Bachelor Informatica

UvA Universiteit van Amsterdam

Dynamic program loading in a shared address space

Leendert van Duijn

June 11, 2012

Supervisor(s): Raphael 'kena' Poss (UvA)

Signed: Raphael 'kena' Poss (UvA)

Abstract

The CSA group at the University of Amsterdam is developing the Microgrid, a platform for research into parallel computing performance. It offers users a single memory management unit per chip, responsible for translation from virtual to physical addresses. In the past the idea of single address space operating systems have been proposed [1], [3], which share a single pool of memory and hardware interfaces via a single address space, offering access protection via capabilities [6]. This contrasts with other processor chip designs, where the MMU is commonly used to isolate programs from each other, offering each program a private address space. Most modern programs rely on being loaded into memory, at a fixed address. This affects chip design, any address needs to be translated to its physical address. Which has to be ready in order to access a cache, requiring a memory management unit per core. Lacking this, the Microgrid loads only a single program at a time. While a single address space architecture do not obviate the need for per-core extra hardware to check capabilities, the logic to do so is cheaper than for a full-fledged translation MMU. We propose to design a loader capable of dynamic program placement onto a Microgrid, into a single shared address space. The resulting system could form the basis for an operating system for the Microgrid where a user can load programs on-the-fly. As our results show, our system can load multiple programs side by side in a running system. Next to program loading, our software also provides an API to loaded programs to access shared resources, such as the loader itself and I/O streams. Our technology could be introduced to other platforms sharing an MMU. With our contribution it has become possible to execute independent programs on the Microgrid, sharing the address space and computational resources, this enables research into real world scenarios where many programs interact with each other.

Contents

CHAPTER 1

Introduction

1.1 Context

Modern computing platforms allow the user to load programs which perform some task or computation. These platforms typically allow not just a single program to run but enable some form of sharing of the available resources. This is typically done by an Operating System, abbreviated as OS. An OS is typically a collection of functions or programs which maintains control over the resources and which ensures any client or userspace activity is granted only those limited rights and controls it needs. Users activities take place in processes. Each process has access to some randomly accessible memory and limited computing time. Along with file interaction process state is maintained and organized by the OS. The execution of a process it done through threads. A thread is a segment of program code which executes sequentially. Each activity consists of one or more threads where threads may be started at any moment during execution and they are executed parallel to each other. All runnable threads are commonly executed concurrently with each other and any other threads running on the system.

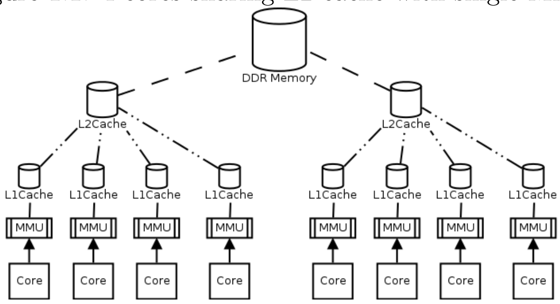
In a single chip, single core processor setup the OS would typically interleave thread execution with that of any other thread the user has initiated. This is a time sharing system where all seemingly parallel execution needs support from the OS. In a multi chip or multi core design this restriction is lifted to some extent, the architecture now allows true parallel computation although in most traditional systems the number of available cores, whether on a single chip or a local network of chips, is outnumbered by the amount of running processes. A trivial example would be the laptop this document was written on, it has 2 processing cores and is typically running about a hundred processes. Most operating systems still interleave the execution of all the processes as they would in a single core environment, with the added benefit of being able to execute several threads truly in parallel.

An example of such multi-core, multi-threaded environment is the Microgrid. The Microgrid environment is a novel platform for the parallel execution of programs with a large number of general purpose processing units capable of OS independent thread management and fast thread creation. This opens up possibilities to make programs massively multithreaded where the traditional overhead is reduced to a minimum.

1.2 Memory architecture and virtual address spaces

In a traditional multi-core chip architecture, where existing core designs previously developed for single-core chips (eg. Intel or ARM cores) are grouped together, each core is equipped with its own MMU. In this context it is assumed each process defines its own virtual address space.

Figure 1.1: 4 cores sharing L2 cache with single MMU



1.2.1 Translation

To prevent homonyms on the cache system the translation of virtual to physical addresses should happen *before* the caches are accessed. This places the MMU *in the way* of memory accesses. As can be seen in Figure 1.1. The architecture must optimize MMU access by, for example investing in complex Translation Look-aside Buffer (TLB) structures, to ensure the time to translate stays as short as possible.

In order to achieve performance the caches could use virtual addresses instead of physical addresses, removing the MMU from the critical path to the fast cache memory. This results in sharing the MMU, as done on the Microgrid. A drawback of this approach is the need to share the address space among all processes, which requires another kind of process management.

1.2.2 Isolation

In conventional systems processes are isolated from each other, by defining them to be in a private address space. This effectively prevents all homonyms. Most systems expand this system to include permissions for this private address space, regions of memory can be marked executable, writeable and readable.

Modern systems implement translation and access control in the same component, the TLB.

When processes share the MMU, using virtual addresses in caches, isolation based on separate address spaces cannot be used. The permissions defined by conventional methods lack the required power to distinct between processes which share an address space. Which voids any isolation.

In order to reintroduce isolation, where programs can only access their own memory and regions which are to be shared, capability checks have to be implemented on the core boundary. A component would, on any memory access, assure the running program has access to this memory. This check can be done in parallel with memory access, thus increasing and opportunity for parallelism, as opposed to the TLB based approach. Due to this key difference, we anticipate a significant performance increase.

The Microgrid is an example chip offering a single virtual address space with capabilities for access control. It supports virtual address translation, under control of a MMU. A Microgrid chip has a single Memory Management Unit per chip. This unit is shared among all on-chip cores. In Figure 1.2 eighth cores can be seen sharing an L2 cache per four cores, where the MMU is only accessed on DDR memory accesses.

In Figure 1.2.2 a flowchart can seen for an access to a shared cache, while at the same time checking the Capability Look-aside Buffer (CLB) whether permission should be granted.

1.3 Problem statement

Using a shared, virtual address space between different programs, while potentially beneficial to performance, requires different support from what is available in contemporary OS like Linux. In particular, the part of the OS in charge of creating processes must not only configure virtual addressing differently. It must also ensure that the segments in program executables are placed

Figure 1.2: 4 cores sharing L2 cache

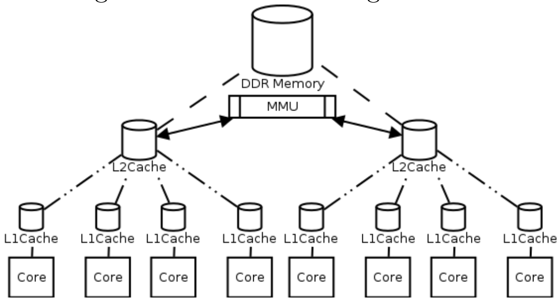
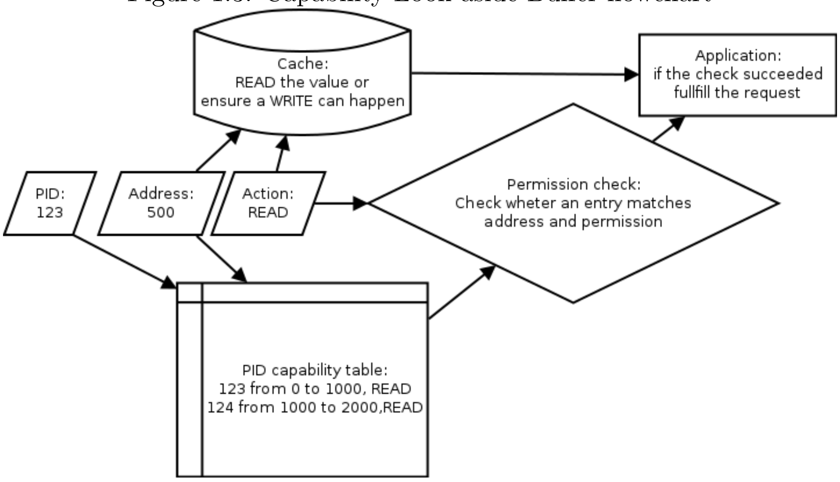


Figure 1.3: Capability Look-aside Buffer flowchart



to different regions of the address space, and configure access capabilities accordingly. We discuss these points further in Section 2.2.1. For a novel architecture like the Microgrid, this implies that existing OS code cannot be reused as-is, and new components must be developed instead.

1.4 Contribution

The goal of our research is to demonstrate the feasibility and benefits of a single virtual address space shared by multiple independent programs, such as the one provided the Microgrid architecture. To achieve this, we implement the components of an operating system for the Microgrid in charge of loading and starting programs in a shared address space. Our proposed components include:

- A memory manager which divides a 64-bit virtual address space in large regions and interacts with the "virtual memory manager"[1] to allocate and deallocate physical memory.
- A process manager which tracks which memory has been allocated per process, and provide separate API handles to each program.
- A program loader which is able to relocate ELF data sections over the shared address space on-the-fly and configure the segments access permissions specified by executable files using the hardware MMU.

We have developed our components as a library of C functions that can be embedded in an operating system, such as the one used on the Microgrid. Using our technology, we are able to show that multiple programs compiled separately can be loaded, share the virtual address

[1] Accelerated in hardware

space and interleave on micro-threaded cores without the overhead of switching address space on context switches between threads.

1.5 Prior work

A single shared address space is not unique, OPAL [1] proposes a distributed system sharing a single address space. The Mungi system [3] is another system which shares an address space among all local programs. These systems are dependent on the hardware to provide address translation on the core boundary, as discussed in Section 1.2.

While some of this existing technology is available, our analysis reveals two fundamental obstacles to reusing existing OS technology on a new architecture like the Microgrid.

Systems developed for a shared 32-bits address space, such as Opal[2], are inherently inadequate for contemporary workloads, which require gigabytes of address space per process. Other systems for 64-bits designs, like Mungi[3], have been typically specialized with custom assembly code to their native platform and ISA, which increases the cost of a software port disadvantageously.

[2] 32-bits MIPS R3000
[3] 64-bits MIPS R4600

CHAPTER 2

Problem analysis and synthesis

2.1 Platform

For this research we use the Microgrid platform. The Microgrid is a many-core architecture developed at the Computer Systems Architecture (CSA) group at the University of Amsterdam which combines hardware multithreading on each core and hardware logic to optimize the distribution of program-defined threads to multiple cores[1]. This platform is designed as a research vehicle for the exploitation of fine-grained , massive parallelism on chip.

In our work we use a software emulator of the Microgrid called MGSim. The emulator implements Microgrids with configurable hardware parameters, such as the number of cores, ISAs and cache sizes. We intend our loader program to be compatible with any Microgrid configuration. However, as a reference configuration we use a Microgrid of 128 cores with each core implementing a 64-bit DEC Alpha ISA.

