up whether a given physical address has a virtual mapping and if that virtual mapping is writable.

Page replacement A page-replacement object manages the allocation of physical pages. This is implemented through a small interface, just requiring a couple of methods: `used(PageIndex)`, `unused(PageIndex)`, and `get_replacement_page()`.

Currently there is only one page-replacement policy: least-recently-used. This is implemented as the index of the least-recently-used physical page, and a vector (indexed by physical page index) that contains page indices for the next and previous element in the order that pages have been used. This allows extremely fast lookup of the least-recently-used page. Since it is doubly-linked the only change required in the common case of moving the least-recently-used page to the most-recently-used page (this occurs during page replacement) is to change the least-recently-used value.

4 Traps and System Calls

`Solemn` intercepts `Tcc` calls (via the `ExternalHandler` interface, not shown in Figure 4), and decides what action to take depending on the trap number, and optionally the system call (if the trap is a system call).

`Solemn` recognises and handles a few non system-call traps that are routinely called. It also handles Solaris 32 and 64-bit system calls (both are handled using the same interface).

Some trap numbers are used by the `Solemn`-specific nucleus to communicate with `Solemn`. These are from `0x60` onwards and are described in §4.4.4, §4.4.5, and §4.5.

Within `Solemn`, many system calls are simply passed on to the host system, possibly translating some values before calling the host system call, and converting results back to the simulation. Some of these are listed in Table 2. File descriptors (fd) are translated to and from the simulated set of files to the host file descriptors via the `Files` interface (§3.2). Whence, open-flags and mode are all translated via the `SolConvert` namespace, which allows conversion to and from the host and Solaris values.

4.1 Solaris types and values

Much of `Solemn` by design is to emulate Solaris system calls. A "port" of `Solemn` for it to emulate the system calls of another operating system (such as SparcLinux or Sparc OpenBSD) would require significant changes. However, since much of the structure and design of these Unix (or Unix-like) kernels is similar some work has been done to ensure that it is possible.

Perhaps the biggest problems are the types, structures and values expected by the kernel. In a real operating system, these are defined in system header files, but with `Solemn` we need to do it differently, since we would like `Solemn` to be able to compiled and run on a non-Solaris (and possibly even non-SPARC) system. In addition, we want to transparently simulate 32 and 64-bit executables; many of the types and structures change size (e.g., `size_t`) and arrangement (e.g., `struct stat`) depending on architecture.

None of the types, structures and values expected by `Solemn` when compiled come from system header files. They are all defined in a set of interface classes (classes that have no state, just types): the `Sol`, `Sol32` and `Sol64` structures depicted in Figure 4.

`Sol` defines the types that are common to both Solaris 32 and 64-bit architectures, mostly the `*64_t` types, but also other enumerated flags such as the open flags (e.g., `o_RDONLY`) or `mmap` flags (e.g., `mmap_MAP_ALIGN`[3]).

`Sol32` and `Sol64` inherit these types and enumerations, and also define the architecture-specific types, such as `size_t` and `struct stat`.

The `Sol`, `Sol32` and `Sol64` types are not made visible to any part of `Solemn`, except through a couple of interfaces: one to convert from simulated values (e.g., `mmap` flags) to a host-agnostic structure (e.g., a simple enumerated class that wraps around the `mmap` flags), and one to convert from host-operating-system values (e.g., `struct stat`) to the equivalent structure in the simulated system.

4.2 Restartable system calls

There are many system calls where there are buffers (e.g., open, ioctl, read, write) that point to data in the simulator to be read from (open, ioctl, write) or written to (ioctl, read). `Solemn` manages this reading from or writing to buffers by going through the calling CPU's MMU. It goes via the MMU since it cannot go directly to the RAM: not only is it possible that the data is in another CPU's cache (so cache-coherency is required) but it is also possible that the data is not in RAM at all[4].

Going via the MMU has its own complications: the MMU may not have a translation for the required page in its TLB (translation lookaside

[3]`MAP_ALIGN` is a new value in Solaris 9; if this flag is used `addr` is no longer used as a hint, but is instead the required alignment of the mapping.

