SMPS: An FPGA-based Prototyping Environment for Multiprocessor Embedded Systems

Ankit Mathur
ankit@netapp.com
Network Appliance,
Bangalore, India

Mayank Agarwal
magarwa2@crhc.uiuc.edu

Soumyadeb Mitra
mitra1@uiuc.edu

Department of Computer Science,
University of Illinois at Urbana-Champaign, USA

Anup Gangwar
anup@cse.iitd.ac.in

M. Balakrishnan
mbala@cse.iitd.ac.in

Subhashis Banerjee
suban@cse.iitd.ac.in

Department of Computer Science and Engineering
Indian Institute of Technology Delhi, India

ABSTRACT

Streaming media applications represent an important class of applications for embedded systems. Recent advances in design-space exploration of architectures for such applications have pointed towards the suitability of Multiprocessor System on Chip (SoC) solutions. Multiprocessor SoCs not only offer higher performance, but can also lead to solutions which are cheaper cost wise. A typical synthesis methodology for such architectures would require a validation stage at the end of final system integration. The wide availability of cheap and large FPGA devices, advances in automatic synthesis from VHDL/Verilog and abundance of high performance computing platforms enables the design of a generic validation system for such Multiprocessor SoCs.

In this paper we present the design and implementation of Srijan Multiprocessor Prototyping System (SMPS). SMPS is a system for rapid prototyping and validation of single chip application specific multiprocessor systems. The individual computing elements are RISC processors, coprocessors which lie in the processor pipeline, and ASICs which connect directly to system bus. The system is a tightly coupled multiprocessor with shared memory and shared address space. A Real-time Operating System (RTOS) provides task scheduling and access to shared resources. SMPS can support this entire design-space. The system is presented as a parameterized VHDL based on the open source Sparc V8 compliant Leon processor and a homegrown lightweight RTOS, RtKer-MP. The entire VHDL is configurable using a GUI, has support for cache coherency, choice of arbitration policy and easy integration of custom processing engines. RtKer-MP allows for a pluggable scheduler, dynamic and static scheduling policies, static and dynamic task migrations domains and variable interruption frequencies for separate processors. The pluggable scheduler interface allows for quick exploration of various scheduling policies for a feedback to the estimation systems.

FPGA '05 Monterey, California USA

1. INTRODUCTION

Embedded multiprocessor platforms are becoming quite popular both in the research domain as well as commercial domain [3] [9] [12] [1] [20] [10] [11]. While commercial ventures are focussing more on the high performance achieved by these multiprocessors, researchers are focussing on architecture customizations on a per application (domain) basis. A driving force is the fact that multiprocessor solutions may lead to solutions which are cheaper cost wise than their uniprocessor counterparts [22] [4].

However, the design-space for multiprocessor systems is huge which makes it practically impossible to explore it exhaustively. Moreover the lack of retargetable tools such as compilers and simulators for such vast design-space complicates the problem further. Researchers try to utilize a top-down approach, with separate tools for each stage [19] [1] [5]. The speed of execution of tools decreases when one moves from top to bottom and the accuracy of results increases. Approach is to achieve quick design-space pruning at the top-level and do a more refined search at the lower levels. Typical tools at the top-level include hardware and software estimators for estimating cost of synthesized components and software performance on customized architecture respectively and partitioners, which decide to which processing element (ASIC or Processor) a particular piece of computation is mapped. At the lower-level a retargetable compilation system coupled with a retargetable multiprocessor simulation system gives the system performance. A final prototyping stage using FPGA devices with all the system components in place serves as a proof-of-concept as well as gives confidence that the system works. More details on application specific multiprocessor synthesis are present in Section 3.

