A Reconfigurable RTOS with HW/SW Co-scheduling for SOPC

Qingxu Deng, Shuisheng Wei, Hai Xu, Yu Han, Ge Yu
Department of Computer Science and Engineering
Northeastern University, China
xhsoldier@163.com

Abstract

Emerging reconfigurable hardware, SOPC (System On Programmable Chip), requires a RTOS to reuse the abundant source code. This paper presents a RTOS with the ability to co-schedule HW/SW, and discusses its architecture in detail for SOPC. The paper addresses an efficient run-time partitioning algorithm for block partitioning of FPGA. At last, a case study will be presented to validate our approach. The RTOS can decreases NRE costs and facilitates integrating hardware and software seamlessly.

1. Introduction

Emerging trends in system design indicate that in the future, more roles will be played by System-on-Chip (SoC) platforms consisting of general-purpose, configurable components [1].For example, it enables designers to utilize a large FPGA that contains both memory and logic elements along with an IP processor core. The increasing densities and reconfiguration modes of SRAM-based field-programmable gate arrays (FPGAs) and configurable systems on a chip (CSoCs) advocate more dynamic uses of these components. These impose strict demands on system implementation technology. High performance requires application-specific architectures, but flexibility and system agility require a programmable, adaptable approach. Qingxu Deng and Hai xu have presented a methodology which is used for helping to design system platform for SOPC [2].

All our efforts are resolving issues of software reuse and hardware reconfiguration. We present a RTOS with HW/SW co-scheduling to manage the system physical resources and to help designers building their software platform. The RTOS forms an abstraction that hides the details of the underlying technology from the developer and decreases Time-to-market and Non-Recurring Engineering (NRE) costs. It also increases the reuse of software and the flexibility of hardware, and facilitates a natural style of hardware/software co-design for embedded system.

The remainder of this paper is organized as follows. Section 2 presents a survey of related work in configurable OS for SoC. Section 3 shows the architecture of our RTOS and some concepts. In section 4, how to manage and utilize the reconfigurable resource is presented, and expatiate a partitioning algorithm. Hardware/software co-scheduling is presented in Section 5. Section 6 gives a case study and concludes the paper.

2. Related Works

In this section, we briefly survey a selection of related work in the area of runtime reconfigurable platform operating system. As the complexity of SOC's architecture and application increases, reconfigurable hardware operating systems become a rather new line of research. The first description of hardware multitasking is due to Brebner [3]. He raised the issue of managing a virtual hardware resource. He proposed decomposing reconfigurable computing applications into swappable logic units (SLUs). Recently, Wigley et al. discussed OS services including device partitioning, placement and routing [4]. In [8], Mignolet et al. introduce relocatable tasks which can be executed either in software or in hardware, depending on the available resources and the performance required. Herbert Walder and Marco Platzner further the study of re-configurable hardware operating systems in a top-down manner [9].

In summary they don't provide feasible and unified hardware and software platform. Some ones only present a prototype or concepts. Our RTOS focus on software reuse and hardware reconfiguration to help the designers building their software platform. At the same time, it stresses real-time response and whole performance, and utilizes least hardware to satisfy the requirements.

Proceedings of the Second International Conference on Embedded Software and Systems (ICESS'05)
0-7695-2512-1/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

3. System Model and Concepts

Commercial RTOSs available for popular embedded processors enable the development of software-based real-time systems and significantly shorten the design period. But they typically take no advantage of hardware to implement any of their functions, probably because processors and custom hardware accelerators have historically resided on separate chips. Although sufficient for a limited number of RISC/DSP processors, these pure software kernels are of limited use for heterogeneous, dynamically reconfigurable designs.

We present a RTOS with HW/SW co-scheduling to combine the software's reuse and hardware's performance. It gives the HW with OS level support. Its architecture is shown in **Figure 1**.

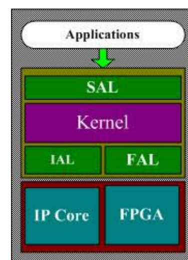

Figure 1. System architecture.

The application should be designed in an uniform HW/SW design environment. The application consists of several software tasks and hardware tasks. Hardware task is actually a circuit that can be configured and executed onto the device without affecting other currently running circuits. As for calling it task is to be consistent with the dynamic nature of software task. The task is the least execution unit in our system. We highly abstract the hardware task and software task with uniform interface to facilitate partitioning and co-scheduling them.