[4]This is actually very common and easy to reproduce. Simply do an anonymous writable mmap, followed by a read into that memory map. An `mmap` does not necessarily make the page directly usable but only potentially usable until there is an attempt to actually access the page. Indeed this will likely cause two traps: firstly a page miss to bring the page into the TLB, then a page fault (protection error), since the page would have been initially mapped read only.

System Call	sim-to-host conversion	host-to-sim conversion (other than `errno`)
lseek, llseek	fd, whence	
close	fd	
open	path, flags, mode	fd
dup	fd	fd
ioctl	fd, request, optional arg	optional arg copy back
fstat, fstat64	fd	buf
read	fd	buf
write	fd, buf	
getuid, getgid		

Table 2: Some directly translating system calls.

buffer). Hence the system call handler must detect the exception, stop system call emulation and return control back to the simulator generating the required exception. The exception will be processed, hopefully installing a TLB entry for the faulting address, and the faulting instruction (which is the *system call*) will be retried. The system call handler will then be called again.

To deal with this case, the simpler system calls (e.g., open, ioctl) assume that the buffer is small and that there is no side affect from doing the operation twice (in the case of ioctl copy-back): these system calls are called *restartable*. The buffer is fully copied from simulator memory to a host memory buffer; for open the maximum size is the Solaris `PATH_MAX` value but is stopped when a zero byte is reached; for ioctl the size is somewhat arbitrary. If an exception occurs during the read from the simulator memory, the system call returns immediately generating the required exception. The exception will get handled by the nucleus which results in a page-miss trap to `Solemn` (this is described in §4.4.4). Assuming the virtual address is valid `Solemn` returns a physical address for the virtual address, the nucleus puts the TLB entry into the D-MMU and retries the offending instruction: in this case the system call. The system call handler will be invoked again, but this time the read from the simulator should get further than the last time[5].

The copy-back for ioctl is similar, but now we must assume that performing the ioctl multiple times is not a problem. This is because if an exception occurs during copy-back (e.g., the page was mapped read only, but it is actually writable) then the entire system call will be redone after the page protection fault is handled. If this turns out to be a problem, then ioctl can easily be changed to behave more like the re-enterable system calls (read and write: see §4.3).

4.3 Re-enterable system calls

The buffers in read and write system calls can be very large: these can be much bigger than the set of all pages that can be in the D-TLB at once. The system calls themselves have a side effect and so are not restartable. These facts mean that these system call handlers need to be *re-enterable*: i.e., return to some state achieved part-way through emulation of the system call.

With the read system call, the entire requested read can be performed at once (and only once) on the host, into a host-allocated buffer. The re-enterable part occurs if an exception occurs during copy back into the simulator. If an exception occurs, the portion of the buffer that has not yet been copied back is stored within `Solemn` and the exception propagated back to the simulator. Upon return to the system call, `Solemn` recognises that it is re-entering the read system call and simply branches to the copy-back procedure. This can continually cause exceptions, but the buffer for the next re-entry gets shorter each time (ignoring page protection faults).

[5]Since both of these buffers are less than a page in size, the number of misses that can occur is usually one, but occasionally two. It is pathologically possible for there to be *three* page misses in this case: when the buffer spans two pages, and both pages are not in the D-TLB, and the first D-TLB insert goes in the first unlocked page, and there are no other unused pages. The second page miss will occur, and the D-TLB page insert will not find any unused pages, all unlocked used pages are set to unused, and the first entry used for the new entry, replacing the first page entry. A third miss will occur when trying to read from the first half of the buffer again. Unless there is only one unlocked page in the D-TLB (which would be really bad, and currently not possible with `Solemn` since the only locked page is the trap-table) then the third page insert will not replace the second page entry; the entire read will now succeed.

With the write system call, the entire buffer is copied from the simulated memory to the host before the write is performed. Like the read system call, the copy re-enters where it last failed, so that very large buffers can be used safely. It would be possible to perform the host write as soon as some part of the buffer had been copied from the simulator, but we decided that this might have peculiar effects when we have threaded simulation working (although a well-written program should probably not rely on the kernel doing this for them).

