A Framework for Investigating Workload Interactions on Multicore Systems

by

Eric Charles Matthews

B.A.Sc., Simon Fraser University, 2009

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Applied Science

In the
School of
Engineering Science

© Eric Charles Matthews 2012

SIMON FRASER UNIVERSITY

Spring 2012

All rights reserved.
However, in accordance with the *Copyright Act of Canada*,
this work may be reproduced, without authorization, under the conditions for
"Fair Dealing." Therefore, limited reproduction of this work for the purposes of
private study, research, criticism, review, and news reporting is likely to be
in accordance with the law, particularly if cited appropriately.

APPROVAL

Name: Eric Charles Matthews

Degree: Master of Applied Science

Title of Thesis: A Framework for Investigating Workload Interactions on Multicore Systems

Examining Committee:

Dr. Bonnie Gray, P.Eng.
Associate Professor of Engineering Science
Chair

Dr. Lesley Shannon, P.Eng.
Assistant Professor of Engineering Science
Senior Supervisor

Dr. Alexandra Fedorova
Assistant Professor of Computing Science
Supervisor

Dr. Arrvindh Shriraman
Assistant Professor of Computing Science
Examiner

Date Defended: January 10, 2012

Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website (www.lib.sfu.ca) at http://summit/sfu.ca and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2011

Abstract

Multicore processor systems are everywhere today, targeting markets from the high-end server space to the embedded systems domain. Despite their prevalence, predicting the performance of workloads on these systems is difficult due to a lack of visibility into the various runtime interactions for shared resources. At the same time, existing support in processors for performance monitoring is primarily limited to capturing single events and thus cannot provide the depth of information required to predict dynamic workload interactions.

This thesis presents a configurable and scalable multicore system framework for enabling new avenues for systems and architectural research. We have outlined and implemented a multicore platform on a Field Programmable Gate Array (FPGA), which supports the Linux Operating System (OS) and contains an integrated profiling unit, ABACUS (a hArdware Based Accelerator for Characterization of User Software), through which runtime profiling of the system can be performed. Support for systems of up to four cores has been demonstrated and the stability of the system verified through various stress tests. A system uptime of over two weeks was obtained while the system was under heavy load, and with the integration of our hardware profiler, we were able to gain insight into the operation of the system through the collection of metrics not readily possible through existing systems and simulators.

Acknowledgments

First, I would like to thank the members of Xilinx's University Program and Research labs for the generous equipment donations; in particular, I wish to acknowledge the time and efforts of Dr. Patrick Lysaght, Mr. Ralph Wittig, and Mr. Rick Ballantyne, who made this research possible. In addition, the support received from many people, especially my family and friends, has helped me make it through. This work would surely not have been possible without the constant support (and nagging) of my parents, whom I can't thank enough. Thank you everyone in the RCL lab, for making the long days both bearable and, often, enjoyable. To my supervisor, Dr. Lesley Shannon, thank you for your support and guidance throughout my time working in your lab. It is because of the rewarding experience I've had working here that I will be continuing on as a Ph.D. student under your supervision.

Finally, I would like to acknowledge the following list of organizations for their financial support and/or equipment donations that have made this research possible: Xilinx, the Natural Sciences and Engineering Research Council, the Canadian Microelectronics Corporation and Simon Fraser University.

Contents

Approval ii

Abstract iii

Acknowledgments iv

Contents v

List of Figures ix

Glossary x

1 Introduction 1

1.1 Motivation 1

1.2 Objective 2

1.3 Contributions 3

1.4 Thesis Organization 4

2 Background 5

2.1 Soft-processors and Processor Emulators on FPGAs 5

2.2 MicroBlaze 7

2.2.1 MicroBlaze and Linux 7

2.2.2 Memory Management 8

2.2.3 Conditional Load Store Instructions 8

2.3 Multicore Computer Architecture 8

2.3.1 Memory Coherency 9

2.3.2 Asynchronous Communication . . . 10
2.4 The Linux Kernel . . . 10
2.4.1 Architecture Support . . . 10
2.4.2 SMP Support . . . 11
2.5 Standard Methods for Workload Performance Analysis . . . 11

3 An Multicore Framework for Systems Research 13

4 Hardware Support for MicroBlaze Based Multicore System 16
4.1 Processor Identification . . . 16
4.2 Atomic Instructions . . . 17
4.3 Special Purpose General Purpose Register . . . 19
4.4 Interrupt support . . . 20
4.5 Timer support . . . 21
4.6 Timer and Interrupt Controller . . . 23
4.6.1 Design Details . . . 24

5 Extending MicroBlaze Linux Support to SMP 28
5.1 Interrupt and Timer support . . . 28
5.1.1 Interrupts . . . 28
5.1.2 Timers . . . 30
5.2 Exception Handling . . . 30
5.3 Memory Management . . . 31
5.4 Atomic operations and Preemption . . . 32
5.5 Boot Process and SMP Support . . . 34
5.5.1 Secondary Processor Start-up . . . 35
5.5.2 Secondary Processor Bring-up . . . 36

6 **ABACUS** 38

6.1 Overview . . . 38

6.1.1 Architectural Overview . . . 39

6.2 Controller . . . 40

6.3 Profiling Units . . . 41

6.3.1 Code Profiling Unit . . . 41

6.3.2 Instruction Mix Unit . . . 41

6.3.3 Reuse Distance Unit . . . 42

6.3.4 Latency Unit . . . 42

6.4 Integration into the MicroBlaze Platform . . . 43

6.5 Device Drivers . . . 43

7 **Platform Investigations** 44

7.1 Test Platform . . . 44

7.1.1 MicroBlaze Configuration . . . 45

7.1.2 ABACUS Configuration . . . 46

7.2 Stress Tests . . . 46

7.2.1 Boot-up . . . 46

7.2.2 Up-time . . . 47

7.3 Bus Read Latency . . . 48

7.4 Design Scalability . . . 51

7.5 Hardware Resource Usage and Scalability . . . 52

7.5.1 Timer and Interrupt Controller . . . 52

7.5.2 MicroBlaze . . . 53

7.6 Kernel Stats . . . 54

8 **Conclusions and Future Work** 55

8.1 Conclusions . . . 55

8.1.1 Multicore MicroBlaze Platform . . . 55

8.1.2 Hardware Profiling . . . 56

8.2 Future Work 56

Bibliography **57**

Appendices **61**

A Example Boot Log 61

B Kernel Code Stats 64

C Example Kernel Source 66

C.1 spinklock.h 66

C.2 atomic.h 73

C.3 intc_timer.c 77

C.4 smp.c 88

C.5 head.S 94

List of Figures

2.1 A Block Diagram of a Linux Capable MicroBlaze System . . . 7

3.1 A high-level overview of our proposed research framework . . . 14
3.2 Test System Configuration . . . 15

4.1 Conditional Load/Store Pair Path Through Processor Pipeline . . . 18
4.2 MFS/MTS Instruction Format . . . 20
4.3 High-level Timer and Interrupt Controller Diagram . . . 24

5.1 Spinlock arch_spin_trylock Implementation . . . 33
5.2 kernel Initialization Code for Secondary Processors . . . 37

6.1 High-level ABACUS Overview . . . 39
6.2 Controller State Machine . . . 40

7.1 Test System Configuration . . . 45
7.2 Log Output of the Single Bootup Failure . . . 47
7.3 Output of /proc/uptime after fourteen days of uptime . . . 48
7.4 Output of /proc/interrutps after fourteen days of uptime . . . 48
7.5 TIC and UART bus read latency . . . 49
7.6 Memory Bus Read Latency . . . 49
7.7 Dual-core TIC Configuration Resource Usage and Operating Frequency . . . 53
7.8 TIC Scalability Across an Increasing Number of CPUs . . . 53

Glossary

ABACUS	hArdware Based Accelerator for Characterization of User Software. 3, 13, 15, 38–41, 43, 44, 46, 48, 55, 56
BRAM	Block-RAM. 31, 34, 35, 45
CIE	Clear Interrupt Enable. 25, 26
CPU	Central Processing Unit. 13, 16, 17, 24, 29, 30, 35, 36, 46
DMA	Direct Memory Access. 40, 43
DSP	Digital Signal Processing. 13
FF	Flip-Flop. 46, 53, 54
FPGA	Field Programmable Gate Array. iii, 3, 5, 6, 15, 44, 46, 52
FPU	Floating Point Unit. 1, 3, 7, 15, 45, 56
GPSPR	General Purpose Special Purpose Register. 19, 20, 31, 35, 53
IER	Interrupt Enable Register. 25, 26
IP	Intellectual Property. 7
IPI	Inter-Processor Interrupt. 10, 11, 21, 24, 25, 28, 29, 32, 35, 36, 50, 52
ISA	Instruction Set Architecture. 8, 9, 17, 19, 32, 53
LUT	Look-up Table. 46, 53
LWX	Load Word Exclusive. 8, 17, 32, 33, 51
MFS	Move From Special Purpose Register. ix, 19, 20
MMU	Memory Management Unit. 7, 8, 10, 19, 31, 32, 34, 35, 54
MSR	Machine Status Register. 19
MTS	Move To Special Purpose Register. ix, 19, 20
OS	Operating System. iii, 1–3, 8, 10, 13–15, 31, 38, 43, 55
PC	Program Counter. 41, 43
PID	Process ID. 31, 32
PLB	Processor Local Bus. 18, 42–45, 51
RISC	Reduced Instruction Set Computing. 7

SDK	Software Development Kit. 7
SIE	Set Interrupt Enable. 25, 26
SMP	Symmetric Multi-Processor. 1, 3–5, 8, 10, 28–32, 44, 54, 66
SWX	Store Word Exclusive. 8, 17, 18, 32, 33, 51
TIC	Timer and Interrupt Controller. ix, 16, 23–25, 27, 29, 44, 47, 49, 50, 52–54
TLB	Translation Look-aside Buffer. 8, 10, 31
UART	Universal Asynchronous Receiver/Transmitter. ix, 7, 49
UP	Uni-Processor. 32

1 Introduction

Modern computing systems typically comprise multiple computing cores, and are thus also referred to as multicore systems. From workstations, to "smart" phones, to "cloud" computing, multicore solutions provide increased computing throughput through increased parallelism as opposed to operating frequency. Although numerous different computing models exist for multicore systems, one of the most common is the Symmetric Multi-Processor (SMP) model.

Modern multicore systems are complex platforms with many shared resources including: caches, buses, Floating Point Units (FPUs), and memory controllers. Hardware resources are often shared to reduce the system's area and power; however, sharing resources between processors results in the loss of predictability of performance for different workloads. In some cases, application performance can be degraded by as much as 150% in a shared workload environment [1]. In addition to application runtime, other aspects of workload performance can also be negatively impacted including: latency, power consumption and quality of service. With the increasing complexity in modern processor systems and future systems expanding into heterogeneous and asymmetric computing [2], performance on these systems is likely to become even more variable. Without visibility into the interactions taking place at runtime in these systems, intelligent decisions on thread scheduling cannot be made for dynamic workloads or fixed workloads on more complex systems.

1.1 Motivation

Research into thread scheduling in systems with resource contention is a difficult area at present due to the scope of the problem. While visibility into low-level interactions in the system is needed, experimental evaluation also requires that the system supports a full OS. This poses a challenge for

two of the primary avenues for systems research, modern commercial multicore systems and system simulators.

For example, it is possible through hardware counters on existing platforms [3, 4] to measure quantities such as number of cache misses and attribute these events to degraded performance. However, understanding the underlying causes is more complex and needs greater visibility into the systems' internals. Therefore, while commercial systems operate at high frequencies to provide results in a reasonable time frame, their fixed hardware limits them. Not understanding the cause of the underlying behaviours limits the application of this research to generic architectures and the conclusions reached may not be applicable to future systems, limiting its influence.

Conversely, software system simulators, such as simics [5], provide a flexible platform for systems research with easy visibility into a system's internals. However, their simulation runs orders of magnitude slower than actual systems and thus, for performance reasons, software simulators tend to focus on a single layer of the system hierarchy. For software simulators, greater oberservability and accuracy comes at the cost of increasing runtimes. This in turn limits their usefulness for experiments involving scheduling at the OS level.

1.2 Objective

The objective of this thesis is to first develop a configurable multicore system framework that supports a full OS for systems research such as optimizing runtime task scheduling for future multicore systems. To support this type of research, the second objective of this thesis is to create a runtime configurable module that is independent from the architecture, yet can be used to snoop runtime behaviours without impacting the system's operation.

The underlying framework requires a multicore system running a full OS, such as Linux, and the integration of a hardware unit capable of profiling the system at runtime. To ensure that we have the observability and configurability available in software simulators without the overwhelming perfor-

mance penalty, the framework's targeted implementation platform is a FPGA. A FPGA based system is well suited for this problem as it allows us to create fully hardware based systems while maintaining full visibility into the workings of the system. In addition, while the design will operate at lower frequencies than high performance computing architectures, it is not limiting, allowing OSs to boot and user software to execute to completion. The hardware and software design will be structured to facilitate design modifications, such as multiple levels of cache hierarchy and asymmetric core configurations, so that future work can look at systems research in asymmetric and heterogeneous computing models.

1.3 Contributions

In this thesis, we present the initial framework for a scalable and configurable multicore system for systems and architectural research. The major contributions of this thesis can be divided into the following items:

- A configurable and scalable multicore system framework with full OS support
- A scalable, independent, modular, hardware profiling unit that can be configured by the user or the OS at runtime.

The MicroBlaze [6] single-core, soft-processor design from Xilinx has been chosen as the basis for this work. It allows configuration of numerous parameters (e.g. inclusion of a FPU, etc.) and supports a full Linux kernel from PetaLogix [7]. This thesis discusses the modifications made to the processor and the system peripherals that are required to support SMP and scale to at least four processors. The necessary modifications to the Linux kernel to support the changes in hardware as well as the extra support needed for SMP are also discussed. Finally, we describe our hardware profiling unit, hArdware Based Accelerator for Characterization of User Software (ABACUS) and its integration into the system. We then evaluate the system framework by performing a series of tests to measure its stability and certain performance aspects using ABACUS.

1.4 Thesis Organization

The remainder of the thesis is structured as follows. Chapter 2 provides background on multicore systems, the MicroBlaze processor and previous work in multicore systems on FPGAs and system simulators. An overview of the desired framework for our heterogeneous multicore systems research is provided in Chapter 3. Chapter 4 presents the MicroBlaze platform and the hardware changes required to support SMP on Linux. The changes required to support the new hardware platform in the Linux OS are detailed in Chapter 5. Chapter 6 describes the architecture and integration of our data profiler into the new platform and Chapter 7 discusses the experiments run to validate our system. Finally, Chapter 8 concludes the thesis and outlines future work.

2 Background

This chapter focuses on the background, related works and concepts necessary to understand the contributions of this thesis. First, previous work on soft-processor systems and multicore emulators on FPGAs are presented, focusing on the MicroBlaze architecture. Next, we cover the hardware requirements of a multicore system, followed by a brief discussion of the Linux kernel architecture, focusing on its SMP support. Finally, the chapter concludes with a discussion on the previous work in workload analysis.

2.1 Soft-processors and Processor Emulators on FPGAs

FPGAs have grown in capacity such that they can implement complex Systems-on-Chip. This has opened the area for research into the acceleration of simulation for traditional architectures [8], as well as research into soft-processor architectures [9]. Multicore processor architecture research utilizing FPGAs includes: the Research Accelerator for Multiple Processors (RAMP) [10] project for multiple lightweight processors on tens and hundreds of FPGAs [11]; and emulation platforms for the design of commercial multicore processors [12, 13].

More recently, open-source, multithreaded, multicore-processor emulators for FPGA platforms [14, 15] enabled a new approach to research into workload behaviours and interactions on multicore architectures, offering several advantages. Specifically, by leveraging access to the Hardware Description Language (HDL) implementation of the emulator, researchers obtain cycle-accurate visibility into the microarchitecture behaviour, while still being able to run a full OS on the system much more quickly than multicore software simulators.

Researchers designing multicore systems for implementation on FPGAs have, like their commercial counterparts, included dedicated hardware for profiling their systems [16]. On-chip, independent, cycle-accurate software profiling units for soft processors have also been investigated for FPGA-based SoCs [17] [18]. However, both these works are aimed at monitoring accesses to specific address ranges for instruction execution or data accesses by single core CPUs and do not support the more complex capabilities required for multicore processors. While the listed works all share a usage of FPGAs for performing simulation or profiling, the approach taken with ABACUS is different in that we focus on providing a generic framework for designing new profiling units, and providing run-time reconfiguration along with a mechanism for communicating with the Operating System.

Currently, we are aware of four soft-processor systems that support Linux 2.6 or later to varying extents. They are: the OpenSPARC [14], the LEON3 [15], the Nios II [19] and the MicroBlaze [6]. The OpenSPARC based systems supports an Ubuntu based Linux distribution for a single core setup and a dual-FPGA board dual-core setup. Unfortunately, the scalability of the OpenSPARC platform is limited due to its memory hierarchy, which relies on a single MicroBlaze to process all memory requests. In addition, scaling the system, even to multiple cores, is made complicated due to the need for a multi FPGA-board setup. The LEON3 is another SPARC-based platform that supports a dual-core setup, but the memory controller support for different boards is not well supported. The LEON3 is also a rather large processor design – only 2 processors can fit on a Virtex 5 110t FPGA.

Both FPGA companies, (Xilinx and Altera) have their own soft-processors. Recently, the NIOS II processor from Altera has gained support for Linux for a single-core setup, but its Linux support is still in the early stages. PetaLogix however, has supported Linux on the MicroBlaze since the 2.6.30 version of the Linux kernel, and currently supports the 2.6.37 version of the kernel through version 2.1 of PetaLinux [20]. While the MicroBlaze only supports a single processor system, it's small size allows for up to 8 cores to fit on a Virtex 5 110t FPGA and its simpler design should allow for easy modification. While the MicroBlaze is a closed source processor, Xilinx has provided us with the source for this work.

2.2 MicroBlaze

As this work builds on top of the MicroBlaze processor we will briefly cover the architecture of this processor relevant to this work.

The MicroBlaze is a simple in-order Reduced Instruction Set Computing (RISC) based processor that is highly configurable in terms of the components included. This includes functionality such as dedicated multiplier and divider, inclusion of a FPU caches and Memory Management Unit (MMU).

2.2.1 MicroBlaze and Linux

As part of the PetaLogix Software Development Kit (SDK) [7], support is provided for some standard Xilinx Intellectual Property (IP) cores. Only the MMU included version is supported by the 2.6 version of the Linux kernel. Figure 2.1 illustrates an example of a Linux-capable MicroBlaze based system.

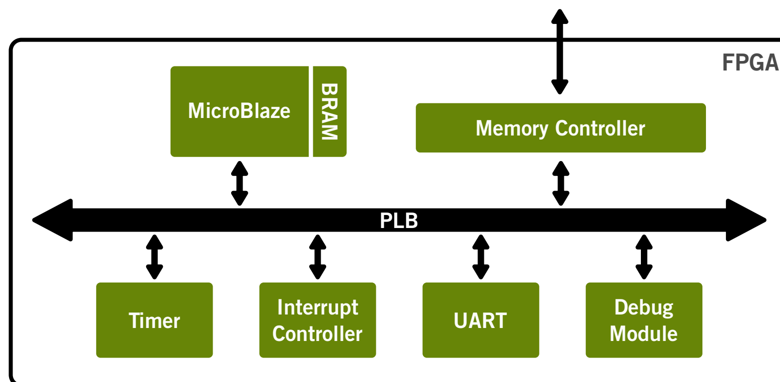

Figure 2.1 A Block Diagram of a Linux Capable MicroBlaze System

Essential components include the MicroBlaze, Interrupt Controller, Timer Unit, UART peripheral and Memory Controller. The Timer Unit is required as the MicroBlaze does not include an internal timer as many architectures do. The UART is used for connecting to the Linux kernel's serial console and the interrupt controller provides an interface to allow the MicroBlaze to handle multiple interrupt sources.

2.2.2 Memory Management

The MicroBlaze MMU is based on the PowerPC 405 [6] MMU and is fully software managed. On Translation Look-aside Buffer (TLB) misses, software routines will add/remove entries in the TLB.

2.2.3 Conditional Load Store Instructions

As part of the MicroBlaze Instruction Set Architecture (ISA) two instructions: Load Word Exclusive (LWX) and Store Word Exclusive (SWX) [6], are provided for performing atomic operations in memory.

The conditional instructions work by setting a reservation bit on the execution of a conditional store instruction (LWX), and by checking this bit when a conditional store (SWX) is to be executed. If the bit has been cleared (by an exception,interrupt or another SWX instruction), then the store will not go through, otherwise the store will complete and the operation was guaranteed to be atomic.

In a single processor environment, the only case where the processor will lose the reservation before the store goes through is in the case where an interrupt or exception occurs. This behaviour will fail in a multicore system as reservations for a particular address must be unique among the processors in the system. As such the reservation mechanism must be replaced to extend the MicroBlaze to support SMP (see Chapter 4).

2.3 Multicore Computer Architecture

At its simplest, a shared-memory multicore system consists of multiple, identical processors connected to a single shared memory. In order for this system to be practical for running an OS, additional features, such as providing memory coherency between the processors and an asynchronous notification system, are required.

