Karlsruher Institut für Technologie

Reducing Overhead in Microkernel Based Multiserver Operating Systems through Register Banks

Studienarbeit
von

Sebastian Ottlik

an der Fakultät für Informatik

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Dipl.-Inform. Raphael Neider

Bearbeitungszeit: 18. April 2010 – 8. Oktober 2010

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 8. Oktober 2010

Sebastian Ottlik

Abstract

Reduced performance is generally considered to be a drawback of microkernel based multiserver operating systems. The reason for this loss of performance is that typical operating system services, performed by the kernel in traditional systems, are provided by the kernel and several user mode servers in a multiserver operating system. As a result, performing these services requires a number of thread switches as well as kernel entries and exits, whereas in a traditional system only one kernel entry and exit usually suffices.

In modern systems with a tagged TLB and a dedicated system call instruction most of the costs of a thread switch or kernel entry result from preserving register contents in memory. Existing approaches to reduce these costs typically improve the performance of kernel entry by delaying register preservation to thread switching. If a kernel entry is followed by a thread switch — a frequent situation in systems using a microkernel — these approaches do not improve total system performance.

In this thesis I present a mechanism that can be used to increase the performance of thread switching and kernel entry and exit by selection of a bank of register file memory instead of exchange of register contents using main memory. I describe in which way it can be used by a kernel and its implementation in the OpenProcessor platform as well as its use by the associated kernel to increase kernel entry and exit performance. Evaluation has shown that it can be used to decrease system call overhead by 10% while providing further possibilities of performance optimization.

Contents

Chapter 1

Introduction

Historically, microkernel based multiserver operating systems [14] have been perceived to have inherent bad performance characteristics that outweigh the advantages of this operating system architecture in most fields of application [2]. Especially first generation microkernels have been shown to have a low performance compared to modern ones [4]. While modern microkernels — such as L4 [6] — have proven that the overhead introduced by a microkernel can be greatly reduced, the encapsulation of operating system services in user mode servers generally leads to a performance overhead in comparison to systems that provide these services using a monolithic kernel.

Part of this performance overhead is a result of the need to perform an increased number of context switches, which I consider to occur on thread switch as well as on kernel entry and exit in this thesis. Thus, improving the performance of context switches can reduce the overhead of a multiserver operating system.

Context switches typically include a change of the address space, the execution mode and the contents of registers. Modern systems reduce the cost of an address space switch by means of translation lookaside buffers (TLBs). The cost of execution mode changes can be reduced by using lightweight methods of doing so like special instructions to perform kernel entry and exit. These measures leave the preservation of register contents in main memory as a considerable part of the costs of context switching.

Existing approaches to avoid the cost of register preservation typically only improve kernel entry performance by delaying the need to preserve register contents using main memory until a thread switch is performed. These approaches do not provide any performance benefit if a kernel entry is followed by a thread switch which is a common occurrence in multiserver operating systems.

In this thesis I present an approach to prevent the memory accesses that are necessary to preserve register contents when a context switch is performed. This approach consists of extending the register file of a processor to provide several banks of register file memory, a mechanism for selecting these register banks and its use by a microkernel. When a context switch is to be performed by the kernel, it can change register contents by selecting a different register bank instead of using main memory. To allow this behavior a bank needs to be assigned to a context beforehand in an operation called bank assignment.

The purpose of this approach is to decouple memory accesses necessary to preserve register contents from context switches by storing register contents associated with multiple contexts on the processor at the same time. This enables the different banks to be used to increase the performance of thread switching in addition to kernel entry and exit thereby decreasing the total number of memory accesses necessary to perform context switching in the whole system.

I describe the implementation of a basic version of the mechanism on the OpenProcessor [8] platform and the extension of the associated kernel to use the mechanism to increase kernel entry and exit performance. Evaluation has shown that this implementation reduces system call overhead by about 10%.

Chapter 2 covers related work and information on the OpenProcessor hardware and operating system. Chapter 3 details the design of the register bank mechanism and its possible applications in the OpenProcessor kernel. Chapter 4 describes my implementation of register banks for the OpenProcessor platform and their usage to increase kernel entry and exit performance. Chapter 5 presents an evaluation of the implementation. Chapter 6 concludes this thesis by summing up its results and providing directions for future work.

Chapter 2

Background

2.1 Related Work

There is work on several topics related to this thesis: first there are a number of approaches to utilize a bigger register file to achieve a general performance increase in a computing system, second there are a number of works addressing the specific problem of preserving register contents across context switches — in particular kernel entries. These categories are not disjoint as most mechanisms could at least be utilized to increase kernel entry and exit performance. In this section I first present works that are closer to the first category, progressively moving towards the second category.

Some early [12] as well as modern [13] RISC processors implement a concept called register windows to increase call performance by utilizing a bigger register file without increasing the number of registers encodeable in an instruction. A similar concept, the Register Stack is used by the IA64 CISC architecture [1]. In these processors, the register file is organized as a ring buffer of overlapping windows. The current window can be moved forwards or backwards on the ring using special instructions with a number of registers local to each window and a number of registers shared with adjacent windows for passing parameters and return values. Usually, there is also a number of registers not affected by bank selection.

When performing a context switch, register windows can be saved and restored lazily. However, memory accesses cannot be avoided completely using such an approach. [5]

It has been shown [15] that a system without register windows can achieve a similar performance for function calls with a smaller register file by increasing the number of registers encodeable in an instruction and improving the register allocation logic of the compiler. In general register windows do not fit the needs of a microkernel very well, but the possibility of preserving register contents lazily could be applied to register banks as well.

Other RISC based processor designs provide a number of shadow registers associated with particular general purpose registers to increase exception, interrupt and system call handling performance. MIPS [7] offers several banks of shadow registers selected by the processor according to an operating system controlled

configuration. Since the kernel can configure hardware bank selection it could utilize banks to increase thread switching performance similarly to the design I present in this thesis. In contrast to my design, access to register contents on another bank is only possible by either switching to it or by using special instructions to access the previously selected bank. This makes MIPS shadow registers inflexible when used in a microkernel, in particular by potentially complicating the implementation of data transfers for inter process communication.

Another popular RISC based processor design employing shadow registers is ARM [11]. On ARM, shadow registers are available for a number of registers when operating in privileged mode. In general, all privileged code parts have a separate shadow register for the stack pointer and the link register. As an exception, Fast Interrupt Handlers(FIQ) have shadow copies of roughly half of the general purpose registers to enable fast handling of some interrupts. Access to shadow registers not associated with the current mode is generally not possible, special instructions to store them to and load them from memory exist instead catering to the needs of exception handlers only.

Some ARM based processors also offer a (further) reduced instruction set, called Thumb Instruction Set, with interesting properties in regard to register access: to reduce the size of instruction words, only the lower half of the general purpose register set is accessible using all instructions. The upper half can only be accessed by a subset of the available instructions. This is interesting with regard to register banks since it can be considered as a form of banking used to reduce binary code size.

The specific problem of preserving register contents across context switches has also been addressed by reducing the number of registers that need to be preserved according to which are currently used [3]. This is made possible by dividing the register set into several subsets and by having the compiler emit instructions that indicate which of these subsets are currently in use. When a context switch occurs, the kernel can evaluate this live set information to determine which subsets need to be preserved.

2.2 The OpenProcessor Platform

This section describes relevant characteristics of the OpenProcessor platform for which I designed the register bank mechanism. The OpenProcessor platform is a minimal, flexible and representative research environment for hardware/software co-design with a focus on operating systems. It consists of hardware components written in the Verilog hardware description language and a microkernel based multiserver operating system implemented on top of the hardware.

The hardware is implemented using a Memec Virtex-4 MB Development Kit, featuring a Xilinx XC4VLX60 [18] Field Programmable Gate Array (FPGA) among other components such as different kinds of memory and I/O facilities. The FPGA is used to implement all further hardware components on top of which the kernel is executed.

The advantage of this system design is that it allows fast development of

hardware changes, since synthesis of a new hardware version can be performed in about half an hour only. The bit stream programming of the FPGA takes a few seconds allowing quick comparisons of different hardware versions.

The hardware implemented in the FPGA consists of a 32-bit RISC processor which is connected by a Wishbone bus [9] to several memory and I/O controllers. Since the system aims at providing a basis for operating system research it has state of the art hardware support for typical operating system functionalities, such as a tagged translation lookaside buffer and special instructions for privileged mode entry and exit.

The kernel implemented on top of the processor is conceptually akin to the L4 microkernel. Especially the paradigms of memory management and inter process communication, which have been described in [6] and [4], have been adopted to the OpenProcessor kernel. On top of the kernel, a few user mode servers provide typical OS services such as file system access and memory management.