To cope with the possibility of multiple threads calling the same system call (e.g., read) but with probably completely different file descriptors, buffers, and lengths, we must allow for multiple re-enterable structures: one for each thread. This is best solved by assuming one thread per CPU, and no thread migration, and storing the re-entry state within `Solemn` indexed by the CPU. This is implemented by an associative array from CPU id (an integer unique to each CPU on the bus) to a `ReentryBuffer`.

4.4 Memory-related system calls

Memory related system calls such as `brk`, `mmap` and `munmap` and memory related system traps (e.g., page miss) are passed down to the memory-manager.

4.4.1 The `mmap` system call

The `mmap` system call asks for a virtual address space that maps to (part of) a file.

The pages are currently implemented in `Solemn` at a fixed 8KB; variable page sizes are planned for a future `Solemn` revision.

An `mmap` system call is emulated by `Solemn` in a number of stages:

1. Firstly, the parameters are checked for validity; the required error number is returned if there is a problem.
2. Then, if the `mmap` request is not `FIXED`, a virtual address range is found for the request. This is done by searching through the free space manager. If there is no space for the required range then `ENOMEM` is returned to the caller.
3. A host `mmap` is performed on the required file of the required size. Assuming the host `mmap` succeeds, the resulting pointer is used as the backing storage for that simulated virtual address range.
4. If the virtual address range contained previous mappings (this can only occur if the `mmap` request was `FIXED`) then the previous mappings are unmapped.
5. The virtual address range is removed from the free space manager.
6. The virtual address range, protections, and backing store are added to the virtual address mappings.
7. The virtual address is returned to the caller as a successful `mmap`.

Note that this does not allocate physical pages for the virtual address range. This is done on demand as page misses occur. This is called demand-paging and is quite a standard mechanism.

If the `mmap` is successful, `Solemn` also checks if it is possibly an `mmap` related to a dynamically loaded library. If it is, and `Solemn` has been given a symbol table object to manage, then the symbol table of the library is loaded; see §4.5.3.

4.4.2 The `munmap` system call

The `munmap` system call asks for all mappings within a virtual address range to be unmapped. Each mapping within that range goes through a number of stages:

1. Any pages mapped to physical RAM are swapped out:
   (a) Firstly, for all CPU's, any TLB entries for that physical page, and cache lines referring to any parts of that physical page are flushed and invalidated. This is done via direct call from `Solemn` to a special hook in the system bus that implements this. In a real system this would probably involve interrupts (for processor-to-processor communication) and displacement loads (for invalidates).
   (b) Then, if the page is dirty (see §4.4.5) the page is written back to the backing store; in the simulation this is simply the location that was `mmap`ed on the host machine.
2. The host backing store is unmapped using a host `munmap`.
3. The virtual address range is removed from the virtual address mappings.

Finally, the entire virtual address range is added to the free space manager.

4.4.3 The `brk` system call

The `brk` system call is used to extend (or shrink) the heap.

If the request is to extend the heap, then a `brk` is equivalent to a fixed `mmap` to an anonymous, private, readable and writable file.

If the request is to shrink the heap, then a `brk` is equivalent to an `munmap`.

4.4.4 MMU-miss system trap

When a page miss occurs, this means that the requested load or store instruction (or instruction fetch) refers to an address which is not mapped within the simulated CPU's TLB (translation lookaside buffer). This causes a `fast-data-access-MMU-miss` (or `fast-instruction-access-MMU-miss`) exception. The `Solemn` nucleus has short exception handlers for these exceptions (§3.1.2) that mainly call this special `Solemn`-specific system trap to do most of the work.

The MMU-miss system trap is given the following parameters: the faulting virtual address, and whether it is a data or instruction MMU miss.

There are three possible cases:

- the virtual address is valid and has already been mapped to a physical address (the page is in RAM); or
- the virtual address is valid and has not been mapped to a physical address; or
- the virtual address is invalid.

To simplify many system calls and to avoid using signals, invalid addresses cause the simulation to halt. Invalid addresses are rarely used in "valid" programs so this should have little practical effect. This also includes invalid addresses passed to system calls: instead of returning `EFAULT`, the simulation will halt. The invalid address case includes protection errors: an instruction miss to a non-executable page or a data miss to non-read/write page.

