Mixed-Architecture Process Scheduling on Tightly Coupled Reconfigurable Computers

Brandon Kyle Hamilton, Michael Inggs
Department of Electical Engineering
University of Cape Town
Cape Town, South Africa
{brandon.hamilton, michael.inggs}@uct.ac.za

Hayden Kwok-Hay So
Department of Electrical & Electronic Engineering
University of Hong Kong
Hong Kong
hso@eee.hku.hk

***Abstract*—The design and implementation of a multitasking runtime system for mixed-architecture applications on a tightly coupled FPGA-CPU platform is presented. The runtime environment and the user applications assume an underlying machine that encompasses multiple computing architectures within a unified machine model. Using this model, a unified process scheduling mechanism was developed that enables concurrent execution of multiple mixed-architecture processes. Scheduling and allocation strategies, including blocking and preemption, were implemented and evaluated with respect to performance and fairness on a Xilinx Zynq platform using a mix of synthetic workloads.**

I. Introduction

While multitasking is routinely supported in conventional CPU-based systems, support for resource sharing in FPGA-accelerated systems has remained limited. With a general lack of resource sharing facilities, existing systems tend to dedicate FPGA accelerators to a single application, blocking other users from accessing them concurrently. Any sharing of accelerator resources must take place within one user process and must rely on ad-hoc user-defined mechanisms to coordinate concurrent access. From the OS's perspective, since the accelerators are treated as I/O devices instead of computing devices, the process scheduler usually does not take into account the computing time an application spent on them, negatively affecting the fairness and responsiveness of the system as a whole [1], [2].

To provide true multiuser support on an FPGA-accelerated platform therefore requires careful coordination among the user processes by the OS, which must treat reconfigurable resources as first-class computing resources of the system. The scheduler must be aware of and be able to adjust its scheduling decision in response to the time and resources consumed by a user process, may it be on the host CPU or in the reconfigurable hardware fabric. This is particularly important in tightly coupled FPGA-CPU systems in which multiple mixed-architecture user processes are going to be executing concurrently, each possibly spending some of their execution time on non-CPU computing resources.

Most existing works in the area have been focusing on the theoretical foundation of spatially scheduling tasks within an FPGA [3]; while others were devoted to the development of system-specific mechanisms for task preemption [4] or task relocation [5]. Partly due to the practical limitations of such existing platforms, substantial portions of the FPGA programming logic resources were often spent on implementing the underlying task management platform instead of on actual computation.

This work presents the design and implementation of an OS process scheduler that is fully aware of the mixed-architectural nature of the user processes. Using an implementation on a Xilinx Zynq processor, different process scheduling strategies under various mixes of workload were studied. The OS scheduler was implemented in a Linux kernel module that employs a *readback-based strategy* via the Configuration Access Port for context-switching of mixed-architecture applications.

With the practical implementation of the proposed scheduler, we consider the main contributions of this work are 1) The first practical implementation of mixed-architecture process scheduler on a tightly-coupled FPGA-CPU platform, and 2) A study on trade-off in blocking and preemptive scheduling decisions in the presence of high-overhead FPGA tasks.

II. Related Work

Various techniques have been proposed to handle context switching and hardware preemption on programmable logic devices. Rupnow et. al. [2] demonstrate three execution policies for hardware multitasking: *block*, *drop* and *rollback*, applied at each software thread context switch interval. A partial reconfiguration-based hardware context-switching method using a database in memory to save and restore the FPGA data is shown in [6]. Koch et. al. [7] extend the concept of software checkpointing to the hardware, in contrast to a readback-based approach. These works separate the programmable logic region into sections, with address scheduling methods carried out by dedicated hardware to effectively managing these regions among competing hardware kernels. In contrast, our system rather views the full reconfigurable hardware region as a possible execution architecture for running applications, thereby allowing the software scheduler to manage the resource utilization.

This work was supported in part by the Research Grants Council of Hong Kong, project ECS 720012E

From the perspective of the operating system, not much work has been devoted to the study of scheduling processes that execute with a mixture of computing architectures during runtime, such as those on a tightly-coupled CPU and FPGA system. Pham et. al [8] propose use of a microkernel-based hypervisor for virtualized execution and management of software and hardware tasks running on a commercial hybrid computing platform. ReconOS [9] provides an execution environment and extends the multi-threaded programming model to reconfigurable hardware using threads. This system requires the explicit control and coordination of the hardware threads within the user application, in contrast to our simplified single-context model.