The rest of this section covers several aspects of the OpenProcessor platform that impact the design and implementation of the register bank mechanism: Section 2.2.1 describes the implementation and usage of registers on the OpenProcessor platform. Section 2.2.2 covers the concept of contexts and its relevance to the OpenProcessor kernel. Finally, Section 2.2.3 describes the processor pipeline with focus on the result and load forwarding logic.

2.2.1 Registers and their Usage

The OpenProcessor's instruction encoding provides 6 bits to address registers, which allows addressing of up to 64 different registers. The lower half of the register address space is used to address general purpose registers, while the upper half is reserved for special purpose registers. Special purpose registers are implemented directly in the FPGA fabric due to their increased performance demands resulting from implicit read and write accesses[1] in several pipeline stages. The general purpose registers are stored in a register file implemented around two of the RAM blocks present on the FPGA.

Each RAM block offers 18 Kibit of memory accessible via two read/write ports. The ports are configured to operate on 32 bit data words with 4 additional unused bits per word for parity checks, resulting in 16 KiBit (or 512 32 bit words) of memory available for the register file. While the capacity of a single block would suffice to hold all general purpose register data, which is 1024 bit total for the 32 general purpose registers, utilizing two blocks is still necessary since the pipeline requires two read and one write port to the register file. The combination of two blocks allows two read operations per cycle, which are each performed on a different block, plus a write operation that is performed synchronously on both blocks to ensure consistency.

The calling conventions employed by the system define most of the lower half of the general purpose registers to be used for function variables ("callee saved") while the upper half is to be used for temporary variables ("caller saved"). When a call is performed, the temporary variable registers are supposed to be saved by

[1] I use this term to denote direct accesses to registers as opposed to more complex ones performed by the Operand Fetch or Write Back stage.

the caller, while the callee is responsible for preserving the contents of function variable registers.

A number of registers are dedicated to a specific purpose. Two of them are defined by the hardware architecture: r0 is constant zero and r31 is the link register, which is used to store the return address when a call is performed. The stack is implemented using r2 as the stack pointer with no frame pointer present. The kernel requires a register (typically r1) dedicated to perform kernel entry. Registers r30 down to r27 are used to pass the first 4 arguments and return results when a function or system call is performed.

2.2.2 Contexts

A context is the set of data that needs to be preserved when a task is interrupted in order to resume it later. While this definition includes memory contents visible to a task and its page table, both are loaded from memory on demand and do not necessarily result in memory accesses when switching contexts on the OpenProcessor platform. The register contents on the other hand need to be preserved in any case — usually using memory.

The OpenProcessor kernel uses one kernel stack per thread. This means that a thread, except when currently executing in user mode, can be considered to have two contexts: a user mode and a kernel mode context. To resume a thread, its kernel mode context is restored which in turn might restore the user mode context. When inter process communication (IPC) is performed, the kernel context of the sender thread accesses the user mode context of the receiver thread in order to transfer the message. This makes it necessary for the kernel to be aware of and differentiate the two contexts associated with a thread.

For the design of the register bank mechanism, a number of operations need to be considered since they work with the contexts in one way or another. In the rest of this section I first present possible ways to enter the kernel from user mode and then describe a number of activities that may be performed by the kernel mode context of a thread.

Two ways to enter the kernel can be distinguished on the OpenProcessor platform: a number of events, such as exceptions or interrupts, implicitly cause a kernel entry or a user mode application can explicitly perform a kernel entry by using the `SYSCALL` instruction. Both cases are handled similarly: first the processor starts execution at an exception vector the address of which depends on whether the processor was executing in privileged mode when the kernel entry was caused and what the cause for it was. Kernel code residing at this address then branches to a vector-specific stub which in turn jumps to the appropriate handler through trampoline code. One difference exists between a kernel entry caused by a `SYSCALL` and the other exceptions: while the `SYSCALL` stores the return address in the link register like a conventional call, the other exceptions store the address of the faulting instruction in a special purpose register.

The responsibility to save and restore the current context is divided between the stub, the trampoline and the handler. The involvement of the handler is a result of the fact that its code is generated by the compiler and is thus subject to the calling conventions, which require that every function preserves the contents

of all callee saved registers. The stub and trampoline code rely on this to reduce the set of registers that need their contents saved and restored to the set of caller saved registers. These registers are saved by the stub on the kernel stack and later restored from it by the trampoline. To make these register contents available to kernel mode code they are stored in a well defined structure, referred to as a *trapframe*. The address of the trapframe on the stack is passed to the handler.

With the register bank mechanism, storing register contents in memory can be avoided, which inherently prevents the creation of the trapframe. For this reason an understanding of how and when it is used is crucial to assess the impact of register banks on the kernel.

When a handler has performed the necessary actions, it needs to read the return address or the address of the faulting instruction and set a special purpose register to prepare the processor for returning to user mode. In the case of a syscall, a number of user mode register may be read to access parameters passed from user mode or they may be written to return results to user mode. If we introduce register banks, avoiding to store register contents in memory, we need to provide other means to access these values.

When a thread switch is performed the thread performing it is already operating in kernel mode on its kernel mode stack and thus only its kernel mode registers need to be backed up. Since the context switch is called as a function from compiler generated code, only the callee saved registers need to be stored, as the caller saved registers have already been stored on the kernel stack according to calling conventions. When the remaining registers have been saved on the kernel stack, the stack pointer is changed to that of the next thread. Then its callee saved register contents are restored and the function returns to the compiler generated code that the kernel context of the now current thread was executing previously which will restore the caller saved registers.

Finally the trapframe plays a vital role in performing IPC. Since the OpenProcessor kernel only supports message items stored in registers the trapframe contains all message items when the kernel performs IPC. All copying is performed by the sending threads kernel context while the receiver is blocked in receiving state. Since the IPC mechanism is quite complex its concept cannot be covered in its entirety here, refer to [4] for an in-depth description.

In conclusion, the kernel requires read and write access to almost all registers stored in the trapframe. This is not only necessary for context switches but also to provide other important kernel functionalities.

2.2.3 Result and Load Forwarding

For the discussion of the register bank design in Chapter 3, a basic knowledge of the OpenProcessor result and load forwarding is required. In particular, different methods of replacing the communication channel in shared registers, which is lost on a full bank switch (a problem introduced later in more detail), require a modification of the forwarding logic.

To understand the result and load forwarding logic in the OpenProcessor a basic understanding of its pipeline is needed: the pipeline has 5 logical stages, of

which those concerned with memory access have been split up into several physical stages depending on the speed of the used translation lookaside buffer (TLB) and cache. The hardware version used as a basis for this thesis has 8 physical pipeline stages.

The first step in the pipeline is the Instruction Fetch stage, which is responsible for loading the next instruction. In a best case scenario (i.e. when no cache or TLB misses occur) it takes 3 cycles to complete and thus is split up into 3 physical stages. When an instruction has been fetched from memory, it is passed on to the Operand Fetch stage, which will decode and load the source operands from the register file in 1 cycle. Afterwards, the Execute stage decodes the instructions, including the destination operand, and performs requested ALU operations in 1 cycle. After the Execute stage has finished, an instruction enters the Data Memory stage which handles memory accesses caused by load and store instructions in 2 cycles. In the last stage, called Write Back, write accesses to the register file are performed to update the destination operand if present.

ALU results can be forwarded from the Execute stage when no load is to be performed. Load forwarding can only be done after an instruction has passed the Data Memory stage. When result forwarding is applied, the register contents are available immediately, while load forwarding results in a stall of up to 3 cycles, if no cache or TLB misses occur.

Chapter 3

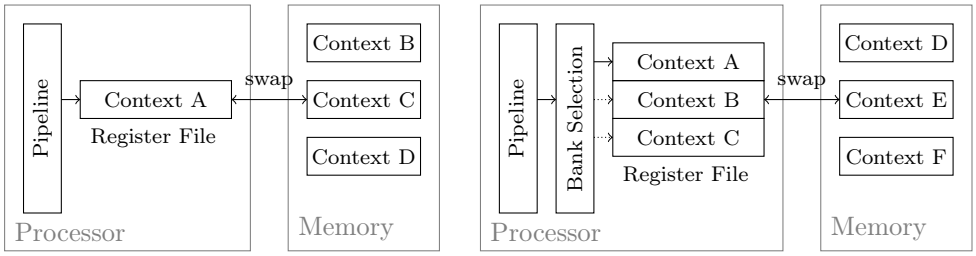
Design

In this chapter I present the design of the register bank mechanism and its impact on the kernel. The purpose of register banks, as has been motivated in Chapter 1, is to reduce context switching overhead by reducing the number of memory accesses resulting from them and as a result to increase the performance of the following typical microkernel operations: (*a*) kernel entry and exit (*b*) thread switch (*c*) inter-process communication (IPC). This is achieved by storing multiple register sets at a time in the register file, instead of only keeping the current one in it and the rest in main memory.