2.1.1 Features

64bits address space

The virtual memory system has a 64 bits address width. All integer registers have a 64-bits size.

General purpose CPU design

The Alpha processor was designed as a true general purpose unit.

Large address space and memory pool

The potential large amount of memory addresses opens up possibilities to run more memory intensive programs concurrently. Typical computation platforms offer the possibility to run several programs either interleaved or in parallel and the main memory is shared among the many running processes.

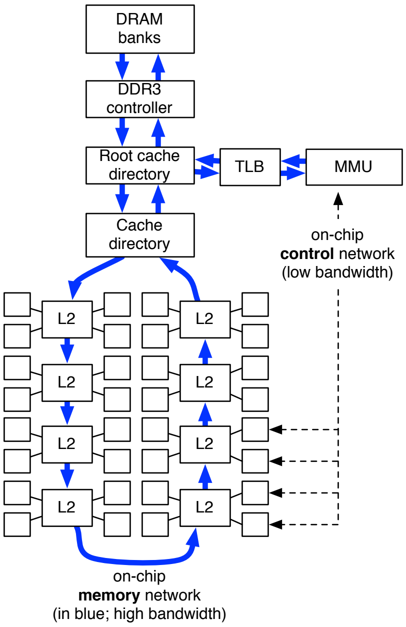
Houses a lot of parallel processing power

Microgrids can be configured with a diversity of hardware parameters, such as the number of cores and cache sizes. The default configuration in our project defines

- 128 D-RISC cores with an Alpha ISA, each with 6KB[2] of L1 cache
- Each ore has between 32 and 256 hardware threads, defined dynamically
- 128KB L2 caches, shared by groups of 4 cores

[1] http://svp-home.org/microgrids
[2] 2KB code, 4KB data

Figure 2.1: A 32 core Microgrid



Courtesy of Raphael 'kena' Poss.

- 4 DDR3-1600 external memory channels

The entire chip, consisting of multiple cores and caches, shares a MMU

As a Microgrid chip houses many cores with several caches it offers a lot of potential for parallel computation. The chip does however have a single Memory Management Unit. This component is responsible for translating virtual address into physical addresses. In a many core configuration this leads to the general issues detailed in Section 1.2. In Figure 2.1.1 a network of these cores can be seen.

Inter-component communications network

In order to facilitate communication between the processing cores and on chip components such as the Memory Management Unit the Microgrid processors have an on chip peer to peer network dedicated to component control and configuration, as such it is optimized for low bandwidth and low latency[3]. This in contrast to the memory network which is optimized for high bandwidth at the expense of high latency[4]. The networks are depicted in Figure 2.1.1.

To send Control messages on this network programs can use special instructions in an Instruction Set Architecture (ISA) extension, specific to the Microgrid. These can be in turn embedded in C program code by using inline assembly.

2.2 Overview of the loading problem

2.2.1 Requirements

The memory manager in our system will need to decide where in the available virtual address range to place a loaded programs memory image. In this context the loader will fulfill the task of an operating system. It will decide which regions of virtual memory can be used to load a program and will ensure this memory can be recycled should the process terminate.

[3] 20 pipeline cycles to control across the chip
[4] hundreds of pipeline cycles to move data across the chip

The availability of on-chip parallelism implies the loader itself could be run simultaneously on multiple cores. However allocation of a memory range and deallocation is bound to several critical sections which prevent it from being fully independent. Therefore the memory manager needs to ensure several actions are executed in an sequential manner. In order to improve performance and utilize parallel capabilities these sections should not include any code which could be safely executed in parallel.

In order to offer timing instrumentation the process manager could be equipped with an interface to time loaded processes, to offer insight into the loaded programs performance and loading overhead.

2.2.2 User input

Loading a program has several phases to go through. At the end of these steps a traditional loader would transfer control to the loaded software. Our system will then create a thread which start executing the loaded code.

The loader needs to fulfill the needs of the user. Most commonly a user will want to tell the loader what programs need to be loaded and what specific parameters or settings should be used. We propose to to do this via a configuration file, although our work can be easily extended to use an API instead. For information on the implemented configuration file see Section 2.3.2.

2.2.3 Location decisions

Programs will share the address space in a parallel fashion, thus a single arbiter, our memory manager, needs to decide where a process can be loaded. This is to prevent independent programs memory overlapping each other.

The memory manager is a sequential component in an otherwise parallel system[5]. In order to retain high performance with an increasing amount of processes a freelist could be maintained. This list points to an administrative entry which is guaranteed to be either available or a truthful indicator that there is no space whatsoever. The entry of a terminating processes can be attached to the front of the freelist ensuring that all memory ranges are accounted for at any time.

The freelist maintains constant complexity over an increasing amount of processes in the system, it does however demand locking/serialization of the requests.

2.2.4 Relocation

During the loading process an exact location is determined for the new program. This location is highly dynamic as it depends on the current memory occupation and deallocation history. When a program is requested to load, any presently loaded program affects its final location. This introduces the need for program relocation.

Relocation is the process of adjusting the program data so that it will run as intended, even though the used addresses can not be known during compilation and linking. This is done by adjusting all pointers, which point to any sort of symbol. These pointers are either absolute pointers, which need to be adjusted to correct for the offset between the initially assumed address and the address where the referenced object is loaded. Or they could be relative pointers, which may reside in instructions as an intermediate value, which need to be adjusted as sections can be placed at variable relative offsets.

The relocation can be split into two important phases. The code relocation and data relocation. The code is not always trivially relocated as pointer of different sizes exist, embedded in instructions. These relative pointers might not be large enough to hold a relocated offset. To limit the scope of this research the loader demands the compiler produce Position Independent Code (PIC). This allows our technology to skip code relocation and focus on data relocation. The required flags are elaborated in Section 3.5.3.

This leaves some data relocation entries to be processed by the loader in order to correct data pointers. For an example and explanation see Section B.1.1. These relocation corrections are

[5]this process could be paralleled to some extend by dividing the possibilities over a set of arbiters and choosing the earliest available arbiter, this introduces overhead and does not solve the need for a single (arbiter) arbiter

done as specified in [2]. This can be summarized as adding the programs base address to each pointer the compiler has flagged for correction. These pointers are full size pointers which places no extra limitations on program location. Some of these pointers could be function pointers for usage by the Position Independent Code. However the function pointers themselves are only data and can be considered as such by the relocation code, with no distinction from other data types.

2.2.5 Loading from ELF

The loader will need to load the program the user has requested. Program code can be packed and stored in many file formats. The ELF file format has been chosen for this loader as it supports some key features, needed for our loader, such as Relocation information and the Dynamic Symbol table. These features are primarily needed for relocation.

There are plenty of implementations for ELF loaders we could have used. We have reused the MGSim implementation. which loads a boot ROM into the memory upon system initialization. We used this code as an inspiration for our loader.

The ELF file format is the defacto standard for executable files. The loading of an ELF executable proceeds generally as specified in [2]. As a reference implementation we have looked into the NetBSD implementation of an ELF loader. We have consulted [4] for general information about the linking and loading processes.

Loading would result in a set of memory ranges being populated with code and data. Administrative features included in the ELF format like the program entry point and symbol data are used beyond this point though they are not necessarily part of a fully loaded program.

2.2.6 Process private memory

Programs may require arguments and environment variables that outlast the parent process, therefore they require their own storage, whose lifetime is bound to the loaded process and not the process invoking the loader. This allocated space is not required for programs which do not follow the C convention of passing command line arguments and environment variables, as such this is optional.

We propose to allocate this storage by making the ELF file format include a special section which reserves space for either arguments, environment or other custom data segments. This section is detected during loading and if present will be used to pass any argument and environment variables. If this section is absent no arguments will be passed to the program. This enables a minor speedup and memory saving for programs which are known not to use these arguments.

2.2.7 Execution

Our process manager governs the transferring control to the loaded program. Its primary tasks in this context are debugging support, timing control, and most importantly transferring partial control of the system to the fully prepared programs memory image. This is done via the Microgrid construct `sl_create` which places the program on the desired cores, taking as parameters the address to start execution, and any arguments to pass.

2.3 User control

The loader can be influenced by a user in several ways. During its preparation and compilation several settings can be tweaked as will be specified in Section 2.3.1. After the loader has been compiled and linked the remaining tweaks need to be done via configuration

Settings are done on a per process basis, settings for one program should not affect other programs in any way other than core occupation.

2.3.1 Preparation and Compilation

The behavior of the system, most specifically the loader, is controlled by both static and dynamic parameters. Static parameters are given by the C preprocessor macros and type definitions in the source code. The dynamic parameters are given via configuration file, described below in Section 2.3.2. The static parameters are:

`base_off`	`loader_api.h`
`base_progmaxsize`	`loader_api.h`
`MAXPROCS`	`elf.c`
`PRINTCORE`	`basfunc.c`
`MEMCORE`	`basfunc.c`
`NODE_BASELOCK`	`elf.c`
`minpagebits`	`basfunc.c`
`maxpagebits`	`basfunc.c`
`ROOM_ARGV`	`loader_api.h`
`ROOM_ENV`	`loader_api.h`

The `base_off` setting allows the tweaking of the base location, this so that the loader may evade certain areas of memory, this also prevents any loaded program from being placed before this address. Possible usage of this setting includes preserving some space in the address range, for future system services or inter process memory.

The `base_progmaxsize` settings is the primary means of assuring that loaded processes will not overflow their allotted memory by selecting a value fitting to the largest needed memory range.

The `MAXPROCS` setting determines the size of the array which holds the process administration blocks. Increasing `MAXPROCS` is a trade off between the memory footprint of the process manager and the upper limit on simultaneous programs.

The effective Maximum number of programs is determined by a simple calculation.

$$Minimum(MAXPROCS, \frac{Memory_{available} - base\_progmaxsize}{base\_progmaxsize})$$

This enables the user to optimize for the administrative memory footprint, per process memory needs, and desired amount of processes running at any time.

The `PRINTCORE` setting is used to pick the core used for blocking prints, as such it should not be used for performance intensive processes.

The `MEMCORE` setting is used to pick the core used for blocking memory accesses needed for atomic structure access, as such it should not be used for performance intensive processes.

The `NODE_BASELOCK` setting is used to pick the core used for PID allocation, it should not be used for performance intensive processes.

Minimum page size is set by `minpagebits`, this is important should the target platform change.

Maximum page size is set by `maxpagebits`, this is important should the target platform change.

The identifier string for argument ELF section is determined by `ROOM_ARGV`, in the same manner `ROOM_ENV` does so for the environment ELF section, specifics can be found in Section 3.5.1.