2.3.1 Memory Coherency

A multicore system with strictly enforced memory consistency between processors (i.e. at no point would one processors view of memory be different from any others) would be highly detrimental to performance. Such a system would not allow any form of memory caching including the caching that is inherent to the processor pipeline. As such, most multicore systems have a minimum time widow in which a processor's view of memory is inconsistent with the other processors in the system. This can vary from just a few cycles in some systems to others where the time frame is indeterminate in size. In a simple system, with write through caches [21] (i.e. all stores are passed through to memory, e.g. SPARC T1 [22] L1 caches), then the synchronization time frame can be as low as a few cycles. This is the simplest system as synchronization of memory happens automatically and the time frame is usually quite small. In a system with writeback caches (eg. Intel/AMD processors [3, 4] L1 caches), memory is not coherent until data is evicted from the local writeback caches. In these systems, special cases must be implemented (usually through software-level support) when atomic operations are issued in order to make sure coherency is forced at these points [3, 4].

Atomic Operations

In addition to keeping memory consistent, as part of the a processor's ISA, there are often a set of instructions to perform atomic operations on memory. These operations will force coherency between the processors in the system as the memory view must be consistent for an operation to be atomic. These instructions allow for the implementation of synchronization primitives such as mutexes and semaphores [23]. Different processors implement this functionality in different ways. In some architectures, atomic compare and exchange instructions are implemented that will perform a store to memory if the existing value at that memory location matches a given value [3, 24, 25]. Most processors also have a pair of load store instructions used to implement conditional atomic instructions where the success of the operation is not guaranteed, but if it does go through then the operation was guaranteed to be atomic [3, 24, 25, 6].

2.3.2 Asynchronous Communication

Current SMP systems also require a form of asynchronous communication between them to be capable of supporting a modern OS such as Linux [26]. This mechanism takes the form of Inter-Processor Interrupts (IPIs) and are the means by which a processor can send an asynchronous notification to any other another processor in the system. With IPIs operations can easily be coordinated between processors such as updating their local TLB entries. Minimally, only a single IPI is needed and the ability to broadcast to all other processors in the system. Software could then be used to control message passing between processors. In practice, however, having a few IPIs and the ability to send an interrupt to any subset of processor in the system will reduce the complexity of the software support.

Virtual Memory

Modern, fully-featured OSs require support for virtual memory; this implies support for a MMU in hardware. The MMU is a per processor hardware block that handles virtual to physical memory mappings. Thus, in a multiprocessor system there must be hardware or software synchronization between these units to ensure that all mappings for processes running in the system are kept up to date.

2.4 The Linux Kernel

This section focuses on details of the Linux kernel [26] that relate to architecture support and SMP support.

2.4.1 Architecture Support

Inside the Linux Kernel, low level architecture support is encapsulated in the `/arch` directory. All architecture specific code exists in this area, except for a few portions of x86 dependent code that are likely legacy code from Linux's origins on x86 based processors. It is here that support for functionality

such as atomic instructions, timekeeping and the management of the memory hierarchy is handled as well as any other functionality that is dependent on the processor architecture.

2.4.2 SMP Support

There are a few areas, within the architecture dependent code, that require custom support for a multicore system. These include, but are not limited to:

- Spinlocks (mutexes), and other atomic operations
- IPIs
- Low level memory management
- Interrupts
- Timekeeping and timers
- Boot-up and bring-up of secondary processors.

All of the listed components have high-level interfaces in the kernel and it is through the architecture level that support is built for this functionality. These issues and how they are addressed will be discussed in detail in Chapter 5.

2.5 Standard Methods for Workload Performance Analysis

Traditionally, hardware counters, software profiling and software simulation have been used to characterize the behaviour of workloads on multicore systems. On existing systems, hardware performance counters [3, 4] can be used, both independently or through software profiling to collect data on workloads at runtime. Hardware counters can be utilized without having a noticeable impact on application performance and have been actively employed in online scheduling algorithms [1, 27, 28]. While they provide a wide range of events that can be captured, their fixed nature and focus on microarchitecture dependent events limits their adoption as a standard tool across platforms. In addition, typically only a few registers are provided for storing hardware counter events. This leads to the use

of sampling [29], decreasing the accuracy of the results and/or incurring a high runtime overhead. Finally, hardware counters are designed to focus on single events, limiting the scope of information they can provide. Specifically, they can detect symptoms of problems (i.e. what happened), but they cannot aid the user in determining the *cause* (i.e why something happened).

Software profilers, such as: oprofile [30], gprof [31], Intel VTune [32], and Sun Studio Performance Analyzer [33], can also take advantage of existing hardware counters, along with code instrumentation, to analyze workload behaviour. Other work has considered using hardware counters to associate events with data structures [29]. However, as software profiling tends to have a significant impact on software performance, it is not typically used for on-line runtime profiling.

Software simulators cover a wide range of tools, from system simulators such as Simics [5], to more microarchitecturally focused simulators such as SimpleScalar [34], as well as including binary instrumentation tools such as Pin [35]. Due to their software implementation, they are not constrained in the amount of data they can collect and can provide a wealth of information on various components in a system. However, increasingly detailed simulation of the interactions in the system comes at the cost of performance, making some simulators extremely slow. In contrast, in the context of providing a research platform for investigating hardware profiling, ABACUS provides a more scalable approach and can be used to provide the same information as a system simulator, but with a 20x speedup [36].

3 An Multicore Framework for Systems Research

Multicore systems are growing in complexity, and for reasons motivated by performance, power and energy consumption [2], future systems will be tending towards asymmetric and heterogeneous platforms. Today, smart phones are already a form of heterogeneous computing system with multiple Central Processing Units (CPUs) and Digital Signal Processing (DSP) cores [37]. However, in these systems, workloads are scheduled statically based on their known behaviours. With a greater variety of applications and constantly changing platforms, this approach will not continue to be viable due to number of combinations that would need to be analyzed. Without statically defined workloads, due to the variability in the system, certain quality of service guarantees, like user interface performance might not be met. This highlights the need for a research platform that can investigate the interactions taking place in these systems so that, the information can be used to fully utilize these platforms.

Figure 3.1 provides an example of what this type of system could look like with multiple processor cores, asymmetric cores and potentially custom hardware accelerators. The system could be configured with a variety of different memory hierarchy configurations including multiple L2 caches that could be shared between different subsets of processors and an optional L3 cache. Omitted from the diagram for clarity are the connections between ABACUS and the system as well as all interrupt signals.

The final goal would be to create a system, such as the one illustrated by Figure 3.1, however, even existing systems pose many challenges to the OS for scheduling due to the complex nature of the interactions that take place for shared resources at runtime. This situation will only worsen as systems scale in size and move towards asymmetric and heterogeneous implementations. Without

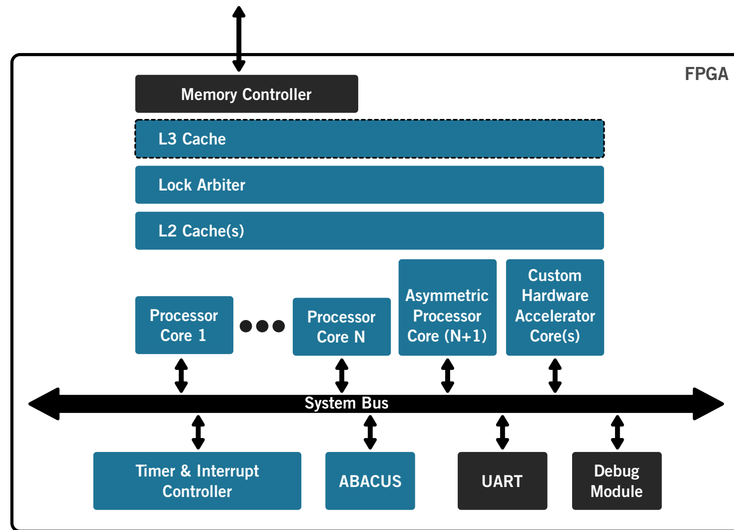

Figure 3.1 A high-level overview of our proposed research framework

providing visibility into the system level interactions to the OS, efficiently utilizing these systems under generic workloads will become an impossible task. At present, without a platform that can model these systems and provide visibility into the interactions taking place inside the system, there are significant challenges in even identifying the metrics necessary to characterize the interactions taking place within the system.

In this work, our objective is to create an extendable framework for performing systems and architectural research that can be readily extended, in the future, to support asymmetric and heterogeneous configurations. The platform will allow for investigations into performance metrics that can be used by the OS or user-level software to make decisions that will optimize for a desired performance target. The two main components of this work are the multicore framework with Linux support and the hardware profiler for collecting runtime data on the interactions taking place within the system.

To be useful for systems research the platform should support a modern OS. In this work, we target the Linux operating system due to its support for the MicroBlaze processor and due to its popularity for systems level research and its open source nature. By selecting Linux as the operating system,

we benefit from the familiarity many systems researchers already have with the OS. The MicroBlaze, as a small and highly configurable processor, is an excellent choice for this work as its configurability will allow us to easily create asymmetric systems (e.g. varying cache sizes, inclusion of a FPU, etc). Furthermore, its small size allows us to include several processors on a single FPGA. One example of an asymmetric system that could be created using the MicroBlaze, would be one in which some of the processors in the system have a dedicated FPU while others do not. In this configuration, the processors without the FPUs would be forced to execute their floating point instructions in software or be rescheduled on another processor.

Without support for monitoring the interactions taking place within the system, this new platform would be of little interest for systems research. Our second contribution is the design and integration of a hardware profiling unit that we can use to collect different self determined metrics that can be used to characterize the behaviour of workloads at runtime and then be used to influence the behaviour/operation of the system. The full system, illustrated in Figure 3.2, consists of more than one MicroBlaze processor, our hardware profiling infrastructure, (ABACUS), and the additional hardware infrastructure necessary for multicore support. The remainder of this thesis will describe the implementation of this platform.

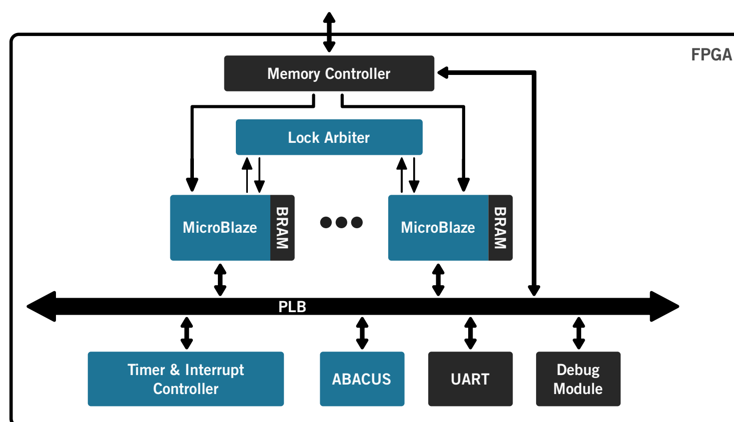

Figure 3.2 Test System Configuration

4 Hardware Support for MicroBlaze Based Multi-core System

In a single-processor system, there are several attributes that simplify the design of the system. There is typically only one master on the system bus, only one core accessing memory, and only one source to handle interrupts and communication with peripherals in the system. Simply stated, very little sharing of any resource takes place in a single-core system compared to a multicore system. It is due to the sharing of different resources, and the complexity that incurs, that additional hardware support is needed in a multicore system. Some hardware support is required such as providing a mechanism for identifying the physical processor index in the system and support for atomic operations. Other components, such as support for distributing interrupts across the cores, while not needed for correct operation, provide beneficial properties that are worth the extra complexity they introduce into the system.

The remainder of this chapter will cover the modifications necessary to convert the current MicroBlaze based Linux platform into a multicore capable system. This includes providing a means to access the processor's physical index in software, support for atomic instructions, additional support for handling exceptions/interrupts within the processor and concludes with a discussion of the changes to interrupt and timer support.

4.1 Processor Identification

In the physical system, each CPU has fixed connections to the various peripherals in the system. For some peripherals, like the Timer and Interrupt Controller (TIC), the CPU number is used to select

the correct offset for the per processor registers. In order to be able to support an arbitrary cumber of CPUs in software (for cases such as CPU hotplugging), each processor must have a way of determining its hardware CPU ID. As part of the MicroBlaze architecture, there exists a user configurable, read-only, Processor Version Register that can be configured on a per processor basis. We use this register to contain the CPU ID, which matches the processors physical connection number to the system bus.

4.2 Atomic Instructions

As part of the existing MicroBlaze ISA [6], there exists a conditional load/store pair of instructions (LWX/SWX) that can be paired together to create atomic operations. The pair of instructions work as follows. Upon execution of the LWX instruction, a reservation bit, internal to the processor, is set. When the accompanying SWX instruction reaches the execute stage, the instruction will be aborted if the reservation bit has been cleared. The reservation bit is cleared under two conditions. The first, is through the execution of any SWX instruction, (note the address of the SWX instruction need not match the proceeding LWX instruction). The second case is if any interrupt or exception has occurred within the processor after setting the reservation bit [6].

In a multicore system, this support is not sufficient to guarantee that an operation is atomic, as an internal reservation bit will not guarantee that another processor is not modifying the same memory location at the same time. What is required is that a reservation is unique for a given address in the system, or even more simply, unique in the system. In our system, we have selected the approach based on a single reservation bit. A per-address based reservation system can be implemented at a later stage along with the integration of the cache hierarchy into the system. In order to handle the arbitration of the reservation between processors an external module (the lock arbiter), as shown in Figure 3.2, has been created for this purpose. The following figure, Figure 4.1, illustrates a successful conditional load store sequence and highlights the involvement of the reservation bit. When LWX instructions enter the execute stage, a request is sent to the lock arbiter. If the lock is free, and no other requests are present, the lock is granted to the processor that requested it. In the case of a

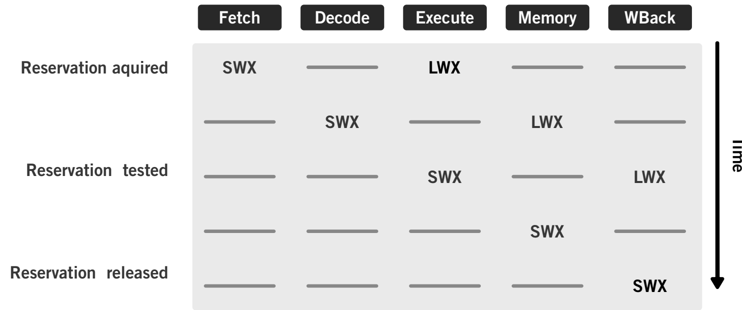

Figure 4.1 Conditional Load/Store Pair Path Through Processor Pipeline

tie, a rotating priority among the processors is chosen. The lock is reserved for this processor and no other processors will be granted a lock until the owner processor releases the lock either by the completion of an SWX instruction, or through an internal exception/interrupt.

As the decision to proceed with the store must be made before the memory stage (the one in which the store actually occurs) to ensure consistency between the processors, the reservation must be held until the memory stage completes. Prior to this point, we do not know the order in which the processors will complete their memory operations as the order is dependent on the arbitration performed by the Processor Local Bus (PLB). As such, a processor, releasing its lock, could be delayed in the memory stage long enough for another processor to have a conditional store go through memory. Without holding the reservation through the memory stage, operations are not guaranteed to be atomic.

Implementing only a single reservation bit is not the most efficient implementation. Per-address reservation bits would allow each processor to have a conditional load/store operation in progress as long as the addresses do not collide. This functionality, however, is better addressed in a memory arbiter and is left for future work.

4.3 Special Purpose General Purpose Register

The MicroBlaze architecture contains 32 registers as part of its main register file [6]. In addition to the general purpose registers, there exist a few special purpose registers such as the Machine Status Register (MSR) as well as some registers for manipulating the MMU. In the existing Linux implementation, there are certain per CPU variables, such as the stack pointer and current task pointer, which are always resident in the register file and need to be saved on any context switch. As the MicroBlaze architecture does not have any free registers with which to save these variables, even temporarily, the Linux kernel implementation works around this by storing these variables into temporary memory locations, using an immediate offset address mode. For the multicore case this implementation is not workable as there is no way to change the immediate address on a per processor basis. The solution then, similar to the approach taken by the PowerPC405 [24], is to add a few (currently four) General Purpose Special Purpose Register (GPSPR) registers, and accompanying support to the ISA of the processor to allow for temporarily storing these register values. Currently, two GPSPR registers are needed for the exception handling support in our multicore framework. We have also included two additional registers to facilitate possible extensions that may be required in the future. However, these two additional registers can also be trivially removed if they are deemed unnecessary.

As part of the MicroBlaze ISA, there exists a Move From Special Purpose Register (MFS)/Move To Special Purpose Register (MTS) pair of instructions for copying a Special Purpose Register into a General Purpose Register and vice versa. We have added support to this instruction pair to address a new set of registers called GPSPRs that can be used by software for any purpose as they have no dedicated use. They are, however, considered privileged registers and can only be accessed in privileged mode if the MMU is active.

Figure 4.2 shows how the new registers are integrated into the MicroBlaze ISA. The MFS/MTS instructions use the immediate mode instruction format. The two most significant bits of the immediate operand are used to select between the MFS/MTS pair and instructions to manipulate the MSR. When

the MFS/MTS instructions are selected the 20th bit is used to select the GPSPRs registers and then bits 30 and 31 are used to select the desired index.

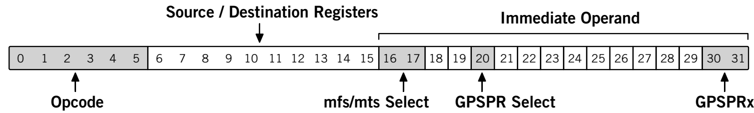

Figure 4.2 MFS/MTS Instruction Format

As these new registers are not general purpose registers, they will not be automatically generated by the compiler and therefore, only used if hand coded in assembly. The compiler platform for the MicroBlaze is GCC and binutils [31] for the assembler. Xilinx provides the source for modifications to binutils and, through the assembler, we added support for these instructions. In assembly, these registers are accessed through the following markup:

```
mrs rD GPSPRx
mts GPSPRx rS
```

with the letter 'x' replaced by the index of the desired GPSPR, rD for destination register and rS for source register. Unfortunately, Petalinux [7] includes a modified version of binutils and thus, these changes to the assembler are not compatible with the kernel build tools. For this case, we have hand coded the machine code directly into the assembly where needed.

4.4 Interrupt support

In a single-processor system, interrupt support is relatively simple. While there may be many sources of interrupts, all interrupts will be served by the same processor. In a multicore system, there are more factors to consider. First, additional support is required to allow processors to send interrupts to each other to provide for asynchronous communication. Secondly, for performance reasons, it can be beneficial to distribute interrupts across available processors in order to reduce interrupt service

latency. In this case, support should be provided to enable interrupts on a per processor basis as interrupts should not be distributed until the processor is online and there may be some situations where we would like to restrict servicing of an interrupt to a single processor.

The existing MicroBlaze system uses a Xilinx provided interrupt controller (LogiCORE IP XPS Interrupt Controller (v2.01a) [38]) to provide support for multiple edge/level sensitive interrupts as the MicroBlaze only accepts a single interrupt request input. While this setup works well for a single-core system, the lack of support for multiple interrupt outputs decreases the usefulness of this interrupt controller for a multicore system. In addition, no mechanism is present to allow for the support of Inter-Processor Interrupts (IPIs).

As with the existing interrupt controller implementation, we are restricting the additional functionality to handle interrupts to the interrupt controller to reduce the complexity of the multicore supported MicroBlaze processor. For our base platform, we satisfy the following list of requirements:

- Support for IPIs
- Support external edge/level interrupts
- Per processor, per interrupt enables
- Interrupt load balancing

4.5 Timer support

In the Linux kernel, support at the hardware level is required for its timekeeping. However, what this support entails is highly variable, and varies greatly among platforms and processor models. Accuracy, counter width and functionality all vary across different platforms, and thus, kernel support for timekeeping is very flexible. With the addition of support for dynamic ticks in the 2.6.21 kernel [39], a standard interface was supplied for timekeeping support in the Linux kernel. Through dynamic ticks, the need for a period timer interval was removed, thus allowing processor cores to stay in idle, and thus low power modes, for longer periods of time. Support for this layer can be provided by a

single monotonically increasing counter, (for time keeping purposes), and a decrementer capable of triggering interrupts (for scheduling purposes). As the free-running counter is required to not roll over, it would ideally be a 64-bit counter, as otherwise additional software support is required to extend the counter to 64-bits in software. A single decrementer with interrupt support can be used for scheduling purposes in a multicore system if events are broadcast to all processors. However, this would prevent processors in idle from going to low power states for as long as they could otherwise and an implementation with per processor decrementer support has lower overhead.

In the PetaLinux system, timer support comes from the Xilinx LogiCORE IP XPS Timer/Counter [40]. Through this peripheral, and its two 32-bit timers, support is provided for the dynamic tick infrastructure in the Linux kernel. One timer in the system is configured to be free running and is used by the scheduler for timekeeping. The second timer is configured as a decrementer and works in periodic (during boot-up) or one-shot mode (once dynamic ticks are enabled towards the end of the boot-up process). When the decrementer underflows, an interrupt is triggered; in periodic mode, the timer value is reloaded and will begin counting again when the interrupt is cleared.