The *register context* of a thread is made up of the set of all register contents which need to be restored in order to resume its execution. While this definition may — depending on the platform — include different registers for different threads, in the case of OpenProcessor the contexts of all threads comprise all 32 general purpose registers. Some of the special purpose registers are also relevant regarding the contexts of a thread. These special purpose registers are small in number compared to the 32 general purpose registers, and the implementations of general and special purpose registers differ severely. Since banking special purpose registers would have a significant impact on the complexity of the solution without providing a significant benefit I did not consider them further. Thus the register context referred to in this thesis is the set of all general purpose register contents associated with the user mode or kernel mode context of a thread.

Microkernel based operating systems usually provide most operating system operations by a number of user mode servers. As a result, many operations that only require a single kernel entry to be performed on traditional operating systems based on monolithic kernels, additionally require a thread switch in systems using a microkernel.

In the case of a monolithic kernel, the performance of many kernel operations like interrupt handling can be increased by providing shadow copies of only a small number of registers. These are used to preserve user mode register contents while the kernel is working. When a thread switch occurs, the contents of these shadow registers need to be preserved in addition to the contents of their currently visible copy. As a result, the total number of registers that need



(a) traditional (b) register banks

Figure 3.1: context switching: traditional and using register banks

to be preserved if a kernel entry is followed by a thread switch does not change through an approach like this.

Since many operating system operations require a kernel entry followed by a thread switch in systems using a microkernel, it is of little use to provide a banking mechanism to only increase kernel entry and exit performance in the case of OpenProcessor. The register bank mechanism described in this chapter addresses this issue by providing several banks of general purpose registers. These banks are referred to as *register banks*.

Figure 3.0 shows how a context switch is performed traditionally and how it can be done using register banks. Traditionally, as shown in Figure 3.0a, the registers encoded in instructions each refer to one word in the register file. When a context switch is performed the register file contents are written to memory to preserve the current context and new values corresponding to another context are loaded into the register file.

Register banks, shown in Figure 3.0b, can be used to decrease the number of situations in which a context needs to be stored in memory and a different one loaded from it. Register banks allow this by providing several banks of registers and translating registers encoded in instructions into an address in the register file according to a bank selector. This allows a decoupling of memory access from a context switch. A context switch can now be performed by simply selecting a different bank.

To assign a bank to a context, the contents of a bank still need to be saved to and loaded from memory, this process is called *bank (re)assignment*. With a sensible bank assignment policy, I assume that bank reassignment is necessary less often than a context switch and thus an increase in total system performance is possible by using register banks.

In order not to restrict which bank assignment policies might be implemented by the operating system, the processor implements the bare mechanism, while it is the responsibility of the operating system to configure the mechanism accordingly.

The rest of this chapter is divided into two parts: while Section 3.1 is concerned with the banking mechanism, Section 3.2 discusses its impact on the kernel.

3.1 Requirement Analysis

Concerning the design of the register bank mechanism — apart from the increase in required register file memory — two important aspects have to be taken into consideration:

- If multiple banks of registers are offered, a translation of a register address encoded in an instruction (referred to as an *encoded register address* in this chapter) and a bank selector to an address in the register file of the processor becomes necessary.
- Assuming a full switch of the register set whenever a context switch occurs, the communication channel previously offered by shared registers needs to be replaced.

A basic translation can be defined as follows: each bank is assigned a unique number in the range of zero to the total number of banks minus one, this number is referred to as the *bank identifier*. The offset of a bank into the register file memory is calculated by multiplying its identifier by the constant bank size. This *bank offset* is then added to an encoded register address to translate it into a register file address.

This basic translation logic has a direct implication for the kernel: previously, register contents were retained by default when a context switch occurred and thus registers were a communication channel between different contexts. It was the responsibility of the kernel to control this channel in a manner that maintained system integrity. For example, if a system call had been performed, the kernel must have cleared kernel data from all registers but the ones used to return results to user mode before returning to it. This was necessary to prevent possibly sensitive data of other contexts from being leaked.

With banks in place, a bank switch can be performed and thus no register contents of other contexts can be visible to user mode, if the bank used has been assigned to the user mode context already and bank switching is controlled by the kernel. The kernel is not freed from the responsibility to control this channel, but the necessity to act is shifted from the frequent context switching to the rare bank reassignment operation.

The communication channel established through shared register contents does not only burden the kernel, but is also used to facilitate communication between different contexts without the need to rely on the slower main memory. For example, if a system call is performed, this channel is used to pass parameters such as the system call number from the user mode context of a thread to its kernel mode context. Likewise the channel can be used to return values from a system call back to user mode. Since this channel is lost in the presence of a full register bank switch, it needs to be replaced either by using main memory or by providing a new way of communication across register banks. Since the purpose of the mechanism is to improve performance a way to communicate using registers should be provided.

When designing a new way to allow inter context communication using registers the following situations need to be considered since they may utilize the

new channel to achieve an increase in performance or need to secure it: (*a*) kernel entry and exit, (*b*) exception handling, (*c*) inter process communication, and (*d*) thread switch.

While kernel entry is neither directly concerned with inter context communication nor must it secure the communication channel since the kernel is trusted, it still needs to be considered since it might setup a way for the exception handlers to access the user mode context. Kernel exit on the other hand needs to make sure no sensitive data may be leaked to user mode.

Exception handlers provided by the OpenProcessor's kernel usually do not access user mode register contents. An exception from this rule is the system call handler which needs at least access to the link register of the user mode as well as a register containing the system call number; passing system call parameters directly in registers is also preferable. Furthermore, the OpenProcessor is constructed in a way that might allows trap & emulate virtualization. This system property would be lost if privileged mode could not access all register contents of unprivileged mode. For these two reasons the new communication channel must enable the kernel to access arbitrary register contents on a semi-regular basis and certain registers on a regular basis.

The IPC implementation used by the kernel relies on registers to contain message items that need to be transferred. Thus kernel mode needs fast access to register contents of two different user mode contexts — sending and receiving context of the message. Since the number of items transferred during IPC is determined at runtime and all items of a message are transferred at the same time, access to a block of registers of an arbitrary user mode context would be an advantage in this situation.

If a thread switch is performed, no data needs to be passed between contexts, but a switch from one context to another must be performed. If a bank has already been assigned to the target thread, this can be accomplished by simply switching the register bank. When no bank is assigned to the target thread one needs to be assigned and filled in properly, which requires access to the register contents of the bank that is to be assigned. When switching threads, securing the new communication channel might be prepared, performed partially or entirely.

The rest of this section covers different ways to establish a new communication channel in Sections 3.1.1 through 3.1.4. Section 3.1.5 describes the final design.

3.1.1 Globally Shared Registers

A number of general purpose registers which are not affected by bank selection could provide the new communication channel between contexts. This could be achieved by either adding new registers into the special purpose register address range or by preventing some of the general purpose registers from being affected by bank selection. The new channel offered by this solution is similar to the one that it is supposed to replace.

This similarity is an advantage in a number of situations: for example, the shared registers could be used to pass parameters to a system call in the same way as before. Assigning a new bank to a thread could be done by using one of

the shared registers to store the memory address of the new register contents while another one could be used to address the function loading the values from memory into the appropriate registers.

A disadvantage of this solution, which also results from its similarity to the communication channel that it is supposed to replace, is that the shared registers necessarily become a part of the register context of a thread, since a thread cannot be resumed without restoring the global registers. As a consequence they need to be handled in the same way as general purpose registers were handled before: they need to be saved and restored using main memory in most cases if a context switch occurs. This is exactly what this design is trying to prevent and thus a major drawback of this solution.

3.1.2 Memory Mapping

Mapping the register file to a region in the memory address range would provide access to the register contents of all banks. This approach has the advantage that the register data of other contexts can be accessed in the same way as before banks were introduced, after the contents have been stored in memory.

To assess this technique, it is important to understand how memory accesses are performed on the OpenProcessor platform and other load/store architectures: generally memory is accessed by a number of instructions typically operating on an immediate and a register to address memory. This results in the need to first load the register contents from the register file in one pipeline stage, then computing the address from it and an immediate in another stage after which the memory can be accessed.

Since main memory is slow and accessing it is a typical bottleneck in an application the resulting minimum delay from instruction to data availability of 3 cycles as compared to 1 cycle for register access is not much of a disadvantage. When accessing the register file on the other hand this delay has a much bigger impact on performance.

Another side effect of this solution is that the stalling and result forwarding logic would need to be extended as there is an additional pipeline stage accessing the register file. Also the 3 cycle delay in itself is a disadvantage since it means access to registers on banks other than the current one would be nearly as slow as memory access in the case of a cache and TLB hit.