2.3.2 Dynamic configuration parameters

The user can guide the loader by using a configuration file. The configuration file contains the following parameters:

- Filename of the ELF file (string)
- Arguments, can be an empty line (newline separated strings)
- Environment, can be an empty line (newline separated strings)

- Verbose, "true" or a numerical value"
- Exclusive, "true" or "false" (Optional, defaults to false)
- Core_start, numerical core number, (Optional)
- Core_size, numerical number of cores, (Optional, defaults to 1)

The file name setting is obligatory, the arguments and environment will need representation in the configuration although these may be represented by an empty line. The remainder of the settings are optional.

Settings Enumeration

Next to the configuration file, software can also offer flags as parameters to the loader API. These flags can be a combination of the following enumeration OR'ed together. The enumeration is defined in `loader_api.h`.

- `e_noprogname`, if true the `argv[0]` will not be passed based on the ELF file.
- `e_timeit`, if true print timing information on termination of a loaded process.
- `e_exclusive`, if true `sl_exclusive` is passed to the MGSim.

Our program manager offers an option to fully reserve the cores for the given program, blocking any core sharing. This setting is set via the `e_exclusive` flag. This flag is intended for applications like a file system driver, which could be used in a micro kernel.

Due to the serializing effects of this option and the absence of intelligent core selection this is disabled by default. By default the program will be started on the core invoking the loader call. A possible side-effect of the `e_exclusive` option could be deadlock, if a program is started on a reserved core. The program would prevent any calls dependent on that core, including those it needs to terminate. A check could be implemented into the process manager prior to accepting this flag, in a performance critical section.

2.4 Implementation considerations and reflection

A perfect design is rare, as not all optimum settings are fully compatible.

2.4.1 Placement of programs in the shared address space

Location

A program needs a location, which cannot be easily changed after the program has started. The run time of a program may be unbounded and as such a single program introduces fragmentation of the memory space. This is largely an allocation problem where prior to execution exact space requirements may not be available.

When loading a program, care has to be taken to ensure no programs are given overlapping address spaces[6].

During the debugging of any collection of programs one would like to know which program is responsible for certain instructions, problems or memory usage. In order to trace a specific memory location to a program we would need some sort of standard procedure.

[6]All programs share the same address space, but to their knowledge the subspace they inhabit is a traditional address space

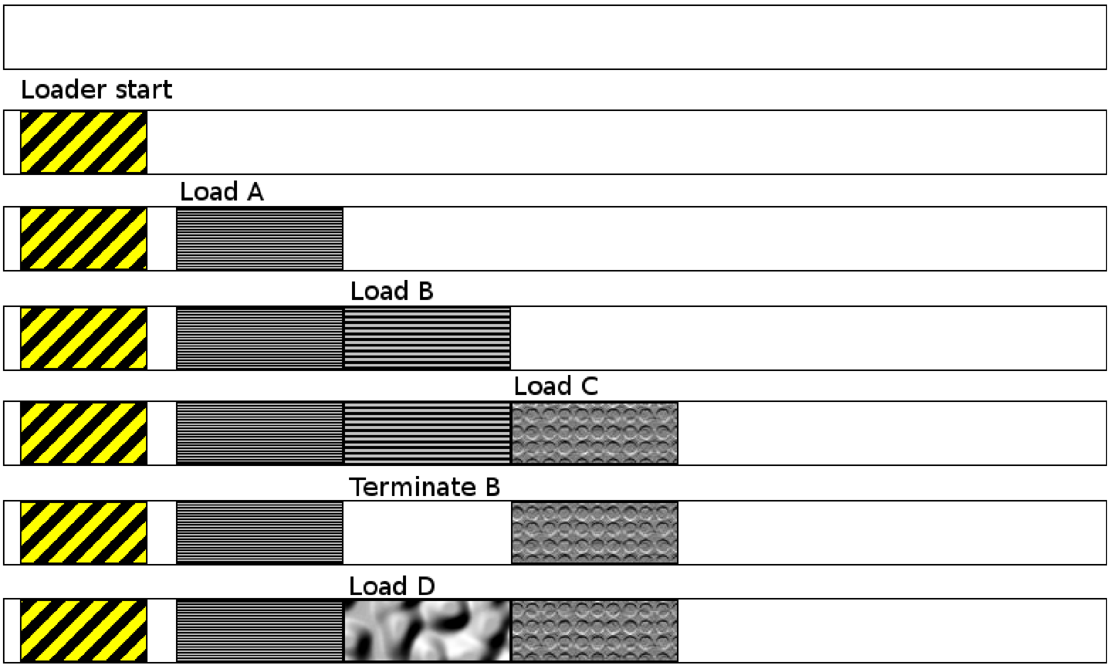
Solution

As our loader is designed with a 64bits address space in mind we have adopted a formula for base address calculation in which a process is given a base address based on its identifier and a predetermined size. This size is the upper limit for any loaded process sub address space. This size can be changed prior to loader compilation.

$$Base = Base_{Global} + (Id_{Process} * Size_{maximumsubspace})$$

This enables us to efficiently determine a base address for a process by a simple calculation based on the Process Identifier.

Figure 2.2: Memory occupancy evolution



An indication of the resulting memory layout is shown in Figure 2.4.1, where the loader is shown to populate several regions of memory after each noted event.

By selecting a value for $Size_{maximumsubspace}$ the maximum amount of memory one loaded process can legally address is determined, in order to prevent overflow we have opted to use a default value of 2^{50} which offers each process more addressable memory than current platforms offer as randomly accessible memory. This value can be tweaked to suit any specific situation as defined in Section 2.3.1

This memory layout has some consequences which limit either the number of processes or their available memory. By using a 16-bits PID and having virtual addresses 32-bits wide, you obtain an upper limit of 64KB of memory per process. The 64-bits systems known as the x86-64 architecture have only 48-bits addresses, which combined with a 16-bits PID this would limit each process to 4GB of memory space. With a 16-bits PID our system could ideally offer each process 48-bits, 281.5TB of address space. This does demand the target platform offer a true 64-bits virtual memory system.

2.4.2 Traceability of problems

As a system runs an increasingly large number of programs, problems tend to become more complex. We propose to use a memory allocation system which allows identification of the owning process by simple calculation, requiring an address and static configuration data. Our proposed method is computationally attractive as it does not require any other data or look-up.

In order to trace errors we can determine ownership by calculating the Id of the memory based on the inverse of our location formula.

$$Id = rounddown((Address - Base_{Global})/Size_{maximumsubspace})$$

2.4.3 Relocation

The loaded program is loaded to an a priory unknown location so the loader has to finish the relocation process. This is done by using relocation information stored in the section headers. The loader parses the section data. [5]

2.5 API

The loaded programs currently lack a full C library which offers functions such as `fopen`, `fprintf` and many others. The incompatibility with our loader stems from the preexisting C runtime used to link and locate the library making false assumptions about the system and its no longer private address space.

In order to offer the loaded applications some of the missing functionality, more importantly, access to several loader related functions an Application Programming Interface (API) structure is defined, which holds pointers to functions as offered by the loader. This API is used to interact with and guide the loader, and other system services such as console I/O.

2.6 Summary

The loader loads initial programs via a configuration file, it allocates the needed memory and after programs terminate cleans everything up. All loaded programs are offered access to the systems API which can be used to spawn other programs.

CHAPTER 3

Implementation

3.1 Assumptions and constraints

In the implementation process of the loader several assumptions had to be made.

- Availability of a working C compiler, `slc`
- Availability of a working linker, `slc`
- ELF file format output for the linker
- Availability of the `-fpic` `-fPIC` and `-shared` flags
- Correctness of loaded code
- Relocateability of loaded code
- Loaded code will not try to harm other loaded programs

3.1.1 C compiler

The CSA group provides a C toolchain to program the Microgrid, fully compatible with the MGSim platform that we target in our work. This toolchain comprises of a C compiler which supports concurrency management extensions to C called SL. The compiler itself is called '`slc`' and uses GNU C as a back-end to produce code, as such slc is nearly fully compatible with C99 as supported by GNU C.

3.1.2 The linker

The process of grouping object code, produced by the compiler, together with libraries to form an autonomous executable file file falls under the responsibility of a linker program, commonly called '`ld`'.

Like `gcc`, the command slc can also be used to drive `ld`, `ld` accepts parameters to tune the linking process, such as whether to include relocation information in the final executable. We explore these in Section 3.1.3.

3.1.3 Flags for linker and compiler

For any loadable program extra restraints exist, the C compiler and linker need to follow specific rules in order to maintain full relocatability and functional correctness. These flags are the `-fpic` `-fPIC` and `-shared` flags. The `-f` flags indicate to the compiler that any executable code needs to be fully relocatable. The `-shared` flags indicated to the linker all data references might be relocated prior to execution and as such administration to support should be included. These flags are needed to compile any program that should reliably run within our loader.

3.2 API

The API is implemented in our loader via an interface akin to a UNIX syscall table, where loaded program are offered a pointer from which, at known offsets certain function pointers are stored. These functions are to be abstracted by the C standard library as they represent system calls directly into the loader which, in this area, could be seen as the operating system.

The loader passes the pointer to all loaded programs which can be used to accessed loader functions, this is done by passing the pointer to the single struct as a parameter in a register. Figure 3.2 gives an overview of the function pointers available for the API.

Figure 3.1: API functions

Name *First argument*	*Description* *Second argument*	*Return value* *Leftover arguments*
`spawn` const char* `ELFFileName`	spawn a program enum `settings`	int 0 on success int `argc`, char **`argv`, char *`env`
`print_string` const char* `PrintedString`	prints in an orderly fashion (blocking) int `WhichOutput`	None
`print_int` int `PrintedNumber`	prints an integer in an orderly fashion (blocking) int `WhichOutput`	None
`print_pointer` void* `PrintedValue`	prints pointer in an orderly fashion (blocking) int `WhichOutput`	None
`load_fromconf` const char* `ConfigFileName`	loads a program from config file	int 0 on success
`load_fromconf_fd` int `FileDescriptor`	loads a program from config from an open file	int 0 on success
`load_fromparam` struct admin_s* `PreparedStruct`	loads from structure with parameters	int 0 on success
`breakpoint` int `IdForPrinting`	loader break point for program, prints and breaks const char* `ForPrinting`	enum `WhoHandledIt`

3.3 Platform dependency

We depend on existing services of the Microgrid for several key functions and constructs. These would need to be replaced if any other platform where to be targeted.

- `sl_create`, for creating the program stack and core allocation, accepting parameters for core placement, exclusive core access and entry point
- `sl_detach`, letting the loader detach from a loaded program logically tied to `sl_create`.
- `mgsim_control`, for sending messages to the MMU concerning range allocation and deallocation, accepting parameters for address, size, permissions and PID
- `mgsim_control`, for sending a message to the simulator concerning a breakpoint

3.4 Configuration

A simple scanner parses keyword value pairs, which are then terminated by a blank line at which point the arguments can be specified. These arguments will be passed as the traditionally called argv, which can only be performed if the necessary space is reserved in a section as detailed in Section 3.5.1. These newline separated arguments are terminated by a blank line. After this blank line the environment variables are written, separated by newlines. The environment variables should be in the form a=b. The environment variables are terminated by a blank line after which any remaining data is left untouched.

