Linking Programs in a Single Address Space

Luke Deller, Gernot Heiser
School of Computer Science & Engineering
University of New South Wales
Sydney 2052, Australia
{luked,gernot}@cse.unsw.edu.au, http://www.cse.unsw.edu.au/~disy

Abstract

Linking and loading are the final steps in preparing a program for execution. This paper assesses issues concerning dynamic and static linking in traditional as well as single-address-space operating systems (SASOS). Related loading issues are also addressed. We present the dynamic linking model implemented in the Mungi SASOS and discuss its strengths and limitations. Benchmarking shows that dynamic linking in a SASOS carries significantly less overhead than dynamic linking in SGI's Irix operating system. The same performance advantages could be achieved in Unix systems, if they reserved a portion of the address space for dynamically linked libraries, and ensured that each library is always mapped at the same address.

1 Introduction

Single-address-space operating systems (SASOS) make use of the wide address spaces offered by modern microprocessor architectures to greatly simplify sharing of data between processes [WSO+92,CLBHL92,RSE+92, CLFL94]. This is done by allocating *all data* in the system, whether transient or persistent, at a unique and immutable virtual address. As a result, all data is *visible* to every process, and no pointer translations are necessary for sharing arbitrary data structures. While the global address space makes all data addressable, a protection system ensures that *access* only succeeds when authorised. Protection relies on the fact that a process can only access a page if it has been mapped to a RAM frame, by the operating system loading an appropriate entry into the translation lookaside buffer (TLB). The operating system thus has full control over which parts of the single address space are accessible to a given process.

As persistent data, i.e., data whose lifetime is independent of that of the creating process, is always mapped into the virtual address space (at an immutable address), SASOS do not need a file system. For compatibility with other systems, a file system interface can be provided, of course, but it represents nothing more than a different way to access virtual memory. All disk I/O is done by the virtual memory paging system.

Such a SASOS is rather different from a system like MacOS, which also shares an address space between all executing programs. The main difference (other than the absence of memory protection) is that the latter system does not ensure that a data item has a unique address for its lifetime. For example, files can be mapped into memory, but each time a file is opened, it will be mapped at a different address. As each object in a SASOS has an immutable address, pointers are perfect object references which do not lose their meaning when passed between processes or stored in files. This greatly facilitates sharing of data between programs in a SASOS, as any data item can always be uniquely identified by its virtual memory address.

Besides sharing, the single address space also significantly simplifies system implementation [WM96] and improves performance even for applications which are not aware of the single address space [HEV+98]. However, the changed notion of address spaces makes it necessary to rethink a number of issues relating to how the system is used in practice. These include preparing programs for execution: how to bind together separately compiled program components ("linking"), and how to get an executable program into a state where the CPU can execute its instructions ("loading").

In this paper we examine the issues of linking and loading of programs in traditional Unix systems, as well as in SASOS. We present the model of (dynamic) linking used in the Mungi SASOS [HEV+98] which is under development at UNSW. The implementation of dynamic linking in Mungi is discussed and its performance compared to that of Unix operating systems.

2 Linking in traditional operating systems

Traditional operating systems, such as Unix systems, generally feature two forms of linkage, static and dynamic.

During *static linking*, all code modules are copied into a single executable file. The location of the various modules within that file implies their location in the memory image (and hence in the address space) during execution, and therefore target addresses for all cross-module references can be determined and inserted into the executable file by the linker.

A *dynamic linker*, in contrast, inserts symbolic references to (library) modules in the executable file, and leaves these to be resolved at run time. In operating systems conforming to the Unix System-V interface [X/O90], such as SGI's Irix, DEC's Digital Unix or Sun's Solaris-2, this works as follows.

For each library module to be linked, the linker allocates a *global offset table* (GOT).[1] The GOT contains the addresses of all dynamically linked external symbols (functions and variables) referenced by the module.[2] When a program referencing such a dynamically linked module is loaded into memory for execution, the imported module is loaded (unless already resident) and a region in the new process' address space is allocated in which to map the module. The loader then initialises the module's GOT (which may require first loading other library modules referenced by the module just loaded).

A variant of this is *lazy loading*, where library modules are not actually loaded until accessed by the process. If lazy loading is used, a module's GOT is initialised at module load time with pointers to stub code. These stubs, when called, invoke the dynamic loader, which loads the referenced module and then replaces the respective entries in the GOT by the addresses of the actual variables and functions within the newly loaded module.

Dynamic linking has a number of advantages over static linking:

1. Library code is not duplicated in every executable image referencing it. This saves significant amounts of disk space (by reducing the size of executable files) and physical memory (by sharing library code between all invocations). These savings can be very substantial and have significant impact on the performance of the virtual memory system.
2. New (and presumably improved) versions of libraries can be installed and are immediately usable by client programs without requiring explicit relinking.
3. Library code which is already resident can be linked immediately, thus reducing process startup latency.

Lazy loading further reduces startup cost, at the expense of briefly stalling execution when a previously unaccessed library module requires loading. For libraries which are only occasionally used by the program, this results in an overall speedup for runs which do not access the library. However, this comes at a cost: Should the referenced library be unavailable (e.g., by having been removed since program link time) this may only become evident well into the execution of the program. Many users will prefer finding out about such error conditions at program startup time.