While this setup works well for a single-processor system, it does not scale well to multiple cores. First, the timer core is limited to two timers and secondly, those two timers share a single interrupt. To support a configuration with per processor decrementers and a single shared free running counter, the timer unit would either have to be replicated per core (only using a single timer) to provide a decrementer interrupt per processor or would have to implement a sharing mechanism between pairs of processors, which would reduce the efficiency of the system. Regardless of the implementation choice, each timer peripheral would increase hardware resource consumption as each would have the overhead of as slave connection to the bus and would consume an interrupt connection to the interrupt controller even though each interrupt would be enabled on only a single processor. Finally, as the free-running counter is not allowed to overflow in the Linux kernel, extra software must be implemented to use the 32-bit hardware counter to increment a 64-bit software counter.

As the existing timer support will not scale well for a multicore system, this provides the opportunity to create hardware support specifically for the Linux kernel. As such, the timer support must meet the following previously discussed criteria:

- 64 bit free-running counter
- Per processor 32 bit one-shot/periodic decrementer with per processor interrupt support

4.6 Timer and Interrupt Controller

To address the functionality required for both interrupts and timers in the system, we integrated the two components into a single peripheral in the system to produce a Timer and Interrupt Controller (TIC), which is roughly based on the implementations of the two Xilinx cores. This approach reduces system complexity and allows for the easy integration of per processor timers into the interrupt infrastructure. In the immediate future, we are only considering systems of 2 to 8 cores, as larger numbers of cores will likely initiate the consideration of a more distributed architecture due to issues such as bus contention. The target feature list of the TIC is as follows:

- Support for 2-8 CPUs
- Configurable number of IPIs
- Global 64-bit free running counter
- Per processor 32-bit one-shot/periodic decrementer with per processor interrupt support
- Per processor enables for interrupts
- Interrupt load balancing
- Lockless operation of interrupts

All but the last item have been previously covered in the Interrupt and Timer Support sections (4.4, 4.5). The last item, *lockless operation*, refers to software access of the peripheral. As each processor will be accessing the peripheral to read their interrupts and set their decrementers, it is desirable to have the the device operate in such a way that software locks are not needed to ensure correct

operation.

4.6.1 Design Details

Figure 4.3 provides a high-level overview of the TIC. All internal registers are 32 bits in width. The design supports one or more IPIs, and *per-CPU decrementers*. The TIC interface is separated into

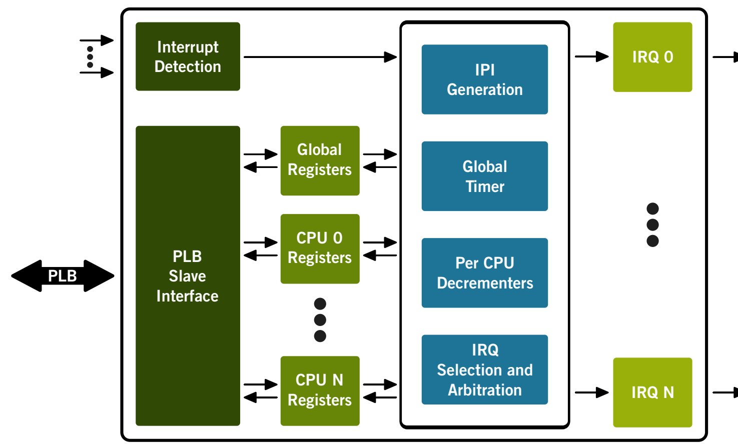

Figure 4.3 High-level Timer and Interrupt Controller Diagram

two components, a set of global registers and a set of of per processor registers.

User Visible Global Registers

The following list of registers are contained within the *Global Registers* block in Figure 4.3:

- the 64-bit free-running counter
- the IPI register (up to 32-bits)
- the Set Interrupt Enable register (up to 32-bits)
- the Clear Interrupt Enable register (up to 32-bits)
- the Master Enable register (up to 32-bits)

The IPI register takes the IPI number in the upper bits, a mask of processors to send the interrupt to in the lower bits and can be written to once per clock cycle. If a specific IPI is already pending for a

given processor and that IPI is written to again, only a single interrupt will be issued. If multiple IPIs of the same type need to be issued to a processor and they cannot be lost, then software must ensure that a previous IPI has been acknowledged before issuing another. This approach is handled in the generic IPI framework in the Linux kernel and is the approach taken in other interrupt controllers such as the MPIC [41] used in older PowerPC platforms.

For enabling/disabling interrupts, a similar approach to the Xilinx interrupt core(LogiCORE IP XPS Timer/Counter (v1.02a) [38]) is taken. There is one global Interrupt Enable Register (IER), however this register is not directly readable/writable and is instead only operated on by the Set Interrupt Enable (SIE) and Clear Interrupt Enable (CIE). These two registers allow for setting/clearing interrupt enables to be performed without having to perform a read-modify-write operation. Each bit in the IER represents a single interrupt with the highest priority interrupt being the IPIs, followed by the timer interrupt, followed by all external interrupts. To set or clear a bit in the master interrupt enable register, the appropriate bit must be set to one in the SIE/CIE register. Finally, the master enable register is simply a single bit that enables interrupt output generation.

User Visible Per-Processor Registers

In addition to the global registers there is a set of per-processor registers contained within the *CPU 1 to N Registers* blocks in Figure 4.3 where N is the number of processors in the system. These registers work in conjunction with the global registers to control the operation of the TIC:

- the 32-bit decrementer configuration register
- the 32-bit decrementer load register
- the Per Processor Set Interrupt Enable (SIE) register (up to 32-bits)
- the Per Processor Clear Interrupt Enable (CIE) register (up to 32-bits)
- the Per Processor Master Enable register (1-bit)
- the Interrupt Vector Register (32-bits)

Each processor has a 32-bit decrementer that can be configured for periodic or one-shot modes and can be set to generate an interrupt on underflow. The starting value for the decrementer can be set by writing the value to the load register. In the case of the one-shot mode, it is of critical importance that no interrupts are lost. If an interrupt is lost, then a processor will never receive its wake-up from idle and the processor will remain in idle state forever. As such, in one-shot mode, if a previous timer interrupt is still being serviced by the processor, the new interrupt is queued until which time the previous local timer interrupt has been served. Just like the global registers, each processor has an IER and a pair of SIE/CIE registers to manipulate it. For an interrupt to be enabled on a processor, the interrupt must be enabled both in the global IER as well as the local processor IER. Through this mechanism interrupts can be enabled on a per processor basis. Finally, each processor has an Interrupt Vector Register, which, contains the current interrupt number when the processor is servicing an interrupt; otherwise, the Interrupt Vector Register contains all ones.

Interrupt Selection

On each cycle only a single interrupt is assigned to any given processor. Selection logic determines the highest priority interrupt request that meets the following criteria:

- the global Master Interrupt Enable bit for the multicore system must be set and
- the interrupt must be enabled in the global IER.

Then for the interrupt to be serviceable on a given processor:

- the specific processor's Interrupt Enable bit must be set,
- the interrupt must be enabled in that processor's IER and
- the processor must not already be busy servicing an interrupt.

If this criteria has been met, and two or more processors are available to service this interrupt, a round-robin arbitration scheme is used to select the processor.

Free-running Counter

The free-running counter is 64-bits in width and counts up from zero after reset. As the system bus and the MicroBlaze processor are 32-bit devices, two reads are required to read the 64-bit value. To ensure that the upper 32 bits do not change while reading the counter the value is snapshotted on a dummy write operation to the base address of the TIC. In addition, within the Linux kernel, software access to the counter is protected by locks to ensure that the read operation is atomic; otherwise, one processor could begin a read and be interrupted after reading only the first 32-bits. Therefore, a non-atomic read of the 64-bit value could cause time to be perceived to flow backwards by the system with unfortunate consequences.

5 Extending MicroBlaze Linux Support to SMP

This chapter focuses on the changes made to the 2.6.37 version of the Linux kernel to add SMP support to the MicroBlaze platform. In adding SMP support to a platform, there are some aspects that are fairly standard such as the support for IPIs, and atomic primitives. Other details, such as the infrastructure needed to support secondary processors, are not well documented. As such, we examined the implementations of other architectures to determine the necessary functionality required. Adding SMP support to the MicroBlaze platform required an iterative process to discover all the details that are needed to support an SMP architecture in the Linux kernel. As this necessitated many changes across the architecture support for the MicroBlaze, this chapter focuses on the key components including: interrupt and timer support, exception handling, memory management support, atomic primitives and the boot-up process.

5.1 Interrupt and Timer support

With the integration of interrupt and timer support into a single hardware block, software support for these two components has also been merged together.

5.1.1 Interrupts

Modifying the software layer to support the new interrupt controller resulted in several cleanups to the existing code base. As part of the generic interrupt infrastructure within the kernel, support is provided for different types of interrupts including level/edge and per cpu interrupts. The different handlers have different behaviours with respect to how the disable/enable and acknowledge of interrupts are handled/supported in order to avoid spurious detection/generation of interrupts during

servicing. However, the interrupt controller for the system, including both the original implementation and the new TIC implementation, hide these details from the processor by supplying only a single level sensitive interrupt to the MicroBlaze. Therefore, all that is required is to call the handler for the active interrupt and then acknowledge the interrupt in the interrupt controller. Implementing interrupt support this way removes the overhead of some locks in the interrupt handling path, some unnecessary enabling/disabling of interrupts, and cleans up the code base.

In the present software implementation, per-CPU interrupt reservations have not been implemented. When an interrupt is enabled/disabled it is done so on all currently online processors. Per-CPU enabling and disabling of interrupts will be added at future point when such support is required.

Inter-Processor Interrupts and Timer interrupts

During the SMP dependent portion of the boot-up phase, the IPI handlers are registered with the TIC and IPIs enabled on the boot CPU. On this platform, there are three types of IPIs: *reschedule*, *call function single*, and *call function many*. The *reschedule* serves as a "no-op," as a scheduling call is part of the return path of the interrupt. *Call function single* is used to call a function on a single CPU and is simply a wrapper for the generic IPI *Call function single*. *Call function many* is used to call the function on two or more CPUs and similarly is simply a wrapper for the kernel's generic implementation. In a platform that supports sending IPIs to any subset of processors, such as our own platform, the overhead of sending the IPIs can be reduced by using *Call function many* instead of multiple sequential calls to *Call function single*.

The timer interrupt handler is also initialized by the boot CPU when the time infrastructure is initialized. This interrupt is also marked as a per CPU interrupt and the handler uses the CPU id to access the correct timer within the TIC.

5.1.2 Timers

Two significant changes have been made to the timer base. First, the free running counter now has 64-bits instead of 32-bits. Secondly, additional support is required to convert the single decrementer implementation into a per-CPU setup.

The free-running counter has been increased to 64-bits in order to avoid extending the value in software. As mentioned in the previous chapter, this counter is required to be a monotonically increasing counter that never overflows. Therefore, if only a 32-bit counter is available, the counter must be extended to 64-bits in software. The original kernel included partial support for extending the 32-bit counter to 64-bits in software, however, it only worked for certain system clock frequencies, which was discovered during testing. Converting this counter to 64-bits removed this issue. During normal system operation, the free-running counter can be read by multiple processors. To ensure that the two 32-bit reads of the counter are performed atomically, reading the counter is protected by a mutex and is accessed with local interrupts disabled.

Expanding the decrementers to a per CPU setup has been a relatively straightforward process. The counter structure has been converted into a perCPU variable and on initialization of the secondary processors, the configuration of the boot CPU's counter is copied and then the per CPU interrupt is enabled. The global interrupt handler for the decrementers uses the CPU id to index the correct data structure for each CPU.

5.2 Exception Handling

Exception handling support, as well as context switching and traps for the MicroBlaze processor, are implemented in assembly and contained within `entry.S` and `hw_exception_handler.S`. As discussed in the previous chapter, the MicroBlaze design had no scratch registers as part of its design and therefore, while servicing interrupts or exceptions, some registers had to be saved to fixed locations in memory in order to free up registers. In an SMP setup, this is not a workable solution

and thus, the GPSPRs have been added to the system. In every instance where the fixed memory locations had been used, the register values are instead saved/restored from the GPSPRs.

For some hardware exceptions, up to 7 temporary registers are required and thus a single scratch space was implemented in memory for this storage. To add SMP support, a per-cpu offset could be added within this address space, however, a simpler modification is possible. As exception handling is performed with the MMU disabled, it is possible to use the Block-RAM (BRAM) that is local to each processor as temporary storage for these registers.

5.3 Memory Management

The MicroBlaze MMU is based off of the PowerPC 405 [24] design and is fully software managed. This requires software to be responsible for adding and removing entries in the TLB. The MMU itself has an 8-bit field for the Process ID (PID), which, limits its support to a maximum of 256 contexts in hardware. Therefore, multiple processes in software may be mapped to the same hardware PID. If any two software processes share a hardware PID, then any TLB entries that are present in the MMU for one of the processes those entries must be invalidated in the MMU when the other is selected to run. An SMP system complicates this a bit more as MMU contexts can now be shared between processors. The new memory management support for the MicroBlaze is heavily based off of the support provided for the PowerPC 4xx/8xx series processors [24], which provide SMP support for fully software managed MMUs.

In an SMP system, we have to ensure that whenever multiple processors share a context, it only gets released when all processors have marked it as inactive. We also have to ensure that when a new software context is created that reuses a hardware context, the old entries are flushed from all processors that have them in their MMUs. Unfortunately, the MicroBlaze does not have a mechanism to flush entries based on the PID; so in this case, the entire MMU must be invalidated. The MicroBlaze does, however, support flushing single pages; so when calls are made in the OS to flush a page, the

whole MMU is not invalidated. In order to synchronize MMU state changes between the processors, an IPI is used to notify all other processors who share a context whenever a PID or a single page must be invalidated.

5.4 Atomic operations and Preemption

Inside the Linux kernel there are many operations that must be performed atomically. In a single-processor system with kernel preemption disabled, the implementation of atomic operations is quite simple. Disabling interrupts over the set of operations that are required to be performed atomically is sufficient. However, this is not sufficient when kernel preemption is enabled as in some locations interrupts cannot be disabled. Similarly, this is not sufficient in a multicore system as multiple cores could be within the same critical section, or even in a single-core system with kernel preemption enabled as kernel preemption forbids the disabling of interrupts under certain circumstances. Therefore, in order to support atomic operations in an SMP system, the infrastructure for atomic operations must be built around the conditional load/store instructions that are a part of the processors ISA.

Inside the Linux kernel, one of the basic primitives for performing mutual exclusion is the spinlock, a type of lock that employs busy waiting while attempting to acquire the lock. In regards to adding SMP support for atomic primitives, it is equivalent to supporting kernel-preemption on a Uni-Processor (UP) system. In the existing Linux kernel, preemption is not supported for the MicroBlaze and no custom implementation for spinlocks is provided. As such, the generic spinlock implementation is used, which creates atomic operations by disabling interrupts on the processor around the operations that need to be atomic. In addition, the original MicroBlaze Linux support did not use the LWX/SWX pair of instructions anywhere other than in the user space support for mutexes (`futex.h`). Unfortunately, their support of a compare/exchange operation was implemented incorrectly, as it would only perform the memory swap if the existing memory value did not match.

Acquiring a spinlock can be performed by two different functions. In the first (trylock), only a single

attempt is made to obtain the lock and the function returns whether it was successful or not. The other function (spinlock) will always obtain the lock and spins until it can obtain the lock. Figure 5.1 shows the implementation of the trylock version of the spinlock. The blocking lock version is built from the trylock and simply spins on the lock value until the value of the lock is zero, in which case it calls the trylock version again, and repeats until it gains the lock. This implementation mirrors the PowerPC implementation of spinlocks but includes differences in behaviour due to how the atomic load store pairs are implemented in our system. If the blocking lock version is not built from the trylock, it would be constantly performing a LWX/SWX pair of instructions and could potentially starve a processor trying to release the lock from obtaining the reservation. This is also why stores must always be paired with loads (address matching not required) to ensure that the reservation is released in a timely manner.

```
static inline int arch_spin_trylock(arch_spinlock_t *lock)
{
   unsigned long prev, tmp, dummy;
   __asm__ __volatile__ (
      "1: lwx   %0, %2, r0;\n" /* load conditional address (&lock->lock) to (prev) */
      "   bnei  %0, 2f;\n"     /* not zero? clear reservation bit and abort */
      "   addi  %1, r0, 1;\n"  /* set lock to one */
      "   swx   %1, %2, r0;\n" /* attempt store */
      "   addic %1, r0, 0;\n"  /* checking msr carry flag */
      "   bnei  %1, 1b;\n"     /* lock available, but store failed. Try again */
      "   bri   3f;\n"         /* done */
      "2: swx   %0, %3, r0;\n" /* clear reservation */
      "3:"
      : "=&r" (prev), "=&r" (tmp)           /* Outputs */
      : "r"   (&lock->lock), "r" (&dummy)   /* Inputs */
      : "cc", "memory"
   );

   /* prev should be zero if lock is available and tmp should be zero if store
      succeeded */
   return (prev == 0 && tmp == 0);
}
```

Figure 5.1 Spinlock arch_spin_trylock Implementation

In addition to spinlocks, the conditional load/store pair of instructions are used to build basic atomic primitives such as add/subtract, test/set and compare/exchange functionality (`atomic.h`, `system.h`). These primitives are used directly in some places in the kernel and also as building

blocks for other mutex and semaphore implementations within the Linux kernel. The last significant usage of the conditional load/store instructions is within `bitops.h`, which contains wrappers to create the atomic bit operations that are used throughout the kernel when bit-masks need to be modified atomically.

In the original MicroBlaze system, spinlocks and atomic operations used the generic implementations with no architecture-specific code. In the generic implementations, all operations are made atomic by saving and restoring the interrupt enable flag of the processor over the operation which is sufficient as long as kernel preemption is not desired. The changes introduced here are applicable to the unmodified MicroBlaze as well, and would enable that system to support Kernel preemption, which helps to reduce latency in the system.

5.5 Boot Process and SMP Support

The boot process for Linux consists of a standard series of steps with many hooks into architecture dependent code for the early initialization process. The entry point for the MicroBlaze is in `head.S`, which first loads the initial kernel pages into the MMU. An initialization function, `early_machine_init` that resides in `setup.c`, is then called to: initialize a few system variables, clear the `.bss` section (statically allocated variables), initialize locks, initialize the *early printk console*, and finally, initialize the hardware exception vector table. The hardware exception table for the MicroBlaze, which contains the jump vectors for all exception handling, is required to exist in the processor's local BRAM starting at physical address zero. Finally, the MMU initialization is handled in a kernel call before the main entry point in the kernel, `start_kernel`, is called from `init\main.c`. From this point onward, the boot-up process architecture-dependent code is called through the standard Linux boot-up process.

The first hook back into the MicroBlaze architecture dependent code is for `setup_arch`, which parses the device tree (that stores information on the full configuration of the system), and initializes

the caches, if present. It is at this point that the secondary processors are marked as both *present* and *possible* in the main CPU masks for the system. From this point, boot-up continues as normal through to `kernel_init`, where the secondary processors are finally brought up at the end of the boot process. First idle processes are created for each of the secondary CPUs present in the system, then the boot CPU brings up the processors one at a time. The number of processors is currently selected at compile time through the kernel config file, which supports an arbitrary number of processors.

5.5.1 Secondary Processor Start-up

To boot-up a secondary processor, some mechanism is required to notify the processor to begin booting. For our system, the IPI mechanism has been selected to handle this. Using IPIs has a direct advantage over another possible approach, spinning on a memory location, as the secondary processors have no impact on the system until brought online. For a processor to be able to receive an IPI, however, it must have the particular interrupt enabled in the interrupt controller; as such, the secondary processors run a simple bootloader (located in their local BRAM) that enables a single IPI and then spins waiting to receive an interrupt. When the interrupt is received, execution jumps to the base address of the kernel where `head.S` is located. In `head.S`, the processor is identified as a secondary processor by checking the processor ID register; if it is non-zero (i.e. secondary processor), then some initialization steps, such as copying the kernel command line arguments, are skipped. From here, the preliminary MMU initialization is handled; initialization of the exception table, the stack, and the current task pointer are saved to the processor's GPSPRs. Finally, execution jumps to the custom `start_secondary` function in the kernel with the MMU running.

start_secondary

The number of operations performed to bring-up a secondary processor after the initialization performed in `head.S` is small and is included in its entirety in Figure 5.2. First, the processor is added to the kernel MMU context, and its CPU info and caches are initialized (if present). Next, the CPU is

set online, the global mask of online CPUs is updated, and finally, IPIs, the timer and interrupts are enabled before calling into the idle function.

