Subordinate Kernels: Application Offloading in Asymmetric Multi-Processor Systems

Ross McIlroy[1,2] and Orion Hodson[1]

[1] Microsoft Research, Redmond, WA 98052, USA
[2] University of Glasgow, Glasgow, G12 8QQ, UK

Abstract. Modern computing systems incorporate a plethora of auxiliary processing cores located on devices such as graphics cards, network devices and raid controllers. These processors are becoming increasingly sophisticated and could provide application offloading if exploitable by third party code. However, to provide an environment for practical general purpose application offloading, the operating system must present a homogeneous programming interface across the heterogeneous processors and provide a safe execution environment. In this paper we present a subordinate kernel operating system structure as a means of practical and safe execution of third party application code on auxiluary processing cores. We expand this concept of subordinate kernels by describing a proof of concept design of application offloading in the Singularity research operating system.

1 Introduction

As well as central processing unit (CPU) cores, modern computing systems include multiple auxiliary *device processors* located on devices such as graphic cards, network devices and raid controllers. These device processors are becoming increasingly sophisticated and can potentially be more powerful than the CPU under certain situations. For example, a typical graphic processing unit (GPU) core has significantly higher floating point performance than most general purpose CPUs [1]. If third party applications were able to exploit these device processors, many compelling application offloading scenarios could emerge, such as voice recognition on GPUs or off-loading of file searching to the disk controller. However, to provide general purpose application offloading, an operating system must provide the following: a safe execution environment for untrusted 3rd party code executing on device processors, with enforced isolation between applications; a unified programming interface and inter-process communication mechanisms across device processors and the main CPU processor; and a management infrastructure to control where applications are loaded.

To tackle these issues we present a subordinate kernel operating system structure. This structure involves an master kernel execution on the main CPU, controlling the overall system, and a subordinate kernel being run on any auxiliary device processors to control execution of third party code on these devices. In this paper we present the design of a subordinate kernel system based upon the Singularity research operating system.

2 Singularity

The Singularity Operating System [2] was created by Microsoft Research in order to re-examine fundamental design decisions common in most contemporary operating systems. The primary goal of the Singularity project is to investigate how an OS and its associated software stack should be structured to improve dependability and trustworthiness. This goal lead to an architecture based upon software isolated processes (SIPs). Operating system services, such as drivers or file-systems, run as separate user level SIPs, with only core OS functionality being located in a micro-kernel. SIPs are written in strongly typed, managed code (a variant of C#), thus Singularity can take advantage of the memory and type safety guarantees provided by this programming language to enforce isolation between processes without relying on a hardware memory management unit (MMU).

To enforce isolation between processes Singularity does not allow SIPs to explicitly share memory. Inter-process communication is instead provided by contract-based channels. A channel is a strongly typed bi-directional message conduit. A Linear type system ensures that only one SIP has ownership of the data within channel messages at any time.

This combination of software isolated processes and channel based inter-process communication provides a well structured basis for general purpose application offloading onto device processors. By isolating processes using software verification, a safe execution environment can be provided even when third party code is running on device without hardware MMU memory protection. Additionally limiting all inter-process communication such that it can only be performed over channels enables the operating system to hide the actual transport

mechanism used to transfer data from one process to another. This provides applications with a homogeneous inter-process communication interface even under a potentially diverse set of underlying communication mechanisms.

3 Subordinate Kernel

Our approach to implementing application offloading on device processors involves the creation of a *subordinate kernel* for any device processor involved in application offloading. A subordinate kernel is extremely small and simple, with even less capability than a typical micro-kernel. It makes decisions local to the device processor which it controls (e.g., management of device local memory) and providing a safe execution environment for third party code, however, it defers any complex decisions to the *master kernel* running on the systems main CPU. The master kernel gives orders to the subordinate kernels to enforce global policies, e.g., which applications are to be run on which particular processing cores.

To provide a homogeneous programming interface to third party applications across device processors and the main CPU processor, a common application binary interface (ABI) must be presented to applications by both the master and subordinate kernels. However, with a limited subset of functionality, a subordinate kernel may not be able to perform all of the operations required by the main kernel's ABI. An ABI shim is therefore inserted between the subordinate kernel and third party applications running on a device processor. Any ABI calls which cannot be serviced directly by the subordinate kernel are delegated to the master kernel.

This ABI shim mechanism provides flexibility in the amount of kernel functionality provided by the subordinate kernel itself. This allows the capabilities of the device and the required turn around time to be taking into account when designing a subordinate kernel for a particular device. For example, an initial subordinate kernel could simply redirect all ABI calls to the master kernel, thereby consisting of only the ABI shim and a communication mechanism to the master kernel. This subordinate kernel could then be gradually enhanced to perform certain ABI calls locally without having to make any modifications to applications running upon it.

4 Application Offloading in Singularity

We are developing a proof of concept implementation of this subordinate kernel approach to application offloading for the Singularity OS. Our target is a commodity x86 PC which includes an ARM based IO board expansion card. The IO board contains a disk controller and a network interface card, therefore, our goal is to offload network and disk based applications from the host x86 processor to the ARM processor on the IO board. This section describes the intended design of this proof of concept OS structure.

A subset of the Singularity micro-kernel has been ported to the ARM architecture to act as a subordinate kernel for the IO board. The host x86 processor runs the master Singularity kernel. Applications are developed as normal irrespective of whether or not they will be offloaded onto the IO board, requiring only recompilation if they are to be offloaded. Offloaded applications are linked to a user level ABI shim, which presents the full Singularity Kernel ABI. This shim either passes ABI calls straight through to the subordinate kernel, or forwards them to a user level offload helper application running on the host processor. The offload helper application services these remote ABI calls, making calls into the master kernel if necessary.

All inter-process communication, including ABI shim remote calls, occur through Singularity's message channels. If the two communicating SIPs are located on the same type of processor then the standard Singularity channel implementation is used (zero copy transfer of ownership of the message data). If a SIP on the device processor is communicating with a SIP on the host CPU, then the channel endpoints are tagged as being remote and messages pass through a channel indirection mechanism located within both the master and subordinate kernels.

The channel indirection mechanism marshals the data within a channel message such that it can be understood by the destination processor. This marshalling process may involve conversion of values between little endian and big endian, and translation of any pointer values if virtual memory addresses are laid out differently for each processor. The channel indirection mechanism then sends the message to the destination kernel using the most appropriate transport mechanism, and signals the destination kernel of this new message.

Since applications running upon Singularity have only two methods of communication – channels and kernel ABI calls – the ABI shim and channel indirection mechanism are the components necessary to create an application offload subordinate kernel. Features such as local memory management and local thread scheduling can be incorporated as necessary to increase overall utilisation of a devices re-

sources and improve the performance of the subordinate kernel's application offloading.

5 Conclusion

As computer systems incorporate devices with increasingly powerful processing capabilities, offloading of third party application code onto these device becomes more compelling. In this paper we propose a subordinate kernel operating system structure as a means of providing practical application offloading onto these devices. We are investigating this OS structure by developing a proof of concept implementation for the Singularity research operating system.

References

1. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3), 2004.
2. G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.