3.1.3 Inter Bank Copy

Access to register file contents could be performed by a new instruction dedicated to this purpose. There are at least three possibilities to encode the register file address: in (a) an immediate, (b) a register or (c) a combination of both.

If the address is encoded in an immediate, the accessed bank and register would have to be determined at compile time, which would remove the ability to dynamically assign banks to threads at runtime. For example this would prevent assigning different banks to multiple instances of the same binary and should therefore be avoided. Storing the address in a register is less problematic because

the address could be computed at runtime from the bank number and register that should be accessed.

Storing the address in a register would mean that the register file access could not be performed until after the register storing it has been read from the register file at which point another instruction might access the register file. This problem could be handled either by introducing stalls to enable the instruction to access the register file from different pipeline stages or the register file would have to be extended to support more accesses in one cycle. This can be done on the OpenProcessor platform by using an additional RAM block to implement the register file for read accesses, while write accesses can be delayed until the Write Back stage.

If a combination of a register and an immediate is used to address register file contents, this solution would be mostly equivalent to the memory mapping based solution presented above. The difference between the two solutions is that a special instruction would establish a new address space whereas the memory mapping based one would dedicate part of the existing memory address range to register file access. The memory mapping based solution has the advantage that no new instruction needs to be added to the processor.

Instead of encoding the address directly it can be encoded indirectly and evaluated using an indirection table stored in the processor. Combined with addresses encoded in an immediate neither of the drawbacks mentioned above would apply. Still a drawback, which applies to all variations of the Inter Bank Copy and Memory Mapping approaches, remains: since the contents of a register on another bank need to be loaded into a register on the current bank prior to their usage, solutions based on this approach would perform similarly to accessing them using memory in that they cannot be operated on directly. They need to be loaded into a register on the current bank instead before they can be operated on.

3.1.4 Direct Access

Instead of mapping to the main memory address space or establishing a new one, there is another address space that could be used: the register addresses encodeable in an instruction form an address space of their own, into which a part of the register file could be mapped in addition to the bank associated with the current context. Since, when using this approach, contents of registers on other banks can be accessed like conventional ones, a number of disadvantages of the other solution are not present.

Since registers on other banks can be accessed like those on the current bank, no extra read or write ports to the register file are needed. Since register file addresses can be calculated in the Operand Fetch stage, result forwarding and stalling logic can be adapted by extending it to take the bank number into account in addition to the encoded register addresses, which means they only need to be extended in bit width instead of changing the actual logic. As a result, an access to registers on a different bank does not introduce different stalls than an access to registers on the current bank. With regard to performance this means that access to register on other banks can be performed as fast as access

to registers on the current one.

A drawback of this solution is that register address space is valuable. This problem could be solved by using more bits to encode registers in instructions. On the OpenProcessor platform — and likely all others — this would require major architectural changes and was therefore discarded.

3.1.5 Direct Access with Split General Purpose Region

The final design of the register bank mechanism uses a variant of the Direct Access approach to enable fast communication between contexts. While the scarcity of register address space is an issue on the OpenProcessor, as can be seen below, this approach was chosen because of its promising performance characteristics.

The design considerations done so far leave a number of important factors undecided. Particularly the following questions need to be answered: (*a*) How many registers should be mapped at one time? (*b*) How many banks should be mapped at one time? (*c*) Where are they supposed to be mapped?

The situations presented in Section 3.1 have diverse requirements in terms of the number of required accessible registers. Thus some flexibility in this regard is preferable considering the scarcity of encodeable register addresses. If a sensible bank assignment policy is implemented by the operating system, bank assignment, which requires full access to the contents of a bank, can be considered to be a rather rare event compared to inter process communication, which has the next bigger demand in terms of the number of registers it needs to access. Thus, concerning the number of registers mappable at one time IPC performance should be the main concern. The largest IPC message sent by components of the operating system should thus serve as a guideline for the minimum number of registers which can be mapped at one time.

The number of mappable registers should also be a power of two to ease implementation and prevent multiple mapping locations for the same register, which would not be intuitive on assembly level.

Thus, concerning the number of mappable registers the following requirements should be met by the design: At least as many registers as are used by the largest IPC message should be mappable at once, with the maximum number of mappable registers being a power of two. The number of mapped registers should ideally be configurable, so the available register address space can be used in the best possible way.

A context always has access to its own and thus at least one register bank. The situation requiring access to the biggest number of banks is inter process communication. The kernel mode context of a thread transfers register contents from the bank assigned to its associated user mode context to the user mode context of another thread. This seems to indicate that access to at least two banks in addition to the current one should be supported, however this is not required considering the following: When the actual copying of register contents is performed by the kernel mode context, apart from a counter variable, it only accesses data on the banks between which the copy is being performed. Thus a routine to perform the copy operation could be implemented using either of the

banks to store the counter and thereby only requires access to two banks at a time.

This trick is only applicable when copying the raw register contents and leads to a bank switching overhead if typed items are to be transferred since typed items require additional work to be performed by the kernel. But, since transferring typed items requires memory access anyway, I expect this overhead to be insignificant. All other situations presented in Section 3.1 only access one other context besides their own and thus only access to a further bank besides the current one suffices.

There are in general two possible regions to which off bank contents could be mapped: the special or the general purpose region. On the OpenProcessor a number of addresses in the special purpose region are unused, but too few to fit the criteria stated above and not aligned in a useful way, thus complicating the implementation. Another option to utilize the special purpose region is to map the off bank register into this region and conceptually treat the special purpose registers as a bank. This would have the advantage that an entire bank of 32 registers could be visible while maintaining full visibility of the current bank.

While mapping to the general purpose area unavoidably leads to some registers of the current bank not being visible, this mapping location offers a great advantage as compared to the special purpose region: when mapping to the special purpose region, any code that needs access to off bank register contents has to be aware of banking whereas mapping to the general purpose region may enable transparent mapping in some cases.

For example, if a system call is performed and the bank assigned to the user mode context is mapped to the special purpose region, access to parameters could be done either by accessing the registers directly in the handler, which would result in copying to a register on the current bank without modifications to the compiler, or the stub could pass the system call parameter as a parameter to the handler by copying it to an appropriate register. When mapping to the general purpose region, the stub could map the user mode registers to the appropriate register address and then call the handler which could access them like a normal parameter without the need to perform a copy or modify the calling conventions. To make use of this advantage, off bank registers are mapped to the general purpose region.

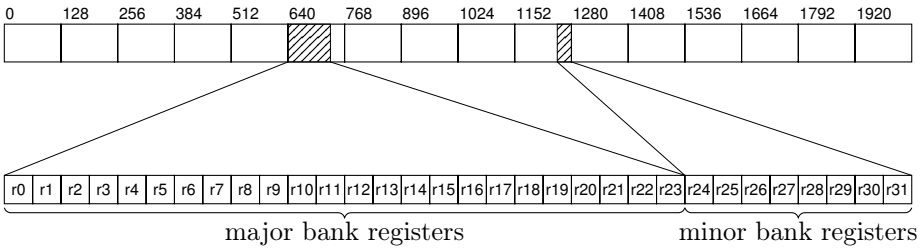


Figure 3.2: encoded register addresses (bottom) mapped to two banks in the register file (top)

The final translation mechanism shown in Figure 3.1 provides this functional-

ity by splitting the general purpose region of the encoded register address space into two parts. The lower part is meant to map to the bank associated with the current context which is referred to as the *major bank*, since generally more registers of the current bank are visible than of the other. The upper part is mapped to the bank the current context needs to communicate with, this bank is referred to as the *minor bank*, since generally less of its registers are visible than of the major bank.

While this design has been sufficient in the implementation, which only used register banks to increase kernel entry and exit performance, the following considerations seem to suggest that adding an offset into register address space for the mapping region might allow a further increase in performance: the design uses mapping to the general purpose region to allow transparent mapping in some situations. One of the situations transparent mapping could be used is if a client performs a call to a server. Instead of copying the register contents from one bank to another the kernel could map the message registers containing the send items into the region specified as receiving message registers by the server. Especially for highly optimized IPC implementations, like fast path IPC of L4 [4], this might offer a substantial increase in performance.

3.2 Kernel Changes

Most changes to the kernel design depend on the presence of a banking mechanism that allows a full switch of registers and the possibility to access contents on other banks. An exception to this is IPC since it is used to perform communication between contexts and thus needs to use the new communication channel presented above.