3.5 ELF loading

3.5.1 Special symbols

The loader searches for some special symbols which it can use to store and pass arguments to programs in an unobtrusive manner. These symbols are generated by including an extra object file during compilation of every loadable program. They have global scope, global to the compiled program, other loaded programs do not see them. The symbols are detected when parsing the dynamic symbol table and include both the size and the unrelocated location. After correcting for relocation the symbol location is stored in the programs administration for later use. The symbols are recognized by their names, these can be changed by altering the definition of `ROOM_ARGV` or `ROOM_ENV` in the file `loader_api.h` and the related C source file which would be either `argroom.c` or `envroom.c`. The latter also permits the size to be modified in order to accommodate for the anticipated amount of arguments.

Size constraints and guidelines

The space both the argument and environment objects require depends on the anticipated input, in order to calculate the most efficient size these formula should be used:

$$Size_{Env} = 1 + \sum_{i \in environment} (1 + strlen(i))$$

$$Size_{Args} = 8 * (Argc + 1) + \sum_{i \in argv} (1 + strlen(i))$$

The space needed for the arguments considers the storage for the `argv` array, the suggested amount of space reserved for environment includes the final null byte. These storage locations are only related in concept and implementation. They are fully independent so one may choose to include any combination of sizes.

In the situation where insufficient space is available for the arguments the loader will print a warning message, unless the verbosity is set to silent. It will then check the same for the environment variables. It will still try to execute the program even if these checks both fail by passing null pointers and an `argc` of zero to indicate no arguments could be passed. It is left up to the developer of the loaded program to decide whether it can successfully execute in their absence.

3.5.2 Algorithm

The ELF file is read into memory where a simple algorithm is followed.

Load the file into memory
if Inspection of the header fails **then**
  Abort the loading
end if
Locate the program headers
Scan the program headers for the base address
$PID \leftarrow Available_{PID}$
$Base \leftarrow Align(basecalculation(PID))$
for all $Header \in Programheaders$ **do**
  $Dest \leftarrow Header.Location + Base$
  $Src \leftarrow Header.FileOffset$
  $Dest[0 : Header.InFileSize] = File[Src : Header.InFileSize]$

$Dest[Header.InFileSize : Header.Memorysize] \leftarrow \{0, \ldots\}$
end for
$Relocations \leftarrow \{\}$
$SymbolTable \leftarrow 0$
Locate the section headers
for all $Header \in SectionHeaders$ **do**
if $Header.type = Relocation$ **then**
$Relocations \leftarrow Relocations, Header$
end if
if $Header.type = SymbolTable$ **then**
$SymbolTable \leftarrow Header$
end if
end for
for all $Symbol \in SymbolTable$ **do**
if $Symbol.Name = ArgumentRoomName$ **then**
$Argroom \leftarrow Symbol$
end if
if $Symbol.Name = EnvironmentRoomName$ **then**
$Envroom \leftarrow Symbol$
end if
end for
for all $Table \in Relocations$ **do**
for all $Entry \in Table$ **do**
$Loc \leftarrow Base + SymbolTable[Entry.Symbol].Offset$
$Value \leftarrow SymbolTable[Entry.Symbol].Value$
$Addend \leftarrow SymbolTable[Entry.Symbol].Addend$
$*Loc \leftarrow Value + Addend + Base$
end for
end for
if Argroom **then**
Copy the Arguments into Argroom
end if
if Envroom **then**
Copy the Environment into Envroom
end if
Call a new thread with the Entrypoint, Arguments and Environment

The loader optionally includes a verbose set of print statements useful for debugging purposes. This can be disabled for performance reasons by changing a macro definition or disabled at runtime by passing a verbosity setting to the loader.

The loader is guided by a configuration file which describes what program should be called with optional arguments and program specific settings. This configuration file is covered in detail in Section 2.3.2. The loader follows two simple algorithms for parsing.

Reading key value pairs,

$Key \leftarrow$ ""
$Value \leftarrow$ ""
$Buffer \leftarrow$ ""
while $X \leftarrow NextCharacter$ **do**
if $X =' =' $ **then**
$Key \leftarrow Buffer$
if $X = NEWLINE$ **then**
if $Key =$ "" **then**
Break While
end if
$Value \leftarrow Buffer$
Call ParseSetting(Key, Value)
else
$Buffer \leftarrow Buffer, X$
end if
end if
end while

At this point all settings have been parsed, the programs file name is known and the settings have been terminated with an empty line. At this point the command line arguments can be set.

$Argc \leftarrow 1$
$Argv[0] \leftarrow ELFFilename$
for all $X \in ToRead$ **do**
  if $X = MEWLINE$ **then**
    if $Argv[Argc][0] = NULLBYTE$ **then**
      Goto Done
    end if
    $Argc \leftarrow Argc + 1$
  end if
  $Argv[Argc] \leftarrow Argv[Argc], X$
end for

The same is the done for the Environment substituting Argc and Argv for EnvC and Evnp.

3.5.3 Program limitations and requirements

Some compilation flags and settings are explicitly required in order to reliably load a program.

- `-fpic`, to tell the compiler to generate position independent code.
- `-fPIC`, to tell the compiler to generate position independent code which could be needed for compilation to SPARC machines[1].
- `-shared`
- `crt_fun.c`
- `-nostdlib`

These flags tell `slc` to compile position independent code, to include data relocation information.

By default `slc` links programs with a simple bootloader and operating system, which assume full control of the chip to that program. The flag `-nostdlib` prevents this automatic bundling, and allows us to link our own initialization code. Which simply calls the 'main' program of the program.

3.6 Spawning an initial program

The loader initial program is loaded by passing arguments which are parsed as configuration files, loading them in sequence on either the default core or the specified cores. These files should adhere to the format as specified in Section 2.3.2. Several examples are included in Section B.2

The loader in this research will behave in ways like loaders normally found in userspace within an operating system environment. As such it will not offer full control of the system as a bootloader would, it will run in userspace and it resides in virtual memory. It is however designed for a system lacking a full operating system, it therefore does not yet implement some, of the more complex features that most operating systems do offer. The most prominent missing features include program exception handling, a loaded program which performs illegal operations is likely to terminate the entire loader and all loaded programs.

The loader lacks dynamic library support, programs lack a way to share libraries dynamically which could increase redundancy as more and more statically linked programs include their own copy of common code.

Our loader does not treat debugging information in any special way, and as such might require expansion if a debugger is introduced to the system.

[1] it concerns Global Offset Table (GOT) maximum size

3.7 In-program Loader calls

In order to allow more complex program structure we offer programs an API through which they can invoke loader functions to spawn further programs recursively.

Another function is the function `load_fromparam`, which allows loading with customized settings without the need for creating writing to a configuration file. in Figure 3.7 a simple program, written in C is shown. The simple example will load a program `a.out` from the current directory. As the program terminates the loaded software may continue to run, as any dependencies on the 'parent' process have been resolved and if required, copied to the newly loaded programs memory.

Figure 3.2: A program spawning a.out

```
//Required for API structure
#include <loader_api.h>

int main(int argc, char **argv, char *env, struct admin_s *api){
  /*Call while not verbose, but do print timing information*/
  cld.verbose=0;
  cld.settings = e_timeit;

  //Specify which to load the program to
  cld.core_start = 64;
  cld.core_size = 1;

  //The file to be loaded
  cld.fname = "a.out";

  //Setup any arguments and environment
  cld.argc = 0;
  cld.argv = NULL;
  cld.envp = "environment=variable\0\0";

  //Call the program
  api->load_fromparam(&cld,0);
  //At this point in code the call has been made and this program may terminate
  return 0;
}
```

In order to compile this program the command `slc -fPIC -fpic -shared -nostdlib crt_fun.o Filename.c -o spawner` could be used, substituting the `Filename` with the path to the input file.

CHAPTER 4

Progress report

As our research progressed we ran into some obstacles we did not foresee.

4.1 Milestones

In order to guard completion of our research we selected several milestones, each of which designed to be an improvement over those before.

4.1.1 Planned milestones

- A loader for single program, this milestone was picked due to the importance of the loader component of our system. Without being able to load a simple program the rest of our research would be impossible. After reaching this milestone this simple program formed the basis of the ELF loader of our system.
- A loader for multiple programs, as our research progressed we introduced those components needed for the loading of more than one single program. These components, though not of much use on their own, cooperate with our ELF loader to allow the loading of programs which can be determined at runtime. This collection of components is the core of our research.
- A loader with in program spawn function, as the system progressed the need for dynamic loading increased. In order to allow loaded software to load other programs we introduced an API which can be utilized by software such as shells.
- A loader with Input and Output redirection, as the number of loaded processes increases the need for regulated input and output arises. Processes are offered functionality to print to both the standard output and standard error streams of the system. This demonstrates the possibility for the system to offer true IO via our API.

4.1.2 Unexpected roadblocks

- Data relocation, we initially assumed that our toolchain would generate code which would handle any data relocation at run-time. The linker, even when passed the position independent flags, picks base addresses for segments to be loaded at. These addresses are then used for all (link-time) data relocation, resulting in an unrelocateable file. The linker had to be called with an extra flag, telling it to include all data relocation information, as noted in Section B.1.1.
- Missing functions, some of the functionality required for the allocation of memory were not readily available. The interface we used did not offer dynamic allocation of executable memory, we have had to temporarily modify the protections settings of the simulator in order to complete our research.

- Libc conflicts, loading an existing libc leads to crashes. The toolchain we used during our research has been adopted to initialize the full system on program start up, this is done by including a simple bootloader in compiled programs. This bootloader had to be disabled by passing a flag to the compiler and linker as noted in Section 3.5.3.
- Runtime cleanup, programs were not being cleaned due to thread termination. Our initial replacement bootloader for the loaded programs included both some initialization and cleanup code. An unwanted side-effect of the cleanup code was the termination of the threads the program was running. The termination effectively meant that any program which was loaded could not be cleaned as the process manager was never signalled that a program was finished. The solution to this problem was replacement of the overzealous bootloader by one that simply called the main function and returned its return value.

4.1.3 Reached milestones

As our research has progressed, so have the intended milestones, resulting in a functional system capable of running programs in parallel and reusing any shared resource.

As we have reached our intended milestones we need to acknowledge some other achievements.

- The loading of a relocatable program, which requires specific data relocation.
- We have implemented loading based on a simple configuration file and in program API calls. This enables users to load programs either by hand or automatically. These methods of loading also let the user start their new program on a specific core.

4.2 Security

As the memory manager is focused on sharing an address space between programs some assumptions where made as seen in Section 6.2.2.