III. Target Machine Model

To facilitate compiling and executing mixed-architecture applications, both the application development framework and the OS scheduler assume a simple underlying machine model with mixed computing architecture. The underlying *Multiple Runtime Architecture Computer* (MURAC) is a unified machine model that views heterogeneous computing architecture systems as a single idealized processor with morphable instruction execution units. It provides an abstraction for utilizing different compute architectures during runtime at a level similar to the instruction set architecture (ISA) of a conventional processor. The model defines the default architecture of the machine, running the OS, as the *primary architecture* (PA), and all other architectures on which a user may choose to execute their code are termed *auxiliary architectures* (AA). Applications may switch the computing architecture between PA and AA during runtime by using the branch-auxiliary-architecture (BAA) and return-to-primary-architecture (RPA) machine instructions.

Implemented on a tightly-coupled FPGA-CPU system, the CPU acts as the primary architecture, and the reconfigurable logic is seen as dynamically adapting auxiliary architectures. Logically, a MURAC system contains only a single unified memory address space, regardless of the number of auxiliary architectures defined. In addition, the MURAC model adopts the broad definition of an instruction to represent any entity that configures the machine so as to carry out proper computation. Using this definition, an FPGA configuration bitfile may equally be treated as an ultra-long instruction word, similar to that proposed by DeHon [10]. With this broad definition, the execution of a program, regardless of the computing architecture, is controlled by a single stream of instruction.

The simple MURAC model is useful as it enables the design of a unified scheduler that takes a holistic view on process scheduling in the presence of heterogeneous computing fabric. It helped to develop the notion of fairness among user processes, and allowed simpler decision making logic within the scheduler. Finally, unified instruction streams aid in streamlining the OS context switch logic in the actual platform implementation.

IV. Process Scheduling and Resource Allocation

In modern operating systems, multitasking allows simultaneous execution of multiple process by interleaving execution. The Completely Fair Scheduler [11] (CFS) implemented in the Linux kernel is modeled after a perfectly fair CPU: if two programs are running simultaneously, they both run at 50% of the CPU power at the same time. This scheduling algorithm is based on the principle of Weighted fair queuing (WFQ).

A. Mixed-Architecture Process Scheduling

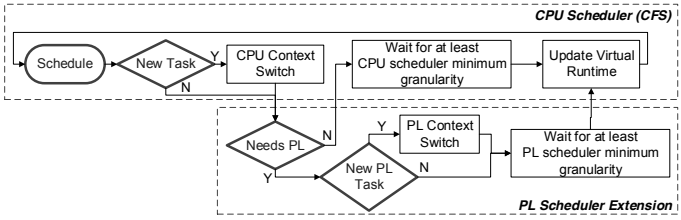

Fig. 1. Mixed-Architecture Process Scheduler

Under normal operation, when the CFS scheduler is triggered it selects the task with the lowest virtual runtime or one that has just become runnable, and performs a CPU context switch if necessary. The new activated task is executed and its dynamic timeslice is calculated. The task will be preempted once it has used it's allocated timeslice, or a new or existing task with a lower virtual runtime is waiting to run. During a context switch, the scheduler updates the task's virtual runtime according to the actual CPU execution time and system load. The CFS scheduler is able to divide the computational resources fairly due to the selection process based on the normalized virtual runtime of the tasks, resulting in every task being executed once per epoch.

As the user application in the MURAC model may execute on multiple system architectures within a single process context, the hw/sw boundary is captured within a single process. This leads to a simplified scheduling model from the OS perspective, as the scheduler does not need to be aware of the underlying execution architecture, but rather just view the whole application as single context task. The programmable logic portion of application can then be modeled as a computationally bound region of the application. In such a mixed-architecture system the main CPU kernel scheduler is used to control overall system scheduling, aided by a programmable logic specific scheduler extension handling mixed-architecture processes that contain programmable logic. Fig 1 illustrates the high level operation of this mixed-architecture scheduler during a scheduler tick.

B. Programmable Logic resource allocation

The allocation of programmable logic to actively executing tasks may consider:

a) Blocking: a process until the PL becomes available. While waiting, the process yields it's timeslice, making it more likely to be scheduled subsequently and receive a comparable share of the processor and programmable logic when it eventually needs it (so called *sleeper fairness*).

b) Preemption: of an exiting running programmable logic (PL) task, forcing a programmable logic context switch to take place. Due to the considerably higher latency of a programmable logic context switch, the concept of a *minimum PL schedule granularity* is used in combination with the *minimum CPU scheduler granularity*. This is needed to avoid thrashing as it ensures that a running PL task will receive a minimum runtime before being preempted, and will also affect the subsequent scheduling of the task as this time will be accounted for in the task's virtual runtime. This strategy effectively models PL execution as a CPU-bound process from the perspective of the OS. In contrast to the blocking strategy, preemption requires the ability to suspend the execution of an ongoing task and to restore a previously interrupted task. The save and restore operations are performed at the software (process/thread) context switch in the OS, carried out by the programmable logic specific scheduler extension.