The main drawbacks of dynamic linking are:

1. Extra work needs to be done at process instantiation to set up the GOT. However, this overhead is easily amortised by loading much less code in average, as some libraries will already be resident.
2. If a dynamic library is (re)moved between link and load time, execution will fail. This is the main reason that Unix systems keep static linking as an option.
3. The location of a dynamically linked module in the process' address space is not known until run time, and the same module will, in general, reside at different locations in different clients' address spaces. This requires that dynamically linked libraries only contain *position-independent code*.[3] Position independence requires that all jumps must

[1] The "global offset table" is Irix terminology, Digital Unix calls it *global address table*.

[2] A further level of indirection is used when an entry references a module exporting a large number of symbols, and for efficiency reasons local symbols are also included.

[3] This is different from *relocatable code* normally produced by compilers. Relocatable code contains addresses which are relative to some yet unresolved symbols. The linker resolves these and replaces them by absolute addresses.

be PC-relative, relative to an index-register containing the base address of the module, or indirect (via the GOT).

The main cost associated with position-independent code is that of locating the GOT. *Every* exported ("public") function in the module must first locate the module's GOT. The GOT is allocated at a constant offset from the function's entry point (determined by the linker) so the function can access it using PC-relative addressing. This code must be executed at the beginning of every exported function. In addition, there is an overhead (of one cycle) for calling the function, as an indirect jump must be used, rather than jumping to an absolute address. These costs will be examined further in Section 6.

Note that the GOT is an example of *private static data*, i.e., data belonging to a module but not shared between different instantiations of the module. Any variable declared "extern" or "static" in the library and not used read-only falls into that category.

It is interesting to note that Digital Unix' *quickstart* facility [DEC94] tries to avoid some of the problems of dynamic linking by reserving a system-wide unique virtual address range for dynamically linked libraries in the Alpha's large address space. This reduces process startup costs by the ability to easily share an already loaded library without address conflicts. However, address clashes cannot be completely avoided, as there is nothing to *enforce* unique virtual address for every library — only a SASOS can give such a guarantee. Consequently, Digital Unix still needs to use position-independent code and pay the overhead this implies. However, Digital's attempt to simulate a single address space for libraries is a good indication of some of the advantages a SASOS offers.

3 Linking in single-address-space systems

A single address space simplifies many things and complicates a few; linking is no exception. Generally speaking, the single address space makes it easy to share data, and difficult not to share. The latter implies some special effort to avoid sharing of private static data.

3.1 Static linking in a SASOS

Static linking by copying all libraries into a single executable is possible in a SASOS exactly as in Unix systems. Consequently, standard static linking in Mungi has the same drawbacks as in Unix: excessive disk and memory use, as well as the requirement to re-link in order to utilise new library versions. Therefore, alternative linking schemes are desirable.

Owing to the fact that all objects in the single address space are at any time fully and uniquely identified by their unchanging address, copying library modules is unnecessary when creating an executable program. Instead, libraries can be executed *in-place*, and the linker only needs to replace references to library modules with (absolute) addresses. No position-independent code is needed, avoiding that source of efficiency loss. This scheme, called *global static linking* was proposed by Chase [Cha95] for the Opal SASOS.

Global static linking is fast and has some of the attractive features of dynamic linking in Unix systems, in particular automatic code-sharing. However, the scheme has two significant drawbacks which limit its applicability in practice:

- As it is a form of static linking, new versions of libraries cannot be used unless programs are re-linked. Note that it is not possible to update a library in-place, as this would break entrypoint addresses in all programs which linked that library. While this could be circumvented by accessing all entrypoints via a jump table, that would not help with currently executing programs which use the library — they would have to terminate prior to replacing the library. To maintain smooth operation, a new library version must be created at a different virtual address, with a naming service pointing to the latest version to be used by the linker.
- Global static linking does not allow private static data. Such data needs to reside at an address where it can be found by the library code, but that address must be different for different instantiations of the library (or the data would not be private to the invoking process).

Private static data must reside in a separate data segment, which must be set up separately for each client process. Similar to dynamic linking in Unix, the problem is how to tell the library code where to find the data segment.

Chase suggests a variation of the GOT used by Unix dynamic linkers: Each process allocates a table for each module containing the addresses of that module's private static data. A dedicated register, the *global pointer*, is loaded with the base address of the address table of the presently executing library module. The difficulty is in loading the global pointer with the correct address. Unlike the Unix case, this cannot be done by PC-relative addressing, as the offset differs between instantiations.

Chase favours (but apparently did not implement) an approach where the called function looks up the correct global-pointer value in a process-specific table (accessed via its thread descriptor, which in Opal is reachable from the stack pointer). The table is indexed by a slot number which is statically assigned to the module containing the function. Given that it is impractical to make this (per-process) table very big, this imposes serious limitations on the use of modules containing private static data — each process can only use a small number of such libraries and even then, clashes between the statically assigned slot numbers preclude importing certain combinations of library modules. Furthermore, at least two memory reads plus some arithmetic is required to obtain the global pointer value on each call across module boundaries.

3.2 Dynamic linking in a SASOS

Even if the problem with private static data is resolved (or ignored), any form of static linking retains one major drawback: A new version of a library cannot be incorporated without relinking client programs. This can only be achieved by dynamic linking.