The kernel of RTOS provides multiple system services: managing resources, co-scheduling tasks, communicating between tasks and so on. Resources management is a general service that includes partitioning the reconfigurable area, managing the memory and mapping I/O devices. The communication service allows tasks interact with messages, regardless of hardware tasks or software tasks. We put forward software abstract layer (SAL) and hardware abstract layer (HAL) to facilitate co-scheduling and communicating. The SAL takes the application as input, and the partitioning of the tasks can be decided in consideration of designer's setting. The decision also can be made automatically for an efficient execution. The HAL is consists of IAL (IP core abstract layer) and FAL (FPGA abstract layer). We utilize it to abstract the IP core and reconfigurable logic to provide uniform interface.Its main aim is to port and utilize them conveniently.

4. Reconfigurable Resource Management

The partitioning and placement is an important issue in reconfigurable resource management. M. Gericota[10] and M. Handa[11] let us know that fragmentation of the FPGA's resources cause low area utilization in the dynamic reconfiguration systems. Bazargan et al. [12] addressed the issue of placing application mappings onto a single device for hardware execution in a reconfigurable computing system. Walder et al. [13] combined an enhanced form of Bazargan's partitioning algorithm and a placement-finding algorithm using 2D-hashing. Brebner and Diessel [14], present a 1D area model where tasks can be allocated anywhere along the horizontal dimension; the vertical dimension is fixed.

In partially reconfigurable hardware platforms, the problems of task and resource management are even more strongly connected than in processor-based platforms, which make the design of runtime systems more challenging. We introduce an efficient partitioning algorithm for runtime reconfigurable platforms to achieve better hardware resources utilization and real-time capability. Although the algorithm seems somewhat simple, it can low the overhead and response time of the kernel and can offer a feasible partitioning solution.

Hai Xu [2] discusses the main issues involved system behavioral description, HW/SW partitioning and mapping them to a set of tasks. It is highly desirable to perform the partitioning aiming at minimizing the overall inter-task communication. The shape of a hardware task is mostly modeled by a rectangle as well as the routing resources used by the task. The granularity of the resulting tasks is in itself an important issue. The larger is the average size of the tasks, the more tractable are the dynamic placement and routing problems. However, smaller tasks create more opportunities for task reuse, and thus potentially minimize. The need for dynamic reconfiguration, our current rule-of-thumb, is that a task should occupy no less than 10

As follows, we simply discuss the algorithm. We adopt 1D area model and partition the horizontal dimension into segments (the size and number of the segment are decided by designer or system), as **Figure 2** shows.

0-7695-2512-1/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

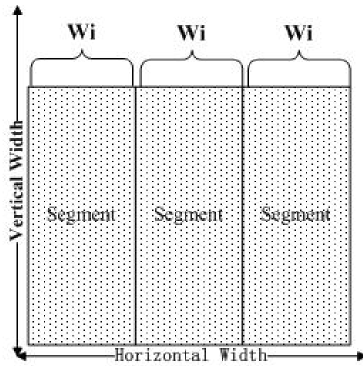

Figure 2. Area partition.

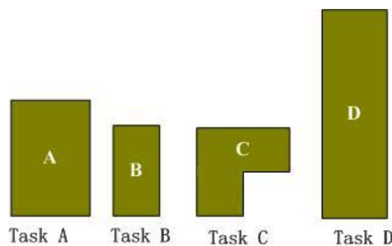

Figure 3. Hardware task blocks.

So we can manage the area in linear memory. For the area in segments, we logically divide them into three types: blank area(no tasks used it before)running area(there is a task running in it)reserved area it is reserved for a periodic task running next time. Every segment has a non-running area interval list which records blank area interval and reserved area interval (stores the interval points in sequence).When a task applies the reconfigurable hardware resource, we check the shape of it. If both the height (h) and the width (w) of a task (h,w) exceed the segments' width(Wi),we consider repartitioning the tasks into small sub-tasks until an adequate task granularity is achieved. We adopts compile and decompile technology for repartitioning the task. Otherwise, we choose a segment that can satisfy the task's requirement for landing edge (width or height that is close to Wi). We search the non-running area list of the segment, and blank areas are considered first. If there are no continuous blank areas, we consider occupying the reserved area (search the similarest reserved area) or merging reasonable blank area and reserved area. If there is still not enough continuous area, we check whether can preempt tasks (discussed in section 6). If width (w) is the landing edge, we assign directly corresponding height (h) of the resource (**Figure 3** task A,C,D). If height is(**Figure 3** task B), we calculate the utilization factor for height and width separately to see whether is good for rotating the task. If the difference of utilization factors, we call it Utilization Factor Difference, is less than the value that designer has set we make it a 90 clockwise rotation. We assume that modules are relocatable, it is possible to change the routing programmed into each cell to reflect the overall rotation of the configuration. We always assign regular resource area (h £ Wi), as **Figure 4** shows. When the task has executed, the periodic task can apply to reserve the area that it has used, and system change the running area into reserved area. So it can decrease the configuration overhead. Other tasks free the resource directly and the area interval is inserted into non-running area interval list. At this time merging blank area interval is involved(one's upper bound is equal to the other's lower bound, eg.[4,5] and [5,10] into [4,10]). **Figure 5** shows the situations of task placement. In consideration of the real-time requirement, only when it is lower than the utilization factor difference, we just rotate the task. If the HW/SW partition is reasonable in the initial stage, and the task granularity and segment's width are appropriate, there are fewer chances to repartition the task.The partitioning algorithmcan be customized with requirements.