The platform could be extended with 'capabilities', such as those suggested in [6], to prevent rogue threads accessing information they should not, though this does require special hardware. For this we envision the introduction of the CLB discussed in Section 1.2.2 as future work.

CHAPTER 5

Experiments

5.1 Testing

During the development of our system several programs were written to test nominal behavior. These programs are designed to make use of several features of the ELF file format which could break on loader malfunction.

5.1.1 Relocation

The program `tinyex.c` prints strings which are globally defined in an array. This array of string pointers requires runtime relocation to ensure they point to the relocated string data. These are full size absolute pointers and as such corrected by simple addition of the program base offset which is unknown at compile time.

Symptoms of malfunction for this program would include illegal memory accesses and the attempted printing of non string data.

5.2 Benchmarking results

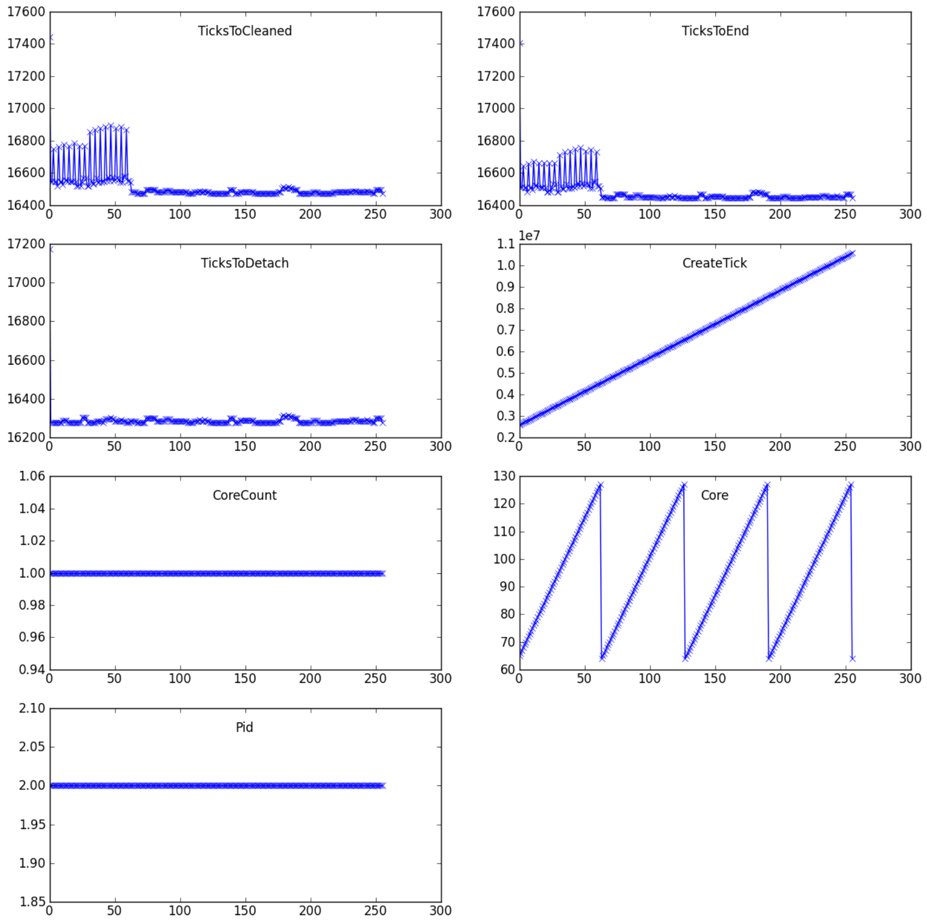
During the benchmarking of our system we have run several programs. Each run of these programs started a number of programs with specific behavior and timing information enabled. The bulk of these programs was started by our test program called `sparmy`. The resulting output and error streams were stored to file, for offline processing. The timing data is printed prior to program termination in a format designed for simple parsing by a python script which we tasked with visualization. The specific format is `<Clocks>%d,%d,%d,%lu,%lu,%lu,%lu</Clocks>` where the following items appear in this order:

`PID`	process identifier
`Core_start`	core the program was started on
`Core_size`	number of allocated cores for this program
`CreateTick`	absolute value of Clock at creation, reference tick for elapsed ticks
`TicksToDetach`	number of elapsed ticks to the actual program entry
`TicksToEnd`	number of ticks elapsed to the programs return
`TicksToCleaned`	number of ticks elapsed to program resources being deallocated

These figures show the performance of several phases of execution, the graphs show the number of 'ticks' between key events. The number of 'ticks' is based on the C function `clock()` which on our platform has a ratio of 1:1 of pipeline cycles to 'ticks'. With our core speed of 1GHz, equals 1 nanosecond per 'tick'. The `TicksToCleaned`, `TicksToDetach` and `TicksToEnd` show the number of 'ticks' since their PID allocation on the Y (vertical) axis. The moment of PID allocation is shown in CreateTick.

The Pid graph shows the numerical PID, the CoreCount graph shows the allocated number of cores and the Core graph shows the first allocated core.

Figure 5.1: Program spawning empty programs



All the graphs share an identical X (horizontal) axis, which depicts the sample number.

These measurements where done on the default MGSim configuration.

Figure 5.1 was gathered from running our test program which called 256 programs. It shows the timing of programs, were compiled from C code, which on the call to main, directly returns 0. Giving us an indication of performance of the loader, process manager, memory manager and printing of timing data.

As can be seen, initial performance shows some irregularities, which weaken as more than 64 programs have been loaded. The initial peaks in performance can be observed more closely in Figure 5.2, where it can be seen that after the highest core has been used, which is core 127, the performance stabilizes. We speculate this behavior is due initialization of cores and L2 caches. The latter due the recurring spike each fourth core.

As we suspect the peaks in our graph, the periodic peaks in executed cycles, to be related to L2 cache initialization we disabled it for several tests. We did this by passing the `-m rbm128` flag to `slr`. These tests are marked with `_noL2`.

in Figure 5.3 we show a comparison between the default settings, and those obtained via the `-m rbm128` flag. Due to the absence of L2 cache we decrease general performance, as any memory access unresolved by the L1 cache now requires access to external memory. This penalty reduces our timed program performance by around 2400 ticks per run, as opposed to the periodic peak