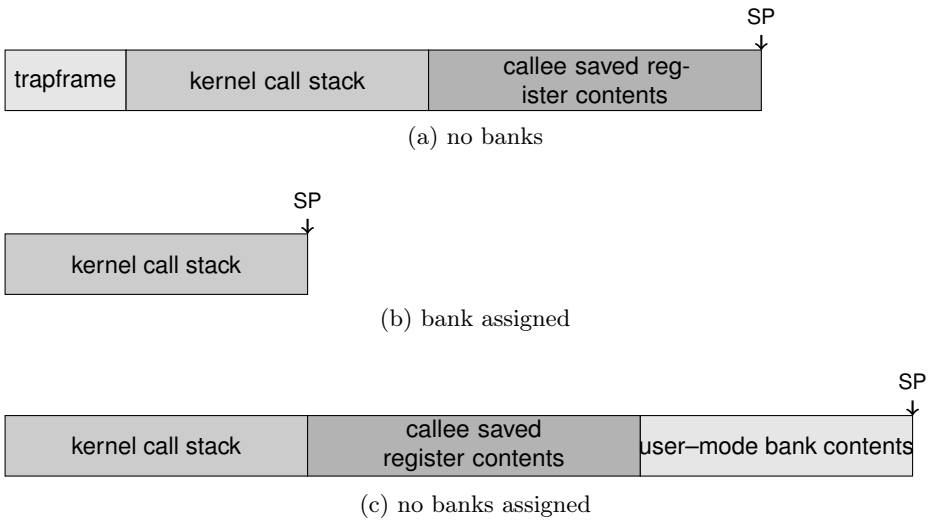
First Section 3.2.1 describes the changes to the context switching parts of the kernel by means of the changed stack layout of a suspended thread. Then the new approach to perform IPC is presented in Section 3.2.2. Finally Section 3.2.3 concludes this chapter by describing approaches to handle kernel reentry in the presence of register banks, a problem neglected so far because reentry is uncommon on the OpenProcessor.

3.2.1 Context Switches

The changes to context switching can best be understood in terms of the changed stack layout of a suspended thread as shown in Figure 3.2. Figure 3.2a shows the layout used by the unmodified kernel.

When the unmodified kernel is entered, the caller saved registers are pushed onto the kernel stack by the stub before jumping to the compiler generated handler code. This is possible since all compiler generated kernel code abides the calling conventions and will thus preserve all callee saved registers on the call stack.

When the thread is to be suspended, the compiler generated code will save the caller saved registers using the kernel call stack before jumping to the switching procedure. The switching procedure will then push the callee saved registers to the stack before switching to the stack associated with the target thread.



(a) no banks

(b) bank assigned

(c) no banks assigned

Figure 3.3: Kernel stack layouts of a suspended thread

With the banking mechanism in place, only the kernel call stack remains as shown in Figure 3.2b. Since all register contents of the user and kernel mode context are preserved by switching the bank, the kernel call stack does neither need to contain callee saved register contents (since their content is never used) nor the caller saved register contents from kernel mode. However, this is not possible without modifications to the compiler, for example by using special calling conventions when generating handler code.

When the banks assigned to a thread need to be freed up to reassign them, not all of their contents need to be saved, since the kernel mode caller saved registers were pushed onto the kernel call stack before the thread was suspended. What remains to be saved are the user mode bank contents and the callee saved register contents from the kernel mode bank. This results in a stack, shown in Figure 3.2c, that is bigger than the one in the unmodified kernel. This could be prevented by mapping the caller saved user mode registers while running in kernel mode, since the compiler generated code will preserve them using the stack anyway. As a result, the stack of a suspended thread would be as big as in the unmodified kernel.

However this approach has a drawback as compared to modifying the compiler to incorporate the fact that kernel mode is entered with these register contents already preserved: transparent mapping results in premature saving of the registers using memory when they could still be stored in the register file. This leads to two problems: since the callee saved registers of the bank assigned to kernel mode are never visible, a fourth of the register file memory is wasted, if a thread always is assigned two banks at once. If a lazy assignment policy is applied that only assigns banks when they are needed, a fourth is a lower bound for wasted register file memory with the upper bound being a half of the memory if all banks are assigned to kernel mode contexts. The other problem occurring is that accessing memory when there is no need to do so degrades performance.

3.2.2 Inter Process Communication

In general, when performing inter process communication in the presence of register banks, four cases can be distinguished, according to which of the communicating contexts currently have a bank assigned to them. If a context does not have a bank assigned to it, the register contents associated with it are stored on its associated kernel stack. To make them accessible, they should be stored in a well defined structure. Using the trapframe for this purpose is convenient as it is already accessed using a pointer in the thread control block and thus no changes to the IPC code in the case of memory to memory IPC were necessary. If the compiler is modified to take banking into account, extending the trapframe or providing a new structure containing all general purpose registers has the advantage that an increased number of items could be transferred.

The remaining three cases should each be handled using a function optimized for the specific case. As stated above, in the OpenProcessor kernel, message register contents are always copied by the kernel context associated with the sending context. Since the kernel provides a combined system call for sending and receiving and the common case is that both are performed, it can be assumed that in general the address of the receiving register is lower than the address of the corresponding sending register.

If neither of the participating contexts has a bank assigned to it, the bank assigned to the senders kernel context can remain the major bank. If the sending context is the one with an assigned bank it can be mapped as the minor bank with all message registers containing send items mapped at once. Since the sending registers always start at the same register address, the copying is best performed by encoding the memory address of the first receiving message register in a register and then jump to the appropriate target in a sequence of store instructions that write each possible message register to the appropriate offset into the memory address of the receiving message register contents of the receiving context. The jump target is calculated depending on the number of registers that need to be transferred.

The case that the sending context has no bank assigned but the receiving context has only occurs in two situations: if typed items had to be transferred or the receiver was not ready to receive the message. These are the only situations in which a context switch, which might have resulted in reassigning the bank of the sender's user mode context, could have occurred between kernel entry and performing IPC. Both are situations in which the IPC operation is already costly and I consider this case to be the least important of the three cases where banks store either of the participating contexts when considering performance.

In this case, copying can be performed by encoding the memory address of the sending message "register" in a register, but in this case the addresses of the participating registers might be offset as well. As a result the trick applied above to encode the number and address of the register, that should be transferred, in the instruction pointer cannot be applied so easily. Encoding the register file offset instead of a bank number would solve this problem, as would an offset for the mapping region, since this would provide a possibility to map the first receiving register to a fixed location. In the absence of such an offset, the solution

I propose for this situation is to either use memory for temporary storage (when initiating the receive operation, in effect freeing up the user mode bank) or provide a table pointing to a number of functions that read needed minor bank registers into a major bank register. Another approach is to use the trick above, but using one function for each possible starting receiving register instead of only one function.

The last case to be considered is when both the sending and receiving context have a bank assigned. As distinguished from the other cases, the kernel context of the sending context cannot remain mapped into the register address space if data is to be copied directly from the banks associated with the sending context to the bank associated with the receiving context. Instead, the bank associated with the sender should be mapped as the minor bank and the bank associated with the receiver as major bank, since message registers containing send items can be expected to have an address greater or equal to their target register. With the design presented above copying would still be cumbersome since there is no way of addressing registers besides a jump table. However, if an offset for the mapping area is used, copying is not necessary since transparent mapping can be applied. Another approach is to allow an offset into the major bank that can be set to map the receiving registers to a fixed starting location in which case the trick used for register to memory IPC can be applied.

3.2.3 Reentry

A side effect of changing the register bank on kernel entry has not been covered so far: if a kernel reentry occurs, the current thread is already operating on its kernel mode context and thus its kernel mode bank. As a result of this it is not possible to switch to the kernel mode bank to preserve the executing kernel mode context. There are two possible approaches to this problem: either switch to another bank or use the kernel stack to store the current register contents as it was done before register banks were introduced.

When switching the bank on reentry the bank switched to can either be reassigned from a different context on demand or a number of banks could be dedicated to this purpose. Reassigning a bank on demand makes the reentry slower than before since not only a register set needs to be stored in memory but a bank switch is performed additionally. Dedicating a number of banks to this purpose has the disadvantage that register file memory is expensive and the maximum reentry depth is limited to the number of banks dedicated to it where before it was limited only by the maximum kernel stack size.

Which of the three solutions is the best fit for a system depends on a number of factors. In general reassigning a bank on reentry is worth the higher cost if threads are suspended on a regular basis when they are reentrant. If reentry is a frequent event, dedicating a number of banks can pay off. However, neither is true for the OpenProcessor platform and thus in this case storing the current register contents on the kernel stack is the best solution.

Storing current register contents on the kernel stack leads to another problem: exception handling code needs to be aware of two different ways to access register contents of the causing context. This problem is solved by duplicating the

handler implementation, with one version for reentry and one for entry from user space. In a system where reentry is more common than on the OpenProcessor platform this would result in a performance overhead due to increased demand on the TLB and cache.