5.5.2 Secondary Processor Bring-up

Secondary processors are brought up by sending them an IPI. The boot CPU then waits for the secondary processor to set a flag (`cpu_callin_map[cpu]`) to know that it is alive. If this flag is not set within a given period of time, the secondary processor is assumed to have failed to start up and execution continues. If the flag has been set, then the boot CPU waits for the secondary processor to mark itself as online before proceeding. This flag is therefore used to prevent the system from becoming deadlocked waiting for a processor that fails to come online.

```
/* Activate a secondary processor. */
int __devinit start_secondary(void)
{
   unsigned int cpu = smp_processor_id();
   int i;

   atomic_inc(&init_mm.mm_count);
   current->active_mm = &init_mm;
   cpumask_set_cpu(cpu, mm_cpumask(&init_mm));
   local_flush_tlb_mm(&init_mm);

   setup_cpuinfo(); // Reads Processor Version Registers
   microblaze_cache_init(); //Initializes caches, if present

   preempt_disable();
   cpu_callin_map[cpu] = 1;

   ipi_call_lock();

   /* hook into kernel arch-independent code to call any functions registered
      to be called when a new CPU is brought online */
   notify_cpu_starting(cpu);
   set_cpu_online(cpu, true);

   for_each_online_cpu(i) {
      cpumask_set_cpu(cpu, cpu_core_mask(i));
      cpumask_set_cpu(i, cpu_core_mask(cpu));
   }
   enable_ipis(cpu);
   microblaze_setup_local_timer();
   ipi_call_unlock();

   local_irq_enable();

   cpu_idle();
   return 0;
}
```

Figure 5.2 kernel Initialization Code for Secondary Processors

6 ABACUS

In this chapter, we present our hardware profiling unit, a hArdware Based Accelerator for Characterization of User Software (ABACUS) [36], our second major contribution and the remaining component of our proposed systems research framework.

6.1 Overview

In order for this platform to be useful for systems research, we need a mechanism to snoop and collect data within the system during runtime and then provide that information to the OS. ABACUS, our hardware block designed for this purpose, is a *collection of profiling units* that aggregates and provides information, relevant for workload analysis, to the OS.

During the design phase of ABACUS, we set the following criteria for its implementation:

- Exist externally from the processor core.
- Be readily portable to different architectures.
- Provide a collection of microarchitecture independent metrics, and
- Easily facilitate the inclusion of new profiling units.

ABACUS is kept external from the processor core by placing it on the system bus and snooping internal signals to create its metrics. By separating ABACUS from the processor core, the design is made more readily portable and lessens the impact on the designers of the processor by removing it from the complex demands of floor planning a processor. Unlike existing performance counters that focus on architecture dependent metrics, one of the design goals of ABACUS is to provide microarchitecture independent metrics; thus, the type of information it collects can be ported to a variety of systems.

In order to make development of new units in ABACUS a straightforward process, ABACUS has been designed as a loose connection of *profiling units* encapsulated in an infrastructure that provides memory access to the collected data, configuration space and coordination of profiling operation.

6.1.1 Architectural Overview

ABACUS consists of three main layers, as shown in Figure 6.1, the *External Interface*, the *Control Logic* and the *Profiling Units*. The *External Interface* is the wrapper that abstracts the majority of the system dependent details and the layer that must be replaced on porting to a new architecture. The next layer, the *Control Logic* contains the infrastructure to coordinate the conditions under which profiling takes place and defines the interface to the system. The final layer, the *Profiling Units*, are a collection of memory mapped units that form the core functionality of ABACUS.

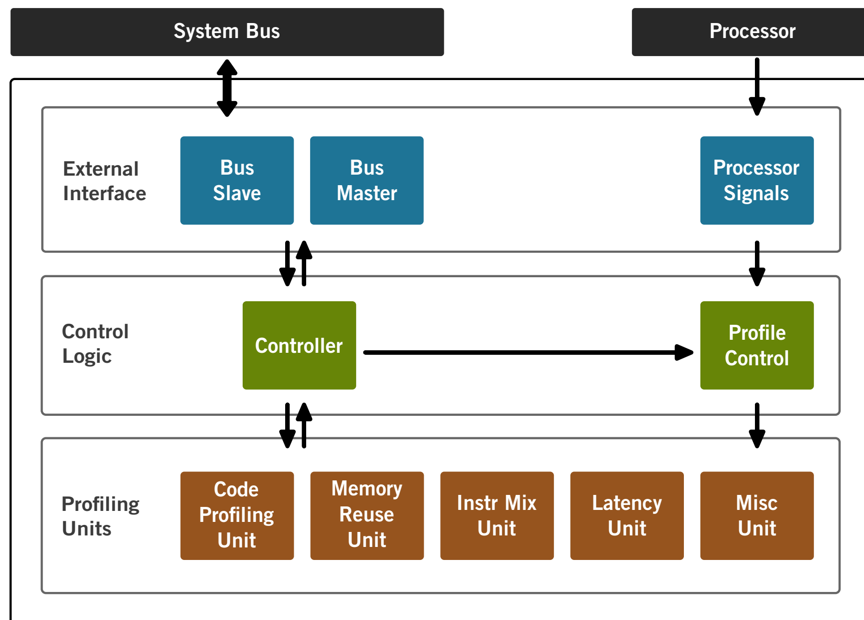

Figure 6.1 High-level ABACUS Overview

6.2 Controller

The controller design within ABACUS is a simple state machine as illustrated in Figure 6.2. Some transitions between states occur only by request of the user (solid lines), while others can be set to be triggered automatically (dotted lines) under certain conditions. For example, ABACUS can be setup to to initiate profiling on the execution of a particular instruction, such as a specific no-op, or start once a specific address has been reached. Similarly, profiling can be halted manually, after a certain number of cycles or instructions, or on a specific instruction or on reaching a specific address. In addition, through the *profile control* block, operation can be limited to profile only certain address ranges or traces of of instructions. Once profiling has been completed, ABACUS can be configured to automatically initiate a Direct Memory Access (DMA) to memory of some specific range of addresses within ABACUS. Finally, profiling can be restricted to just user space, kernel space or both.

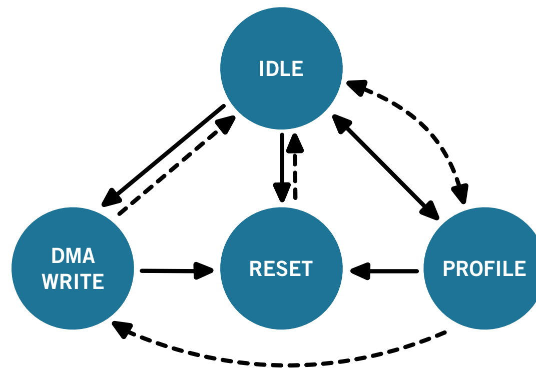

Figure 6.2 Controller State Machine: Solid lines indicate user initiated transitions, dotted line indicate automatic transitions

Within the controller, some basic information is also accounted for, such as the number of cycles spent profiling, the number of instructions, the time spent in DMA (in cycles) and the number of DMA transfers.

6.3 Profiling Units

At the core of ABACUS is a collection of independent *profiling units*. Each unit is connected to ABACUS by registering an address range for memory mapping the data it collects and, optionally, a range for configuration options. Specific signals to create each unit's metrics are passed on from the *External Interface* and are synchronized with each other to ensure that events from different parts of the system coincide. The first three units: the *Code Profiling Unit*, *Instruction Mix Unit* and the *Reuse Distance Unit* have been previously introduced in [36] and, since they are not used in this work, they are only briefly described. The *Latency Unit* is, however, new to this work and used in our analysis of the system.

6.3.1 Code Profiling Unit

The *code profiling unit* is based on the idea of SnoopP [17] with some further extensions. The module described SnoopP allowed for counting the number of cycles for which the Program Counter (PC) is within a specific address range. Through this approach, non-intrusive and cycle accurate code profiling can be preformed, not unlike software profilers such as gprof [31]. In the *Code Profiling Unit*, this functionality has been expanded to include offering *trace profiling* as well. In *trace profiling*, cycle counting begins when a certain address for the PC is reached and does not stop until an end address is reached, which may be the same as the start address. This allows for the profiling of paths of code that branch through many functions, allowing a function, and all the code it calls, to profiled with a single counter. The two types of counters can be combined together in any number within this unit.

6.3.2 Instruction Mix Unit

The *Instruction Mix Unit* is a simple unit that takes the opcode from the instruction and divides it into a number of configurable categories. This unit is one of the few units that is architecture dependent and needs to be customized for a specific architecture. The proposed use behind this unit would be in an asymmetric system where different processors had different performance profiles for

certain classes of instructions, or even in systems where processors share a floating point unit. By categorizing the instruction mix the thread can then be scheduled to reduce resource contention or be matched to the most suitable processor in the system.

6.3.3 Reuse Distance Unit

The *Reuse Distance Unit* has been designed to provide a measure of the locality of memory accesses by creating a reuse profile. The reuse profile has uses in understanding cache contention between multiple threads [42], and although it depends upon the existence of an Least Recently Used (LRU) stack, the reuse profile can be estimated in hardware with a low area overhead [43].

6.3.4 Latency Unit

The *Latency Unit* is an example of a profiling unit that implements functionality not typically available in system simulators. With the *Latency Unit*, we can measure the variable latency of memory requests and classify the latency into its constituent sources, including bus contention and DRAM latency. As storing the unique latency for every event is not feasible, the latency is binned into a histogram with a configurable number and width of bins for each constituent component. The latency unit has potential usages in monitoring bus contention, interrupt service time, or any other operation in the system that has variable latency.

Bus Latency Unit

For the investigations performed on the system presented in this work the latency unit has been modified to support multiple sources of latency. As only one read operation can complete on the PLB bus in a given cycle, with a pipelined design, we can amalgamate the events from each processor into a single histogram.

6.4 Integration into the MicroBlaze Platform

In our previous porting of ABACUS to the OpenSPARC [14] platform, we moved the external interface of ABACUS over to the PLB, which is also used by the MicroBlaze based system. The other architectural differences, such as the PC width are handled by existing generic parameters within ABACUS. To demonstrate integration into the MicroBlaze platform, we specifically looked at the behaviour of locks and bus contention using ABACUS.

6.5 Device Drivers

The final aspect of integrating a hardware profiler into the system is to include a mechanism that the OS can use to communicate with the device. This interface is supplied through kernel device drivers. As ABACUS occupies a given address range on the system bus, it can be memory mapped by the OS, allowing registers to be read or written through pointer referencing/de-referencing. The ABACUS device drivers allow access through `ioctl` functions for reading/writing single registers as well as via `mmap` to access the full address space of ABACUS. For DMA support, the drivers enable the allocation of a page of kernel memory into which ABACUS can copy its collected data. No higher level abstraction is provided at present as the details of ABACUS's interface are still evolving. However,in the long term, a higher-level interface can be provided to make interaction with ABACUS through software an easier process.

7 Platform Investigations

This chapter focuses on testing our newly created platform, and demonstrating through these tests, that it is able to support performing systems research. First, the system is put through two stress tests to measure the stability of our hardware and software support for SMP. We then perform some additional tests with ABACUS to examine the impact on bus/memory read latency when additional cores are added to the system. This chapter concludes with an examination of the scalability of the Timer and Interrupt Controller (TIC) and the changes made to the Linux code base.

7.1 Test Platform

Figure 7.1 illustrates our test system; it is the same as shown in Figure 3.2, but is included here for completeness. A set hardware configuration, consisting of four cores, is used for all tests, independent of how many processors are actually booted. This can be done without changing the behaviour of the system as the TIC and PLB both use round-robin arbiters that have the same behaviour regardless of the number of active cores. This simplifies testing as only one configuration has to be created.

Our test platform has been implemented on a Virtex 5 LX110t FPGA on an ML505 development board. The system operates at 100MHz, with the exception of the memory controller, which operates at 200MHz and interfaces with the DDR2 memory on the board. Communication between the host workstation and the FPGA system is performed through the serial console, with the Linux kernel configured to use this as its default console on boot up.

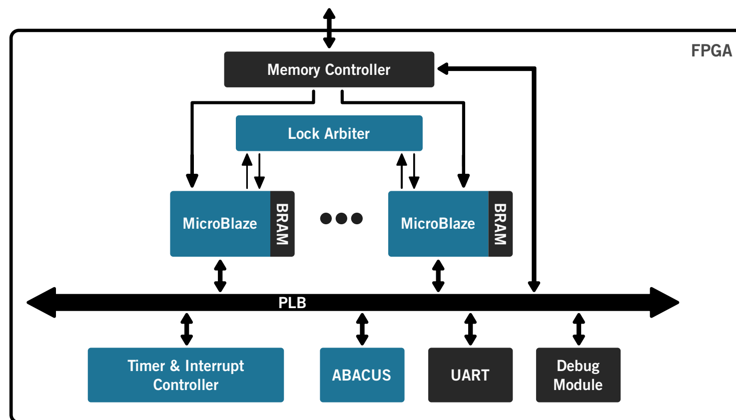

Figure 7.1 Test System Configuration

Memory Access Behaviour

To provide additional context as to how memory operations are performed in this system, all memory accesses performed are 32-bits in size. In the case of a write, as this system has a 64-bit DDR2 interface, a 32-bit write is transformed into a read-modify-write operation by the memory controller. In addition, if a DRAM refresh takes place prior to a memory operation that particular read/write will have additional latency. Finally, if multiple writes are issued without any interceding reads, then the memory controller may buffer the writes.

7.1.1 MicroBlaze Configuration

Each MicroBlaze in the system has the same configuration, including a basic FPU. The instruction side interface has a direct connection to the memory controller in order to lessen contention for the main system bus; however, the data side is connected to the PLB. The data side connection to the bus is required, as the memory controller does not support transaction ordering between different ports and that would break memory coherency for our system. In addition, each MicroBlaze has a local 8KB BRAM, which contains the bootloader code and stores the exception table for the processor.

7.1.2 ABACUS Configuration

ABACUS has been configured with only the units required for the experiments presented in this chapter, specifically the latency unit, which is configured to measure bus read latency.

Resource usage for ABACUS is highly dependent on the selected configuration (i.e. what *profiling units* are included). The configuration used for the experiments presented in this chapter includes only the Bus Latency Unit and the Misc Unit and requires 1333 Flip-Flops (FFs) and 1700 Look-up Tables (LUTs) and has a maximum operating of 162MHz, which is well above the 100MHz system frequency.

7.2 Stress Tests

In order to provide a measure of the stability/reliability of the system, we performed two basic tests. In the first, we repeatedly boot the system to see how reliable the boot-up processes is. In the second test, we stress the system by launching many compute intensive benchmarks and measure the system uptime. As there are a greater number of possible interactions in a system with more processor cores, both of these tests were performed on the four core system only.

7.2.1 Boot-up

For the boot-up test, a simple script was written to automate the programming of the FPGA. After programming the FPGA, the script waits a predetermined length of time to allow the boot-up to complete, then it attempts to login to the system. A successful boot-up is considered one in which the login prompt is reached, login is successful, and the running of a simple command, (e.g. `ls`), is successful. This test procedure was repeated 103 times during which one failure occurred (number 99). The one boot-up that failed did so between displaying the PetaLinux banner and the login prompt as shown in Figure 7.2 (To see an example complete boot log please see Appendix A).

From the message printed during the bootup process, the most likely scenario is that, CPU one did

```
Welcome to
 -----       _           _      _
| ___ \     | |         | |    (_)
| |_/ / ___ | |_   __ _ | |     _  _ __   _   _ __  __
|  __/ / _ \| __| / _` || |    | || '_ \ | | | |\ \/ /
| |   |  __/| |_ | (_| || |____| || | | || |_| | >  <
\_|    \___| \__| \__,_|\_____/|_||_| |_| \__,_|/_/\_\

on PolyBlaze


PolyBlaze login: INFO: rcu_sched_state detected stall on CPU 1
(t=15000 jiffies)
INFO: rcu_sched_state detected stalls on CPUs/tasks: { 1}
(detected by 2, t=15002 jiffies)
```

Figure 7.2 Log Output of the Single Bootup Failure

not wake-up from the brief period in which the whole system goes to idle before the login prompt is displayed. During the original system development process, a hardware bug in the TIC caused this type of error to occur. However, since that particular bug has been fixed, this suggests the source of the error may be in software, but without more details, it is not possible to know what caused this failure. Despite this one failure, the system has been booted successfully over a hundred times and since this one failure, no other faults have occurred.

7.2.2 Up-time

In order to gain a measure of the stability of the system, a compute intensive benchmark, (Dhrystone [44]), was selected to place the system under heavy load. Due to the system architecture not having data caches, this causes the behaviour to be closer to a memory intensive benchmark. More instances of the benchmark than processors are created in order to increase the amount of context switching taking place within the system. After approximately fourteen days, as shown in Figure 7.3, the system was still responsive and all processors were still active. Included, in Figure 7.4, is the output of `proc/interrupts`, which indicates the number of interrupts each processor has received while the system has been running. The system up-time test, with the Dhrystone benchmark, provides

a basic measure of the stability of the system. More intensive benchmarks, such as parallel workloads with a significant portion of data sharing can be tested on the system in the future.

```
~ # cat /proc/uptime
1250621.86 1415635.52
```

Figure 7.3 Output of /proc/uptime after fourteen days of uptime

```
~ # cat /proc/uptime
1250621.86 1415635.52
~ # cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
 0:       24592      37500      32229      29820  per_cpu  ipi reschedule
 1:      535851     733253     742347     750536  per_cpu  ipi call function
 2:      133376     103843      76843      65000  per_cpu  ipi call function single
 4:  223766892  224338911  228982339  220920627  per_cpu  timer
 5:         775        614        580        504  per_cpu  serial
```

Figure 7.4 Output of /proc/interrutps after fourteen days of uptime

7.3 Bus Read Latency

The next experiment provides insight into the performance of the system as we scale the number of cores from a single core up to four cores. In this test, ABACUS is used to monitor the bus traffic in the system and capture a cycle accurate histogram of all bus read latencies. The latency unit has been configured with 512 bins to allow the recording of individual latencies from 1 to 512 cycles. Latencies greater than 512 cycles are accumulated in the highest bin. In this particular system, without caches, most periods of execution create a reasonable amount of bus traffic. The only requirement of this test is that each core in the system is frequently accessing memory. This is particularly the case during the boot-up of the system, which we have selected as the time window in which to collect our data.

The data collection window begins upon exiting reset and terminates at the display of the login prompt. We also measured the boot time during these tests as a second point of reference. As a 512 cycle window is too large to plot on a single page, the results have been subdivided into two

figures, Figure 7.5 and Figure 7.6. These two figures show the number of times a given latency occurred during a bus read for the single, dual and quad core configurations. Figure 7.5, shows the accesses to the UART (12 cycles) and TIC (8 cycles) peripherals, while Figure 7.6 shows primarily memory accesses. Due to contention, some UART and TIC accesses will be captured in Figure 7.6, however, as memory accesses are more than two orders of magnitude more frequent these UART and TIC accesses due not impact the distribution we see for memory accesses. In Figure 7.6, bin 51 includes all latencies greater than or equal to 51 cycles.

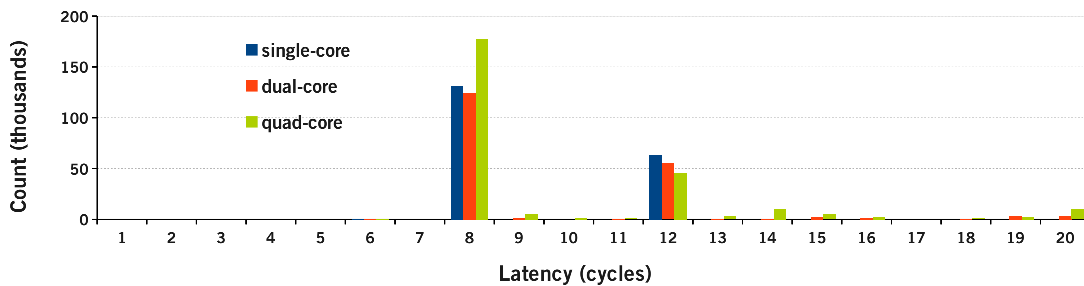

Figure 7.5 TIC and UART bus read latency

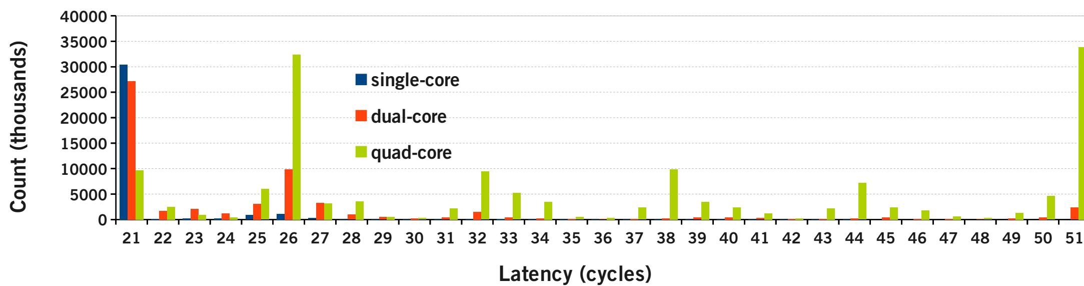

Figure 7.6 Memory Bus Read Latency

Single-core

The single-core system provides a baseline for the latency measurement as there is no bus contention in the system as the processor is the only master on the bus. This allows us to measure the memory access time and see how much variability there is with no bus contention. Sources of which include,

DRAM refresh cycles and instruction side accesses to memory. Comparing results between two runs illustrated that they varied by less than one tenth of a percent for the largest bin (21 cycles) for the single-core test. Boot-time was measured to be approximately 49 seconds.

Dual-core

In the dual-core system, we can see the impact of bus contention for the memory controller. The majority of reads still take only 21 cycles, but there is a significant portion of the memory accesses that require 26 cycles and the distribution of memory access latencies has a longer tail, compared to the single-core case. This impact can be seen in the boot-up time as well, which now takes approximately 52 seconds.