The IBM AS/400, a SASOS which, in its former guise as System/38 [Ber80], goes back to the mid 1970's, originally *only* supported dynamic linking ("late binding" in IBM terminology) [Sol96]. Static linking ("early binding by copy") was only introduced in 1993 *for performance reasons* resulting from the calling overhead. At the same time they introduced "early binding by reference" which essentially is global static linking. According to Soltis, while there is some initialisation overhead when first accessing a bound-by-reference library module, performance of subsequent calls are "about the same" as in the bound-by-copy case. No further information could be found on how the AS/400 implements dynamic linking, but the reference to performance problems of dynamic linking seems to indicate that it does not have a particularly good solution.

Roscoe [Ros95] presents a dynamic linking scheme for Nemesis. Each invocation of a library modules has its own *state buffer*, containing private static data. Modules are accessed via *interface references*, which are instantiated from *module interfaces* when resolving the symbolic reference to the module. An interface reference points to a structure containing a pointer to an *operations table* and the state buffer. A function is invoked by an indirect jump via the operations table, and the interface reference is passed as a parameter. As a result, three memory reads are required to find the address of the function to be called.

4 Linking in Mungi

Mungi supports static linking (by copying) as well as a version of dynamic linking, designed to retain the full flexibility dynamic linking offers in Unix system while minimising run-time overheads.

During execution of a program in Mungi, a *data segment* containing private static variables is associated with every instantiation of a dynamically linked module. While executing such a module's code, its data segment's address is held in the *data segment register*.[4] The data segment also contains *module descriptors* of imported modules. A module descriptor contains pointers to all functions imported from the module, plus a pointer to the data segment of the exporting module. This is shown in Figure 1.

4.1 Initialisation of module descriptors

Module descriptors are allocated in the importing module's data segment by the linker. To initialise a module descriptor at run time, the importing module calls the exporting module's *constructor*, passing the address of the descriptor as a parameter. In order to avoid multiple instantiation of modules which are imported by several other modules (such as `libc` in Figure 1), the constructor is also passed a pointer to a table of already instantiated modules. This table is held in the main module's data segment.

After verifying that its module is not already instantiated, the constructor

[4] On the SGI Indy we use the *global pointer register* which Irix uses to point to the GOT, hence the number of registers available to the compiler does not change with respect to Irix.

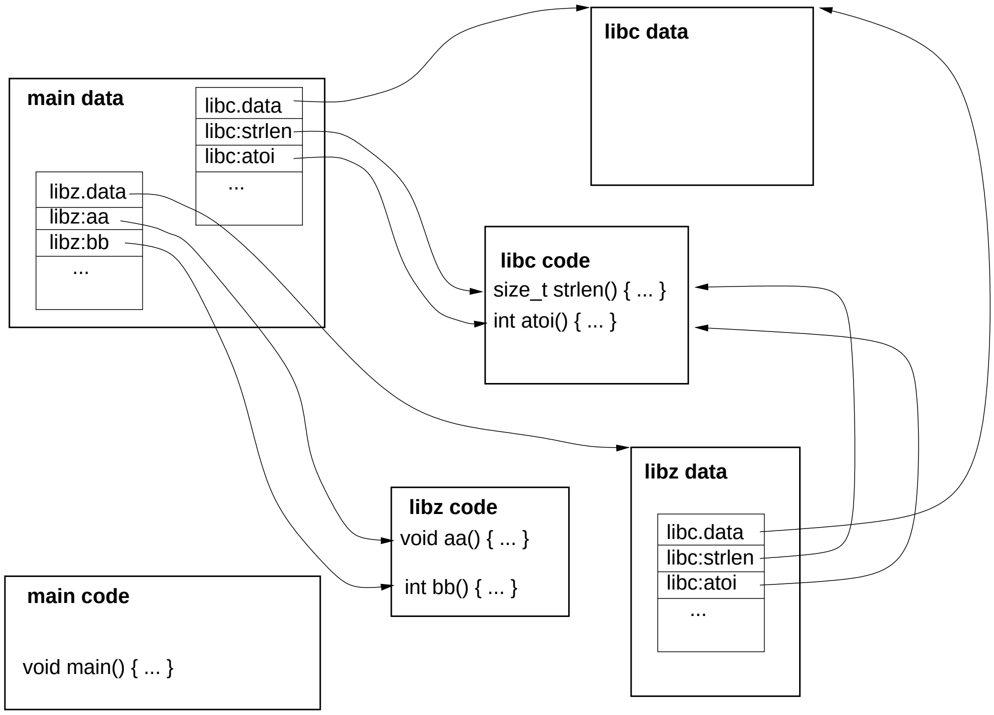

Figure 1: Memory objects (bold boxes) and module descriptors (other boxes) during execution of a dynamically linked Mungi program.

- allocates and initialises a new data segment,
- initialises the module descriptor passed by the caller, and
- updates the table of instantiated modules.

Only the second step needs to be performed for a module which has already been instantiated, in which case the address of the module's data segment is obtained from the table of instantiated modules.

The constructor is passed a third parameter, the expected size of the module descriptor. The constructor will only initialise the descriptor up to this specified size. This supports evolution of library modules — further entrypoints can be added to a library without breaking existing applications, provided that new entrypoints are added at the end.