Figure 5.2: Program spawning empty programs, closeup

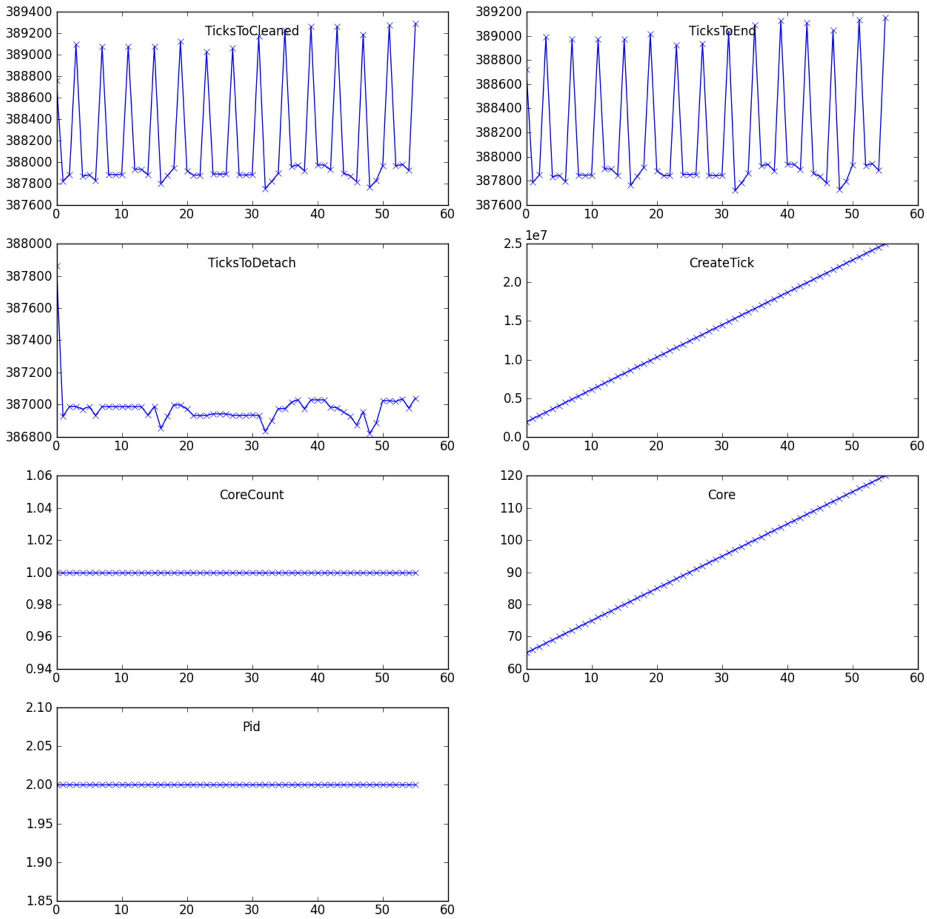


Figure 5.3: Comparison of empty programs, L2

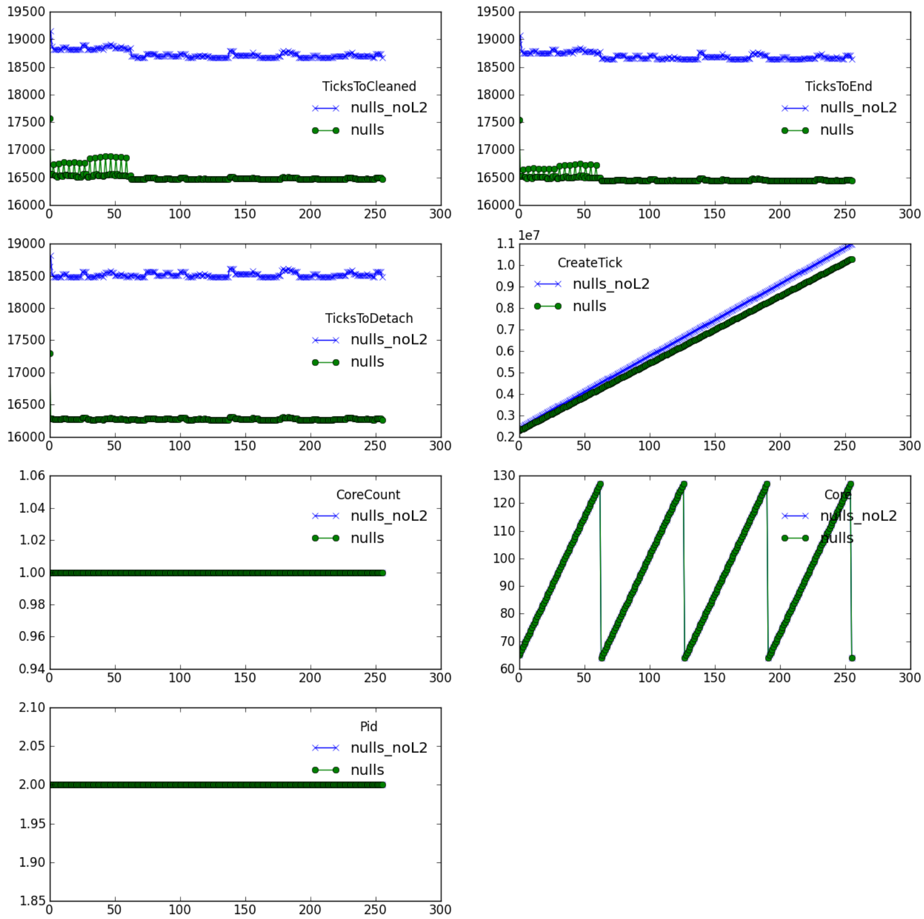


Figure 5.4: Comparison of empty programs, L2, closeup

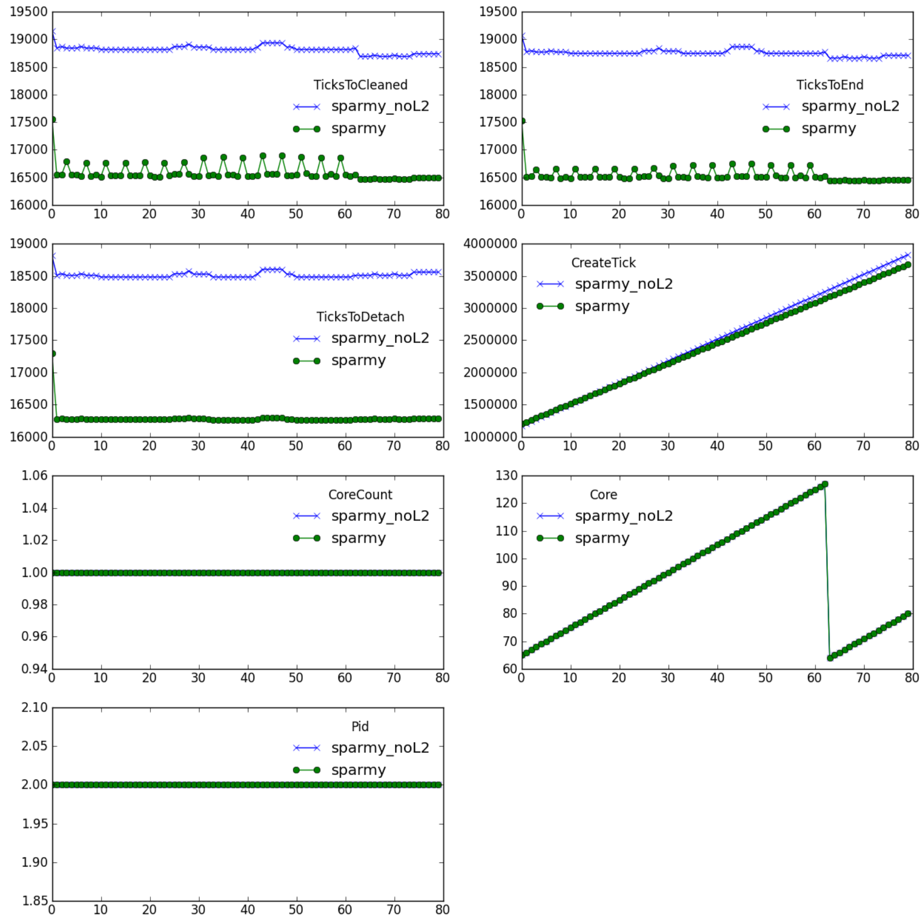
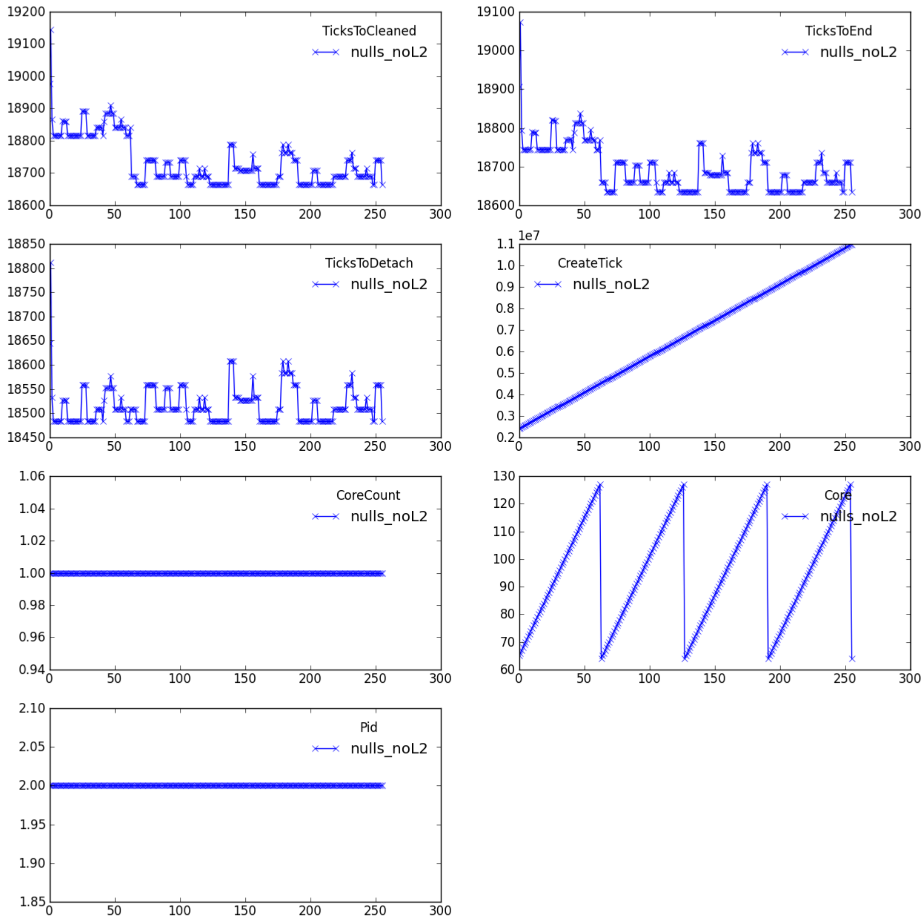


Figure 5.5: Program spawning empty programs, No L2 Cache



of no more than 300 cycles.

Our test does confirms that the presence, or absence of the L2 cache, is vital to the periodic dips in performance. This can be seen in Figure 5.4 and 5.3, where the initial 64 programs, running on the L2 deprived system, have nearly identical performance.

As can be seen in Figure 5.5 and 5.6 the initial performance hit still exists when leaving out the L2 cache entirely. We suspect the minor abnormalities, roughly a hundred ticks, are due to memory allocation which requires exclusive access to a designated core. If we take these into account we are left with an initialization penalty, which without the L2 cache is fifty to hundred ticks and, with L2 cache integrated into the system, around 30 to 400 ticks.

These findings support our theory that these periodic performance dips are related to the initialization of the L2 cache, they do however not warrant removal of the L2 cache, considering the increased performance of localized memory access.

In Figure 5.7 the creation of the tested programs shows linear performance. The program used for this measurement was designed to have a stable runtime which exceeds the overhead observed for empty programs. In Figure 5.8 and 5.9 we observe the same trends.

Figure 5.6: Program spawning empty programs, No L2 Cache, closeup

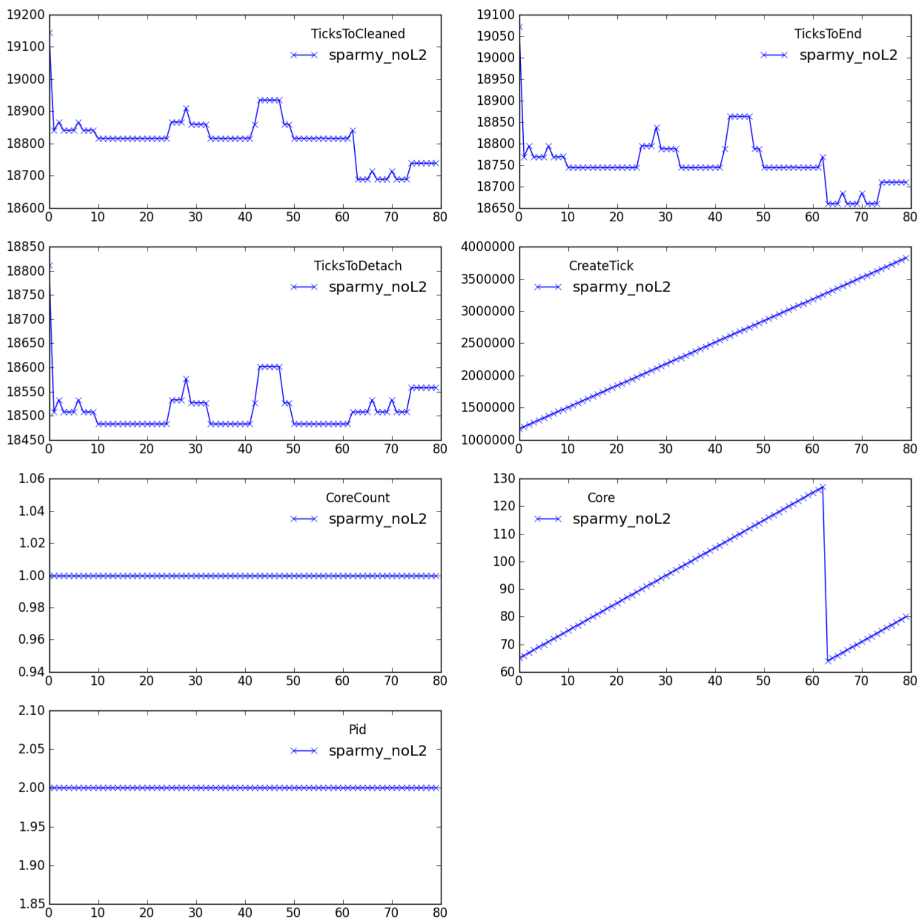


Figure 5.7: Program spawning counters

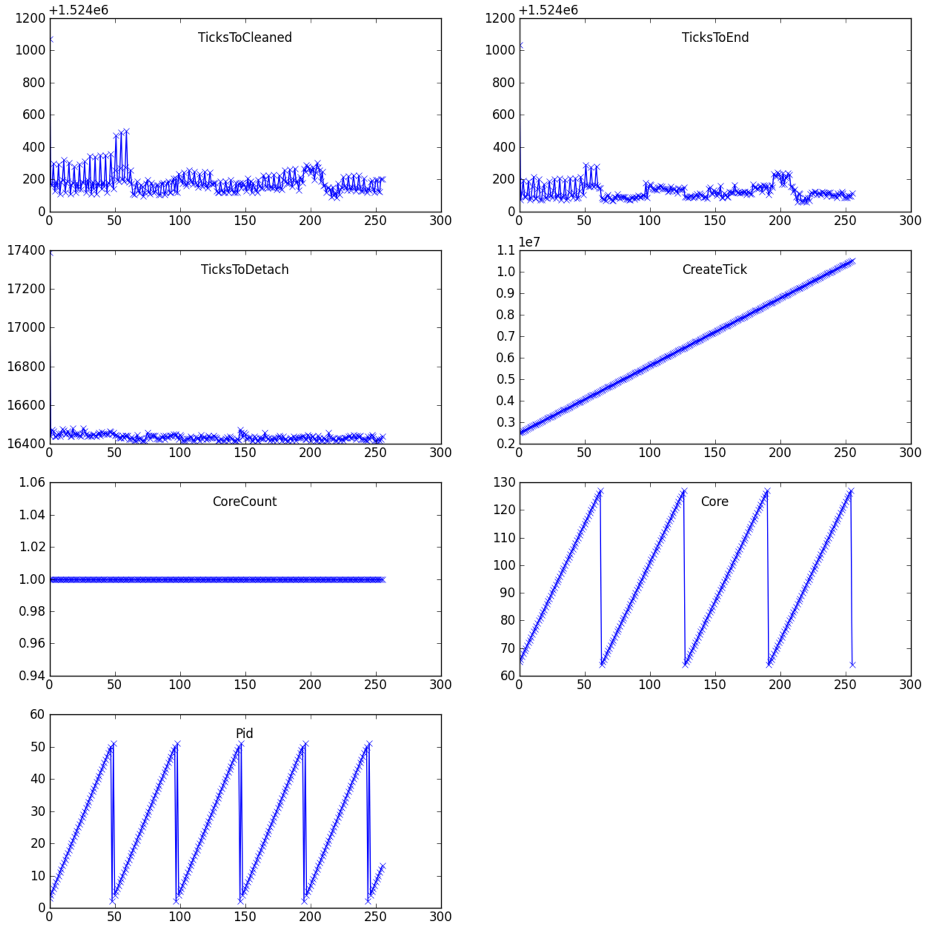


Figure 5.8: Performance of counting programs, no L2 cache

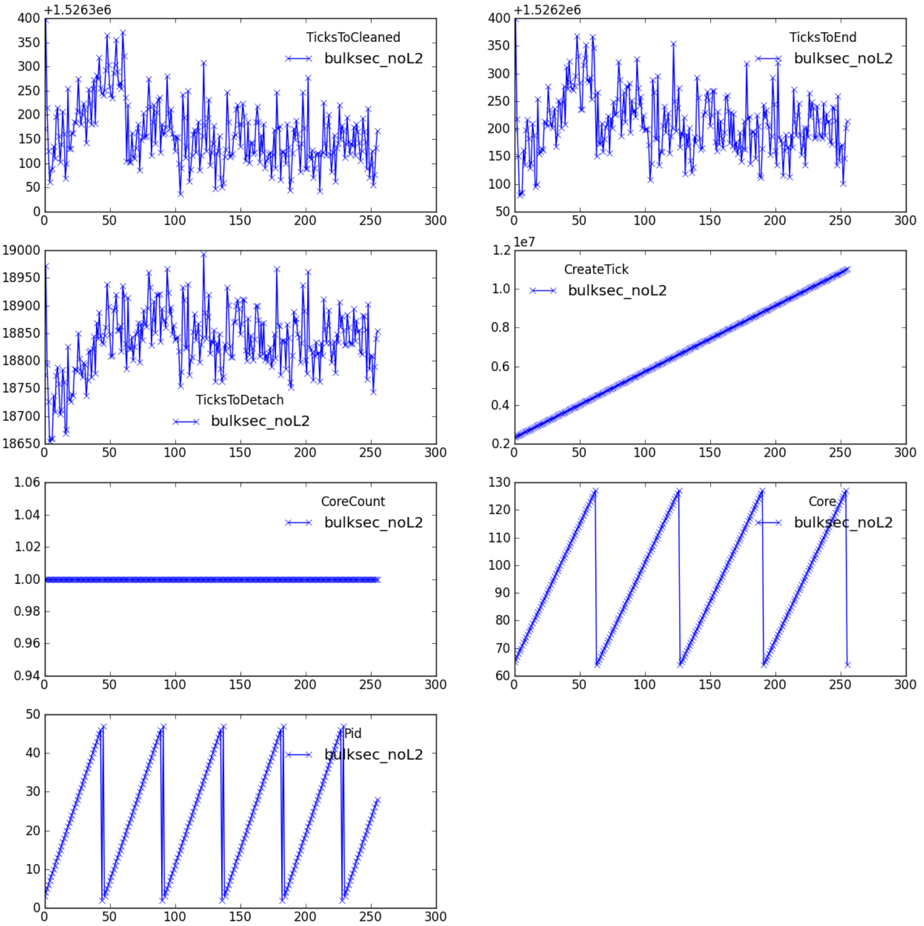
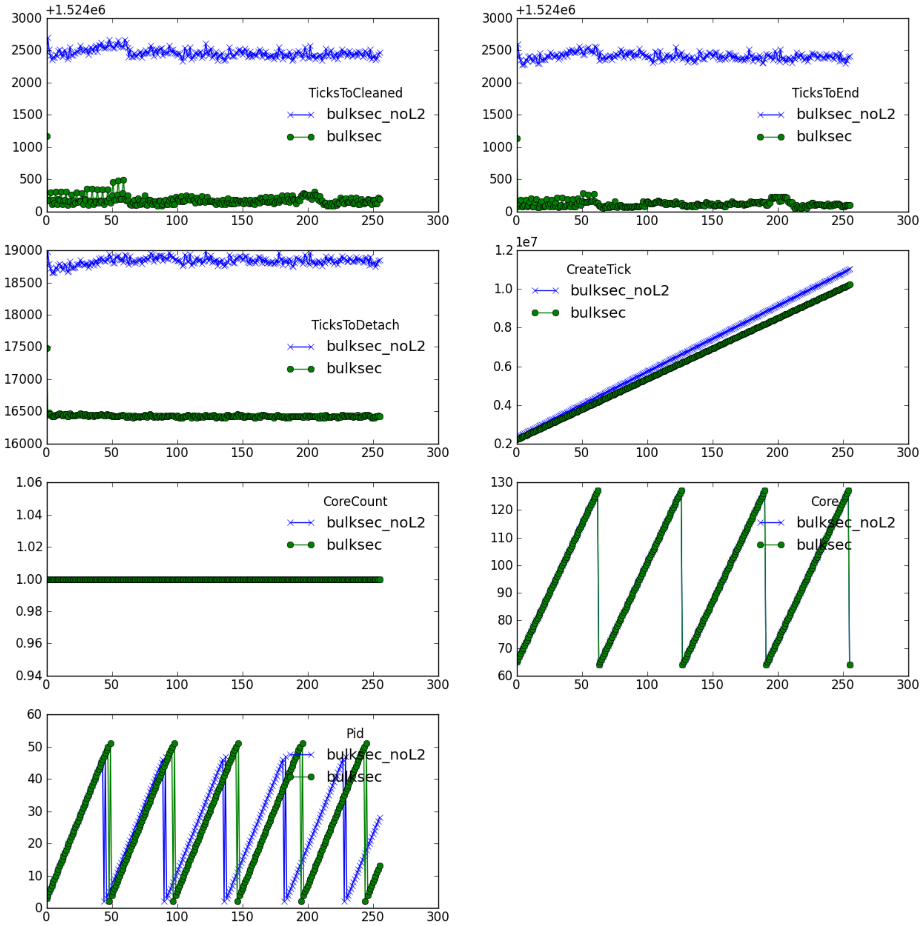


Figure 5.9: Comparison of counting programs, L2



CHAPTER 6

Conclusions

6.1 Overview

Our system has shown that a collection of separately compiled programs can run in parallel harmony, the loaded programs themselves can request the loader to load more programs and these will continue their execution even after their invoking process has finished. The process identifiers used to mark processes can be efficiently recycled, allocated memory is cleaned up after a process has terminated as to ensure the loader can remain active after initial processes have terminated.

We predict the system can load as many processes as the available memory allows, the upper limit for the amount of processes can be influenced by adjusting the amount of reserved per process memory range and setting the appropriate setting in `MAXPROCS`, which is found in `elf.c`.

6.2 Limitations and future work

6.2.1 Stability

The loader lacks some of the protection mechanisms required for the stable execution of untrusted code. However under normal execution the loader is quite stable and handles any errors it can by terminating the offending program and offering several handles for debugging purposes. One of these is the `breakpoint` function which signals the MGSim system that a breakpoint is encountered. In interactive emulation mode, which can be invoked by passing the flag `-Ws,-i` to `slr`. The user can inspect the system whenever a breakpoint is encountered, optionally continuing execution by issuing the `run` or `step` command.

6.2.2 Permissions

In order to explore possible problems extra test programs for corner cases were constructed. These have been used during testing to improve the loader. There is however another class of programs, malicious and erroneous programs, which attempt to access memory which was allocated for another loaded program or even the loader itself. We did not consider this class as our platform did not support access control yet.

A means of protecting loaded programs from each other is documented in [6]. This protection works by introducing hardware which can define capabilities on word-size granularity[1] by introducing hardware which checks permissions every time memory is accessed. Their system could implement access control and secure the system and programs from ill written programs if not malicious programs.

[1] 64-bits in our system

6.2.3 Library sharing

The loader could be extended to dynamically support libraries in such a manner that multiple programs can share them in existing operating systems. This has shown to decrease program sizes and reduce memory needs.

Programs could all use the same copy of program code, where the loader would fill in all unresolved symbols, at run-time, by locating the symbols from single copy of the library into each program being loaded.

6.2.4 Microgrid OS

The loader, process manager and the manager could be integrated into the Microgrid OS, offering users a more complete system without the need for manual configuration.

6.3 Applications

The loader offers a platform which could be extended to allow dynamic task execution and placement. A shell program could be used to offer a dynamic interface. Combined with other programs and daemons a simple operating system could be realized.

In other words, the implementation of a loader program is a foundational stepping stone which bootstraps the implementation of a fully fledged Operating System on this platform.

Bibliography

[1] Jeff Chase, Miche Baker-Harvey, Hank Levy, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. *SIGOPS Oper. Syst. Rev.*, 26:9–, April 1992.

[2] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) Specification, Version 1.2*, May 1995.

[3] 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, 1998.

[4] John R. Levine. *Linkers and Loaders (The Morgan Kaufmann Series in Software Engineering and Programming)*. Morgan Kaufmann, 1999.

[5] John D. Polstra. *FreeBsd Source src/sys/sys/elf64.h, version 1.9, 5 May 2012, http://freebsd.active-venture.com/FreeBSD-srctree/newsrc/sys/elf64.h.html*, September 1999.