Quad-core

It is not until the quad-core system that a significant impact is seen on read latencies. Here, the bin with the largest count is at 26 cycles, suggesting that there is often one outstanding read in progress when a processor attempts to read from memory. The tail of the latency distribution is also much longer than that of the dual-core system and is extends past 512 cycles (although the number of memory access requiring more than 512 cycles is around $100/135x10^6$), or less than one one hundred thousandth of a percent). It is also only once four cores are reached that we start to see concurrent reads to the TIC having an impact on bus read latency, which can be seen in Figure 7.5. For the TIC, the only case where multiple reads will be taking place is if multiple interrupts occur simultaneously and are assigned to different processors within a few cycles, the most frequent case being a broadcast IPI. For the quad-core system, a substantial increase in boot-time was seen to approximately 91 seconds (an approximately 86% increase in boot-time). In addition to the memory requests captured here, each core has a connection to the memory controller for its instruction requests, which are also increasing the load on the memory controller, and thus the latency as well.

7.4 Design Scalability

As can be seen in the quad-core setup, there is often one outstanding read transaction on the bus at any given time. As the system scales beyond four cores, this becomes increasingly the case until the point is reached in which every processor has an outstanding bus read request in progress. A side effect of this will be that memory accesses will become completely serialized in the system due to the round-robin arbitration process in the PLB. In our current system, this is a serious issue as the reservation design can fail if LWX/SWX sequences between processors become serialized. One scenario can occur when a processor trying to release a lock fails to obtain a reservation because the LWX for a processor trying to obtain the lock always occurs first. While in some cases, interrupts can be expected to reintroduce randomness into the system, some locks are held under cases where interrupts are disabled. No reservation contention issues were encountered in system with up to four cores, however, systems with 5-8 cores would become unresponsive during boot-up. Investigations showed that multiple processors became stuck within the spinlock functions. In one case, with an 8 core system, halting one of the cores and then resuming it through the hardware debugger allowed the system to reach the login prompt, however, while attempting to login, the system became unresponsive again. In order to address this issue, an alternative behaviour is likely needed for the reservation system, the design of which is already underway. To remove the potential for starvation, every processor will have its reservation bit set on a LWX instruction. SWX stores would remain conditional on the reservation bit being set but the decision to proceed would be made as the stores are committed to memory and not within the processor. By structuring the implementation this way, we know the ordering of memory operations and will know at this point if any other stores have already been performed to this memory address since the reservation bit was set. If a store, from any processor with a matching address, is processed before the paired SWX instruction arrives, the reservation bit will be cleared and the store aborted. This alternative approach removes the need to pair LWX/SWX instructions to release a lock, and thus, the starvation issue that can occur in our existing implementation when the owner of the lock is unable to obtain the reservation bit.

7.5 Hardware Resource Usage and Scalability

In this section, we examine the resource usage and operating frequency of the newly implemented hardware blocks and their scalability as well as the resource usage impact of our changes to the MicroBlaze processor. Note, all resource usage and operating frequency statistics are for an implementation on the Virtex 5 LX110t FPGA.

7.5.1 Timer and Interrupt Controller

The TIC was designed to scale to 8 cores and support up to 32 interrupts (including IPIs, a timer interrupt and external interrutps). The TIC is configurable across the number of IPIs supported, the number of cores and the number of external interrupt sources. In our analysis, we have fixed the number of IPIs implemented to four (the number used by the kernel in our implementation). The plot shown in Figure 7.7 provides the resource usage and operating frequency for the dual-core configuration across a varied number of external interrupts, up to the maximum of 27. In this configuration, the maximum operating frequency of the core is 142MHz, which is well above the 100MHz target of the system. The next two Figures 7.8 (a) and (b), illustrate how the TIC scales as the number of cores is increased with a single external interrupt, and with the full 27 external interrupts.

For the 8-core system, by the time the design reaches 8 external interrupts (13 total) the operating frequency has dropped below 100MHz to 91MHz. The critical path in the design being the selection and arbitration of interrupts which currently occurs in a single cycle. The design could be readily modified to pipeline this process by splitting up the interrupt selection and arbitration phases without otherwise impacting the behaviour or operation of the TIC. However, at present we do not have a system with these requirements so this change will be future work.

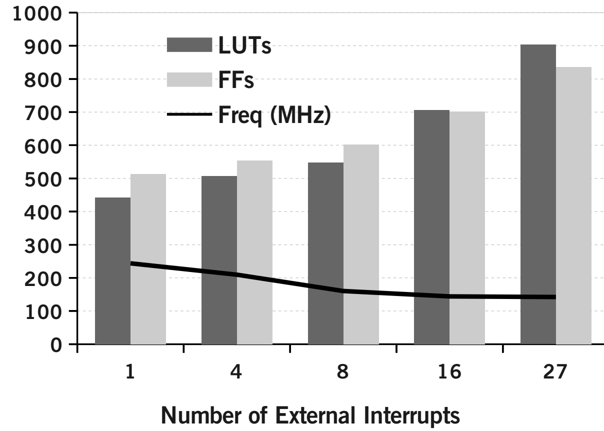

Figure 7.7 Dual-core TIC Configuration Resource Usage and Operating Frequency

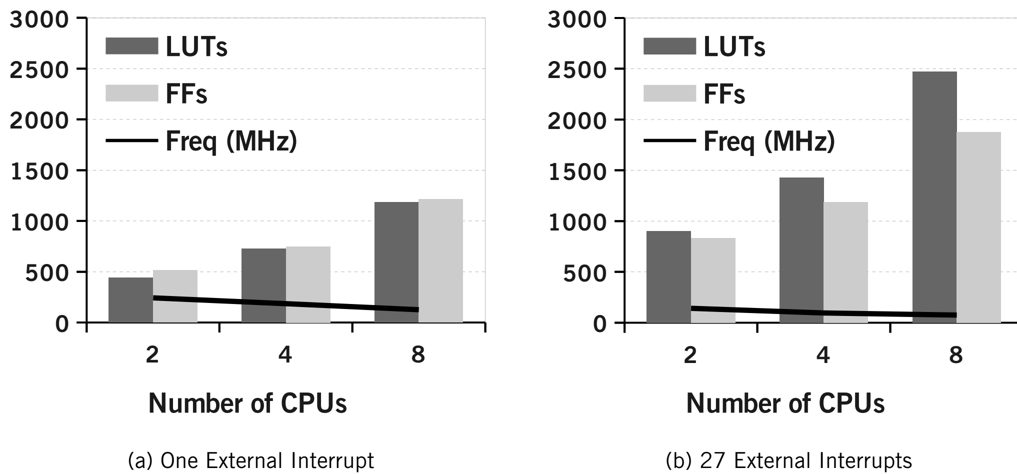

Figure 7.8 TIC Scalability Across an Increasing Number of CPUs

7.5.2 MicroBlaze

In updating the MicroBlaze to a scalable, multicore architecture, we modified its ISA to support an additional set of registers, the General Purpose Special Purpose Registers (GPSPRs). Our implementation supports four of these resisters and each resister has a width of 32 bits. Compared against the unmodified MicroBlaze, LUT usage has been increased by approximately **2%** (86 LUTs) and for FFs,

by approximately **7%** (224 FFs). The increase in FFs is higher as each register bit requires a FF, but the extra control logic ties into the existing infrastructure.

7.6 Kernel Stats

In this section, we report some statistics on the code changes made in adding SMP support to the MicroBlaze platform. In total, 5 new files were added, 48 files modified, and the static code size grew from 464.5KB to 525.2KB (13% increase). For a full diffstat of the code changes please see Appendix B. The largest changes were focused on adding atomic primitive support, MMU changes and support for the new combined TIC. The previous support for atomic operations in the MicroBlaze architecture used only interrupt disabling to ensure the atomicity of operations. With atomic support supplied through conditional load/store pairs, many operations need not have interrupts disabled, therefore, our work would also enable kernel-preemption support in a single-core system.

8 Conclusions and Future Work

Today's multicore systems provide many challenges to the OS, particularly in the area of efficiently utilizing these systems. Due to the complex interactions that take place within the system, without visibility into the internals of the system, we do not have the ability to understand performance ahead of time and thus make effective scheduling decisions. Ideally, future systems will contain a hardware block able to monitor system interactions at runtime, which the OS can then use to more effectively utilize the resources of the platform.

8.1 Conclusions

In this work, we have presented a framework that enables new avenues for systems research. We have outlined and implemented a multicore platform that supports the Linux OS and contains an integrated profiling unit, ABACUS, through which non-invasive runtime profiling of the system can be performed.

8.1.1 Multicore MicroBlaze Platform

We have taken a single-core MicroBlaze processor and extended the platform with multicore support. Our support includes both modifications to the hardware and to the Linux kernel. Careful consideration of design parameters ensures that the framework is scalable. In our initial implementation, we have created a system that can scale from one to four processors and have demonstrated its reliability and stability through our test cases. The four core system has been demonstrated to be stable over a two week period under heavy load and during stress testing of the boot-up process, only a single failure was encountered in over 100 attempts.

8.1.2 Hardware Profiling

A configurable hardware profiling unit, ABACUS, has been created to provide visibility into the internal workings of the system. We have demonstrated how ABACUS can be integrated into a multicore system to provide unique insight on workload interactions. At present, the metrics we collect are meant to demonstrate some basic functionality, however, our framework allows, and enables, investigations into more complex metrics in the future. Furthermore, we have illustrated how using a processor emulator provides a model that enables us to consider additional metrics that require greater visibility into the system, such as latency profiles for memory accesses.

8.2 Future Work

As this work has created an extendable framework for performing systems and architectural research, there are many avenues for future work. Currently, work is underway to scale the system up to at least 8 processors by addressing the scalability issues found in the conditional load/store implementation. Separate work is also underway to provide the mechanisms required for a coherent memory hierarchy consisting of multiple levels of caches. From this point, investigations can then turn to creating systems with asymmetric processors, an example of which would be a system where not all cores have a FPU. Another example could be as system where the processor cores have varying cache sizes and replacement policies. Finally, future work could also include integrating custom accelerator cores into the system to create a heterogeneous platform.

Bibliography

[1] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Contention on Multicore Processors via Scheduling," in *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[2] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33 –38, july 2008.

[3] *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3B: System Programming Guide, Part 2.* . [Online]. Available: www.intel.com/Assets/PDF/manual/253669.pdf

[4] *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*. [Online]. Available: support.amd.com/us/Processor_TechDocs/31116.pdf

[5] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[6] Xilinx Inc., *MicroBlaze Processor Reference Guide*. [Online]. Available: www.xilinx.com/support/documentation/sw_manuals/xilinx12_4/mb_ref_guide.pdf

[7] "About — PetaLogix." [Online]. Available: http://www.petalogix.com/about

[8] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *Proc. of the 40th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, 2007, pp. 249–261.

[9] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of fpga-based soft processors," in *2005 Int'l Conf. on Compilers, architectures and synthesis for embedded systems*, 2005, pp. 202–212.

[10] "RAMP - Research Accelerator for Multiple Processors." [Online]. Available: ramp.eecs.berkeley.edu/

[11] Daniel Burke, John Wawrzynek, Krste Asanovic, Alex Krasnov, Andrew Schultz, Greg Gibeling, Pierre-Yves Droz, "Ramp blue: Implementation of a multicore 1000 processor fpga system," in *Reconfigurable Systems Summer Institute*, Urbana, IL, 2008.

[12] P. H. Wang, J. D. Collins, C. T. Weaver, B. Kuttanna, S. Salamian, G. N. Chinya, E. Schuchman, O. Schilling, T. Doil, S. Steibl, and H. Wang, "Intel® atom processor core made fpga-synthesizable," in *The ACM/SIGDA Int'l Symp. on FPGAs*, 2009, pp. 209–218.

[13] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang, "Intel nehalem processor core made fpga synthesizable," in *The ACM/SIGDA Int'l Symp. on FPGAs*, 2010, pp. 3–12.

[14] "OpenSPARC FPGA | Field-Programmable Gate Array | OpenSPARC." [Online]. Available: www.opensparc.net/fpga/index.html

[15] *GRLIB IP Core User's Manual*. [Online]. Available: www.gaisler.com/products/grlib/grip.pdf

[16] G. G. F. Lemieux, "Hardware performance monitoring in multiprocessors," Master's thesis, ECE Dept. University of Toronto, 1996.

[17] L. Shannon and P. Chow, "Using reconfigurability to achieve real-time profiling for hardware/-software codesign," in *Proc. of ACM/SIGDA 12th Int'l Symp. on FPGAs*, 2004, pp. 190–199.

[18] M. Finc and A. Zemva, "A systematic approach to profiling for hardware/software partitioning," *Comput. Electr. Eng.*, vol. 31, no. 2, pp. 93–111, 2005.

[19] Altera Inc., *(2011, May) The NIOS Soft CPU Family*. [Online]. Available: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[20] "PetaLinux System Development Kit — PetaLogix." [Online]. Available: www.petalogix.com/products/petalinux

[21] J. Handy, *The cache memory book*. San Diego: Academic Press, 1998.

[22] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded sparc processor," *Micro, IEEE*, vol. 25, no. 2, pp. 21 – 29, march-april 2005.

[23] E. W. Dijkstra, "Cooperating sequential processes," 1968. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF

[24] IBM Corp., *PPC405Fx Embedded Processor Core User's Manual*. [Online]. Available: www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/D060DB54BD4DC4F2872569D2004A30D6/$file/ppc405fx_um.pdf

[25] *The SPARC Architecture Manual Version 9*. [Online]. Available: developers.sun.com/solaris/articles/sparcv9.pdf

[26] "The Linux Kernel Archives." [Online]. Available: kernel.org

[27] D. Tam, R. Azimi, and M. Stumm, “Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 47–58, 2007.

[28] A. Snavely and D. M. Tullsen, “Symbiotic jobscheduling for a simultaneous multithreaded processor,” *SIGARCH Comput. Archit. News*, vol. 28, no. 5, pp. 234–244, 2000.

[29] A. Pesterev, N. Zeldovich, and R. Morris, “Locating cache performance bottlenecks using data profiling,” in *European Conf. on Computer Systems*, 2010.

[30] “OProfile - A System Profiler for Linux (News).” [Online]. Available: http://oprofile.sourceforge.net/news/

[31] “GNU’s Not Unix! The GNU Project and Free Software Foundation (FSF).” [Online]. Available: http://www.gnu.org

[32] “VTune™ Amplifier XE 2011 from Intel - Intel® Software Network.” [Online]. Available: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/

[33] “Sun Studio 12: Performance Analyzer.” [Online]. Available: http://docs.oracle.com/cd/E19205-01/819-5264/

[34] T. Austin, E. Larson, and D. Ernst, “Simplescalar: An infrastructure for computer system modeling,” *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[35] C.-K. Luk *et al.*, “Pin: building customized program analysis tools with dynamic instrumentation,” in *PLDI*, 2005, pp. 190–200.

[36] E. Matthews, L. Shannon, and A. Fedorova, “A configurable framework for investigating workload execution,” in *Field-Programmable Technology (FPT), 2010 International Conference on*, 2010, pp. 409 –412.

[37] “TI unveils multi-core OMAP 4.” [Online]. Available: www.eetimes.com/electronics-products/processors/4110441/TI-unveils-multi-core-OMAP-4

[38] Xilinx Inc., *LogiCORE IP XPS Interrupt Controller (v2.01a)*. [Online]. Available: www.xilinx.com/support/documentation/ip_documentation/xps_intc.pdf

[39] “Clockevents and dyntick [LWN.net].” [Online]. Available: http://lwn.net/Articles/223185/

[40] Xilinx Inc., *LogiCORE IP XPS Timer/Counter (v1.02a)*. [Online]. Available: www.xilinx.com/support/documentation/ip_documentation/xps_timer.pdf

[41] IBM Corp., *Multiprocessor Interrupt Controller Data Book*, 2006. [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/F27971551C9EED8E8525774A0048770A/$file/mpic_db_05_16_2011.pdf

[42] Cascaval, Calin and Padua, David A., “Estimating cache misses and locality using stack distances,” in *in Int’l Conf. on Supercomputing*, 2003, pp. 150–159.

[43] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *MICRO 39: in 39th Annual IEEE/ACM Int’l Symp. on Microarchitecture*, 2006, pp. 423–432.

[44] R. P. Weicker, “Dhrystone: a synthetic systems programming benchmark,” *Commun. ACM*, vol. 27, pp. 1013–1030, October 1984. [Online]. Available: http://doi.acm.org/10.1145/358274.358283

A Example Boot Log

```
CPU 1 waiting for IPI
CPU 2 waiting for IPI
CPU 3 waiting for IPI
early_printk_console is enabled at 0x83e01000
Ramdisk addr 0x00000003, Compiled-in FDT at 0xc01a4de8
MMU: Allocated 1088 bytes of context maps for 255 contexts
Linux version 2.6.37.4-00531-g2db5587-dirty (ematthew@sally)
   (gcc version 4.1.2) #1530 SMP Fri Dec 9 13:16:45 PST 2011
setup_cpuinfo: initialising
setup_cpuinfo: Using full CPU PVR support
cache: wt_msr_noirq
setup_memory: max_mapnr: 0x10000
setup_memory: min_low_pfn: 0x50000
setup_memory: max_low_pfn: 0x60000
On node 0 totalpages: 65536
free_area_init_node: node 0, pgdat c020f100, node_mem_map c07de000
  Normal zone: 512 pages used for memmap
  Normal zone: 0 pages reserved
  Normal zone: 65024 pages, LIFO batch:15
PERCPU: Embedded 7 pages/cpu @c09e1000 s6464 r8192 d14016 u32768
pcpu-alloc: s6464 r8192 d14016 u32768 alloc=8*4096
pcpu-alloc: [0] 0 [0] 1 [0] 2 [0] 3
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 65024
Kernel command line:  console=ttyS0,115200
PID hash table entries: 1024 (order: 0, 4096 bytes)
Dentry cache hash table entries: 32768 (order: 5, 131072 bytes)
Inode-cache hash table entries: 16384 (order: 4, 65536 bytes)
Memory: 251448k/262144k available
Hierarchical RCU implementation.
NR_IRQS:32
xlnx,xps-intc-1.00.a #0 at 0xd0000000, num_irq=6, edge=0x0
requesting IPIs
cpu: 0 enable_or_unmask: 0
cpu: 0 enable_or_unmask: 1
cpu: 0 enable_or_unmask: 2
CPU: 0, microblaze_timer_set_mode: shutdown
CPU: 0, microblaze_timer_set_mode: periodic
cpu: 0 enable_or_unmask: 4
Calibrating delay loop... 2.24 BogoMIPS (lpj=4496)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 512
Waking CPU 1
setup_cpuinfo: initialising
setup_cpuinfo: Using full CPU PVR support
cache: wt_msr_noirq
```

```
Processor 1 found.
CPU: 1, microblaze_timer_set_mode: shutdown
CPU: 1, microblaze_timer_set_mode: periodic
Waking CPU 2
setup_cpuinfo: initialising
setup_cpuinfo: Using full CPU PVR support
cache: wt_msr_noirq
Processor 2 found.
CPU: 2, microblaze_timer_set_mode: shutdown
CPU: 2, microblaze_timer_set_mode: periodic
Waking CPU 3
setup_cpuinfo: initialising
setup_cpuinfo: Using full CPU PVR support
cache: wt_msr_noirq
Processor 3 found.
CPU: 3, microblaze_timer_set_mode: shutdown
CPU: 3, microblaze_timer_set_mode: periodic
Brought up 4 CPUs
NET: Registered protocol family 16
bio: create slab <bio-0> at 0
Switching to clocksource microblaze_clocksource
CPU: 1, microblaze_timer_set_mode: oneshot
CPU: 2, microblaze_timer_set_mode: oneshot
CPU: 3, microblaze_timer_set_mode: oneshot
CPU: 0, microblaze_timer_set_mode: oneshot
NET: Registered protocol family 1
Skipping unavailable RESET gpio -2 (reset)
GPIO pin is already allocated
JFFS2 version 2.2. (NAND) (SUMMARY)  © 2001-2006 Red Hat, Inc.
msgmni has been set to 491
Block layer SCSI generic (bsg) driver version 0.4 loaded (major 254)
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
90000000.debug: ttyUL0 at MMIO 0x90000000 (irq = -1) is a uartlite
83e00000.serial: ttyS0 at MMIO 0x83e01003 (irq = 5) is a 16550A
console [ttyS0] enabled
cpu: 1 enable_or_unmask: 5
Freeing unused kernel memory: 5836k freed
Mounting proc:
Mounting var:
Populating /var:
Running local start scripts.
Mounting sysfs:
mdev: initialising /dev
Mounting /etc/config:
Populating /etc/config:
```

```
flatfsd: Nonexistent or bad flatfs (-48), creating new one...
flatfsd: Failed to write flatfs (-48): No such file or directory
flatfsd: Created 7 configuration files (226 bytes)
Mounting devpts:
Setting hostname:
Bringing up network interfaces:
ifconfig: socket: Address family not supported by protocol
ifconfig: socket: Address family not supported by protocol
ifconfig: socket: Address family not supported by protocol
route: socket: Address family not supported by protocol

Welcome to
 _____       _           _      _
| ___ \     | |         | |    (_)
| |_/ / ___ | |_   __ _ | |     _  _ __   _   _ __  __
|  __/ / _ \| __| / _` || |    | || '_ \ | | | |\ \/ /
| |   |  __/| |_ | (_| || |____| || | | || |_| | >  <
\_|    \___| \__| \__,_|\_____/|_||_| |_| \__,_|/_/\_\

on PolyBlaze


PolyBlaze login: root
Password:
~ # ls
bin      etc      init     linuxrc  proc     sys      usr
dev      home     lib      mnt      sbin     tmp      var
~ #
```