Initialisation of the main module is somewhat different. Like a constructor, the startup code needs to allocate and initialise a data segment. In addition, the startup code must initialise the table of already instantiated modules. There is no problem with a module having startup code as well as a constructor, such a module can then be imported by other modules as well as being executed as a program.

4.2 Lazy initialisation

The first step above, allocation and initialisation of the new data segment, involves calling the constructors of all imported modules. To avoid the obvious recursion problem with cyclic reference, the constructor must at this stage mark its module's entry in the table of instantiated modules as partially initialised.

Alternatively, modules can be instantiated lazily, in analogy to "lazy loading" of library modules in Unix systems.[5] As lazy loading in Unix, this reduces task startup cost and reduces total overhead if a module is linked but not actually accessed at run time (at the cost of delaying fatal "module not found" errors until well into the execution).

[5] While there is some similarity to lazy loading, it is important to note that there is no explicit "loading" step in a SASOS — everything is already in virtual memory, and is made resident by the demand paging system on access. As far as physical memory is concerned, lazy loading is normal in a SASOS, but does not have the same drawback of delaying irrecoverable errors when libraries are removed.

A lazily initialised module has its descriptor point to initialisation stubs rather than module entry points. Each stub calls the lazy initialisation routine (which is statically linked to the module), passing it an index to its own position in the module descriptor. On the MIPS R4600, such a stub looks as follows:

```
1:   li     $reg, constIndex
2:   b      lazyInitialiser
```

The initialiser, after setting up the module's data segment, replaces the pointer to the stub by the address of the appropriate entrypoint in the library module. The stubs require an extra 8 bytes of space per entrypoint — really a negligible space overhead.

4.3 Invoking library functions

To invoke a function called `printf` from the library module `libc`, the following code is executed on the MIPS R4600:

```
1:   ld     $temp,libc_descr+\
                   printf_index($dseg)
2:   sd     $dseg,const($sp)
3:   ld     $dseg,libc_descr($dseg)
4:   jalr   $temp
5:   ld     $dseg,const($sp)
```

The first line loads the address of the `printf` function into a temporary register. This address (relative to the beginning of the data segment) is determined by the linker, and is at a constant offset from the data segment register. The next line saves the data segment register of the calling module on the stack, and line 3 sets up the segment register for the called module. Line 4 invokes the library function and line 5 restores the data segment register after its return. This code executes in 5 cycles on the R4600 (single-issue) CPU.

Note that on the R4600, a jump to a constant immediate 64-bit address (as would be used in a naive implementation of static linking) takes 7 cycles. Irix reduces this to 2–3 cycles (depending on the ability to make use of load delay slots) by using a *global pointer register* pointing to a table of entry point addresses. Comparison to the 5 cycles required for calling a dynamically linked function in Mungi indicates that the run-time overhead of dynamic linking in Mungi is very small, 2–3 cycles per call of an exported function.

The above invocation code only works if the `printf` entry is less than 64kB from the beginning of the data segment. This allows for about 8,000 exported functions (actually less, as some space is required for private static data). If the table becomes bigger, a somewhat longer code sequence is required, which takes 7 cycles to execute:

```
1:   ld     $temp,libc_interf_ptr
                   ($dseg)
2:   sd     $dseg,const($sp)
3:   ld     $tmp2,printf_index($temp)
4:   jalr   $tmp2
5:   ld     $dseg,0($temp)
6:   ld     $dseg,const($sp)
```

Note that the CPU stalls after line 3, although the compiler may be able to schedule some other instruction into the load delay slot.

We found that the biggest libraries generally used in Unix systems, `libc` and `libX11`, have between 1,000 and 1,500 entry points. (Many of these are actually internal and would not be exported if the C language provided better control over export of functions from libraries. Mungi's *module descriptors*, described in Section 5 provide such control and will therefore result in smaller interfaces for the same functionality.) We therefore expect that the shorter code sequence will almost always suffice.

5 Discussion

One remaining issue is that of modules exporting variables. For example, POSIX [POS90] specifies that the global variable `errno` is used to inform clients of the reason for the failure of an operation. This cannot be supported by Mungi's dynamic linking scheme.

It is, of course, always possible to avoid this problem by resorting to static linking — a highly unsatisfactory way out. However, exporting global variables from library modules is very bad practice, as it is not multi-threading safe. For that reason, Unix systems must break POSIX compliance if they want to support multithreaded processes. Modern Unix systems inevitably[6] use a construct like

[6] We checked Irix, Digital Unix, Solaris and Linux.

```
extern int *__errno();
#define errno (*(__errno()))
```

to declare errno when multithreading is supported. The same works in Mungi without problems.

Another issue concerns function pointers, which are used, for example, by the C qsort() utility and to implement virtual functions in C++. Function pointers can no longer be simply entrypoint addresses, as invoking a function requires loading the data segment register prior to branching to the entry point. Hence a "function pointer" must consist of an (address, global pointer) pair. This does not cause problems with portability of properly written application code, as the C standard [ISO90] makes no assumption about the size of function pointers and explicitly prohibits casts between function pointers and other pointers. Unfortunately, most compilers do not enforce this rule and, consequently, there exists plenty of non-conforming code. However, "bug-compatibility" is not a design goal of Mungi, and we therefore do not consider this a significant problem. The format change of function pointers is the only change required to standard Unix compilers to allow them to support Mungi's dynamic linking scheme.