If the virtual address is valid and the page is already mapped to a physical address, then the system simply returns with the physical address as the return value. The nucleus inserts the new translation table entry into the TLB and retries the faulting instruction.

If the virtual address is valid and the page is not mapped to a physical address, then the virtual page must be assigned a physical page.

Firstly, a physical page is chosen. This is via a least-recently-used page replacement policy (where "used" is defined as when a page miss or protection fault occurs).

In general, this physical page may have already been assigned a virtual page. So this existing page must be swapped out: this involves the same steps as the page swapping in the `munmap` system call emulation (§4.4.2).

The faulting virtual page is copied from the backing store to the physical page (the RAM), and the system call returns with the physical address.

Note: with a data miss, the new TLB entry that the nucleus inserts is set to be read-only. Write-access checks are done during protection traps.

4.4.5 Page protection error system trap

When a store is done to a page that has a TLB entry is not marked as writable, a page protection exception occurs (`fast-data-access-protection`). As for the miss exception handling case, the `Solemn` nucleus performs a special system trap to ask `Solemn` to check whether the write should be allowed.

An invariant in the page protection case is that the page is already mapped to a physical address. This invariant is maintained during page replacement by flushing existing translations to the page.

The page protection system call simply checks the permissions on the virtual page. If writing is not allowed, simulation halts. If writing is allowed, then the physical page is marked as dirty (so future page replacements will flush back to stable storage) and the system call returns.

The `Solemn` nucleus replaces the previous TLB entry with one that also enables writing, and retries the faulting instruction.

4.5 Initialisation and Program Loading

The initialisation of `Solemn` occurs when the nucleus calls the `solemn-init` system trap during the boot phase (§3.1.1).

This sets up the file and virtual memory subsystems, then loads the program into virtual memory (see §4.5.1) and loads the dynamic linker as well if required (see §4.5.2).

Finally, the initial process stack is created as shown in Figure 2.

4.5.1 Program loading and dynamic linking

Program loading involves opening the executable file, and parsing the ELF (executable and linking format) structure contained within the file. An ELF executable contains the following information in its ELF header:

type The type of file. Must be either `EXEC` (a normal executable) or `DYN` (a dynamic library; this is unusual, but allows for directly running `/usr/lib/ld.so.1`). Other possible types include `REL` (a relocatable object file) and `CORE` (a core file).

machine The required architecture. This must be one of the `SPARC` variants.

entry The virtual address of the entry point.

flags Processor-specific flags for the file; this can give requirements on the platform to be able to run this program (e.g., the program may contain UltraSPARC-III specific instructions).

An ELF executable also contains a program table with loadable program table entries (a program table may also contain other, unloadable entries). Each loadable program table entry contains information about a *segment* within the executable. This information includes:

offset The offset from the beginning of the file to the first byte of the segment.

vaddr The virtual address at which the first byte of the segment resides (i.e., to where the segment must be loaded). The rest of the segment is contiguous.

filesz The number of bytes in the file image of the segment.

memsz The number of bytes in the memory (loaded) image of the segment. If this is greater than filesz, then the remainder should be zero.

flags Permissions for the segment: whether the segment should be readable, writable, and/or executable.

Each segment of the executable is loaded using the emulated `mmap` (§4.4.1).

4.5.2 Dynamically linked executables

A program is dynamically linked if the program has an *interpreter* (this is true if and only if the program file contains a `PT_INTERP` program table entry). If this is the case, then the path of the interpreter is extracted from the program file.

The executable is loaded as per normal, but then the interpreter is also loaded.

Since the interpreter is a dynamic library, it can be loaded at any available address space. Different segments within the interpreter must, however, be located at the same relative position as their offsets require. Hence, we need to know the entire range of virtual addresses that the interpreter covers in order to ensure that we reserve a large enough space. This information (which we could call the *span* of the library) is not explicitly recorded within the ELF structure; it requires a pass through the program table to calculate the range of addresses required.