B Kernel Code Stats

```
$ diff -rwB microblaze_ori microblaze_new | diffstat
 microblaze_new/Kconfig                          |   25 +
 microblaze_new/include/asm/atomic.h             |  164 +++++++++++-
 microblaze_new/include/asm/bitops.h             |  251 +++++++++++++++++
 microblaze_new/include/asm/cpuinfo.h            |    6
 microblaze_new/include/asm/entry.h              |    2
 microblaze_new/include/asm/futex.h              |    4
 microblaze_new/include/asm/intc.h               |only
 microblaze_new/include/asm/irqflags.h           |   11
 microblaze_new/include/asm/mmu.h                |    6
 microblaze_new/include/asm/mmu_context_mm.h     |  106 ++-----
 microblaze_new/include/asm/percpu.h             |    8
 microblaze_new/include/asm/pgalloc.h            |  195 +++-----------
 microblaze_new/include/asm/pgtable.h            |   19 -
 microblaze_new/include/asm/setup.h              |    3
 microblaze_new/include/asm/smp.h                |only
 microblaze_new/include/asm/spinlock.h           |only
 microblaze_new/include/asm/spinlock_types.h     |only
 microblaze_new/include/asm/system.h             |  107 ++++++-
 microblaze_new/include/asm/tlb.h                |   12
 microblaze_new/include/asm/tlbflush.h           |   41 +--
 microblaze_new/include/asm/types.h              |    6
 microblaze_new/kernel/Makefile                  |    5
 microblaze_new/kernel/asm-offsets.c             |    2
 microblaze_new/kernel/cpu/cache.c               |  134 +++++----
 microblaze_new/kernel/cpu/cpuinfo-pvr-full.c    |    2
 microblaze_new/kernel/cpu/cpuinfo-static.c      |    2
 microblaze_new/kernel/cpu/cpuinfo.c             |   13
 microblaze_new/kernel/cpu/mb.c                  |   65 ++--
 microblaze_new/kernel/cpu/pvr.c                 |    1
 microblaze_new/kernel/early_printk.c            |    7
 microblaze_new/kernel/entry.S                   |  150 +++++++----
 microblaze_new/kernel/head.S                    |  152 +++++++++++
 microblaze_new/kernel/hw_exception_handler.S    |  161 +++++++----
 microblaze_new/kernel/intc_timer.c              |only
 microblaze_new/kernel/irq.c                     |   18 -
 microblaze_new/kernel/process.c                 |    1
 microblaze_new/kernel/setup.c                   |   67 +++-
 microblaze_new/kernel/signal.c                  |    1
 microblaze_new/kernel/smp.c                     |only
 microblaze_new/kernel/vmlinux.lds.S             |    5
 microblaze_new/mm/Makefile                      |    2
 microblaze_new/mm/consistent.c                  |    4
 microblaze_new/mm/init.c                        |   10
 microblaze_new/mm/mmu_context.c                 |  367 +++++++++++++++++++++++++--
```

```
microblaze_new/mm/pgtable.c                    |  156 +++++++++++
microblaze_new/mm/tlb_nohash.c                 |only
microblaze_ori/kernel/intc.c                   |only
microblaze_ori/kernel/timer.c                  |only
48 files changed, 1738 insertions(+), 553 deletions(-)
```

C Example Kernel Source

This appendix contains a representative (but not complete) selection of source files modified/added in bringing SMP support to the MicroBlaze platform.

C.1 spinklock.h

```
#ifndef __ASM_SPINLOCK_H
#define __ASM_SPINLOCK_H
#ifdef __KERNEL__
/*
 * Unlocked value: 0
 * Locked value: 1
 */

#include <linux/console.h>
#include <asm/irqflags.h>
#define arch_spin_is_locked(x)      ((x)->lock != 0)
#define arch_spin_unlock_wait(lock) \
   do { while (arch_spin_is_locked(lock)) cpu_relax(); } while (0)

/*
static inline void arch_spin_lock(arch_spinlock_t *lock)
{
   ////eprintk("arch_spin_lock\n");
   unsigned long tmp;
   __asm__ __volatile__ (
      "1: lwx   %0, %1, r0;\n" // load conditional address in %1 to %0
      "   bnei  %0, 1b;\n"     // not zero? try again
      "   addi  %0, r0, 1;\n"  // increment lock by 1
      "   swx   %0, %1, r0;\n" // attempt store
      "   addic %0, r0, 0;\n"  // checking msr carry flag
      "   bnei  %0, 1b;\n"     // store failed (MSR[C] set)? try again

      : "=&r" (tmp)            // Outputs   : temp variable for load
          result
      : "r"   (&lock->lock)    // Inputs    : lock address
      : "cc", "memory"
   );
}*/

static inline int arch_spin_trylock(arch_spinlock_t *lock)
{
   //eprintk("arch_spin_trylock\n");
   unsigned long prev, tmp, dummy;
   __asm__ __volatile__ (
```

```
      "1: lwx   %0, %2, r0;\n" /* load conditional address in %2 to %0
         */
      "   bnei  %0, 2f;\n"     /* not zero? clear reservation */
      "   addi  %1, r0, 1;\n"  /* increment lock by one if lwx was
         sucessful*/
      "   swx   %1, %2, r0;\n" /* attempt store */
      "   addic %1, r0, 0;\n"  /* checking msr carry flag */
      "   bnei  %1, 1b;\n"     /* store failed (MSR[C] set)? try again
         */
      "   bri   3f;\n"         /* jump to check */
      "2: swx   %0, %3, r0;\n" /* attempt store */
      "3:"
      : "=&r" (prev), "=&r" (tmp) /* Outputs : temp variable for load
         result */
      : "r"   (&lock->lock), "r" (&dummy)        /* Inputs  : lock
         address */
      : "cc", "memory"
   );

   /*tmp should be zero if lock is available and tmp should be zero if
      store
     suceeded */
   return (prev == 0 && tmp == 0);
}

static inline void arch_spin_lock(arch_spinlock_t *lock)
{
   while (1) {
      if (likely(arch_spin_trylock(lock)))
         break;
      do { } while (unlikely(arch_spin_is_locked(lock)));
   }
}

#define arch_spin_lock_flags(lock, flags) arch_spin_lock(lock)

/*static inline
void arch_spin_lock_flags(arch_spinlock_t *lock, unsigned long flags)
{
   unsigned long flags_dis;

   while (1) {
      if (likely(arch_spin_trylock(lock)))
         break;
      local_save_flags(flags_dis);
      local_irq_restore(flags);
      do {
      } while (unlikely(arch_spin_is_locked(lock)));
```

```
      local_irq_restore(flags_dis);
   }
}*/

/* Ick, we have to lock on clearing too as we always have to pair LWX/
    SWX
   together, otherwise some other processor could have the lock on this
       memory
   location (not the spinlock itself just the mem location) and
       therefore a write
   would be cleared when the swx went through on the first processor and
        the lock
   would never be released as the write by the second processor was "
       lost"
*/
static inline void arch_spin_unlock(arch_spinlock_t *lock)
{
   ////eprintk("arch_spin_unlock\n");
   unsigned long tmp;
   __asm__ __volatile__ (
      "1: lwx   %0, %1, r0;\n" /* load conditional address in %1 to %0
          */
      "   nop;\n"
      "   swx   r0, %1, r0;\n" /* attempt clear */
      "   addic %0, r0, 0;\n"  /* checking msr carry flag */
      "   bnei  %0, 1b;\n"     /* store failed (MSR[C] set)? try again
          */

      : "=&r" (tmp)            /* Outputs   : temp variable for load
          result */
      : "r"   (&lock->lock)    /* Inputs    : lock address */
      : "cc", "memory"
   );

}

/*
 * RWLOCKS
 */
#define WRITE_LOCK_VALUE -1
/*
static inline void arch_write_lock(arch_rwlock_t *rw)
{
   //eprintk("arch_write_lock\n");
   unsigned long tmp;
   __asm__ __volatile__ (
      "1: lwx   %0, %1, r0;\n" // load conditional address in %1 to %0
      "   bneid %0, 1b;\n"     // not zero? try again
```

```
      "   addi  %0, r0, -1;\n" // set tmp to -1
      "   swx   %0, %1, r0;\n" // attempt store
      "   addic %0, r0, 0;\n"  // checking msr carry flag
      "   bnei  %0, 1b;\n"     // store failed (MSR[C] set)? try again

      : "=&r" (tmp)            // Outputs   : temp variable for load
         result
      : "r"   (&rw->lock)      // Inputs    : lock address
      : "cc", "memory"
   );
}*/

static inline int arch_write_trylock(arch_rwlock_t *rw)
{
   //eprintk("arch_write_trylock\n");
   unsigned long prev, tmp;
   __asm__ __volatile__ (
      "1: lwx   %0, %2, r0;\n" /* load conditional address in %1 to tmp
         */
      "   bnei  %0, 2f;\n"     /* not zero? clear conditional in case we
          got it */
      "   addi  %1, r0, -1;\n" /* set tmp to -1 */
      "   swx   %1, %2, r0;\n" /* attempt store */
      "   addic %1, r0, 0;\n"  /* checking msr carry flag */
      "   bnei  %1, 1b;\n"     /* store failed (MSR[C] set)? try again
         */
      "   bri   3f;\n"         /* jump to check */
      "2: swx   %0, %3, r0;\n" /* attempt store */
      "3:"
      :"=&r" (prev), "=&r" (tmp) /* Outputs : temp variable for load
         result */
      : "r"  (&rw->lock), "r" (&tmp)         /* Inputs    : lock address
          */
      : "cc", "memory"
   );
   /*prev value should be zero and MSR should be clear*/
   return (prev == 0 && tmp == 0);
}

static inline void arch_write_lock(arch_rwlock_t *rw)
{
   while (1) {
      if (likely(arch_write_trylock(rw)))
         break;
      do { } while (unlikely(rw->lock != 0));
   }
}
```

```
/* Ick, we have to lock on clearing too as we always have to pair LWX/
    SWX
   together, otherwise some other processor could have the lock on this
       memory
   location (not the spinlock itself just the mem location) and
       therefore a write
   would be cleared when the swx went through on the first processor and
        the lock
   would never be released as the write by the second processor was "
       lost"
*/
static inline void arch_write_unlock(arch_rwlock_t *rw)
{
   //eprintk("arch_write_unlock\n");
   unsigned long tmp;
   __asm__ __volatile__ (
      "1: lwx   %0, %1, r0;\n" /* load conditional address in %1 to %0
          */
      "   nop;\n"
      "   swx   r0, %1, r0;\n" /* attempt clear */
      "   addic %0, r0, 0;\n"  /* checking msr carry flag */
      "   bnei  %0, 1b;\n"     /* store failed (MSR[C] set)? try again
          */

      : "=&r" (tmp)            /* Outputs   : temp variable for load
          result */
      : "r"   (&rw->lock)      /* Inputs    : lock address */
      : "cc", "memory"
   );
}

/* write_can_lock - would write_trylock() succeed? */
#define arch_write_can_lock(x)      ((x)->lock == 0)

/*
 * Read locks
 */
/*
static inline void arch_read_lock(arch_rwlock_t *rw)
{
   //eprintk("arch_read_lock\n");
   unsigned long tmp;
   __asm__ __volatile__ (
      "1: lwx   %0, %1, r0;\n" // load conditional address in %1 to %0
      "   bltid %0, 1b;\n"     // < 0 (WRITE LOCK active) try again
      "   addi  %0, %0, 1;\n"  // increment lock by 1 if lwx was
          sucessful
```

```
    "   swx   %0, %1, r0;\n" // attempt store
    "   addic %0, r0, 0;\n"  // checking msr carry flag
    "   bnei  %0, 1b;\n"     // store failed (MSR[C] set)? try again

    : "=&r" (tmp)           // Outputs   : temp variable for load
        result
    : "r"   (&rw->lock)     // Inputs    : lock address
    : "cc", "memory"
  );
}*/


static inline void arch_read_unlock(arch_rwlock_t *rw)
{
  //eprintk("arch_read_unlock\n");
  unsigned long tmp;
  __asm__ __volatile__ (
    "1: lwx   %0, %1, r0;\n" /* load conditional address in %1 to tmp
        */
    "   addi  %0, %0, -1;\n" /* tmp = tmp - 1*/
    "   swx   %0, %1, r0;\n" /* attempt store */
    "   addic %0, r0, 0;\n"  /* checking msr carry flag */
    "   bnei  %0, 1b;\n"     /* store failed (MSR[C] set)? try again
        */

    : "=&r" (tmp)            /* Outputs   : temp variable for load
        result */
    : "r"   (&rw->lock)      /* Inputs    : lock address */
    : "cc", "memory"
  );
}

static inline int arch_read_trylock(arch_rwlock_t *rw)
{
  //eprintk("arch_read_trylock\n");
  unsigned long prev, tmp;
  __asm__ __volatile__ (
    "1: lwx   %0, %2, r0;\n" /* load conditional address in %1 to %0
        */
    "   blti  %0, 2f;\n"     /* < 0 bail, release lock */
    "   addi  %1, %0, 1;\n"  /* increment lock by 1*/
    "   swx   %1, %2, r0;\n" /* attempt store */
    "   addic %1, r0, 0;\n"  /* checking msr carry flag */
    "   bnei  %1, 1b;\n"     /* store failed (MSR[C] set)? try again
        */
    "   bri   3f;\n"         /* jump to check */
    "2: swx   %0, %3, r0;\n" /* attempt store */
    "3:"
```

```
      : "=&r" (prev), "=&r" (tmp) /* Outputs   : temp variable for load
          result */
      : "r"   (&rw->lock), "r" (&tmp)      /* Inputs    : lock address
          */
      : "cc", "memory"
   );
   return (prev >= 0 && tmp == 0);
}

static inline void arch_read_lock(arch_rwlock_t *rw)
{
   while (1) {
      if (likely(arch_read_trylock(rw)))
         break;
      do { } while (unlikely(rw->lock < 0));
   }
}

#define arch_read_can_lock(rw)      ((rw)->lock >= 0)

#define arch_read_lock_flags(lock, flags) arch_read_lock(lock)
#define arch_write_lock_flags(lock, flags) arch_write_lock(lock)

#define arch_spin_relax(lock) cpu_relax()
#define arch_read_relax(lock) cpu_relax()
#define arch_write_relax(lock)   cpu_relax()

#endif /* __KERNEL__ */
#endif /* __ASM_SPINLOCK_H */
```

C.2 atomic.h

```
#ifndef _ASM_MICROBLAZE_ATOMIC_H
#define _ASM_MICROBLAZE_ATOMIC_H

#include <linux/types.h>

#ifdef __KERNEL__

#define ATOMIC_INIT(i)  { (i) }
#include <linux/compiler.h>
#include <asm/system.h>

#define atomic_read(v)  (*(volatile int *)&(v)->counter)

static inline void atomic_set(atomic_t *v, int i)
{
   unsigned long tmp;

   __asm__ __volatile__ (
      "1: lwx   %0, %1, r0;\n" /* load conditional address in %1 to %0
          */
      "   nop;\n"              /* bubble */
      "   swx   %2, %1, r0;\n" /* attempt store */
      "   addic %0, r0, 0;\n"  /* checking msr carry flag */
      "   bnei  %0, 1b;\n"     /* store failed (MSR[C] set)? try again
          */

      : "=&r" (tmp)
      : "r"   (&v->counter), "r" (i) /* Inputs    : counter address */
      : "cc", "memory"
   );
}

static inline int atomic_add_return(int i, atomic_t *v)
{
   int result;
   unsigned long tmp;

   __asm__ __volatile__ (
      "1: lwx   %0, %2, r0;\n"  /* load conditional address in %2 to %0
          */
      "   add   %0, %0, %3;\n"  /* increment counter by i*/
      "   swx   %0, %2, r0;\n"  /* attempt store */
      "   addic %1, r0, 0;\n"   /* checking msr carry flag */
      "   bnei  %1, 1b;\n"      /* store failed (MSR[C] set)? try again
          */

      : "=&r" (result), "=&r" (tmp)  /* Outputs   : result value */
```

```
      : "r"   (&v->counter), "r" (i) /* Inputs    : counter address */
      : "cc", "memory"
   );

   return result;
}

static inline void atomic_add(int i, atomic_t *v)
{
   atomic_add_return(i,v);
}


static inline int atomic_sub_return(int i, atomic_t *v)
{
   int result;
   unsigned long tmp;

   __asm__ __volatile__ (
      "1: lwx   %0, %2, r0;\n"  /* load conditional address in %2 to %0
          */
      "   rsub  %0, %3, %0;\n"  /* decrement counter by i*/
      "   swx   %0, %2, r0;\n"  /* attempt store */
      "   addic %1, r0, 0;\n"   /* checking msr carry flag */
      "   bnei  %1, 1b;\n"      /* store failed (MSR[C] set)? try again
          */

      : "=&r" (result), "=&r" (tmp)  /* Outputs   : result value */
      : "r"   (&v->counter), "r" (i) /* Inputs    : counter address */
      : "cc", "memory"
   );

   return result;
}

static inline void atomic_sub(int i, atomic_t *v)
{
   atomic_sub_return(i,v);
}

#define atomic_cmpxchg(v, o, n) (cmpxchg(&((v)->counter), (o), (n)))
#define atomic_xchg(v, new) (xchg(&((v)->counter), new))

/**
 * atomic_add_unless - add unless the number is a given value
 * @v: pointer of type atomic_t
 * @a: the amount to add to v...
 * @u: ...unless v is equal to u.
```

```
 *
 * Atomically adds @a to @v, so long as it was not @u.
 * Returns non-zero if @v was not @u, and zero otherwise.
 */
static inline int atomic_add_unless(atomic_t *v, int a, int u)
{
   int result;
   unsigned long tmp;

   __asm__ __volatile__ (
      "1: lwx   %0, %2, r0;\n"  /* load conditional address in %2 to %0
          */
      "   cmp   %1, %0, %3;\n"  /* compare loaded value with old value*/
      "   beqid %1, 2f;\n"      /* equal to u, don't increment */
      "   add   %1, %0, %4;\n"  /* increment counter by i*/
      "   swx   %1, %2, r0;\n"  /* attempt store of new value*/
      "   addic %1, r0, 0;\n"   /* checking msr carry flag */
      "   bnei  %1, 1b;\n"      /* store failed (MSR[C] set)? try again
          */
      "   bri   3f;\n"          /* jump to check*/
      "2: swx   %0, %5, r0;\n"  /* attempt store of new value*/
      "3:"
      :"=&r" (result), "=&r" (tmp)                /* Outputs   : result
          value */
      : "r"   (&v->counter), "r" (u), "r" (a), "r" (&tmp) /* Inputs    :
           counter address, old, new */
      : "cc", "memory"
   );

   return result != u;
}

/*
 * Atomically test *v and decrement if it is greater than 0.
 * The function returns the old value of *v minus 1, even if
 * the atomic variable, v, was not decremented.
 */
static inline int atomic_dec_if_positive(atomic_t *v)
{
   int result, tmp;

   __asm__ __volatile__ (
      "1: lwx   %1, %2, r0;\n"  /* load conditional address in %2 to %1
          */
      "   blei  %1, 2f;\n"      /* if <= 0 write old value*/
      "   addi  %0, %1, -1;\n"  /* decrement counter by i*/
      "   swx   %0, %2, r0;\n"  /* attempt store of new value*/
      "   addic %1, r0, 0;\n"   /* checking msr carry flag */
```

```
    "   bnei  %1, 1b;\n"      /* store failed (MSR[C] set)? try again
        */
    "   bri   3f;\n"          /* jump to check */
    "2: swx   %1, %3, r0;\n"  /* attempt store of old value*/
    "3: "
    : "=&r" (result), "=&r" (tmp)   /* Outputs   : result value */
    : "r"   (&v->counter), "r" (&tmp)           /* Inputs    : counter
         address */
    : "cc", "memory"
  );

  return result;
}


#define atomic_inc_not_zero(v) atomic_add_unless((v), 1, 0)

#define atomic_inc(v)       atomic_add(1, v)
#define atomic_dec(v)       atomic_sub(1, v)

#define atomic_inc_and_test(v)   (atomic_add_return(1, v) == 0)
#define atomic_dec_and_test(v)   (atomic_sub_return(1, v) == 0)
#define atomic_inc_return(v)    (atomic_add_return(1, v))
#define atomic_dec_return(v)    (atomic_sub_return(1, v))
#define atomic_sub_and_test(i, v) (atomic_sub_return(i, v) == 0)

#define atomic_add_negative(i,v) (atomic_add_return(i, v) < 0)

#define smp_mb__before_atomic_dec() smp_mb()
#define smp_mb__after_atomic_dec()  smp_mb()
#define smp_mb__before_atomic_inc() smp_mb()
#define smp_mb__after_atomic_inc()  smp_mb()

#include <asm-generic/atomic64.h>
#include <asm-generic/atomic-long.h>

#endif /* __KERNEL__ */
#endif /* _ASM_MICROBLAZE_ATOMIC_H */
```