In this paper we present Srijan Multiprocessor Prototyping System (SMPS), which can be efficiently utilized for validation of application specific multiprocessor systems. Our system consists of a tightly coupled multiprocessor architecture, Leon-MP and a light weight retargetable parameterized RTOS, RtKer-MP. The entire system is easily downloaded onto reasonably large FPGA devices such as Xilinx XC2V3000-FF-1152-5. The ready availability of large and fast FPGA devices coupled with cheap and efficient synthesis tools makes such a system extremely practical. Also, SMPS can be used to provide accurate feedback to estimation tools and partitioners while the algorithms for these tools are still evolving.

The rest of this paper is organized as follows: Section 2, discusses the previous work done in this domain. Section 3, gives an overview of a synthesis methodology for multiprocessor embedded systems, Srijan. Section 4, presents details about the multiprocessor architecture and parameterized components of the hardware part of SMPS. Section 5, discusses RtKer-MP, a lightweight parameterized RTOS with a pluggable scheduler. Section 6, gives more details on the implementation of this system. Section 7, discusses some results and Section 8 presents conclusions and gives directions for future work.

2. RELATED WORK

A variety of Rapid Prototyping systems are available in both research as well as commercial domain [15], [21] RPM Rapid Prototyping System [6] [18] etc. Also there are software which allow one to explore system-level alternatives [16], COSSAP/CoCentric System Studio from Synopsys, Signal Processing Workbench (SPW) from Cadence and are termed as rapid prototyping systems. However, our focus is on actual prototyping and validation which is comparable to systems offering rapid prototyping facilities using FPGA devices.

Predominantly the systems reported either in research or commercial domain (e.g. [15]) are suitable for one application (or domain). In such systems user can rapidly change the system configuration using a few available alterntatives. On the other hand there are *general* prototyping platform (e.g. [6], [18] etc.) which allow users to prototype large systems using inexpensive large number of FPGA devices. However, as large FPGA devices are now available inexpensively, the simplicity of prototyping using a single large FPGA device, coupled with widely available CAD tools speaks heavily in their favour. In principle we are also offering a parameterized system for a special domain, which allows configuration using a wide variety of parameters, similar to [15], since such systems are specific to application, no comparison is possible with other such application specific or general prototyping platforms. In our knowledge, no such system as is being reported here, exists.

3. DESIGN FLOW FOR MULTIPROCESSOR EMBEDDED SYSTEMS

Figure 1, shows the architecture design space being targeted under the Srijan project. Processing elements can be plain RISC processors, Application Specific Instruction Processors (ASIPs) or ASICs. The system uses shared bus and shared address space for communication amongst processing elements. Memories can either be connected to processing elements (local memories) or connected to the bus using memory controllers. Application representation under Srijan is using Kahn Process Network semantics (KPN) [17].

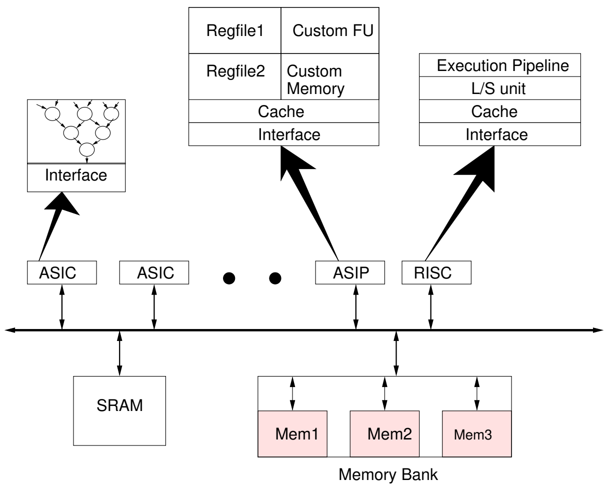

Figure 1: Srijan Architecture Design Space

In KPN processes communicate using FIFO channels which are assumed to be unbounded (infinite size). As is easily observable there is tremendous amounts of architectural flexibility. However, to properly utilize this flexibility effecient design-space exploration mechanisms need to be devised.

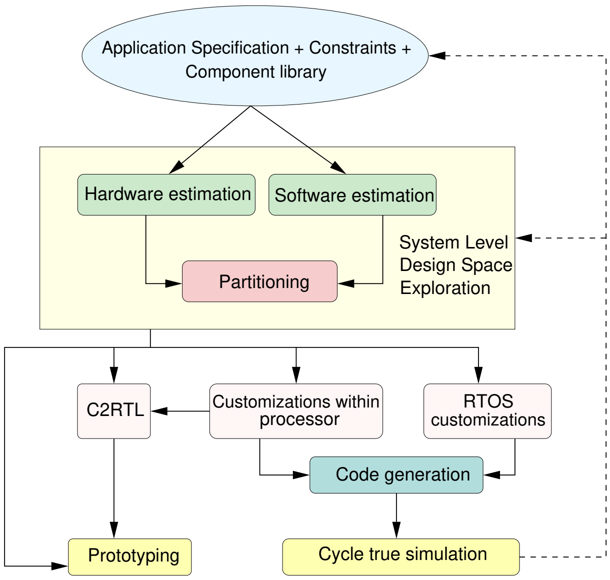

Figure 2: Srijan Synthesis Flow

Figure 2, gives the synthesis flow for a methodology for synthesis of application specific multiprocessor systems, Srijan [1]. Application, represented using KPN semantics, system level constraints such as those on total power, area and performance and component library containing processing elements, memories etc. are the inputs to this synthesis system. The system level design space exploration is performed using hardware and software estimation systems and the system level partitioner. Partitioner invokes hardware and software estimators to evaluate individual design alternatives at a very high level of abstraction details of this are present in [11].

Once the system level alternatives have been decided, the choices are passed onto the various subsystems. These include the C-to-VHDL translator, for synthesis of various coprocessors and ASICs, processor customization subsystem, RTOS customization etc. Processor customization can include customization of processor datapath width, addition of coprocessor which lie in the processor pipeline etc. RTOS customization includes formation of task migration domains (more details are present in Section 5), decision on interruption frequency for individual processors etc. It need to be noted that for design-space exploration C-to-VHDL translation is not required, a retargetable code generator, coupled with a retargetable simulation system provides the necessary performance numbers. After the design-space exploration is over, final system integration with all the synthesized VHDL modules is performed and a FPGA prototype is formed.

4. LEON-MP: A PARAMETERIZED SCALABLE MULTIPROCESSOR SYSTEM

Let us now look at the hardware portion of our prototyping system in more detail. Leon-MP, the hardware component of SMPS, is a tightly-coupled shared-memory shared address-space multiprocessor system. It is based on the open source Sparc V8 compliant Leon uniprocessor core. LEON2 uniprocessor [13] is a synthesizable VHDL model of a 32-bit RISC processor compliant with the SPARC-V8 Instruction Set Architecture. The integer unit consists of a 5-stage pipeline, and there are separate instruction and data caches. Full source code is available under the GNU LGPL license, allowing free and unlimited use in both research and commercial applications. The model is highly configurable, and particularly suitable for system-on-a-chip (SOC) designs. Extending this work to develop Leon-MP for SMPS, however, presented its own set of multiprocessor-related issues and challenges. Here we present the design and features of Leon-MP and try to capture the major issues encountered during its development. We also describe the major customization opportunities offered by our system.

4.1 Design of the Leon-MP System

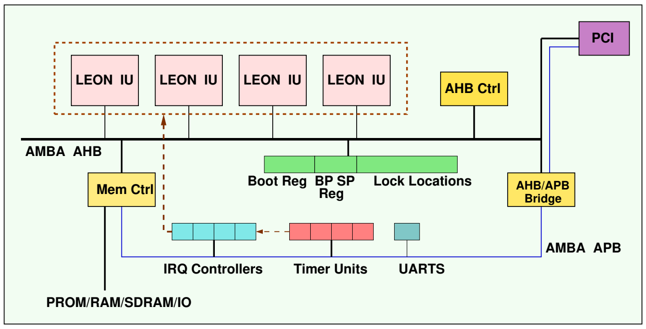

Figure 3: Leon-MP system architecture

Figure 3 shows the block diagram of Leon-MP system. It is a tightly coupled multiprocessor system with a a variable number of Leon[14] processors attached to the shared AHB bus. It provides two snoopy protocols mechanisms for cache coherence, invalidate and update. Separate IRQ controllers and timers are associated with each processor to provide variable timer interruption frequency and reside on APB bus. Also, these IRQ controllers can be used for interfacing with external peripherals. The timers and IRQ controller configuration registers are accessible at unique locations on the APB bus. Also, the application designer may choose to deliver certain interrupts to certain processors only. All processors share the two UARTs which provide communication with external world. The multiprocessors currently does not contain any memory management unit (MMU). Access to memory is managed through a memory controller. Also, there is debug support on the system, and one can attach a (optional) Debug Support Unit (DSU) to any Leon IU, which allows non-intrusive debugging on the testbed. It can communicate with an outside debugger such as *gdb*.

Some registers are added to the AHB bus as slaves. Two of these, the Boot Register, and the Boot Processor Stack Pointer Register, are used for synchronization during the booting process. Slave processors wait for the contents of these registers to be written by the Boot Processor so that they can start operation. Some other registers also reside on the AHB, referred to as the Lock Locations. These are non-cacheable memory locations and are used to implement atomic locks. Atomic instructions such as *swap* and *ldstub* are guaranteed to behave properly when using these memory locations. These lock locations have high-speed, low-latency access, as they reside on the AHB. They can be used to implement the semaphore operations in Operating Systems. Each processor is assigned some $CPU\ Id$ by means of a number hard wired in a reserved field of a CPU register. Based on these Ids, one of the processors is designated as the $Boot\ Processor$ (BP), which takes control during the booting process, after which it functions as a normal processor. Rest of the processors act as $Application\ Processors$ (AP).

4.2 Issues and Challenges in developing Leon-MP

There were a number of challenges and issues encountered in the design and development of Leon-MP system. Our design decisions were motivated by high performancee as well as flexibility and ease of customizability. Simple protocols were chosen since they consume less logic and in such a constrained environment, they are empirically found to perform well. Here are some of the major issues encountered:

Booting: As soon as the processors are given a reset (external input), they start executing from absolute location 0x0. The boot code residing at this location recognizes each processor separately using a unique processor ID which is hardcoded during synthesis inside each processors processor status register (PSR). While all the processors with IDs other than zero (slaves) go into a loop, the processor with ID zero (boot processor) performs a number of tasks. The slaves busy wait on a register synthesized onto the AHB bus, this is shown in Figure 3 as boot register. The boot processor is responsible for initialization of peripherals, detecting the on-chip memory and finally waiting for a program to be loaded into memory using the serial port. Once the program is loaded into memory the boot processor stores start address of the program in the boot register, and stores its stack pointer in the stack pointer register, which is shown in Figure 3, as *BP SP Reg*. As soon as the slaves see a non-zero value into the boot register, they read the *BP SP Reg.* and set their own stack pointer using the following formula: $Stack\ Pointer = BP\ SP\ Reg. - 20 * 1024 * Processor\ ID$. This provides an initial separation of 20KBytes to each of the processor'ss stack pointers. Having set their stack pointer, the processors jump into the code location pointed to by the boot register.