V. System Implementation

Our system has been implemented and tested on the Xilinx Zynq XC7Z020 based Zedboard platform: a dual Cortex-A9 Processing System (PS) tightly integrated with programmable logic (PL).

A. Mixed-Architecture Applications

The mixed-architecture application compilation toolflow is demonstrated in Fig 2. The reconfigurable logic portions of the application are synthesized using standard FPGA tools. The resulting configuration bitfiles are then embedded directly into the software application source code at the desired point of execution within the program flow. This is implemented at the ISA level as an architectural branch instruction followed by the configuration data (demonstrated in Listing 1). This mechanism enables the entire mixed-architecture application to be compiled by the unmodified CPU toolchain, producing a single executable binary application file.

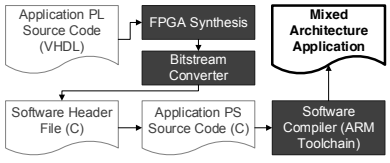

Fig. 2. Mixed-Architecture Application Compilation

In this way, user applications are able to take advantage of implementation-independent access to multiple alternative compute architectures available at runtime. The configuration, co-ordination of control of a slave accelerator device that would normally be required is eliminated, leaving a simple explicit application control flow path when running code on the programmable logic.

```
#define BAA_CONFIG_IMPL_PARTIAL(PERIPHERAL_BASE) \
  asm volatile("mov r1,%[value]" : :
       [value]"r"(PERIPHERAL_BASE) : "r1"); \
  asm volatile("mrc 1,0,R15,c1,c2"); // BAA instruction \
  asm volatile( // Embedded FPGA configuration \
    ".word 0x52554dff\n\t" \
    ".word 0x1004341\n\t" \
    ".word 0x30600300\n\t" \
    ...

int main(int argc, char *argv[]) {
   unsigned int reg; reg = atoi(argv[1]);
   printf("Branching to FPGA\n");
   BAA_CONFIG_IMPL_PARTIAL(0x60A00000)
   printf("Output: 0x%x\n", reg);
   return 0;
}
```

Listing 1. Example mixed-architecture application source listing

A synthetic workload application demonstrates the usage of the MURAC abstraction in Listing 1, which is used to simulate a variable length runtime mixed-architecture process in our system. This illustrates the simplified programming model for switching execution between alternate compute architectures with the use of a single instruction, as well as the explicit control flow of accelerator utilization from the point of view of the application.

B. Programmable Logic Resource Allocator

A linux kernel module implements the programmable logic specific scheduler extension and enables runtime support for the execution of mixed-architecture binary applications. This module is responsible for the configuration of the programmable logic region as well as co-ordination and management of task scheduling, transparent to the user process. Upon being loaded into the operating system kernel, the module hooks into the kernel instruction handler to enable device configuration and scheduling upon execution of an *architectural branch* (BAA) instruction in a user process, as well as loading a base FPGA configuration with an AXI connected *Internal Configuration Access Port* (ICAP) module. In addition, the user design must support a mechanism to signal completion of the hardware task, indicating the RPA instruction to the waiting kernel module in order to continue the task execution on the primary architecture.

The running programmable logic design does not need to be controlled by software as is typical in the master-slave accelerator model. Application data is accessible directly from the programmable logic region through the unified virtual address space, allowing the software to fully suspend on the primary architecture while the auxiliary architecture is employed. In our implementation, the information about the current process memory address space (stack) is passed to the programmable logic via a register exposed over the AXI bus. This mechanism thus enables the explicit single-context control flow visible to the application.

The context-switch operation is implemented through a readback-based strategy by writing commands to the configuration access port, namely *capture* (IOBs and CLB contents) and *readback*. Although reading from the FIFO of the ICAP

via the AXI bus is achieved through the processor subsystem rather than DMA, this does not affect our implementation as the processor would not be used concurrently for any other processing in the MURAC model.

VI. Implementation Results

Workloads consisting of 10 concurrently executing mixed-architecture processes were run under the blocking and preemption scheduler strategies. In our implementation, the partial bitstream (219776 bytes) leads to context switch latency of 45ms. The processes are all run with the *SCHED_NORMAL* process priority class. Under the preemption strategies tested, the *minimum CPU scheduler granularity* and *minimum PL scheduler granularity* are varied to illustrate the effect on scheduler performance based on workload (as described in Section IV). These parameters are indicated as P(*minimum CPU scheduler granularity* in ms, *minimum PL scheduler granularity* in ms) in the figures below. The standard CFS scheduler in most common linux distributions has a default *minimum CPU scheduler granularity* of 1.5ms.