An anonymous, unreadable, address space of size *span* is then `mmap`-ed, giving us the base address of the interpreter. The segments of the interpreter are loaded using *fixed* `mmap`s relative to the base address.

The auxiliary vector table (this follows the argument and environment lists on the stack, see Figure 2) is populated with the following information:

platform The type of platform (as a string; e.g., “`SUNW,Ultra-1`”).

execname The file name of the executable.

phdr If the executable’s program header table contains a program header entry, then the auxiliary vector contains the value of the program header entry (which is defined to be the address of the program header table[6]). It also contains:

> **phent,phnum** The size of each entry, and number of entries in the program header table.
>
> **entry** The entry point of the executable.

execfd If the program header table does not contain a program header entry, then the file descriptor of the executable is added to the auxiliary vector.

base The address that the interpreter was loaded at.

flags Processor flags (the type of processor on the current platform).

pagesz The page size.

sun-uid,ruid,gid,rgid Various user and group ids.

[6]This means that the program header table must be part of a loadable segment.

sun-hwcap Sun-specific hardware capabilities of the current platform.

The auxiliary vector is used by the dynamic linker (the interpreter) to allow it to do the required dynamic linking. The entry point of the user-level program is the dynamic linker's entry point. After linking, the dynamic linker branches to the original program's entry point, which is (usually) in the auxiliary vector.

4.5.3 Symbol tables

`Solemn` can optionally be given a symbol table (class `SymbolTable`) object to manage, in which case `Solemn` will load symbols of the simulated executable into the symbol table as required. This provides for easier debugging and observation of what exactly is happening during execution. In addition, if a call tracer is active it means the call tracer can recognise the symbols of the dynamic linker and other dynamic libraries.

During loading of the executable and dynamic linker this is trivial since it is obvious that we have just loaded something that has symbols.

Loading a dynamic object's symbols (such as the dynamic linker) requires we pass the base address so the symbol table loader knows where the objects in the library have been loaded.

It is harder to detect when dynamic libraries are loaded by the dynamic linker. This is because the dynamic linker simply uses the `mmap` system call to load the object. After a handling a successful `mmap` system call (§4.4.1) `Solemn` uses a series of heuristics to determine whether an `mmap` is actually loading a dynamically linked library, and hence loads the symbol table. These heuristics include:

- The `mmap` was not anonymous, was private, and was executable.
- The file is a valid ELF object, and is either a dynamic object or an executable.
- If the file is a dynamic object, then we do some other checks. We have observed that the dynamic linker does dynamic loading in several phases:
    1. Firstly, it loads the first 8K of the object, presumably in order to read its ELF information.
    2. Then, if the object has only one program table entry and fits within the 8K page, it simply uses that loaded page as the library and links to it.
    3. If the object does not fit within the 8K page, then it loads the entire file with an *unfixed* `mmap` covering the entire *span* of the library (that is, the address range that the library covers; the span is likely to be much larger than the size of the file). It does this so that later segments of the object can be loaded into the correct offset relative to the earlier segments. The initially loaded 8K page is kept as scratch space for later loads of other dynamic objects.
    4. The other segments of the dynamic object are loaded using *fixed* `mmap`s, within the address range from the initial, *span*-covering `mmap`.

  We can detect the initial, *span*-covering `mmap`, by checking the offset and length of the `mmap` with the dynamic object's *span*.
- If the object is an executable, then the `mmap` must be *fixed* and located at the base address of the object.

If all of these tests pass, then we assume that the dynamic linker is loading an object that will be linked and possibly called, so the symbol table of the object is loaded and added to the global symbol table.

5 The Development of `Solemn`

The development of `Solemn` is a follow on from the `UserSim` [2] module in Sparc Sulima. `UserSim` used the C library from RSIM[6], and so the executable to be simulated needed to be statically linked with that C library. As a result, there was a limit to the sorts of programs that could be simulated within `UserSim`. In particular, programs that used `mmap` could not be simulated.

We investigated the possibility of extending `UserSim` and the C library to support `mmap` and dynamically linked executables. This idea was discarded to using an existing full C library, so we did not have to reinvent the wheel. Solaris was chosen as the platform since we were already familiar with it, and our main program of interest, Gaussian, was optimised for it.