More changes are required for linking. We decided not much was to be gained by porting a Unix linker, and instead implemented a Mungi linker from scratch. Its size is about 4,000 lines of C.

The mechanics of preparing code for execution differs somewhat between Mungi and Unix. The main reason for this is the need to generate a different calling sequence for functions exported by dynamically linked libraries. In order to do this, the assembler must know which entrypoints will be loaded from a dynamic library.

This is supported by a *module description object* (MDO) associated with each library module. The MDO is a simple text object (which is presently created manually, although tools will automate this in the near future). It contains a list of entry points exported by the module, and a list of imported modules. Figure 2 gives examples of module descriptions.

C source objects are presently compiled to assembly language by an unmodified GNU C compiler.[7] The gcc output is then processed by the GNU assembler, which we modified to generate the proper calling sequence for cross-module calls and to access private static data from the module's data segment. The assembler reads the MDO in order to identify cross-module calls and produces relocatable binary objects.

```
libc.mm               main.mm
[IMPORTS]             [IMPORTS]
                      libc.mm
[EXPORTS]             libz.mm
strlen
atoi                  [EXPORTS]
...
                      [OBJECTS]
[OBJECTS]             main.o
c1.o                  sub.o
c2.o                  ...
...
```

Figure 2: Sample module descriptions: At the left, a typical description (libc.md) of a library module libc.mm is given, while at the right the module description (main.md) of a program module main.mm is shown. Names correspond to Figure 3.

When creating a library, the Mungi linker is used to (statically) link all of the library's relocatable objects into a single library module object; the linker determines the exported entry point from the MDO. It also adds the initialisation code for the library module, as well as the initialisation stubs which invoke it and the module constructor.

When preparing an executable module, the linker reads the MDOs of all imported libraries and creates the appropriate initialisation stubs for all imported functions, and (statically) links all remaining relocatable modules into the new program module, which is then ready for execution. Unlike Unix, no "run-time linker/loader" is required, as each module has its own initialisation code. The mechanics of Mungi linking are shown in Figure 3.

The Macintosh on the PowerPC uses a similar approach to dynamic linking [App94]. Function references in MacOS are represented by a "transition vector", which consists of the entrypoint address and the address of a "table of contents" (TOC), essentially the module's data segment.[8] The TOC contains pointers to imported functions, in the form of transition vector addresses, as well as pointers to the module's static data. C language function pointers are also represented as transition vector addresses.

[7] As indicated a few paragraphs earlier, this implies that it is presently not possible to invoke function arguments outside their own module without modifications to the C source.

[8] The Macintosh terminology for modules is "fragments" but in order to avoid a proliferation of terms we will stick to calling them "modules".