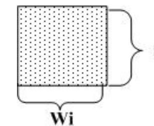

Figure 4. Resource block.

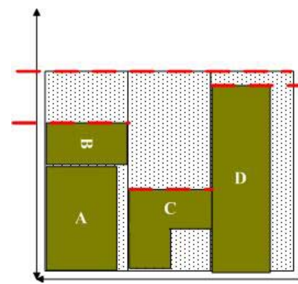

Figure 5. Task placement.

5. Hardware/Software Co-scheduling

5.1. Online Scheduling Issues

One of the first descriptions of hardware multitasking is due to Brebner [3]. The preemption of hardware tasks was investigated by Simmler et al. [5]. Diessel [16] discusses how to deal with partial reconfiguration and multi-tasking for rearrangement of tasks on FPGAs, which lacks real-time capabilities. H. Walder[15] propose an online scheduling system that allocates tasks to a block-partitioned reconfigurable device. However, they didn't refer to HW/SW co-scheduling.

IEEE COMPUTER SOCIETY

There are several advantages for multitasking in FPGA: First, parallel execution of several tasks in the same FPGA is possible. Second, the programmer does not need to care about resource sharing and so writing FPGA designs is much easier, because each FPGA design has the complete set of I/O resources available. But the tasks have some restriction conditions for hardware multitasking: relocatability, independence, required cycles and so on. So we implement these tasks in hardware and the others in software that are not feasible for hardware multitasking. Our goal is holding implementation efficiency and design flexibility.

The RTOS keeps track of the HW/SW tasks by means of a uniform task information structure list. Every task instantiation is linked to such a task information structure. We divide tasks into types: period/aperiodic, static/dynamic, preemptive/non-preemptive, and every task have an exclusive priority. There is a task state graph to show the kernel how to schedule the HW/SW tasks, **Figure 6**.

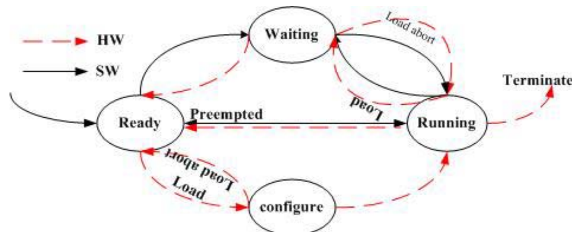

Figure 6. Task state graph.

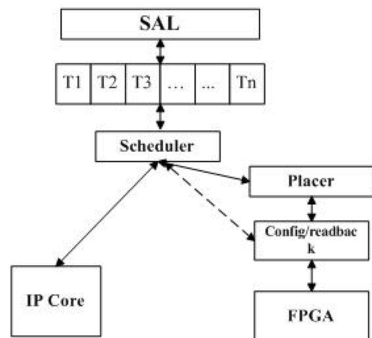

Figure 7. Co-scheduling structure.

The problems of scheduling tasks are strongly interrelated with resource management in hardware platforms. We discuss how to manage the reconfigurable hardware resource and how to place the tasks in section 4. We show the HW/SW co-scheduling structure in **Figure 7**. Actuallywe suggest that should decrease the chance of preempt the hardware task, because it is a work with heavy time cost and memory cost.

5.2. Uniform Communication between HW/SW Tasks

By providing a uniform communication scheme for hardware and software tasks, the kernel we developed hides the complexity of co-scheduling tasks. In our approach, communication between tasks is based on message passing. Messages are transferred from one task to another in a common format for both hardware and software tasks, and have a priority. Both the operating system and the hardware architecture should support this kind of communication. At the same time we provide event, mailbox and semaphore service. All the communication services are achieved by sharing memory. In our communication scheme, there are three distinguished types of message passing between tasks.