Chapter 4

Implementation

In this chapter I cover the implementation of the register bank mechanism and its kernel support as described in Chapter 3. While I fully implemented the basic banking mechanism in the processor, the operating system modifications are limited to preserve and restore register contents on kernel entry and exit, removing any need for memory access to preserve the user mode context. Using register banks to perform thread switching or inter process communication has not been implemented.

The changes to the processor were tested using Ikarus Verilog [16] to simulate the hardware which executed a special application that mimicked the entry and exit code of the original kernel — the full kernel code was too big for simulation. Once simulation results indicated functional hardware, the hardware was synthesized to run the kernel on the FPGA development board. Synthesis was performed by Xilinx ISE [17].

Since the kernel execution could not be simulated, a combination of simulation of code snippets, debug output by the kernel and output from the OpenProcessor's trace unit was used for debugging purposes.

4.1 Processor

As stated above, the basic version of the register bank mechanism design described in Section 3.1 was implemented. It is based on bank selection using bank numbers, as opposed to register file offsets, for major and minor bank and without an offsettable mapping region. This section describes all changes done to the implementation following a bottom up approach.

The register file of the unmodified processor already offered a 4 bit set selector for each of its three ports. As stated in Chapter 2, every port takes a 5 bit address for access to the register file memory, allowing to address the 32 general purpose registers. The block RAM used to implement the register file is byte addressable using 11 bit addresses. The existing register file addresses the block RAM by wiring the set selector to bits 10–7, the register address to bits 6–2 and constant zero to bits 1 and 0. This addressing scheme provides access to the whole block RAM divided into 16 non-overlapping sets of 32 32 bit words. Since this is exactly what is required to implement the design, no changes to the

register file were necessary.

4.1.1 Register Address Translation

The two read ports of the register file are accessed from the Operand Fetch stage while the Write Back stage accesses the write port. The Operand Fetch stage is also responsible for decoding the source operand and thus neither the source register addresses nor their set number need to be tracked to further pipeline stages. In contrast, the destination operand is decoded in the Execute stage and thus its address and set number need to be tracked to the Write Back stage.

Both stages evaluate the set selector for each operand according to the following formula, where m is the minor bank number, M is the major bank number, c is the minor bank register count and e is the encoded register address:

$$\text{set} = \begin{cases} 0 & e \geq 32 \implies \text{special purpose register} \\ \begin{cases} m & e > 31 - c \\ M & \text{else} \end{cases} & \text{else} \implies \text{general purpose register} \end{cases}$$

While setting the set selector to zero on access to a special purpose registers is not necessary, since no access to the register file needs to be performed, it simplifies adapting the forwarding logic to register banks, since special purpose register do not need to be handled differently by it.

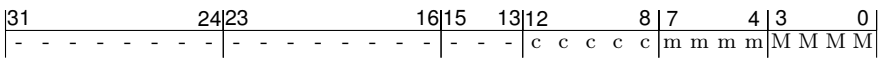
The forwarding logic is the reason why the set selector for result registers needs to be evaluated in the Execute stage — otherwise the translation configuration would need to be tracked and the forwarding logic would need to calculate the selector for every stage between Execute and the one that finally evaluates the set selector. Another approach would be to operate on the same register bank configuration in every pipeline stage in effect requiring a pipeline flush whenever it changes.

With the set selector available as soon as the operands are decoded and tracking the destination operand set selector and address from the Execute to the Write Back stage, the forwarding logic can be adapted to register banks by augmenting it with a comparison of the set selector in addition to the operand address.

All the aforementioned changes to the pipeline make it in effect operate on register file memory addresses instead of encoded register addresses.

4.1.2 Translation Control Register

To encode the configuration of the translation logic, a special purpose register was added to the processor. Its encoding is shown in Figure 4.0.



c: minor bank count; m: minor bank id; M: major bank id; -: unused

Figure 4.1: Control Register Encoding

As stated in Section 2.2.1, special purpose registers on the OpenProcessor are implemented directly in the FPGA fabric instead of block RAM. This is necessary due to implicit accesses to them in several pipeline stages. As a side effect of its implementation the forwarding logic does not apply to implicit reads of special purpose registers. As a result, explicit updates to special purpose registers are only visible to implicit reads after the instruction that updates the special purpose register has completed the Write Back stage. This leads to a delay of 4 cycles for the new value to take effect after the instruction writing to it has passed the Operand Fetch stage.

For the existing special purpose registers, this delay is of little significance because they are rarely and in some cases never updated explicitly. If explicit updates are performed, they are usually done well before the registers are evaluated. This is different for the bank translation control register.

First, all writes to it are explicit since banking is controlled by the operating system, while it is read implicitly nearly every time an instruction enters the Operand Fetch stage.

Second, updates to it can be frequent at times since parts of the kernel need random access to minor bank register contents in the current implementation. This is performed by changing the minor bank register count, reading the appropriate minor bank register into a major bank register and then resetting the minor bank register count, as described in Section 4.2. While the design with all extensions to the register address translation would remove this need, a pipeline stall until the control register value is committed would still reduce kernel entry and exit performance. Evaluation has shown that forwarding the control register reduces the time to perform a kernel exit by one cycle.

Third, a bank switch occurs when a kernel entry is performed. Since this cannot be predicted in every case and unprivileged mode cannot write the control register for security reasons, the visibility delay always results in a number of instructions that operate on the old configuration. To make matters worse, it cannot be predicted how many of the instructions operate on the old configuration if some of these instructions might result in a pipeline stall or even an exception.

Since in most cases it has to be assumed that an instruction can cause an instruction fetch stall, predicting which instructions still work on the old configuration is rarely an option. This leaves the option to use instructions not affected by bank selection after a change to the control register. These fall into two categories: (*a*) instructions not operating on registers, in the case of OpenProcessor this is true only for instruction pointer relative branches operating only on an immediate; (*b*) instructions which operate only on registers the location of which is independent of the change, e.g. the special purpose registers. Neither of these instructions were found to be useful after a change to the control register, let alone four of them to fill all cycles in which bank configuration was uncertain. In an early stage of the implementation, `NOP` instructions were added instead after each update of the control register.

To address this issue, without the need to add `NOP` instructions after most bank switches and thereby increase binary code size unnecessarily, two approaches were taken: one solution is to stall the pipeline until the register has been updated, which still has the same performance characteristics but keeps code size down.

To improve performance, the next approach was to add dedicated forwarding logic to complement the stalling logic.

It is a problem with the forwarding logic that it resulted in a critical path that reduced the safe operating frequency of the resulting system from 50 MHz to 37 MHz. This critical path is a result of complex ALU operations to a considerable extend the results of which are used in the same cycle to translate a register address. Since no actual wrong system behavior was observed when operating the system at 50 MHz, this issue was not investigated further. I assume it could safely be resolved by introducing a one cycle stall or delay after the new control register value has been determined, if restructuring the translation logic or avoiding the use of certain ALU operations with the control register does not suffice.

Lastly, tracking the control register value to the Execute stage has not been covered yet. The reason is that this is not done and is actually not necessary as long as no load instructions are used to update the control register. Not tracking the control register results in the possibility that an instruction uses a translation of its destination operand different from the translation of the source operands. However this can only occur if the control register is updated while an instructions is in the Execute stage. With stalling until the new value has been committed this can never occur. If forwarding is used for the control register this happens when the instruction in the Execute stage writes to the control register, in which case the control register does not need to be evaluated.

Another possible location for control register updates is the Write Back stage, if a load instruction was used to update the control register. In this case the operation in the Execute stage might in fact be affected by the change and use a different configuration for its source and destination operands, if forwarding is used for the control register. Thus this approach is safe as long as no load instructions are used to update the control register, which was not necessary for the implementation of the kernel changes.

4.2 Kernel

To prove the feasibility of the kernel design described in Section 3.2 I changed the OpenProcessor kernel to use banks to preserve the user mode context on kernel entry. This is achieved by assigning the same two banks to any thread when it gets scheduled, in effect having one bank reserved for user mode and one for kernel mode. When a thread is executing on its user mode context, it always has full visibility of the associated bank, while the kernel code responsible for context switches operates on an intermediate state where parts of the user mode bank are visible. Other kernel code accesses the user mode bank contents by temporarily mapping the user mode bank for each register it needs to access.

Even though this limited implementation only improves kernel entry and exit performance, all code parts of the kernel that directly deal with threads had to be adapted. Section 4.2.1 describes how banks are utilized to increase the performance of kernel/user mode transitions and the resulting changes to exception handlers. The necessary changes to the rest of the kernel are covered

in Section 4.2.2.