The main issue with developing `Solemn` has been deficiencies in the documentation of Solaris. Some examples of this are:

- The initial process stack [8, pp 3P-25 – 3P-27] and dynamic linking [10, pp 248–255] is

mode	build	#instructions
`UserSim`	32-bit static	258
`Solemn`	32-bit static	649
`Solemn`	32-bit dynamic	224913
`Solemn`	64-bit dynamic	220064

Table 3: The number of instructions evaluated by Sparc Sulima when simulating different builds of a trivial C program (`int main() { return 0; }`).

reasonably well documented, although the required values for various auxiliary vector entries was only determined by reading the Solaris kernel source.

- While the mapping from system call number to name is well documented, the fact that some systems calls are multiplexed, and the system call error method are not. Judicious reading of Solaris kernel and C library source was and continues to be required.

Debugging such a beast is, not surprisingly, quite difficult. This is particularly evident during dynamic linking, when a huge amount of instructions are simulated. The principle techniques used were:

- Targetted debugging information, including debugging levels and masks.
- Small test programs which test different components of `Solemn`, in particular different system calls.

5.1 Current status

Currently, `Solemn` can emulate a single-threaded Solaris executable, 32 or 64-bit, and statically or dynamically linked. `Solemn` recognises and handles about 25 system calls, although this is increasing as we test programs that use them.

The effect dynamic linking has on process startup is made quite evident when simply counting the number of instructions simulated. Table 3 shows the number of instructions evaluated when simulating a trivial C program.

6 Conclusions and Future Work

`Solemn` extends Sparc Sulima to allow it to simulate unmodified, dynamically linked, 32 or 64-bit Solaris executables. With this support, the behaviour of many more interesting programs can be examined than was previously possible.

Since `Solemn` provides true virtual memory, including swapping, it is possible to examine the effect of running programs in limited RAM. This is something we believe is not possible in previous user-level simulators.

We are currently working on extending the system calls supported by `Solemn`, in particular thread support. This will allow us to simulate many commercial and scientific workloads.

Sparc Sulima (including `Solemn`) is open source, with source code available under the GNU GPL at `http://cap.anu.edu.au/cap/projects/sulima/`.

Acknowledgements

The author is a member of the CC-NUMA Project, which is supported by Australian Research Council linkage grant LP0347178, with Sun Microsystems Inc and Gaussian Inc as industry partners. The author thanks Peter Strazdins for his comments and feedback in developing `Solemn` and in producing this paper.

References

[1] Bill Clarke. Solemn: Solaris emulation mode for Sparc Sulima. In *37th Annual Simulation Symposium*, Arlington VA, USA, April 2004. The Society for Modeling and Simulation International, IEEE Computer Society Press.

[2] Bill Clarke, Adam Czezowski, and Peter Strazdins. Implementation aspects of a SPARC V9 complete machine simulator. In Michael Oudshoorn, editor, *Computer Science 2002*, volume 4 of *Conferences in Research and Practice in Information Technology*, pages 23–32, Monash University, Melbourne, January 2002. Australian Computer Society.

[3] Robert F. Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[4] Peter S. Magnusson, Fredrik Dahlgren, Hekan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim

Nilsson, Per Stenstrvm, and Bengt Werner. SimICS/sun4m: A Virtual Workstation. In *Usenix Annual Technical Conference*, June 1998.

[5] Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Components*. Prentice Hall, 2001.

[6] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997. Also appears in IEEE TCCA Newsletter, October 1997.

[7] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, Fall 1995.

[8] SPARC International, Santa Clara, CA. *SPARC Compliance Definition 2.4.1*, July 1999.

[9] Sun Microelectronics, Palo Alto, California. *UltraSPARC User's Manual: UltraSPARC-I and UltraSPARC-II*, July 1997.

[10] Sun Microsystems, Palo Alto, CA. *Linker and Libraries Guide*, January 2001.

[11] Sun Microsystems. Solaris 9 operating environment source, SPARC platform edition, August 2002.