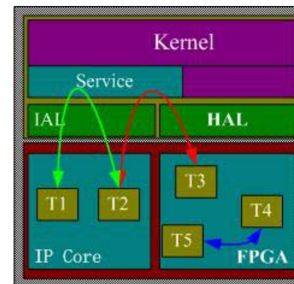

Fig. 8. Message passing between tasks.

Table 1. Benchmark loop information.

Example	Size instr	Loop Instr	Loop Time(%)	CSL Size
g3fax	2188	24	62	450
adpcm	3820	76	30	938
crc	2120	34	65	92
des	3058	180	52	1032
engine	2216	32	28	266
jpeg	2980	58	10	314
summin	2068	50	48	424
v42	3194	30	23	466
Average	2075	60	40	498

As **Figure 8** shows, messages between two tasks that are scheduled in the soft CPU (T1 and T2), are routed solely based on memory address and do not pass the HAL. Communication between a software task and a hardware task (T2 and T3) must pass through the hardware abstraction layer. In this case, an address mapping is performed by the communication service. The task's physical address allows the HAL to determine which task will receive the message.

5.3 Task Context Switch Issues

Consequently, the state of a preempted task can be fully saved by pushing all the registers on the task

0-7695-2512-1/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

stack.This approach is not usable for a hardware task. A FPGA design normally does not have a code or data descriptor like a CPU. Instead of the handful of registers and required to specify the state of a software task (in addition to its memory contents), the hardware process may have a large number of internal registers. Rather, an FPGA holds data in several registers scattered over the FPGA. It depicts its state in a completely different way: state information is held in several registers, latches and internal memory, in a way that is very specific for a given task implementation. There is no simple, universal state representation, as for tasks executing on the CPU. Nevertheless, the operating system will need a way to extract and restore the state of a task executing in hardware, since this is a key issue when enabling implement preemptive scheduling. A way to extract and restore state when dealing with tasks executing on the recon¯gurable logic, is described in[5]. State extraction is achieved by getting all status information bits out of the read back bitstream. We should do our best to decrease the situation of preempting the hardware tasks, because it affects the real-time capability and needs lots of memory. Furthermore, the FPGA was configured within 70 ms and readback takes 800 ms. The memory requirement is up to 350 Kbytes for a modern FPGA device. Although we extract the state bits that are then stored and form the basis for the reconstruction, and the memory requirement is also about 300 Kbytes. We use a high level abstraction of the state information for hardware tasks, and the preemptive hardware task has a memory area to store the state bits. When partitioning the hardware and software tasks, we should cautiously consider which task is preemptive.

6. Preliminary Results

We use Memec Virtex-4 LC Development Kit in our experiment. The clock speed for the MIPS is 100 MHz at a supply voltage of 1.2V, and we used Xilinx's Virtex Power Estimator [22] to estimate power for each example. Here, we use an operating system, and this cause the result di®erent from those of no operating system supporting as for most of the test cases. We examined several examples from Motorola's Powerstone [21] benchmark suite: a voice encoder (adpcm), a cyclic redundancy check (crc), a data encryption standard (des), an engine controller (engine), a fax decoder (g3fax), a JPEG decoder (jpeg), a handwriting recognizer (summin), and a modem encoder/decoder (v42). We implement each example, using the input vectors in Powerstone, on an instruction set simulator for an MIPS microprocessor, augmented to output instruction traces. Table 1 summarizes the relevant loop data for our benchmarks. Size indicates the total number of instructions in the program, and Loop Instr is the number of instructions in the region(s) moved to hardware. Loop Time is the percentage of total execution time taken by the region(s). CSL Size is the number of configurable logic blocks required by those regions. The test results for our method are summarized in Table2. Sw is the total number of cycles to execute the example completely in software. Loop in sw is the total cycles required by the loop when running in software. Loop in CSL is the number of cycles required by the loop when running in custom hardware. Sw/CSL is the number of cycles required to execute the entire program after partitioning. Speed-up is the resulting speedup after partitioning. Performance of Co-scheduling RTOS and General RTOS are the test results of executing all the bench marks at one time.

Table 2. Benchmark test results.