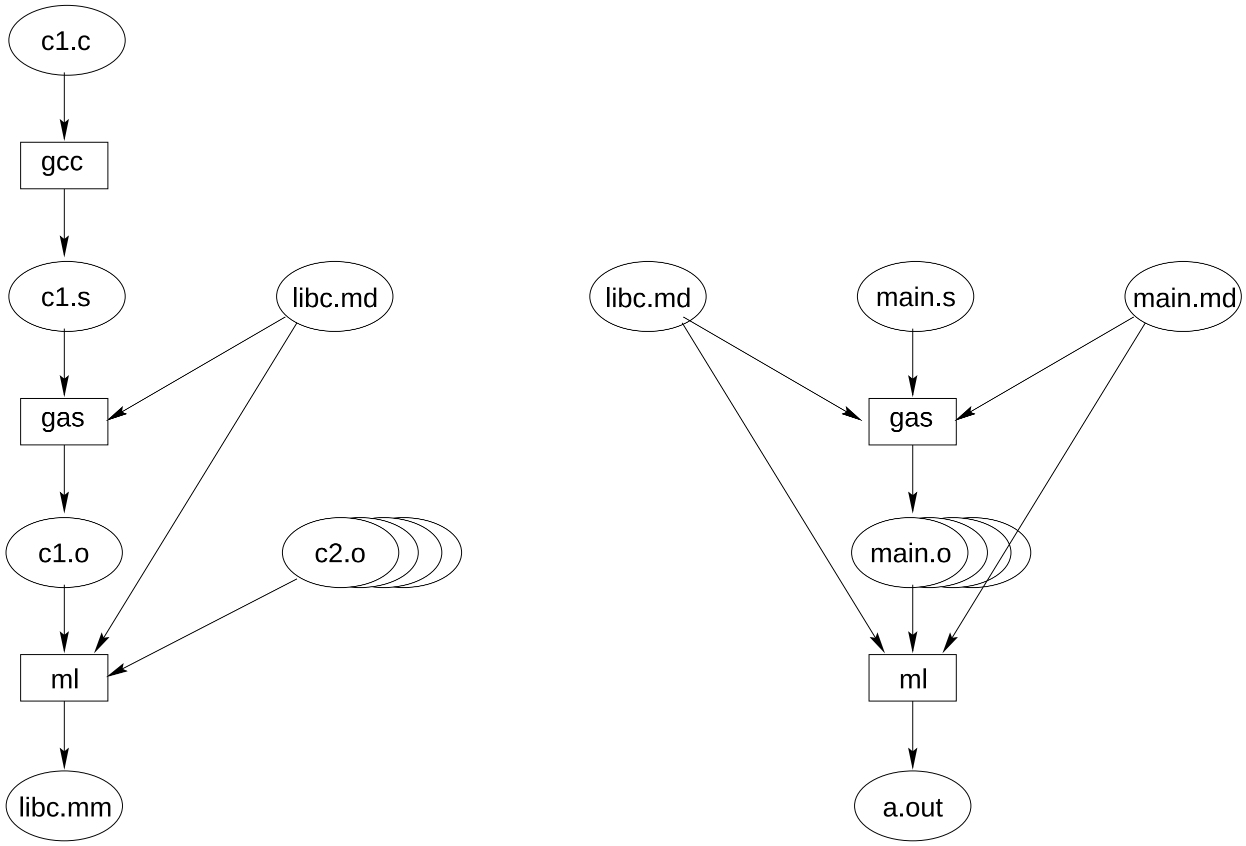

Figure 3: Dataflow for linking in Mungi. Left: A source object `c1.c` is compiled and linked with other relocatables into a dynamically linkable library object `libc.mm`. Right: A program `main.c` is compiled and linked into an executable module `a.out`.

The present module's TOC address is contained in the "table of contents register" (RTOC, equivalent to our data segment register). The invocation sequence for imported functions uses a double indirection. The caller loads the RTOC with the address of the callee's transition vector (i.e., the contents of the function pointer). The callee then loads the RTOC with the new TOC address by an indirect load through the present RTOC value. The main difference to our scheme is the extra indirection.

While the MacOS scheme is similar to ours, this does not obviate the need for position-independent code (called "pure executable code") on the Macintosh. This is because MacOS is not a SASOS, and can therefore not ensure that a dynamic library module is always linked at the same address.

6 Performance

Performance figures are shown in Table 1. The table gives execution times of a benchmark program (OO1 [CS92] run as a single process as in [HEV+98]) for static and dynamic linking under Irix 6.2 and Mungi. All runs were performed on the same hardware running either Irix or Mungi in single-user mode. Lazy loading (for Irix) and lazy initialisation (for Mungi) were turned off to make timings more consistent. (As explained earlier, lazy loading/initialisation does not normally reduce overall runtime, only start-up latency.) Irix runs used the Irix 6.2 C compiler, assembler and linker, while Mungi runs used GCC version 2.8.1, the GNU assembler version 2.8.1 (modified to support dynamic linking) and our linker.

OO1 tests operations which are considered typical for object-oriented databases: lookup and insertion of objects, and traversal of inter-object references. The specification of OO1 requires a database server running on a

	Irix/32-bit/SGI-cc			*Mungi/64-bit/GCC*		
	static	*dynamic*	*dyn/stat*	*static*	*dynamic*	*dyn/stat*
lookup	7.26(3)	8.02(3)	1.104(10)	7.568(6)	8.199(4)	1.083(3)
forward traverse	4.77(3)	5.17(4)	1.084(15)	6.013(6)	6.040(3)	1.004(6)
backward traverse	5.13(2)	5.68(4)	1.107(12)	6.976(4)	7.011(4)	1.005(1)
insert	4.61(2)	5.02(2)	1.087(10)	4.528(4)	4.755(3)	1.051(1)
total	21.7(1)	23.9(1)	1.097(12)	25.08(1)	26.00(1)	1.037(1)

Table 1: Average execution times in ms of OO1 operations (single-process version, see [HEV$^+$98]). Figures in parentheses indicate standard deviations in units of the last digit. Irix figures are for 32-bit execution using the SGI C compiler, while Mungi figures are for 64-bit executing using GCC.

machine different from the client, and also specifies that all updates are to be flushed to disk at certain points. As we are here only interested in the cost of function invocation, we ignored that part of the specification and ran everything in memory, without any I/O, and in a single process. In every other respect we followed the OO1 specification.

We implemented the "server part" of the database in a library module which is invoked by application code via normal function calls. In line with the OO1 specification, server invocations pass a function pointer to the database which the database invokes to obtain further data or pass data back to the client. The *lookup* part of the benchmark consists of 1000 server invocations, each looking up a different object, and each call invoking a client function passed as a parameter. The *forward traversal* operation consists of a single invocation of the server code, which invokes a client procedure 3,280 times (for different objects directly or indirectly linked to the object referenced in the server invocation). *Backward traversal* is similar; the actual invocation counts are different but, in average, the same as for forward traversal. The *insert* benchmark consists of 100 calls to the database, each inserting a new object into the database and in the process calling a client procedure three times. Hence the total benchmark performs about 8960 cross-module calls. The *lookup* and *insert* operations mainly exercise the index data structure (a B^+ tree in our implementation) while the *traversal* operations mostly follow internal links and thus perform mostly random accesses to the data without much use of the index structure.

This benchmark was selected as it is "tough" in the sense that it is dominated by cross-module calls to functions performing very little work. As it is the cross-module tasks which bear the dynamic linking overhead in Mungi, this test stresses the overheads of the SASOS dynamic linking scheme. A benchmark consisting of the same number of function calls, with a larger fraction of calls being inter-module, would reduce the total overheads in Mungi while leaving Irix' overheads unchanged.