[6] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proc. 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, ASPLOS-X, pages 304–316, New York, NY, USA, 2002. ACM.

APPENDIX A

Data

A.1 Terminology

- OS, Operating System
- CSA, Computer Systems Architecture
- MMU, Memory Management Unit
- TLB, Translation Look-aside Buffer
- CLB, Capability Look-aside Buffer
- GOT, Global Offset Table
- PID, Process IDentifier

APPENDIX B

Problems and in depth solutions

B.1 Bugs

B.1.1 Implementation details

At an early stage in the loaders development progress all data relocation was done at compile time. The assumption was made the compiler would generate code to correct data pointers included in initialized variables. However this is not the case as was concluded when a simple program designed to print an array of strings was run and it became clear that these strings where assumed to be at a fixed location. The error in the location could be expressed as the loaded programs base. The programs continued to show this behavior when compiled with the `-fPIC` and `-fpic` compiler flags.

In order to solve this problem, which is a symptom of an incomplete relocation process the data pointers need to be corrected. In order to know which data needs to be corrected for the actual base, the compiler needs to be told that the loader will finish the loading process. This is done by passing it the -shared flag. This flag prevents the compiler from incorrectly assuming a value for the definitive base address and includes relocation information into the produced ELF file.

Included is the information printed by `mtalpha-linux-gnu-objdump`, which can print information about section and relocation records present in the given file.

`tinyex` without the `-shared` flag

```
mtalpha-linux-gnu-objdump -xr tinyex

tinyex:     file format elf64-mtalpha
tinyex
architecture: mtalpha, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000010002c4

Program Header:
    LOAD off    0x0000000000010000 vaddr 0x0000000001000000 paddr 0x0000000001000000 align 2**16
         filesz 0x00000000000003e8 memsz 0x00000000000003e8 flags r-x
    LOAD off    0x00000000000103e8 vaddr 0x00000000010103e8 paddr 0x00000000010103e8 align 2**16
         filesz 0x0000000000000050 memsz 0x0000000000000060 flags rw-
EH_FRAME off    0x0000000000010364 vaddr 0x0000000001000364 paddr 0x0000000001000364 align 2**2
         filesz 0x0000000000000024 memsz 0x0000000000000024 flags r--
   STACK off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**3
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000300  0000000001000000  0000000001000000  00010000  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000061  0000000001000300  0000000001000300  00010300  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .eh_frame_hdr 00000024  0000000001000364  0000000001000364  00010364  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .eh_frame     00000060  0000000001000388  0000000001000388  00010388  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00000038  00000000010103e8  00000000010103e8  000103e8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  5 .got          00000010  0000000001010420  0000000001010420  00010420  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  6 .sdata        00000008  0000000001010430  0000000001010430  00010430  2**3
                  CONTENTS, ALLOC, LOAD, DATA, SMALL_DATA
  7 .sbss         00000010  0000000001010438  0000000001010438  00010438  2**3
                  ALLOC, SMALL_DATA
  8 .comment      00000024  0000000000000000  0000000000000000  00010438  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
```

```
0000000001000000 l     d  .text  0000000000000000 .text
0000000001000300 l     d  .rodata  0000000000000000 .rodata
0000000001000364 l     d  .eh_frame_hdr  0000000000000000 .eh_frame_hdr
0000000001000388 l     d  .eh_frame  0000000000000000 .eh_frame
00000000010103e8 l     d  .data  0000000000000000 .data
0000000001010420 l     d  .got 0000000000000000 .got
0000000001010430 l     d  .sdata 0000000000000000 .sdata
0000000001010438 l     d  .sbss  0000000000000000 .sbss
0000000000000000 l     d  .comment 0000000000000000 .comment
0000000001010438 l      O .sbss  0000000000000004 id.1396
0000000001000004 g      F .text  000000000000004c 0x88 bp
00000000010002c4 g      F .text  000000000000002c 0x88 _start
00000000010103e8 g      O .data  0000000000000038 strs
0000000001010438 g        *ABS*  0000000000000000 __bss_start
0000000001010440 g      O .sbss  0000000000000008 lbp
0000000001010438 g        *ABS*  0000000000000000 _edata
0000000001010448 g        *ABS*  0000000000000000 _end
0000000001000084 g      F .text  0000000000000230 0x88 lmain
0000000001010430 g      O .sdata 0000000000000008 a
```

The same program With `-shared` passed to the compilation toolchain. Since a dynamic executable is produced we can include the `-R` flag for `mtalpha-linux-gnu-objdump`, so it prints relocation entries[1].

```
mtalpha-linux-gnu-objdump -xrR tinyex_shared

tinyex_shared:     file format elf64-mtalpha
tinyex_shared
architecture: mtalpha, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000010002c4

Program Header:
    LOAD off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**16
         filesz 0x00000000000004a0 memsz 0x00000000000004a0 flags r--
    LOAD off    0x0000000000010000 vaddr 0x0000000001000000 paddr 0x0000000001000000 align 2**16
         filesz 0x00000000000003e8 memsz 0x00000000000003e8 flags r-x
    LOAD off    0x00000000000103e8 vaddr 0x00000000010103e8 paddr 0x00000000010103e8 align 2**16
         filesz 0x00000000000001c8 memsz 0x00000000000001d8 flags rwx
 DYNAMIC off    0x00000000000103e8 vaddr 0x00000000010103e8 paddr 0x00000000010103e8 align 2**3
         filesz 0x0000000000000120 memsz 0x0000000000000120 flags rw-
EH_FRAME off    0x0000000000010364 vaddr 0x0000000001000364 paddr 0x0000000001000364 align 2**2
         filesz 0x0000000000000024 memsz 0x0000000000000024 flags r--
   STACK off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**3
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-

Dynamic Section:
  HASH        0x190
  STRTAB      0x358
  SYMTAB      0x268
  STRSZ       0x28
  SYMENT      0x18
  PLTGOT      0x1010510
  PLTRELSZ    0x30
  PLTREL      0x7
  JMPREL      0x470
  RELA        0x380
  RELASZ      0xf0
  RELAENT     0x18
  RELACOUNT   0x7

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000300  0000000001000000  0000000001000000  00010000  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .hash         000000d8  0000000000000190  0000000000000190  00000190  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .dynsym       000000f0  0000000000000268  0000000000000268  00000268  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynstr       00000028  0000000000000358  0000000000000358  00000358  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .rela.dyn     000000f0  0000000000000380  0000000000000380  00000380  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .rela.plt     00000030  0000000000000470  0000000000000470  00000470  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .rodata       00000061  0000000001000300  0000000001000300  00010300  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .eh_frame_hdr 00000024  0000000001000364  0000000001000364  00010364  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .eh_frame     00000060  0000000001000388  0000000001000388  00010388  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .dynamic      00000120  00000000010103e8  00000000010103e8  000103e8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 10 .plt          00000038  0000000001010510  0000000001010510  00010510  2**4
                  CONTENTS, ALLOC, LOAD, CODE
 11 .data         00000038  0000000001010548  0000000001010548  00010548  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 12 .got          00000028  0000000001010580  0000000001010580  00010580  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 13 .sdata        00000008  00000000010105a8  00000000010105a8  000105a8  2**3
                  CONTENTS, ALLOC, LOAD, DATA, SMALL_DATA
 14 .sbss         00000010  00000000010105b0  00000000010105b0  000105b0  2**3
                  ALLOC, SMALL_DATA
 15 .comment      00000024  0000000000000000  0000000000000000  000105b0  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
0000000001000000 l     d  .text  0000000000000000 .text
0000000000000190 l     d  .hash  0000000000000000 .hash
0000000000000268 l     d  .dynsym  0000000000000000 .dynsym
0000000000000358 l     d  .dynstr  0000000000000000 .dynstr
0000000000000380 l     d  .rela.dyn  0000000000000000 .rela.dyn
0000000000000470 l     d  .rela.plt  0000000000000000 .rela.plt
0000000001000300 l     d  .rodata  0000000000000000 .rodata
0000000001000364 l     d  .eh_frame_hdr  0000000000000000 .eh_frame_hdr
0000000001000388 l     d  .eh_frame  0000000000000000 .eh_frame
```

[1]this option can only be used on dynamic, or 'shared', objects

```
00000000010103e8 l     d  .dynamic 0000000000000000 .dynamic
0000000001010510 l     d  .plt 0000000000000000 .plt
0000000001010548 l     d  .data  0000000000000000 .data
0000000001010580 l     d  .got 0000000000000000 .got
00000000010105a8 l     d  .sdata 0000000000000000 .sdata
00000000010105b0 l     d  .sbss  0000000000000000 .sbss
0000000000000000 l     d  .comment 0000000000000000 .comment
00000000010105b0 l      O .sbss  0000000000000004 id.1396
00000000010103e8 l      O *ABS*  0000000000000000 .hidden _DYNAMIC
0000000001010510 l      O *ABS*  0000000000000000 .hidden _PROCEDURE_LINKAGE_TABLE_
0000000001010580 l      O *ABS*  0000000000000000 .hidden _GLOBAL_OFFSET_TABLE_
0000000001000004 g      F .text  000000000000004c 0x88 bp
00000000010002c4 g      F .text  000000000000002c 0x88 _start
0000000001010548 g      O .data  0000000000000038 strs
00000000010105b0 g        *ABS*  0000000000000000 __bss_start
00000000010105b8 g      O .sbss  0000000000000008 lbp
00000000010105b0 g        *ABS*  0000000000000000 _edata
00000000010105c0 g        *ABS*  0000000000000000 _end
0000000001000084 g      F .text  0000000000000230 0x88 lmain
00000000010105a8 g      O .sdata 0000000000000008 a


DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000001010548 RELATIVE          *ABS*+0x000000000100032c
0000000001010550 RELATIVE          *ABS*+0x0000000001000331
0000000001010558 RELATIVE          *ABS*+0x0000000001000336
0000000001010560 RELATIVE          *ABS*+0x000000000100033a
0000000001010568 RELATIVE          *ABS*+0x000000000100033f
0000000001010570 RELATIVE          *ABS*+0x000000000100034b
00000000010105a8 RELATIVE          *ABS*+0x000000000100035a
0000000001010588 GLOB_DAT          strs
0000000001010590 GLOB_DAT          lbp
00000000010105a0 GLOB_DAT          a
0000000001010580 JMP_SLOT          bp
0000000001010598 JMP_SLOT          lmain
```

As can be seen passing the `-shared` flag results in relocation entries, which the loader can then use to complete the loading process. No longer being bound to the fixed address the linker would have selected.

B.2 Example Configurations

These configurations load a single program each with the specified arguments, environment and specific loader settings.

```
verbose=true
filename=/path/to/file/elffile

argv1
argv2
argv3

env0=1
env1=cookies
env3=needs more ducktape
env4=sudo su
env5=make sandwhich -j9001
```