Example	Performance(cycles)				
	SW	Loop in sw	Loop in CSL	Sw/CSL	Speed-up
g3fax	7,800,000	2,360,000	299,500	5,739,500	1.33
adpcm	56,500	14,650	2,720	4,457	1.24
crc	2,520,000	1,740,000	230,400	1,010,000	2.46
des	71,000	35,350	7,550	43,200	1.60
engine	457,500	72,500	14,050	399,050	1.11
jpeg	3,950,000	323,000	85,500	3,712,500	1.02
summin	1,460,000	635,000	133,000	958,000	1.50
42	1,925,000	423,000	108,000	1,610,000	1.18
average					1.43
Execute all at one time	Co-scheduling RTOS		General RTOS		Speed-up
	0.08635s		0.18759s		2.17

7. Conclusions and Future Work

As shown in the table 2, we conclude that using such a RTOS for SOPC can greatly improve the performance of embedded system. The higher abstraction is, the better reuse will be. Such a HW/SW co-scheduling RTOS will reuse the abundant resource of source code, so it can decreases Time-to-market and Non-Recurring Engineering (NRE) costs. The flexibility of the hardware, reconfigurable system, make the system meet many different kinds of application requirements, and only making a few changes will enable the system qualify for another kind of application.

In the future, we expect the kernel support transforming HW/SW tasks mutually to make best use of software and hardware. SAL needs further abstraction for HW/SW tasks, so less need for designer to concern whether a task is SW or HW. We also plan

IEEE COMPUTER SOCIETY

to extend our research to the configuration of heterogeneous multi-processor systems.

References

[1] Krishna Sekar, Kanishka Lahiri, Sujit Dey. “Configurable Platforms with Dynamic Platform Management: An Efficient Alternative to Application-Specific System-on-Chips”. *Proceedings of the 17th International Conference on VLSI Design (VLSID'04)*, IEEE, 2004

[2] Qingxu Deng, Hai Xu, etc. “An Embedded SOPC System Using Automation Design”. ICPP05, IEEE, 2005

[3] G. Brebner. “A Virtual Hardware Operating System for the Xilinx XC6200”. *In Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL)*, Springer,1996.

[4] G. Wigley, D. Kearney. “Research Issues in Operating Systems for Reconfigurable Computing”. *In Proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms(ERSA)*, CSREA Press, 2002.

[5] H. Simmler, L. Levinson, R. Manner. "Multitasking on FPGA Coprocessors". *In Proceedings of the 10th International Workshop on Field Programmable Gate Arrays (FPL)*, pages 121-130. Springer, 2000.

[6] G. Brebner, O. Diessel. “Chip-Based Reconfigurable Task Management”. *In Proceedings of the 11th International Workshop on Field Programmable Gate Arrays (FPL)*, pages 182-191. Springer, 2001.

[7] H. Walder, M. Platzner. “Online Scheduling for Block-partitioned Reconfigurable Devices”. *In Proceedings of Design, Automation and Test in Europe (DATE)*, pages 290-295. IEEE Computer Society, March 2003.

[8] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, V. S., R. Lauwreins. “Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip”. *In Proceedings of Design, Automation and Test in Europe (DATE)*, pages 986-991. IEEE Computer Society, March 2003.

[9] Herbert Walder, Marco Platzner. “Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations”. *Engineering of Reconfigurable Systems and Algorithms*, pp: 284-287, 2003.

[10] M. Gericota, G. Alves, M. Silva, J. Ferreira. “Run-Time Management of Logic Resources on Reconfigurable Systems”, *In Proc. of Design, Automation and Test in Europe*, Mar. 2003.

[11] M. Handa, R. Vemuri. “Area Fragmentation in Re-configurable Operating Systems”. *In Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms*. CSREA Press, Jun. 2004.

[12] Kiarash Bazargan, Ryan Kastner, Majid Sarrafzadeh. “Fast Template Placement for Reconfigurable Computing Systems”. *In IEEE Design and Test of Computers*, volume 17, pp 68-83, 2000.

[13] HerbertWalder, Marco Platzner. “Fast On-line Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing”. *17th International Parallel and Distributed Processing Symposium*, 2004.

[14] Gordon Brebner, Oliver Diessel. “Chip-Based Reconfigurable Task Management”. *In Proc. 11th Intel Workshop on Field Programmable Gate Arrays (FPL)*, pages 182-191, 2001.

[15] H. Walder, M. Platzner. “Online scheduling for block-partitioned reconfigurable devices”. *In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, Mar. 2003. 17.

[16] Diessel, O., H.ElGindy (1997). “Partial FPGA Rearrangement by Local Repacking”. *Technical Report 97-02, Dept. of Comp. Sci. and Software Engr., Univ. of Newcastle, Australia.*

0-7695-2512-1/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY