Hell Freezes Over:
Freezing Limbo modules to reduce Inferno's memory footprint

David Boddie
david@boddie.org.uk

ABSTRACT

While Inferno is capable of running on systems with a modest amount of memory, it is sometimes useful to be able to reduce the memory footprint of the running system. This is particularly true of embedded systems with only a few hundred kilobytes of RAM. This paper explores an approach where Dis virtual machine code is retained (''frozen'') in flash memory instead of being loaded into precious RAM.

Introduction

Inferno was designed to have minimal hardware requirements, with a stated baseline of 1 MB of memory and no memory mapping hardware [1]. It was suggested in early publicity [2] that about 512 KB each of RAM and ROM would be enough to ''run an interesting system.'' Ports of Inferno to several systems [3] were made and explored, including some with low amounts of memory [4]. It was also shown to be possible to run Inferno with only 512 KB of RAM using a custom memory allocator [5].

Today, microcontrollers with several hundred kilobytes of RAM and flash memory can be readily obtained. For example, microcontrollers using cores in the ARM Cortex-M4 series can be found with up to 256 KB of RAM, and those in the Cortex-M7 series can be found with 1 MB or more, making them potential targets for Inferno ports [6]. The discussion below focuses on targets at the lower end of this range, specifically the STM32F405RGT6 microcontroller [7] which has 192 KB of RAM and 1 MB of flash memory.

Challenges

The first challenge a port to a microcontroller faces is the availability of a compiler. In this case [8], the Thumb compiler was retrieved from the commit history of the Inferno repository and updated slightly. Beyond this, the two main challenges are the amounts of flash memory and RAM available to use.

The amount of flash memory constrains the size of the operating system and limits the features it can include. In the worst case it may not even be possible to fit a system into the space available. Around 256 KB may be required for a minimal set of features, so the target under discussion has plenty of space for a reasonable system.

The amount of RAM, however, is more limiting. Most Inferno systems tend to run in several megabytes of RAM, and many are copied into RAM on boot-up. Although it was found [5] that 512 KB could be enough to run a useful system if the kernel is run from flash memory, problems can still occur when memory allocation occurs in the running system. In the case of the STM32 target hardware these problems appear in the form of heap allocation errors when trying to run programs.

There are two obvious strategies for dealing with the shortage of RAM in this case:

1. Reduce the amount used by the operating system to begin with.
2. Reduce the amount of memory requested when loading Limbo [9] modules in the form of `.dis` files.

The second approach seemed to be a useful starting point and would still be a beneficial approach for systems that have enough free RAM to begin with, but which use large Limbo modules, since they will still encounter limits to the amount of memory they can allocate.

One strategy for dealing with the shortage of RAM suggested itself when investigating how the Dis virtual machine loads Limbo modules.

Loading Limbo modules

When the Dis virtual machine [10] loads a Limbo module from a file, it performs a number of steps:

1. In the `readmod` function, raw module data is loaded into a newly allocated memory buffer. The `parsemod` function is then called with a new `Module` object to create the suitable data structures needed to run the module code.
2. The `parsemod` function checks for a suitable magic number at the start of the data, returning if unsuccessful.
3. Each section in the module data is decoded and the amount of memory required to represent it is allocated. The first section contains code, which is decoded and expanded into a sequence of `Inst` structures suitable for execution.
4. The members of the `Module` object are filled in and the module is linked to an existing list of modules.

Dis normally patches the code as it decodes it into its runnable form, adjusting the instructions that perform branches to refer to the addresses of their target instructions in memory. This patching is necessary because it is not possible to know ahead of time where the code will be allocated.

When processing the code section of the module, two things stand out. The amount of memory used to store the instructions is typically larger than the size of the corresponding code in the original file. In addition, once the instructions have been decoded, they never change. This suggests that they could be retained in flash memory in a pre-decoded form through a process of "freezing" at build-time.

Freezing code

Support for frozen modules requires three things: a tool to create frozen module data at build-time, a way for the virtual machine to recognize frozen modules, and an extension to the virtual machine that can load frozen module data.

A `freeze` tool was created to perform the task of reading an existing module file and writing data structures that can be compiled into the operating system. Reusing code from the virtual machine, the tool decodes the instructions in the module's code section, ready to be executed at run-time, but writes the other sections in their original form for the virtual machine to handle in the usual way.

The necessary patching of code might appear to be a problem. However, the address of frozen code is fixed at build-time, allowing the branch addresses to be pre-calculated and avoiding any need to modify the instructions at run-time. Each sequence of instructions is represented by a sequence of 32-bit words written in assembly language. Branches within the code are represented as references to offsets from the start of each sequence.

Another purpose of the `freeze` tool is to create a placeholder module file for each original module. This contains the magic number, `0xFD15`, and the path of the original file within the root file system, allowing the virtual machine to recognize that it refers to frozen data for a particular module. There is no need to include any of the original data.

At run-time, the virtual machine recognizes a frozen module by its magic number, causing it to divert control to the `loadfrozen` function which looks up the address of the frozen data using the path encoded in the file. If frozen data is found, it is processed in much the same way as regular module data, except that the address of the code in flash memory is used unchanged and the `frozen` flag is set in the `Module` object.