C.3 intc_timer.c

```
/*
 * Copyright (C) 2011 Eric Matthews
 * Copyright (C) 2007-2009 Michal Simek <monstr@monstr.eu>
 * Copyright (C) 2007-2009 PetaLogix
 * Copyright (C) 2006 Atmark Techno, Inc.
 *
 * This file is subject to the terms and conditions of the GNU General
    Public
 * License. See the file "COPYING" in the main directory of this archive
 * for more details.
 */

#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/param.h>
#include <linux/interrupt.h>
#include <linux/profile.h>
#include <linux/irq.h>
#include <linux/err.h>
#include <linux/clk.h>
#include <linux/clocksource.h>
#include <linux/clockchips.h>
#include <linux/delay.h>
#include <linux/sched.h>
#include <linux/spinlock.h>
#include <linux/cnt32_to_63.h>
#include <linux/io.h>
#include <linux/bug.h>
#include <linux/smp.h>
#include <asm/page.h>
#include <asm/prom.h>
#include <asm/irq.h>
#include <asm/smp.h>
#include <asm/cpuinfo.h>
#include <asm/setup.h>
#include <asm/system.h>
#include <asm/intc.h>

static unsigned int intc_baseaddr;
#define INTC_BASE intc_baseaddr

unsigned int nr_irq = 0;
unsigned int freq_div_hz;
unsigned int timer_clock_freq;

/* No one else should require these constants, so define them locally
   here. */
```

```
/* Write Interface*/
#define IPI_WADDR 0x00
#define SIE_WADDR 0x04
#define CIE_WADDR 0x08
#define MER_WADDR 0x0C
#define GLB_TIMER_SNAPSHOT_WADDR 0x10

#define NUM_BASE_REGS 8

#define PER_CPU_CIE_WADDR 0x20
#define PER_CPU_SIE_WADDR 0x24
#define PER_CPU_MER_WADDR 0x28
#define PER_CPU_IAR_WADDR 0x2C
#define PER_CPU_TIMER_CFG_WADDR 0x30
#define PER_CPU_TIMER_LOAD_WADDR 0x34

#define NUM_PER_CPU_REGS 8

/* Read Interface */
#define GLB_TIMER_RADDR_L 0x0
#define GLB_TIMER_RADDR_H 0x4
#define PER_CPU_IVR_RADDR 0x8
#define PER_CPU_TIMER_RADDR 0xC

#define PER_CPU_READ_REGS 2

#define NUM_IPI 4
#define TIMER_IRQ NUM_IPI


#define PER_CPU_REG(cpu, reg) (INTC_BASE + reg + (cpu*NUM_PER_CPU_REGS
   *4))

#define PER_CPU_READ_REG(cpu, reg) (INTC_BASE + reg + (cpu*
   PER_CPU_READ_REGS*4))

static DEFINE_PER_CPU(struct clock_event_device, ltime_events);

static DEFINE_RAW_SPINLOCK(rw_lock);

static int timer_initialized;


void send_ipi(int ipi_number, const struct cpumask *cpu_mask)
{
   //unsigned long cpu_id = smp_processor_id();
   //eprintk("CPU: %d sends IPI: %d to cpu mask: %x\n", cpu_id,
      ipi_number, cpumask_bits(cpu_mask)[0]);
```

```
    unsigned int ipi_shift = 0x80000000;

    out_be32(INTC_BASE + IPI_WADDR, ((ipi_shift >> ipi_number) |
        cpumask_bits(cpu_mask)[0]));
}


void intc_enable_or_unmask(unsigned int irq)
{
    unsigned long cpu = smp_processor_id();
    unsigned long mask = 1 << irq;
    int i;
    printk("cpu: %d enable_or_unmask: %d\n", cpu, irq);

    out_be32(PER_CPU_REG(cpu,PER_CPU_IAR_WADDR), mask);
    out_be32(PER_CPU_REG(cpu,PER_CPU_SIE_WADDR), mask);


    for_each_online_cpu(i) {
        out_be32(PER_CPU_REG(i,PER_CPU_SIE_WADDR), mask);
    }

    out_be32(INTC_BASE + SIE_WADDR, mask);
}

void intc_disable_or_mask(unsigned int irq)
{
    unsigned long mask = 1 << irq;
    unsigned long cpu = smp_processor_id();
    printk("cpu: %d disable: %d\n", cpu, irq);

    out_be32(INTC_BASE + CIE_WADDR, mask);
    out_be32(PER_CPU_REG(cpu,PER_CPU_CIE_WADDR), mask);
}

void end_interrupt(unsigned int irq)
{
    unsigned long cpu = smp_processor_id();
    unsigned long mask = 1 << irq;

    out_be32(PER_CPU_REG(cpu,PER_CPU_IAR_WADDR), mask);
}


void enable_ipis(int cpu)
{
    int i;
    unsigned long mask;
```

```
   for(i=0; i < NUM_IPI; i++) {
      mask = 1 << i;
      out_be32(PER_CPU_REG(cpu,PER_CPU_IAR_WADDR), mask);
      out_be32(PER_CPU_REG(cpu,PER_CPU_SIE_WADDR), mask);
      out_be32(INTC_BASE + SIE_WADDR, mask);
   }
   out_be32(PER_CPU_REG(cpu,PER_CPU_MER_WADDR), 0x00000001);
}

void enable_timer_interrupt(int cpu)
{
   unsigned long mask = 1 << TIMER_IRQ;

   out_be32(PER_CPU_REG(cpu,PER_CPU_IAR_WADDR), mask);
   out_be32(INTC_BASE + SIE_WADDR, mask);
   out_be32(PER_CPU_REG(cpu,PER_CPU_SIE_WADDR), mask);
}


unsigned int get_irq(struct pt_regs *regs)
{
   int irq;
   unsigned long cpu = smp_processor_id();


   irq = in_be32(PER_CPU_READ_REG(cpu, PER_CPU_IVR_RADDR));

   pr_debug("get_irq: %d\n", irq);
   return irq;
}


static inline void microblaze_decrementer_stop(void)
{
   int cpu = smp_processor_id();
   out_be32(PER_CPU_REG(cpu,PER_CPU_TIMER_CFG_WADDR), 0x0);

}

static inline void microblaze_decrementer_start_periodic(unsigned long
    load_val)
{
   int cpu = smp_processor_id();

   if (!load_val)
      load_val = 1;
```

```
	out_be32(PER_CPU_REG(cpu,PER_CPU_TIMER_LOAD_WADDR), load_val); /*
	    loading value to timer reg */

	/* Load timer & set auto reload & set down count & enable interrupt &
	     enable*/
	out_be32(PER_CPU_REG(cpu,PER_CPU_TIMER_CFG_WADDR), T_ENABLE | T_LOAD
	    | T_AUTO_RELOAD | T_GEN_INT);

}

static int microblaze_timer_set_next_event(unsigned long delta, struct
    clock_event_device *dev)
{
	int cpu = smp_processor_id();

	if (!delta)
		delta = 1;

	pr_debug("%s: next event, delta %x\n", __func__, (u32)delta);


	out_be32(PER_CPU_REG(cpu,PER_CPU_TIMER_LOAD_WADDR), delta); /*
	    loading value to timer reg */

	/* Load timer & set auto reload & set down count & enable interrupt &
	     enable*/
	out_be32(PER_CPU_REG(cpu,PER_CPU_TIMER_CFG_WADDR), T_ENABLE | T_LOAD
	    | T_GEN_INT);


	return 0;
}

static void microblaze_timer_set_mode(enum clock_event_mode mode, struct
     clock_event_device *evt)
{
	int cpu = smp_processor_id();

	switch (mode) {
	case CLOCK_EVT_MODE_PERIODIC:
		printk(KERN_INFO "CPU: %d, %s: periodic\n", cpu, __func__);
		microblaze_decrementer_start_periodic(freq_div_hz);
		break;
	case CLOCK_EVT_MODE_ONESHOT:
		printk(KERN_INFO "CPU: %d, %s: oneshot\n", cpu,  __func__);
		break;
	case CLOCK_EVT_MODE_UNUSED:
		printk(KERN_INFO "%s: unused\n", __func__);
```

```
      break;
   case CLOCK_EVT_MODE_SHUTDOWN:
      printk(KERN_INFO "CPU: %d, %s: shutdown\n", cpu,  __func__);
      microblaze_decrementer_stop();
      break;
   case CLOCK_EVT_MODE_RESUME:
      printk(KERN_INFO "%s: resume\n", __func__);
      break;
   }
}

static struct clock_event_device clockevent_microblaze_timer = {
   .name           = "microblaze_clockevent",
   .features       = CLOCK_EVT_FEAT_ONESHOT | CLOCK_EVT_FEAT_PERIODIC,
   .shift          = 8,
   .rating         = 300,
   .set_next_event = microblaze_timer_set_next_event,
   .set_mode       = microblaze_timer_set_mode,
};


void __devinit microblaze_setup_local_timer(void)
{
   struct clock_event_device *evt = &__get_cpu_var(ltime_events);

   memcpy(evt, &clockevent_microblaze_timer, sizeof(*evt));
   evt->cpumask = cpumask_of(smp_processor_id());

   clockevents_register_device(evt);
   enable_timer_interrupt(smp_processor_id());
}

static irqreturn_t timer_interrupt(int irq, void *dev_id)
{
   struct clock_event_device *evt = &__get_cpu_var(ltime_events);

#ifdef CONFIG_HEART_BEAT
   heartbeat();
#endif
   evt->event_handler(evt);
   return IRQ_HANDLED;
}


static __init void microblaze_clockevent_init(void)
{
   clockevent_microblaze_timer.mult = div_sc(timer_clock_freq,
      NSEC_PER_SEC, clockevent_microblaze_timer.shift);
```

```
    clockevent_microblaze_timer.max_delta_ns = clockevent_delta2ns((u32)
        ~0, &clockevent_microblaze_timer);
    clockevent_microblaze_timer.min_delta_ns = clockevent_delta2ns(1, &
        clockevent_microblaze_timer);

    microblaze_setup_local_timer();

}

static DEFINE_SPINLOCK(timebase_lock);
static cycle_t microblaze_read(struct clocksource *cs)
{
    u64 a, b;
    cycle_t c;
    unsigned long flags;

    spin_lock_irqsave(&timebase_lock, flags);
        out_be32(INTC_BASE + GLB_TIMER_SNAPSHOT_WADDR, 0x0);
        a = in_be32(INTC_BASE + GLB_TIMER_RADDR_L);
        b = in_be32(INTC_BASE + GLB_TIMER_RADDR_H);
        c = (cycle_t)(a | (b << 32));
    spin_unlock_irqrestore(&timebase_lock, flags);

    //eprintk("t: %llu\n", c);

    return c;
}


static struct timecounter microblaze_tc = {
    .cc = NULL,
};

static cycle_t microblaze_cc_read(const struct cyclecounter *cc)
{
    return microblaze_read(NULL);
}


static struct cyclecounter microblaze_cc = {
    .read = microblaze_cc_read,
    .mask = CLOCKSOURCE_MASK(64),
    .shift = 8,
};

static struct irqaction timer_irqaction = {
    .handler = timer_interrupt,
    .flags = IRQF_DISABLED | IRQF_PERCPU | IRQF_TIMER,
```

```
    .name = "timer",
    .dev_id = &ltime_events,
};

static struct clocksource clocksource_microblaze = {
    .name       = "microblaze_clocksource",
    .rating     = 300,
    .read       = microblaze_read,
    .mask       = CLOCKSOURCE_MASK(64),
    .shift      = 8,
    .flags      = CLOCK_SOURCE_IS_CONTINUOUS,
};

static struct irq_chip intc_dev = {
    .name     = "Timer and Interrupt Controller",
    .unmask   = intc_enable_or_unmask,
    .mask     = intc_disable_or_mask,
    .eoi      = end_interrupt,
};


int __init init_microblaze_timecounter(void)
{
    microblaze_cc.mult = div_sc(timer_clock_freq, NSEC_PER_SEC,
        microblaze_cc.shift);
    //eprintk("\n\nmult2: %d\n\n",microblaze_cc.mult);
    timecounter_init(&microblaze_tc, &microblaze_cc, sched_clock());

    return 0;
}


static int __init microblaze_clocksource_init(void)
{
    clocksource_microblaze.mult = clocksource_hz2mult(timer_clock_freq,
        clocksource_microblaze.shift);
    //eprintk("\n\nmult: %d\n\n",clocksource_microblaze.mult);
    if (clocksource_register(&clocksource_microblaze))
       panic("failed to register clocksource");

    /* register timecounter - for ftrace support */
    init_microblaze_timecounter();
    return 0;
}

unsigned long long notrace sched_clock(void)
{
    if (timer_initialized) {
```

```
        struct clocksource *cs = &clocksource_microblaze;
        return clocksource_cyc2ns(cs->read(NULL), cs->mult, cs->shift);
    }
    return 0;
}

/*
 * We have to protect accesses before timer initialization
 * and return 0 for sched_clock function below.
 */
void __init time_init(void)
{
#ifdef CONFIG_HEART_BEAT
    setup_heartbeat();
#endif
    microblaze_clocksource_init();
    microblaze_clockevent_init();

    setup_irq(TIMER_IRQ, &timer_irqaction);

    timer_initialized = 1;
}

void __init init_IRQ(void)
{
    u32 i, j, intr_type;
    struct device_node *intc = NULL;
    unsigned long cpu = smp_processor_id();
    const void *prop;
    const char * const intc_list[] = {
                "xlnx,xps-intc-1.00.a",
                NULL
            };

    for (j = 0; intc_list[j] != NULL; j++) {
        intc = of_find_compatible_node(NULL, NULL, intc_list[j]);
        if (intc)
            break;
    }
    BUG_ON(!intc);

    intc_baseaddr = be32_to_cpup(of_get_property(intc,
                            "reg", NULL));
    intc_baseaddr = (unsigned long) ioremap(intc_baseaddr, PAGE_SIZE);
    nr_irq = be32_to_cpup(of_get_property(intc,
                    "xlnx,num-intr-inputs", NULL));

    intr_type =
```

```
   be32_to_cpup(of_get_property(intc,
                "xlnx,kind-of-intr", NULL));
if (intr_type >= (1 << (nr_irq + 1)))
   printk(KERN_INFO " ERROR: Mismatch in kind-of-intr param\n");


/* If there is clock-frequency property than use it */
prop = of_get_property(intc, "clock-frequency", NULL);
if (prop)
   timer_clock_freq = be32_to_cpup(prop);
else
   timer_clock_freq = per_cpu(cpu_info,0).cpu_clock_freq;

freq_div_hz = timer_clock_freq / HZ;

printk(KERN_INFO "%s #0 at 0x%08x, num_irq=%d, edge=0x%x\n",
   intc_list[j], intc_baseaddr, nr_irq, intr_type);

/*
 * Disable all external interrupts until they are
 * explicity requested.
 */

out_be32(intc_baseaddr + CIE_WADDR, 0xFFFFFFFF);
out_be32(PER_CPU_REG(cpu,PER_CPU_CIE_WADDR), 0xFFFFFFFF);

/* Acknowledge any pending interrupts just in case. */
out_be32(PER_CPU_REG(cpu,PER_CPU_IAR_WADDR), 0xffffffff);

/* Turn on the Master Enable. */
out_be32(intc_baseaddr + MER_WADDR, 0x00000001);
out_be32(PER_CPU_REG(cpu,PER_CPU_MER_WADDR), 0x00000001);


for (i = 0; i < nr_irq; i++) {
   set_irq_chip_and_handler_name(i, &intc_dev, handle_percpu_irq,
       intc_dev.name);
   irq_desc[i].status |= IRQ_PER_CPU;
   eprintk("irq: %d of type: IRQ_PER_CPU , name: %s\n", i, intc_dev.
       name);
}

printk(KERN_INFO "requesting IPIs\n");
for (i = 0; i <  MICROBLAZE_NUM_IPIS; i++) {
   smp_request_message_ipi(i, i);
}

enable_ipis(cpu);
```

```
}
```

C.4 smp.c

```
/*
 * SMP support for MicroBlaze, borrowing a great
 * deal of code from the PowerPC implementation
 *
 *      This program is free software; you can redistribute it and/or
 *      modify it under the terms of the GNU General Public License
 *      as published by the Free Software Foundation; either version
 *      2 of the License, or (at your option) any later version.
 */

#undef DEBUG

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/smp.h>
#include <linux/interrupt.h>
#include <linux/delay.h>
#include <linux/init.h>
#include <linux/spinlock.h>
#include <linux/cache.h>
#include <linux/err.h>
#include <linux/sysdev.h>
#include <linux/cpu.h>
#include <linux/notifier.h>
#include <linux/topology.h>

#include <asm/ptrace.h>
#include <asm/atomic.h>
#include <asm/irq.h>
#include <asm/page.h>
#include <asm/pgtable.h>
#include <asm/prom.h>
#include <asm/smp.h>
#include <asm/cputable.h>
#include <asm/system.h>
#include <asm/intc.h>
#include <asm/pvr.h>
#include <asm/cpuinfo.h>
#include <asm/sections.h>
#include <asm/tlbflush.h>

struct thread_info *secondary_ti;

DEFINE_PER_CPU(cpumask_var_t, cpu_core_map);
EXPORT_PER_CPU_SYMBOL(cpu_core_map);
```

```
volatile unsigned int cpu_callin_map[NR_CPUS];


static irqreturn_t call_function_action(int irq, void *data)
{
   //eprintk("CPU %d received call many\n",smp_processor_id());
   generic_smp_call_function_interrupt();
   return IRQ_HANDLED;
}

static irqreturn_t reschedule_action(int irq, void *data)
{
   /* we just need the return path side effect of checking need_resched
       */
   return IRQ_HANDLED;
}

static irqreturn_t call_function_single_action(int irq, void *data)
{
   generic_smp_call_function_single_interrupt();
   return IRQ_HANDLED;
}

#define NUM_IPI 3
static irq_handler_t smp_ipi_action[] = {
   [MICROBLAZE_MSG_RESCHEDULE]            = reschedule_action,
   [MICROBLAZE_MSG_CALL_FUNCTION]         = call_function_action,
   [MICROBLAZE_MSG_CALL_FUNCTION_SINGLE] = call_function_single_action,
};

const char *smp_ipi_name[] = {
   [MICROBLAZE_MSG_RESCHEDULE]            = "ipi reschedule",
   [MICROBLAZE_MSG_CALL_FUNCTION]         = "ipi call function",
   [MICROBLAZE_MSG_CALL_FUNCTION_SINGLE] = "ipi call function single",
};


int smp_request_message_ipi(int irq, int msg)
{
   int err;

   if (msg < 0 || msg >= MICROBLAZE_NUM_IPIS) {
      return -EINVAL;
   }

   err = request_irq(irq, smp_ipi_action[msg], IRQF_DISABLED|IRQF_PERCPU
       , smp_ipi_name[msg], 0);
```

```
   WARN(err < 0, "unable to request_irq %d for %s (rc %d)\n",irq,
       smp_ipi_name[msg], err);

   return err;
}


void smp_send_reschedule(int cpu)
{
   //eprintk("CPU %d reschedule on: %d\n",smp_processor_id(), cpu);
   if(cpu_online(cpu))
      send_ipi(MICROBLAZE_MSG_RESCHEDULE, cpumask_of(cpu));
}

void arch_send_call_function_single_ipi(int cpu)
{
   //eprintk("CPU %d call single on: %d\n",smp_processor_id(), cpu);
   if(cpu_online(cpu))
      send_ipi(MICROBLAZE_MSG_CALL_FUNCTION_SINGLE, cpumask_of(cpu));
}

void arch_send_call_function_ipi_mask(const struct cpumask *mask)
{
   //eprintk("CPU %d call many on: %x\n",smp_processor_id(),
       cpumask_bits(mask)[0]);
   send_ipi(MICROBLAZE_MSG_CALL_FUNCTION, mask);
}


static void stop_this_cpu(void *dummy)
{
   /* Remove this CPU */
   set_cpu_online(smp_processor_id(), false);

   local_irq_disable();
   while (1)
      ;
}

void smp_send_stop(void)
{
   smp_call_function(stop_this_cpu, NULL, 0);
}

struct thread_info *current_set[NR_CPUS];

static void __devinit smp_store_cpu_info(int id)
{
```

```
    //per_cpu(cpu_pvr, id) = get_single_pvr(0, pvr0);
}

static void __init smp_create_idle(unsigned int cpu)
{
    struct task_struct *p;

    /* create a process for the processor */
    p = fork_idle(cpu);
    if (IS_ERR(p)) {
        panic("failed fork for CPU %u: %li", cpu, PTR_ERR(p));
        printk(KERN_ALERT "failed to create cpu %d idle\n", cpu);
    }


    current_set[cpu] = task_thread_info(p);
    task_thread_info(p)->cpu = cpu;
}

void __init smp_prepare_cpus(unsigned int max_cpus)
{

    unsigned int cpu;
    /*
     * setup_cpu may need to be called on the boot cpu. We havent
     * spun any cpus up but lets be paranoid.
     */
    BUG_ON(boot_cpuid != smp_processor_id());

    /* Fixup boot cpu */
    cpu_callin_map[boot_cpuid] = 1;

    for_each_possible_cpu(cpu) {
        zalloc_cpumask_var_node(&per_cpu(cpu_core_map, cpu),
                  GFP_KERNEL, cpu_to_node(cpu));
    }

    cpumask_set_cpu(boot_cpuid, cpu_core_mask(boot_cpuid));

    max_cpus = NR_CPUS;

    for_each_possible_cpu(cpu) {
        if (cpu != boot_cpuid) {
            smp_create_idle(cpu);
        }
    }
}
```

```
void __devinit smp_prepare_boot_cpu(void)
{
   BUG_ON(smp_processor_id() != boot_cpuid);
   current_set[boot_cpuid] = task_thread_info(current);
}

int __cpuinit __cpu_up(unsigned int cpu)
{
   int c;

   secondary_ti = current_set[cpu];

   /* Make sure callin-map entry is 0 (can be leftover a CPU
    * hotplug
    */
   cpu_callin_map[cpu] = 0;

   /* The information for processor bringup must
    * be written out to main store before we release
    * the processor.
    */
   smp_mb();

   /* wake up cpus */
   printk(KERN_ALERT "Waking CPU %d\n", cpu);
   //printk(KERN_ALERT ": %d: current %x\n", cpu, current);
   //printk(KERN_ALERT "mem_loc: %x\n",current_thread_info());
   //printk(KERN_ALERT "mem_loc_sec_ti: %x\n",secondary_ti);
   //printk(KERN_ALERT "mem_loc_sec_ti_task: %x\n",secondary_ti->task);

   if(cpu < 4)
      send_ipi(0, cpumask_of(cpu));
   else
      return -ENOENT;

   if (system_state < SYSTEM_RUNNING)
      for (c = 5000; c && !cpu_callin_map[cpu]; c--)
         udelay(100);


   if (!cpu_callin_map[cpu]) {
      printk(KERN_ERR "Processor %u is stuck.\n", cpu);
      return -ENOENT;
   }

   while (!cpu_online(cpu))
      cpu_relax();
```

```
   printk(KERN_ALERT "Processor %u found.\n", cpu);

   return 0;
}

asmlinkage void __devinit secondary_machine_init(void)
{
   unsigned long *src, *dst;
   unsigned int offset = 0;
   /* Do not copy reset vectors. offset = 0x2 means skip the first
    * two instructions. dst is pointer to MB vectors which are placed
    * in block ram. If you want to copy reset vector setup offset to 0x0
         */
#if !CONFIG_MANUAL_RESET_VECTOR
   offset = 0x2;
#endif
   dst = (unsigned long *) (offset * sizeof(u32));
   for (src = __ivt_start + offset; src < __ivt_end; src++, dst++)
      *dst = *src;
}



/* Activate a secondary processor. */
int __devinit start_secondary(void)
{
   unsigned int cpu = smp_processor_id();
   int i;

   atomic_inc(&init_mm.mm_count);
   current->active_mm = &init_mm;
   cpumask_set_cpu(cpu, mm_cpumask(&init_mm));
   local_flush_tlb_mm(&init_mm);

   //eprintk(KERN_ALERT "cpu ID alivE: %d: current %x\n", cpu, current);

   setup_cpuinfo();
   microblaze_cache_init();

   preempt_disable();
   cpu_callin_map[cpu] = 1;

   ipi_call_lock();
   notify_cpu_starting(cpu);
   set_cpu_online(cpu, true);

   for_each_online_cpu(i) {
      cpumask_set_cpu(cpu, cpu_core_mask(i));
```

```
        cpumask_set_cpu(i, cpu_core_mask(cpu));
    }
    enable_ipis(cpu);
    microblaze_setup_local_timer();
    ipi_call_unlock();

    local_irq_enable();

    cpu_idle();
    return 0;
}

int setup_profiling_timer(unsigned int multiplier)
{
    return 0;
}

void __init smp_cpus_done(unsigned int max_cpus)
{ }
```