The table shows that on Irix there is an average 10 % penalty for using dynamically linked libraries, while on Mungi the penalty is less than 4 %. The average overhead due to dynamic linking of an inter-module function invocation in Mungi comes to about 0.1μs or about ten cycles on the R4600. This is more than the number of extra instructions required in the calling sequence for dynamically linked code. The difference can be explained with an increased number of cache misses.

The significantly lower overhead of dynamic linking in Mungi as compared to Irix is mostly due to the fact that the Mungi scheme does not require position-independent code. The somewhat higher overhead of inter-module function invocation in Mungi (2–3 cycles more than in Irix) is more than compensated by not requiring position-independent code. Furthermore, the overhead in the Mungi scheme only applies to inter-module invocations, where in Irix inter-module calls to exported functions have the same overhead.

We also attempted to measure the initial overhead of dynamic libraries, i.e. the invocation overhead of the constructor which initialises the data segment. However, this overhead is so small that we could not measure it reliably for either Irix or Mungi. In both cases it is at most a few tens of microseconds.

While Mungi has a significantly lower penalty for dynamic linking than Irix, a seemingly disturbing observation from Table 1 is that code seems to execute generally slower under Mungi than under Irix. However, it must be kept in mind that all Mungi executions are true 64-bit, while Irix only supports 32-bit execution on the Indy. 32-Bit code is inherently faster on the MIPS

	Irix	*Mungi*		*Mungi/Irix*	
		good	*bad*	*good*	*bad*
lookup	7.367	7.169	7.452	0.973	1.012
forward traverse	5.904	6.085	6.079	1.031	1.030
backward traverse	6.796	6.992	6.991	1.029	1.029
insert	4.755	4.724	4.801	0.993	1.010
total	24.822	24.970	25.323	1.006	1.020

Table 2: Average execution times in ms of OO1 operations on for 64-bit execution of statically linked code on Irix and Mungi. Code is compiled with GCC and assembled with the SGI assembler and finally linked with the native (SGI or Mungi) linker. "Good" vs "bad" in the Mungi numbers refers to the cache friendliness of the stack alignment and the last two column give Mungi execution time normalised to Irix times.

R4600 as loading a constant address requires more cycles for 64-bit than for 32-bit addresses. The tests were also run with different compilers: Mungi benchmarks could only be compiled with GCC, as SGI's C compiler/assembler/linker toolchain does not support our dynamic linking scheme, while the GNU assembler and linker do not support Irix. Finally, different implementations of the `strcpy()` C library functions are used.

In order to eliminate the effects of 32-bit vs. 64-bit execution and differing tool chains and C libraries, we did a direct comparison, running an identically compiled version of the statically linked benchmark code in 64-bit mode on both systems. This meant compiling and assembling the code, including the C library, using GCC and the SGI assembler, and then linking it for Irix with the SGI linker, and for Mungi with our linker. As the benchmarks only time user code (no system calls are performed between time stamps, and the timer overhead is subtracted), this means that identical instructions are executed on both systems.

As Irix does not support 64-bit code on our platform, we had to patch the executable to pretend to the loader that it was a 32-bit image. This approach works under certain circumstances (as long as only a very limited set of system calls are used), but only for statically linked code. One required system call where extra work was required is `gettimeofday()`: As the format of the `timeval` struct differs between the 32-bit and the 64-bit Irix APIs, we had to use the 32-bit C library interface for this call.

In order to verify that these modifications do not affect performance of the Irix executable, we ran the "proper" 64-bit image as well as the patched one on an SGI machine supporting 64-bit executions. We found that the execution times of the two versions were identical.

The results of running the same code in 64-bit mode on Irix and Mungi are shown in Table 2. Two sets of Mungi results are presented: "good" and "bad", which differ only in the address at which the user stack is allocated. The stack address affects the results as it affects conflict misses in the Indy's data cache. The R4600 features separate on-chip instruction and data caches, both 2-way set associative and 16kB big [R4k95]. The Indy does not have secondary caches and thus has a high cache-miss penalty. The Mungi execution times recorded as "good" and "bad" in Table 2 correspond to the most and least cache friendly stack layout, respectively. They differ by about 1.5 %, which gives an indication of the impact of cache effects on the results. Irix runs used the default layout (which is cache friendly).

Comparing the Irix times with the "good" Mungi times it can be seen that they are very close. Mungi is between one and three percent faster on *lookup* and *insert*, and about three percent slower on the *traverse* benchmarks. For the total benchmark time these almost average out, with Mungi being 0.6 % slower. Given the fact that Mungi is several percent faster on some benchmarks and Irix on others, that overall difference is negligible and insignificant. They are much smaller than the performance gain of Mungi's dynamic linking scheme compared to the one used in Irix.

It is nevertheless interesting to speculate about the sources of these remaining differences. We can think of two possible reasons for the observed discrepancies in execution times: TLB misses and other cache effects.

The R4600 has a software-loaded, fully associative, 48-entry tagged TLB; each entry maps a pair of virtual pages [R4k95]. Hence the TLB can map a maximum of 96 pages, or 384kB. As the total database is about 4MB in size, and the benchmark is designed to access its contents randomly, a significant number of TLB misses is expected, particularly in the *traverse* operations.