4.2.1 Kernel Entry and Handler Code

Before banking was implemented, the kernel passed the same parameters to any exception handler: the faulting instruction pointer, the fault address and a pointer to the trapframe. Instead of using the stack to preserve user mode caller saved register contents when the kernel is entered, a switch to the kernel mode bank is performed. As a result, no trapframe is constructed and thus no trapframe address can be passed to the handlers as a parameter. This requires a new approach to access user mode register contents, particularly from within the system call handler.

Since the trapframe is not available anymore, it was removed as a parameter and replaced by two parameters to directly pass user mode arguments in the case of a system call. While it would be possible to transparently map the argument registers to kernel mode, they are copied instead for the following reasons: first, the implementation of the banking mechanism does not allow an offset into the minor bank or for the mapping region and as a result the minor bank always contains r31, which is the link register. Since its contents, unlike the argument registers, needs to be preserved to return to user mode, a memory access would become necessary. Second the code providing random access to user mode register contents, described below, resets the minor bank register count to zero.

In the case of reentry, as described in Section 3.2.3, a trapframe is still constructed. This makes it necessary to distinguish two methods of access to user mode register contents in some parts of the kernel and all handler code. Deciding which method to use every time a user mode register is accessed would result in a branch for each access. Since this would result in overhead in terms of both code size and performance, two versions of each handler are generated by the C preprocessor: one version uses banks for normal entry and the other one uses the trapframe for use on reentry. While this approach has an even bigger impact on code size than a case by case decision, it can be expected to have a lower impact on performance and the cache, since half of the code, the handlers invoked on reentry, is rarely executed and does not have an impact on the cache if not invoked, if it is linked properly.

Before banking support was implemented, kernel code that needs access to user mode register contents, invoked by the system call handler, would get the trapframe passed as a parameter to acquire the parameters passed to the system call and return results. However most functions called from the handler only need access to the four argument registers and one register to return results. Thus I could adapt functions called from within the system call handler to take their parameters and return a result directly instead of passing them in the trapframe.

An exception to this is the function performing IPC, since it needs access to up to 14 user mode registers. This is handled by passing a value indicating the absence of a trapframe instead of an address to it to the function. This is sensible when considering performance since IPC does not need random access to

register content instead it accesses user mode register contents in one block. The changes to the code performing IPC are described in more detail in Section 4.2.2.

Since the registers an exception handler needs to access are known at compile time, access to them is facilitated by a number of inline functions, in effect requiring 4 instructions for each access. This approach was taken because the inline functions were required to provide access to generic user mode register contents for debugging purposes already and thus was simple to implement. However all the registers accessed had their content in fact available on the kernel mode bank, but access to those values would have required a distinction of case in the preprocessor, which is not possible using simple means, or less maintainable handler code.

4.2.2 Threading Code

Other locations in the kernel code that required modification for the implementation of register bank support without directly benefiting from it are: thread creation, thread startup, thread switch and inter process communication.

Thread creation code needed to be adapted because the user mode register contents were set completely on thread switch in this implementation. As a result, the stack layout of a new thread changed. However, if the transparent mapping technique described in Section 3.2.1 had been applied, no change would have been necessary.

Since user mode register contents are already restored on thread switch, thread startup code does not need to set their initial values anymore. The startup code was thus reduced to perform an initial kernel exit.

In this implementation, bank assignment is the responsibility of the thread switching function. In addition to backing up and restoring callee saved kernel mode registers and the kernel mode stack pointer, it is now responsible for preserving the user mode bank. This is performed by backing both banks up to and restoring them from the kernel stack of their associated thread.

The last kernel change that needs to be covered is the changed IPC code. Since in this implementation the sender would always have a bank assigned to its user mode context whereas the receiver would not have a bank assigned, only this case needed to be covered as described in Section 3.2.2.

At the point the IPC code was changed it was done to create a runnable system to test the changed kernel entry and exit code and thus was implemented with a fast implementation and not performance in mind. Thus the trapframe as a structure to hold the message registers of the receiver was not removed but rather created and restored on demand. A better approach to this would have been to create it appropriately on thread switch, since its contents are saved on the kernel stack anyway due to bank reassignment.

Chapter 5

Evaluation

In this chapter I present an evaluation of the register bank mechanism implementation described in Chapter 4. Since I only implemented banks to increase kernel entry and exit performance and the current kernel implementation in general is not optimized for performance, only two micro benchmarks were evaluated.

One benchmark measures the number of cycles to perform a null operation system call to allow an evaluation of the impact on actual system call performance. The other benchmark measures the number of cycles necessary for kernel entry and exit, including register backup and restoration, using the `SYSCALL` instruction to perform kernel entry.

In the rest of this chapter first Section 5.1 describes the hardware/software combinations that were benchmarked. Finally Section 5.2 presents and discusses the benchmark results.

5.1 Combinations

All benchmarking was done on the following hardware/software combinations:

Vanilla This combination consists of the unmodified operating system, extended by testing code. The hardware version used has no influence on results since they behave exactly the same when the control register is not changed. Vanilla benchmarks provide reference values for the evaluation of the other pairs.

Banked This combination consists of the modified operation system and a hardware build with register bank support but no special forwarding logic for the control register. The results of this benchmark allow an evaluation of performance increases due to using register banks to increase kernel entry and exit performance.

Forwarded This combination is similar to the Banked one except that forwarding for the control register was enabled in the processor. This combination allows evaluation of further increases in performance due to forwarding.

I evaluated the hardware version without control register forwarding separately because the forwarding logic did not meet all timing constraints according

to the synthesis tools [17]. While I observed no errors when using control register forwarding, the results obtained when it was used should be viewed critically. I present them to provide an estimation of the upper bound of possible increases in performance by improving the hardware. For example, if the control register was set implicitly by `SYSCALL` and `RTU` in a way that the system call stub and exception trampoline did not need to modify it, performance of the benchmarked operations would be similar to "Forwarded" even without forwarding of the control register.

5.2 Results

The benchmarks show an absolute reduction of 20 cycles (cyc.) in kernel entry, 27 cyc. in kernel exit and 53 cyc. in null operation system call cost for Banked compared to Vanilla. Forwarded shows a further improvement of 8 cyc. in kernel entry, 1 cyc. in kernel exit and 9 cyc. in null operation system call performance. Figure 5.0 shows the individual benchmark results.

Two observations resulting from the benchmarks are of particular interest: first, the number of cycles saved on kernel exit exceeds the savings on kernel entry in Banked even though the original kernel restores fewer registers on kernel exit than are saved on kernel entry. Second, the increase in performance for the null operation system call exceeds the sum of the increases in entry and exit, proving that further improvements of the system call performance are possible by using register banks.

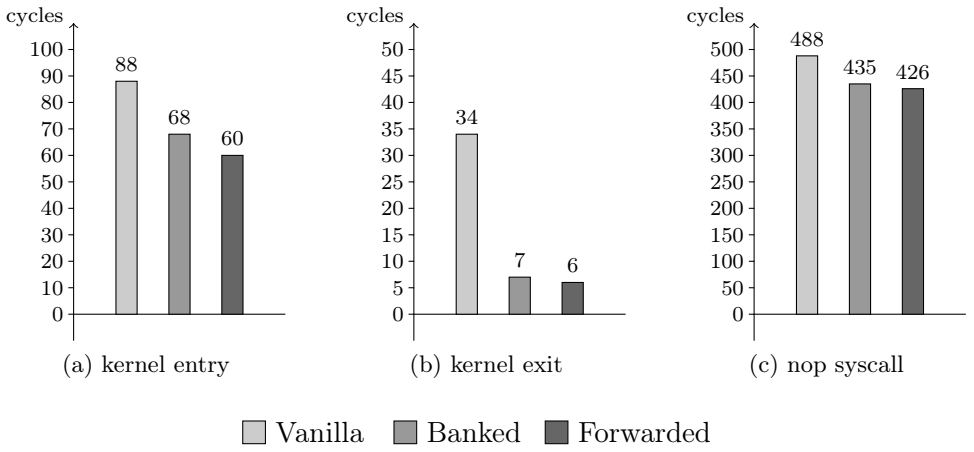


Figure 5.1: Benchmark Results

The (unexpected) difference in performance improvements of kernel entry and exit are in part a result of optimizing the entry code for system call performance. Instead of only preserving the contents of user mode registers, the stub now moves the link register contents to a special purpose register that indicates the faulting instructions address in the case of other exceptions. The advantage of this is that register access to the system call return address from the system call

handler is possible without having to switch to the user mode bank. Additionally, the contents of argument registers r30 and r29 of the user mode bank are copied to r28 and r27 on the kernel mode bank, effectively passing them as additional arguments to the handler. As a result, an additional change of the bank control register is necessary.