The build system for the STM32 port was updated with `mk` rules to run the `freeze` tool and build the generated code. The kernel's `main` function was updated to call a function to register the frozen modules with the virtual machine. The modules to be frozen are specified in the root section of the configuration file, using sources that end with the `.fdis` suffix, as in these examples:

```
/dis/env.dis     /dis/env.fdis
/dis/ls.dis      /dis/tiny/ls.fdis
```

These intermediate files are created then processed by the `mkroot` script as normal.

Issues with freezing code and other data

The virtual machine assumes that the resources held by modules are allocated at run-time, so issues can occur when these resources are no longer needed. The most obvious place where a change was needed is in the `freemod` function which was modified to only free sequences of instructions for non-frozen modules. However, problems still occurred when the `ps` command was run. This was tracked down to the `devprog` device that tries to obtain the size of a running program, with assumptions made in the allocator's `poolmsize` function about where the instructions are stored. Checks for the module's `frozen` flag were used to work around issues like these.

Experiments with freezing other sections of modules were only partially successful. As with frozen code, problems occur when data structures created by the loader are assumed to be managed by the allocator. An analysis of how each section is used by the virtual machine is needed to ensure that there are no surprises at run-time. Some sections are less amenable to freezing than others, with potential savings being only slight for those sections that contain only small amounts of data. A minimal approach that only freezes instructions offers most of the benefits of the technique with relatively few changes to the surrounding virtual machine.

Summary of memory savings

The use of frozen modules results in two main ways in which memory usage is reduced. The obvious one is the lack of a need to allocate memory for instructions at run-time. Additional savings are made by the use of placeholder files to represent frozen modules instead of including full-size module files in the root file system, leading to smaller allocations when reading them.

A side effect of the use of placeholder files is the reduction in the size of the root file system itself. Since this data is copied into RAM in the STM32 port, moving the contents of modules into the text section of the kernel means that only references to them in the root file system use up RAM. This means that using frozen modules also reduces the initial RAM requirements of the system. A different approach to handling immutable data would lead to similar benefits for all kinds of data, making this particular benefit obsolete.

Conclusion and future work

The work described above shows how a conceptually simple approach, supported by small changes to the Limbo module loader in the virtual machine, can reduce the size of memory allocations at run-time and reduce the overall memory footprint of an embedded Inferno system.

Further improvements could be made by applying the same approach to other sections of modules, where applicable, noting that further memory savings may be smaller than those already obtained.

It may be possible to extend the technique to compile virtual machine code to native machine code ahead of time. This approach may work well for devices with slow processors and reasonable amounts of flash memory, and could be used as an alternative to just-in-time (JIT) compilation for frequently used modules.

The source code associated with this paper can be found in the following repository:
https://github.com/dboddie/inferno-os/tree/stm32f405

References

1. S. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey, P. Winterbottom, ''The Inferno Operating System'', *Bell Labs Technical Journal*, Vol. 2, No. 1, Winter 1997, pp. 5-18
   https://www.vitanuova.com/inferno/papers/bltj.pdf
2. ''Lucent brews Inferno'', *InfoWorld*, October 1996
   https://www.infoworld.com/article/2077261/lucent-brews-inferno.html
3. Vita Nuova, ''Inferno Ports: Hosted and Native'', *Vita Nuova Limited*, 27 April 2005
   https://www.vitanuova.com/inferno/papers/port.pdf
4. Salva Peiró, ''Inferno DS: Inferno port to the Nintendo DS'', *3rd International Workshop on Plan 9*, University of Thessaly, Volos, Greece, 2008
   http://3e.iwp9.org/iwp9_proceedings08.pdf
5. Brian L. Stuart, ''Inferno in Embedded Space: Porting to the Sun SPOT'', *8th International Workshop on Plan 9*, Athens, Georgia, USA, 2013

http://8e.iwp9.org/sunspot.pdf

6. Petter Duus Berven, "Porting Inferno OS to ARMv7-M and Cortex-M7", *Master's thesis in Computer Science*, Norwegian University of Science and Technology (NTNU), January 2022
https://hdl.handle.net/11250/3034870
7. "STM32F405xx STM32F407xx Datasheet", *STMicroelectronics*, 2020
https://www.st.com/en/microcontrollers-microprocessors/stm32f405rg.html
8. David Boddie, "STM32F405 Port of Inferno", accessed on 8 February 2023
https://github.com/dboddie/inferno-os/tree/stm32f405
9. Dennis M. Ritchie, "The Limbo Programming Language", *Inferno Programmer's Manual, Volume Two, Vita Nuova*, 2005
https://www.vitanuova.com/inferno/papers/limbo.pdf
10. Lucent Technologies Inc, Vita Nuova Limited, "Dis Virtual Machine Specification", *Lucent Technologies*, 1999, *Vita Nuova Limited*, 2003
https://www.vitanuova.com/inferno/papers/dis.pdf