Mungi is implemented on top of the L4 microkernel [EHL97], hence TLB misses are handled by L4. The microkernel's TLB miss handler is highly optimised and loads the TLB from a software cache [BKW94, EHL99] which is big enough to hold all page table entries required for the benchmark. However, the need to support 64-bit address spaces makes L4's TLB miss handler inherently slower than what can be achieved in a system only supporting 32-bit address spaces. Slightly slower handling of TLB misses in L4, and thus Mungi, is a likely explanation for the somewhat slower Mungi execution in the *traverse* benchmarks (which particularly exercise the TLB).

Other cache effects which could impact on the results are instruction cache conflicts. While we made certain that the same user-mode instructions are executed in both benchmarks, the layout of the executable still differs as a result of linking different system libraries and the linkers using different strategies for collecting relocatable modules. These differences can lead to different cache miss patterns. The *traverse* benchmarks contain the largest number of cross-module invocations (and hence non-local jumps) and are most likely to be affected.

7 Conclusions

In this paper we have examined the issues relating to linking in a single address space, and have compared them to linking in a Unix system. We have presented a dynamic linking scheme for Mungi and have discussed its merits and limitations. Benchmarking shows that the run-time overhead of Mungi's dynamic linking scheme is less than half of dynamic linking in Irix, in a scenario which favours Irix.

The performance advantages of a dynamic linking scheme as that used in Mungi could be obtained in traditional systems too, *if they used a global address space for dynamically-linked libraries*. As in *quickstart*, a region of the address space must be reserved for library modules, and each participating module must always be linked at the same address. Such a scheme would obviate the need for position independent code even in traditional systems. It requires a system-wide manager which hands out address regions for linking libraries, and each of a participating library's clients must follow the protocol of always linking the library at the same address. A single-address-space operating system guarantees this automatically, as in such a system *every* object is always mapped to the same virtual memory address.

8 Acknowledgements

Luke Deller gratefully acknowledges his School of Computer Science & Engineering Vacation Scholarship under which most of the work presented here was performed. The project also received support from the Australian Research Council under the Small Grants Scheme.

9 Availability

Mungi will be freely available in source form under the GNU General Public License from http://www.cse.unsw.edu.au/~disy/Mungi.html.

References

[App94] Apple Computer Inc. *Inside Macintosh: PowerPC System Software*. Addison-Wesley, 1994.

[Ber80] Viktors Berstis. Security and protection in the IBM System/38. In *Proceedings of the 7th Symposium on Computer Architecture*, pages 245–250. ACM/IEEE, May 1980.

[BKW94] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 243–253, Monterey, CA, USA, 1994. USENIX/ACM/IEEE.

[Cha95] Jeffrey S. Chase. *An Operating System Structure for Wide-Address Architectures*. PhD thesis, University of Washington, 1995. URL http://www.cs.duke.edu/-chase/research/thesis.ps.

[CLBHL92] Jeff S. Chase, Hank M. Levy, Michael Baker-Harvey, and Edward D. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, 1992. URL ftp://ftp.cs.washington.edu/tr/-1992/03/UW-CSE-92-03-02.PS.Z.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12:271–307, November 1994.

[CS92] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17:1–31, 1992.

[DEC94] Digital Equipment Corp. *DEC OSF/1 Programmer's Guide*, 1994. Order No AA-PS30C-TE.

[EHL97] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual — MIPS R4x00*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 1997. UNSW-CSE-TR-9709. Latest version available from http://www.cse.unsw.edu.au/~disy/.

[EHL99] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. Page tables for 64-bit computer systems. In *Proceedings of the 4th Australasian Computer Architecture Conference*, pages 211–226, Auckland, New Zealand, January 1999. Springer Verlag. Available from URL http://www.cse.unsw.edu.au/~disy/.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[ISO90] *International Standard, ISO/IEC 9899, Programming Languages — C*, 1990.

[POS90] *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*, 1990. IEEE Std 1003.1-1990, ISO/IEC 9945-1:1990.

[R4k95] Integrated Device Technology. *IDT79R4600 and IDT79R4700 RISC Processor Hardware User's Manual*, April 1995.

[Ros95] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. Phd thesis, University of Cambridge Computer Laboratory, April 1995. TR-376, URL http://www.cl.cam.ac.uk/ftp/papers/-reports/TR376-tr-multi-service-os.ps.gz.

[RSE+92] Stephen Russell, Alan Skea, Kevin Elphinstone, Gernot Heiser, Keith Burston, Ian Gorton, and Graham Hellestrand. Distribution + persistence = global virtual memory. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 96–99, Dourdan, France, September 1992. IEEE.

[Sol96] Frank G. Soltis. *Inside the AS/400*. Duke Press, Loveland, CO, USA, 1996.

[WM96] Tim Wilkinson and Kevin Murray. Evaluation of a distributed single address space operating system. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 494–501, Hong Kong, May 1996. IEEE.

[WSO+92] Tim Wilkinson, Tom Stiemerling, Peter E. Osmon, Ashley Saulsbury, and Paul Kelly. Angel: A proposed multiprocessor operating system kernel. In *European Workshop on Parallel Computing*, pages 316–319, Barcelona, Spain, 1992.

[X/O90] X/Open. *System V Application Binary Interface*, 3.1 edition, 1990.