These four operations increase the time necessary to perform kernel entry by at least 7 cycles (3 for the register accesses, 4 for changing the bank control register) and should be attributed to calling the handler rather than to the kernel entry proper, which was not measured for the kernel entry in Vanilla. If we add these cycles to the saving on kernel entry in Banked, the performance improvement for kernel entry are equal to those of kernel exit. However, one would expect the improvements in kernel entry time to exceed those of the kernel exit time, since in Vanilla more registers are saved on entry than are restored on exit. This can be explained when considering that Forwarded only improves exit times by 1 cycle compared to Banked: `RTU` introduces a number of pipeline stalls, which do not affect the instruction changing the control register since it is executed in the branch delay slot of `RTU`. Thus switching to the user mode bank results in 1 additional stall cycle, whereas switching to the kernel mode bank results in a stall of 4 cycles.

Results of the nop system call benchmark should be considered with caution, because they do not capture the full increase in performance possible from register banks. Since the compiler generated handler code preserves 13 registers (r4–r11) using the stack, which at that point do not contain any values that ever are used. With a slight change to the calling conventions used for the handlers these 26 unnecessary memory accesses for saving and restoring these 13 “dead” registers can be prevented and at least 26 cycles saved when performing a system call. Since the total performance improvement in the system call benchmark is only 53 cycles for Banked and 60 cycles for Forwarded another improvement of at least 26 cycles due to register banks is significant.

Chapter 6

Conclusions and Future Work

6.1 Conclusions and Closing Remarks

In this thesis I presented a mechanism to reduce context switching overheads by utilizing increased register file memory. Experiments have shown that the mechanism can be utilized to decrease system call overhead by at least 10%. I also presented a number of possible improvements to the OpenProcessor platform and banking mechanism that are likely to provide further performance increases.

The presented banking mechanism distinguishes itself from similar approaches by taking two properties of microkernels using one kernel stack per thread into account: (*a*) if a kernel uses one kernel stack per thread, a thread can be considered to have two associated contexts; (*b*) many operations that only require a kernel entry in monolithic kernels, additionally require a thread switch on a microkernel. Thus the performance of these operations cannot be improved by traditional approaches like shadow registers [11] that improve kernel entry performance but increase the cost of thread switching by a similar amount.

The design is based on a number of assumptions: (*a*) performance critical kernel code accesses user mode register contents in continuous blocks; (*b*) registers can be ordered and used so that kernel code that accesses one user mode register also needs access to all user mode registers with a higher encoded register address, at the same time it does not need to access these registers on the current bank; (*c*) the compiler can be extended and the kernel source code modified accordingly to take into account that neither handlers need to preserve callee saved registers nor do caller saved registers need to be preserved before performing a thread switch; (*d*) kernel reentry is rare and not performance critical.

Assumptions (*a*) and (*b*) have been discussed in Chapter 3 and are true if a special purpose register instead of the link register is used to store the return address when a `SYSCALL` is issued. An exception from this, for assumption (*b*), is when a thread with a register bank assigned to its user mode context receives an IPC message from a thread with no bank assigned to its user mode context. In this case there are unneeded registers with an encoded address higher than needed registers. This problem can be solved by changing this assumption and for example adding a offset for the mapping area.

Assumption (*c*) can be supported for kernel entry by the fact that GCC [10] offers function attributes for similar purposes on several architectures. For thread switches this could be achieved by using inline assembly to perform the call to the thread switching function. Assumption (*d*) has been supported by further development of the OpenProcessor platform, which was done independently of this thesis and eliminated kernel reentry under regular conditions.

6.2 Future Work

Since I did not fully implement usage of the register bank mechanism in the kernel and described a number of improvements which remain unimplemented, the results of this thesis should be viewed as a preliminary evaluation of the feasibility of the concept. In this section I will first describe a number of changes to the OpenProcessor platform that could allow a more accurate evaluation of the concept, then I will list questions about the concept in general that arise from this thesis but remain unanswered.

Possible improvements that allow a better utilization of the register bank mechanism are:

Storing the `SYSCALL` return address in `fault_next_ip`
: instead of the link register would be beneficial for a lot of reasons, the most important ones are: (*a*) system calls could be handled more similar to other exceptions; (*b*) it could be accessed from within the system call handler directly without the need of passing it as a parameter and thus allow (*c*) to pass (all) four system call parameters in argument registers to the system call handler.

Changing the link register to a lower register
: allows to use r31-r28 as argument registers and thus mapping them transparently when the system call handler is executed.

Eliminating the exception stubs
: would improve kernel entry performance and is possible when using register banks. This is because the kernel stack can now be loaded in the trampoline, reducing the functionality performed by the stubs to switching to the kernel mode bank, set a register to the handler address and then branch to the trampoline. Since this is a small number of instructions, this could be performed by the exception vectors, if their size was increased. As a result one branch on kernel entry could be eliminated, the unused branch delay slot in the vector could be utilized and r1 might be freed to be used in user mode.

Removing unneeded register preservation on the kernel stack
: would improve general handler performance. If banks are employed to perform fast thread switches, additional savings would also become possible.

Implicit bank switches on kernel entry and exit
: were excluded from the design to not restrict the bank assignment policies

implementable by the operating systems. If the bank assignment policy is known however, depending on the policy, implicit bank switches on kernel entry and exit might become feasible.

More general questions with regard to the register bank mechanism are as follows:

How much can banks increase thread switching performance?
Since the register bank mechanism was designed with thread switching in mind, the possible increase in thread switching performance are interesting.

How much can banks increase IPC performance?
IPC was a big consideration in the design of the register bank mechanism the questions of how and by how much IPC performance can be increased is important.

Evaluation of the improvements mentioned above
Since at least 26 cycles are wasted in the current system call handler to perform unneeded work, the question arises how much all of the improvements mentioned above can increase the performance of kernel entry.

What is an efficient bank assignment policy?
As with any scarce resource in a computing system its impact on total system performance is highly dependent on the used assignment policy. Such a policy has so far only been assumed to exist for register banks. In a small and predictable system the banks might be assigned statically to a number of threads but in a more complex one dynamic reassignment of banks based on run time information is most likely needed.

How much can register banks increase total system performance?
Assuming a perfect assignment algorithm, how much can register bank increase the performance of an actual system? The impact is clearly dependent on the actual system, but a methodology to determine a best case performance improvement allows an evaluation of available assignment algorithms.

Bibliography

[1] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 2: System Architecture*, May 2010. Revision 2.3.

[2] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, St. Malo, France, October 5–8 1997.

[3] Kun-Yuan Hsieh, Yung-Chia Lin, Chien-Chin Huang, and Jenq-Kuen Lee. Enhancing Microkernel Performance on VLIW DSP Processors via Multiset Context Switch. *J. Signal Process. Syst.*, 51(3):257–268, 2008.

[4] Jochen Liedtke. Improving IPC by Kernel Design. In *Proceedings of the 14th Symposium on Operating System Principles (SOSP-14)*, Asheville, NC, December 1993.

[5] Jochen Liedtke. *Lazy Context Switching Algorithms for Sparc-like Processors*. Technical Report 776, September 1993.

[6] Jochen Liedtke. On Microkernel Construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995.

[7] MIPS Technologies Inc. *MIPS© Architecture For Programmers Volume III: The MIPS32® and microMIPS32™ Privileged Resource Architecture.* 955 East Arques Avenue, Sunnyvale, CA 94085-4521, July 2010. Version 3.05.

[8] Raphael Neider. OpenProcessor. Karlsruhe Institute of Technology.

[9] Richard Herveille (Steward), OpenCores. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores.*

[10] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection.* GNU Press, Free Software Foundation, 51 Franklin Street, Fifth Floor Boston, MA 02110-1301 USA. For gcc version 4.5.0.

[11] David Seal. *ARM Architecture Reference Manual.* Boston, MA, USA, second edition, 2001.

[12] Carlo H. Séquin and David A. Patterson. *Design and Implementation of RISC I*. Technical Report UCB/CSD-82-106, EECS Department, University of California, Berkeley, October 1982.

[13] SPARC International, Inc. *The SPARC Architecture Manual V9*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[14] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems : design and implementation*. The MINIX book. Prentice Hall, Upper Saddle River, NJ, third edition, 2006.

[15] D. W. Wall. Register windows vs. register allocation. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 67–78, New York, NY, USA, 1988. ACM.

[16] Stephen Williams. Icarus Verilog. http://www.icarus.com/eda/verilog/. Verilog HDL synthesis and simulation tool.

[17] Xilinx Inc. Xilinx ISE 11.4. HDL synthesis toolchain.

[18] Xilinx Inc. *Virtex 4 FPGA User Guide*, 2008. version 2.6.