C.5 head.S

```
/*
 * Copyright (C) 2011 Eric Matthews
 * Copyright (C) 2007-2009 Michal Simek <monstr@monstr.eu>
 * Copyright (C) 2007-2009 PetaLogix
 * Copyright (C) 2006 Atmark Techno, Inc.
 *
 * MMU code derived from arch/ppc/kernel/head_4xx.S:
 *    Copyright (c) 1995-1996 Gary Thomas <gdt@linuxppc.org>
 *      Initial PowerPC version.
 *    Copyright (c) 1996 Cort Dougan <cort@cs.nmt.edu>
 *      Rewritten for PReP
 *    Copyright (c) 1996 Paul Mackerras <paulus@cs.anu.edu.au>
 *      Low-level exception handers, MMU support, and rewrite.
 *    Copyright (c) 1997 Dan Malek <dmalek@jlc.net>
 *      PowerPC 8xx modifications.
 *    Copyright (c) 1998-1999 TiVo, Inc.
 *      PowerPC 403GCX modifications.
 *    Copyright (c) 1999 Grant Erickson <grant@lcse.umn.edu>
 *      PowerPC 403GCX/405GP modifications.
 *    Copyright 2000 MontaVista Software Inc.
 * PPC405 modifications
 *      PowerPC 403GCX/405GP modifications.
 *    Author: MontaVista Software, Inc.
 *          frank_rowand@mvista.com or source@mvista.com
 *          debbie_chu@mvista.com
 *
```

```
 * This file is subject to the terms and conditions of the GNU General
    Public
 * License. See the file "COPYING" in the main directory of this archive
 * for more details.
 */

#include <linux/init.h>
#include <linux/linkage.h>
#include <asm/thread_info.h>
#include <asm/page.h>
#include <linux/of_fdt.h>     /* for OF_DT_HEADER */

#ifdef CONFIG_MMU
#include <asm/setup.h> /* COMMAND_LINE_SIZE */
#include <asm/mmu.h>
#include <asm/processor.h>
#include <asm/asm-offsets.h>

#define SAVE_ENTRY_SP_R1 \
   .word 0b10010100000000011100100000000000; nop;

#define RESTORE_ENTRY_SP_R1 \
   .word 0b10010100001000001000100000000000; nop;

#define SAVE_ENTRY_SP_R11 \
   .word 0b10010100000010111100100000000000; nop;

#define RESTORE_ENTRY_SP_R11 \
   .word 0b10010101011000001000100000000000; nop;

#define SAVE_ENTRY_SP_R31 \
   .word 0b10010100000111111100100000000000; nop;

#define RESTORE_ENTRY_SP_R31 \
   .word 0b10010111111000001000100000000000; nop;

#define SAVE_CURRENT_SAVE_R1 \
   .word 0b10010100000000011100100000000001; nop;

#define RESTORE_CURRENT_SAVE_R1 \
   .word 0b10010100001000001000100000000001; nop;

#define SAVE_CURRENT_SAVE_R31 \
   .word 0b10010100000111111100100000000001; nop;

#define RESTORE_CURRENT_SAVE_R31 \
   .word 0b10010111111000001000100000000001; nop;
```

```
#define SAVE_ENTRY_SP(reg) \
   mts   gspr0, reg;

#define RESTORE_ENTRY_SP(reg) \
   mfs   reg, gspr0;

#define SAVE_CURRENT_SAVE(reg) \
   mts   gspr1, reg;

#define RESTORE_CURRENT_SAVE(reg) \
   mfs   reg, gspr1;

.section .data
.global empty_zero_page
.align 12
empty_zero_page:
   .space   PAGE_SIZE
.global swapper_pg_dir
swapper_pg_dir:
   .space   PAGE_SIZE
#ifdef CONFIG_SMP
temp_boot_stack:
   .space   1024
#endif /* CONFIG_SMP */
#endif /* CONFIG_MMU */

.section .rodata
.align 4
endian_check:
   .word 1



   __HEAD
ENTRY(_start)
#if CONFIG_KERNEL_BASE_ADDR == 0
   brai  TOPHYS(real_start)
   .org  0x100
real_start:
#endif

   mfs   r1, rmsr
   andi  r1, r1, ~2
   mts   rmsr, r1
/*
 * According to Xilinx, msrclr instruction behaves like 'mfs rX,rpc'
 * if the msrclr instruction is not enabled. We use this to detect
```

```
 * if the opcode is available, by issuing msrclr and then testing the
    result.
 * r8 == 0 - msr instructions are implemented
 * r8 != 0 - msr instructions are not implemented
 */
   msrclr   r8, 0 /* clear nothing - just read msr for test */
   cmpu  r8, r8, r1 /* r1 must contain msr reg content */


   /* skip FDT copy if secondary */
   mfs   r11, rpvr00
   andi  r11, r11, 0xFF
   bnei  r11, _setup_initial_mmu


/* r7 may point to an FDT, or there may be one linked in.
   if it's in r7, we've got to save it away ASAP.
   We ensure r7 points to a valid FDT, just in case the bootloader
   is broken or non-existent */
   beqi  r7, no_fdt_arg       /* NULL pointer?  don't copy */
/* Does r7 point to a valid FDT? Load HEADER magic number */
   /* Run time Big/Little endian platform */
   /* Save 1 as word and load byte - 0 - BIG, 1 - LITTLE */
   lbui  r11, r0, TOPHYS(endian_check)
   beqid r11, big_endian /* DO NOT break delay stop dependency */
   lw r11, r0, r7 /* Big endian load in delay slot */
   lwr   r11, r0, r7 /* Little endian load */
big_endian:
   rsubi r11, r11, OF_DT_HEADER  /* Check FDT header */
   beqi  r11, _prepare_copy_fdt
   or r7, r0, r0     /* clear R7 when not valid DTB */
   bnei  r11, no_fdt_arg         /* No - get out of here */
_prepare_copy_fdt:
   or r11, r0, r0 /* incremment */
   ori   r4, r0, TOPHYS(_fdt_start)
   ori   r3, r0, (0x4000 - 4)
_copy_fdt:
   lw r12, r7, r11 /* r12 = r7 + r11 */
   sw r12, r4, r11 /* addr[r4 + r11] = r12 */
   addik r11, r11, 4 /* increment counting */
   bgtid r3, _copy_fdt /* loop for all entries */
   addik r3, r3, -4 /* descrement loop */
no_fdt_arg:

#ifdef CONFIG_MMU

#ifndef CONFIG_CMDLINE_BOOL
/*
```

```
 * handling command line
 * copy command line to __init_end. There is space for storing command
   line.
 */
   or r6, r0, r0     /* incremment */
   ori   r4, r0, __init_end   /* load address of command line */
   tophys(r4,r4)        /* convert to phys address */
   ori   r3, r0, COMMAND_LINE_SIZE - 1 /* number of loops */
_copy_command_line:
   lbu   r2, r5, r6 /* r2=r5+r6 - r5 contain pointer to command line */
   sb r2, r4, r6     /* addr[r4+r6]= r2*/
   addik r6, r6, 1      /* increment counting */
   bgtid r3, _copy_command_line  /* loop for all entries       */
   addik r3, r3, -1     /* descrement loop */
   addik r5, r4, 0      /* add new space for command line */
   tovirt(r5,r5)
#endif /* CONFIG_CMDLINE_BOOL */

#ifdef NOT_COMPILE
/* save bram context */
   or r6, r0, r0           /* incremment */
   ori   r4, r0, TOPHYS(_bram_load_start) /* save bram context */
   ori   r3, r0, (LMB_SIZE - 4)
_copy_bram:
   lw r7, r0, r6     /* r7 = r0 + r6 */
   sw r7, r4, r6     /* addr[r4 + r6] = r7*/
   addik r6, r6, 4      /* increment counting */
   bgtid r3, _copy_bram    /* loop for all entries */
   addik r3, r3, -4     /* descrement loop */
#endif
   /* We have to turn on the MMU right away. */

_setup_initial_mmu:
   /*
    * Set up the initial MMU state so we can do the first level of
    * kernel initialization.  This maps the first 16 MBytes of memory
       1:1
    * virtual to physical.
    */
   nop
   addik r3, r0, MICROBLAZE_TLB_SIZE -1   /* Invalidate all TLB entries
      */
_invalidate:
   mts   rtlbx, r3
   mts   rtlbhi, r0        /* flush: ensure V is clear   */
   bgtid r3, _invalidate      /* loop for all entries       */
   addik r3, r3, -1
   /* sync */
```

```
/* Setup the kernel PID */
mts   rpid,r0        /* Load the kernel PID */
nop
bri   4

/*
 * We should still be executing code at physical address area
 * RAM_BASEADDR at this point. However, kernel code is at
 * a virtual address. So, set up a TLB mapping to cover this once
 * translation is enabled.
 */

addik r3,r0, CONFIG_KERNEL_START /* Load the kernel virtual address
   */
tophys(r4,r3)        /* Load the kernel physical address */

/* start to do TLB calculation */
addik r12, r0, _end
rsub  r12, r3, r12
addik r12, r12, CONFIG_KERNEL_PAD /* that's the pad */

or r9, r0, r0 /* TLB0 = 0 */
or r10, r0, r0 /* TLB1 = 0 */

addik r11, r12, -0x1000000
bgei  r11, GT16 /* size is greater than 16MB */
addik r11, r12, -0x0800000
bgei  r11, GT8 /* size is greater than 8MB */
addik r11, r12, -0x0400000
bgei  r11, GT4 /* size is greater than 4MB */
/* size is less than 4MB */
addik r11, r12, -0x0200000
bgei  r11, GT2 /* size is greater than 2MB */
addik r9, r0, 0x0100000 /* TLB0 must be 1MB */
addik r11, r12, -0x0100000
bgei  r11, GT1 /* size is greater than 1MB */
/* TLB1 is 0 which is setup above */
bri tlb_end
GT4: /* r11 contains the rest - will be either 1 or 4 */
ori r9, r0, 0x400000 /* TLB0 is 4MB */
bri TLB1
GT16: /* TLB0 is 16MB */
addik r9, r0, 0x1000000 /* means TLB0 is 16MB */
TLB1:
addik r2, r11, -0x0400000 /* must be used r2 because of substract if
   failed */
bgei  r2, GT20 /* size is greater than 16MB */
```

```
   /* size is >16MB and <20MB */
   addik r11, r11, -0x0100000
   bgei  r11, GT17 /* size is greater than 17MB */
   /* kernel is >16MB and < 17MB */
GT1:
   addik r10, r0, 0x0100000 /* means TLB1 is 1MB */
   bri tlb_end
GT2: /* TLB0 is 0 and TLB1 will be 4MB */
GT17: /* TLB1 is 4MB - kernel size <20MB */
   addik r10, r0, 0x0400000 /* means TLB1 is 4MB */
   bri tlb_end
GT8: /* TLB0 is still zero that's why I can use only TLB1 */
GT20: /* TLB1 is 16MB - kernel size >20MB */
   addik r10, r0, 0x1000000 /* means TLB1 is 16MB */
tlb_end:

   /*
    * Configure and load two entries into TLB slots 0 and 1.
    * In case we are pinning TLBs, these are reserved in by the
    * other TLB functions.  If not reserving, then it doesn't
    * matter where they are loaded.
    */
   andi  r4,r4,0xfffffc00  /* Mask off the real page number */
   ori   r4,r4,(TLB_WR | TLB_EX) /* Set the write and execute bits */

   beqi  r9, jump_over /* TLB0 can be zeroes that's why we not setup it
      */

   /* look at the code below */
   ori   r30, r0, 0x200
   andi  r29, r9, 0x100000
   bneid r29, 1f
   addik r30, r30, 0x80
   andi  r29, r9, 0x400000
   bneid r29, 1f
   addik r30, r30, 0x80
   andi  r29, r9, 0x1000000
   bneid r29, 1f
   addik r30, r30, 0x80
1:
   ori r11, r30, 0

   andi  r3,r3,0xfffffc00  /* Mask off the effective page number */
   ori   r3,r3,(TLB_VALID)
   or r3, r3, r11

   mts     rtlbx,r0     /* TLB slow 0 */
```

```
   mts   rtlblo,r4      /* Load the data portion of the entry */
   mts   rtlbhi,r3      /* Load the tag portion of the entry */

jump_over:

   beqi  r10, jump_over2 /* TLB0 can be zeroes that's why we not setup
      it */

   /* look at the code below */
   ori   r30, r0, 0x200
   andi  r29, r10, 0x100000
   bneid r29, 1f
   addik r30, r30, 0x80
   andi  r29, r10, 0x400000
   bneid r29, 1f
   addik r30, r30, 0x80
   andi  r29, r10, 0x1000000
   bneid r29, 1f
   addik r30, r30, 0x80
1:
   ori r12, r30, 0

   addk  r4, r4, r9  /* previous addr + TLB0 size */
   addk  r3, r3, r9

   andi  r3,r3,0xfffffc00  /* Mask off the effective page number */
   ori   r3,r3,(TLB_VALID)
   or r3, r3, r12

   ori   r6,r0,1        /* TLB slot 1 */
   mts     rtlbx,r6

   mts   rtlblo,r4      /* Load the data portion of the entry */
   mts   rtlbhi,r3      /* Load the tag portion of the entry */

jump_over2:
   /*
    * Load a TLB entry for LMB, since we need access to
    * the exception vectors, using a 4k real==virtual mapping.
    */
   ori   r6,r0,3        /* TLB slot 3 */
   mts     rtlbx,r6

   ori   r4,r0,(TLB_WR | TLB_EX)
   ori   r3,r0,(TLB_VALID | TLB_PAGESZ(PAGESZ_4K))

   mts   rtlblo,r4      /* Load the data portion of the entry */
   mts   rtlbhi,r3      /* Load the tag portion of the entry */
```

```
	/*
	 * We now have the lower 16 Meg of RAM mapped into TLB entries, and
	    the
	 * caches ready to work.
	 */
turn_on_mmu:
	ori	r15,r0,start_here
	/*
	 * Read PVR and mask off all but CPU id bits to use to select boot
	    sequence
	 */
#ifdef CONFIG_SMP
	mfs	r4, rpvr00
	andi	r4, r4, 0xFF

	beqi	r4, finish
	ori	r15,r0,start_secondary_cpu
finish:
#endif /* CONFIG_SMP */
	ori	r4,r0,MSR_KERNEL_VMS
	mts	rmsr,r4
	nop
	rted	r15,0		/* enables MMU */
	nop

start_here:
#endif /* CONFIG_MMU */

	/* Initialize small data anchors */
	la r13, r0, _KERNEL_SDA_BASE_
	la r2, r0, _KERNEL_SDA2_BASE_

	/* Initialize stack pointer */
	la r1, r0, init_thread_union + THREAD_SIZE - 4
	SAVE_ENTRY_SP_R1

	/* Initialize r31 with current task address */
	la r31, r0, init_task
	SAVE_CURRENT_SAVE_R31
	/*
	 * Call platform dependent initialize function.
	 * Please see $(ARCH)/mach-$(SUBARCH)/setup.c for
	 * the function.
	 */
	la r11, r0, machine_early_init
	brald r15, r11
	nop
```

```
#ifndef CONFIG_MMU
   la r15, r0, machine_halt
   braid start_kernel
   nop
#else
   /*
    * Initialize the MMU.
    */
   bralid   r15, mmu_init
   nop

   /* Go back to running unmapped so we can load up new values
    * and change to using our exception vectors.
    * On the MicroBlaze, all we invalidate the used TLB entries to clear
    * the old 16M byte TLB mappings.
    */
   ori   r15,r0,TOPHYS(kernel_load_context)
   ori   r4,r0,MSR_KERNEL
   mts   rmsr,r4
   nop
   bri   4
   rted  r15,0
   nop

   /* Load up the kernel context */
kernel_load_context:
   # Keep entry 0 and 1 valid. Entry 3 mapped to LMB can go away.
   ori   r5,r0,3
   mts     rtlbx,r5
   nop
   mts   rtlbhi,r0
   nop
   addi  r15, r0, machine_halt
   ori   r17, r0, start_kernel
   ori   r4, r0, MSR_KERNEL_VMS
   mts   rmsr, r4
   nop
   rted  r17, 0      /* enable MMU and jump to start_kernel */
   nop
#endif /* CONFIG_MMU */



#ifdef CONFIG_SMP

/* Entry point for secondary processors */
start_secondary_cpu:
```

```
/* Initialize small data anchors */
la r13, r0, _KERNEL_SDA_BASE_
la r2, r0, _KERNEL_SDA2_BASE_

/* Initialize stack pointer */
la r1, r0, temp_boot_stack + 1024 - 4

/*
 * Initialize the exception table.
 */
la r11, r0, secondary_machine_init
brald r15, r11
nop

lwi   r1, r0, secondary_ti

/* Initialize r31 with current task address */
lwi   CURRENT_TASK, r1, TI_TASK
SAVE_CURRENT_SAVE_R31

/* Initialize stack pointer */
addi  r1, r1, THREAD_SIZE-4
swi   r0, r1, 0

// Initialize MMU
ori r11, r0, 0x10000000
mts rzpr, r11

ori   r15,r0,TOPHYS(kernel_load_context_secondary)
ori   r4,r0,MSR_KERNEL
mts   rmsr,r4
nop
bri   4
rted  r15,0
nop

/* Load up the kernel context */
kernel_load_context_secondary:
# Keep entry 0 and 1 valid. Entry 3 mapped to LMB can go away.
ori   r5,r0,3
mts     rtlbx,r5
nop
mts   rtlbhi,r0
nop
addi  r15, r0, machine_halt
ori   r17, r0, start_secondary
ori   r4, r0, MSR_KERNEL_VMS
```

```
    mts   rmsr, r4
    nop
    rted  r17, 0      /* enable MMU and jump to start_kernel */
    nop


#endif /* CONFIG_SMP */
```