The workload profiles evaluated were: a) equal granularity long-running PL task processes, b) equal granularity very short-running PL task processes, c) mixed granularity (uniformly distributed long- and short-running PL task) processes.

The results of these experiments, as illustrated in Fig 3, show the total runtime to complete all processes in the workload combined, normalized per workload. Fig 4 illustrates the overall scheduler overhead experienced for each of these workloads.

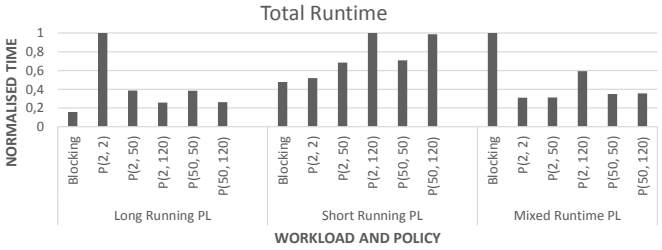

Fig. 3. Normalised total runtime

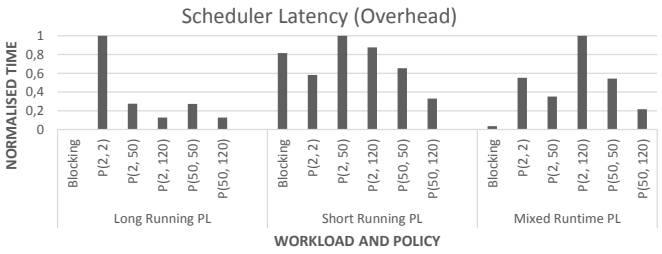

Fig. 4. Normalised scheduler overhead

Based on the results, it can be seen that *blocking* provided better overall performance for equal granularity tasks, as it minimizes turnaround time. This indicates that *blocking* allocation is preferable for batch processing that is CPU bound. *Blocking* minimizes expensive scheduler latency, providing better throughput. However, when running a uniformly distributed mixed workload, the *preemption* strategy performs better, as it ensures more fairness for the shorter running tasks. It can be observed that the scheduler performs best when the *minimum PL scheduler granularity* is higher than the scheduler latency for these workloads.

VII. Conclusions & Future Work

In this paper, we have presented the design and implementation of a runtime process management system for mixed-architecture processes in a tightly-coupled FPGA-CPU system. By using the MURAC model, the scheduler takes a holistic view on process scheduling regardless of the exact compute fabric a process utilizes. This allows a simple scheduler design in the Linux kernel and improves fairness among users. Blocking and preemptive scheduler strategies were investigated. Based on our current implementation on a Xilinx Zynq system, the blocking scheduling policy is best when the input processes are batch type processes with similar processing time using the FPGA fabric. On the other hand, preemptive scheduling policy is best when there is a mix of long and short applications running on the programmable fabric.

With a working basic scheduling framework, we will further explore trade-offs between fairness, interactivity and performance on mixed-architecture process in the future and improve on context-switch efficiency on existing platform.

References

[1] W. Fu and K. Compton, "Balanced allocation of compute time in hardware-accelerated systems," in *ICECE Technology, 2008. FPT 2008. International Conference on*, 2008, pp. 241–248.

[2] K. Rupnow, W. Fu, and K. Compton, "Block, Drop or Roll(back): Alternative Preemption Methods for RH Multi-Tasking," in *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, 2009, pp. 63–70.

[3] W. Fu and K. Compton, "Scheduling intervals for reconfigurable computing," in *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, 2008, pp. 87–96.

[4] L. Levinson, R. Männer, M. Sessler, and H. Simmler, "Preemptive multitasking on FPGAs," in *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 301–.

[5] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," in *Field Programmable Logic and Applications, 2005. International Conference on*, 2005, pp. 223–228.

[6] T.-Y. Lee, C.-C. Hu, L.-W. Lai, and C.-C. Tsai, "Hardware context-switch methodology for dynamically partially reconfigurable systems," *Journal of Information Science and Engineering*, vol. 26, no. 4, pp. 1289–1305, 2010.

[7] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: concepts, overhead analysis, and implementation," in *Proceedings of the 15th international symposium on field programmable gate arraysn (FPGA '07)*. New York, NY, USA: ACM, 2007, pp. 188–196.

[8] K. D. Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell, "Microkernel hypervisor for a hybrid ARM-FPGA platform."

[9] E. Lübbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 1, pp. 8:1–8:33, Oct. 2009.

[10] A. DeHon, "DPGA utilization and application," in *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, ser. FPGA '96. New York, NY, USA: ACM, 1996, pp. 115–121.

[11] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, "Towards achieving fairness in the Linux scheduler," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 34–43, Jul. 2008.