Synchronization: An interesting feature of the testbed is Lock Locations. These are five registers which are sitting on the AHB bus and provide one cycle access for both read and write. These locations are used by the RTOS running on this setup to provide mutual exclusion (mutexing) to threads using semaphores. During the course of development of this testbed, we realized the high logic area required for maintaining hardware cache coherence. In this testbed, we have SRAMs acting as the shared memory. These provide three cycle standard read and write access and 2 cycle burst mode read and write access. In view of this, all the caches are write through. To maintain cache coherence all that is required is to update the values in caches while data is being propagated through the shared AHB bus. However, to avoid race conditions during mutexing, high amount of additional logic is required which makes little sense for Embedded Systems. An empirical study by us revealed that the Embedded Multiprocessor applications make use of RTOS provided mechanisms to achieve mutexing. Moreover, the number of locks required by RTOS to achieve this is not high. In view of the above discussion, lock locations are a meaningful addition to such multiprocessor Embedded Systems.

Cache Coherence: For reasons identified above, the current system has the implementation of a very simple protocol, which is based on a write-through mechanism. We give the choice of having an invalidate or an update protocol. This protocol is justified since the cache size is very small, and moreover, the penalty of fetching from the main memory is not too large (3 cycle normal and 2 cycle burst). Future directions could include implementation of more complex protocols with many more states such as the Berkeley protocol and Illinois protocol. Here, however, we would need to weigh the benefits obtained against the cost of extra logic consumed.

Bus Arbitration: A round-robin arbitration policy is implemented. Initially the bus is allotted to processor with highest index (3 in case of quad processor) if it is requesting the bus. Once this processor releases the bus, then it is checked whether there are any requests from the just next priority processor (index 2 here), followed by processor with index 1 and subsequently with processor 0. After servicing the processor with index 0, it again starts with processor with index 3 and this keeps repeating.

Testing and Debugging: The VHDL model of the LEON processor comes along with a SPARC disassembler in the *DEBUG* package. It is used by the test bench to disassemble the executed instructions and print them to stdout (if enabled). Thus, as the VHDL model of the processor executes instructions, the SPARC disassembler implemented in the debug package pops up the instruction being executed over the simulator console. However, in case of multiprocessor, one also needs to know the processor on which it is being executed. This information is important for understanding the flow of instructions on different processors and the effect of the arbitration policy on bus allocation. To achieve this, we have extended the disassembler to suffix each instruction with the processor number on which it is being executed. This can be used to study the flow of events in the system.

4.3 Customization Opportunities in Leon-MP

Leon-MP offers numerous architecture customization opportunities. Using a GUI, we can select a number of parameters such as the number of CPUs, cache sizes of the different processors, whether to add a hardware multiplier/divider unit to the processors. The GUI can also be used to select the desired bus arbitration policy, as also the choice of cache coherency protocols between write-invalidate and write-update protocols. In addition, the testbed contains many components which can be programmed at run-time to achieve heterogeneous configurations. There are separate timers for each of the processors and these can be configured to have separate interruption rates for individual processors. Thus, application portions which need high throughput can have slow interruption rates and big time slices, whereas components which need to respond to stimuli fast can have high interruption rates. Separate programmable IRQ controllers allow the ability to select the type of interrupts that are to be delivered to a particular processor. For example, one processor can service all the I/O interrupts, another can service network, etc. Since the IRQ controllers are memory-mapped, processors can also interrupt each other by means of Inter-Processor Interrupts. Also, IRQ controllers can be simplified based on the types of interrupts they are going to service.

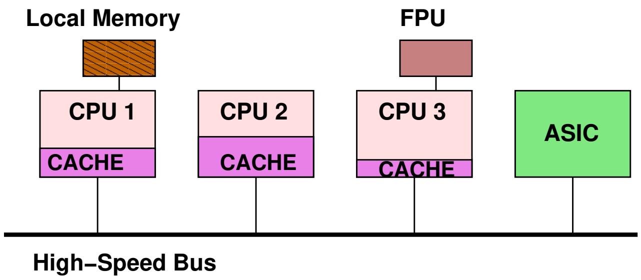

Figure 4: A Heterogeneous Multiprocessor Configuration

Leon-MP is easy to extend by adding custom policies and components. For instance, user-provided ASIC modules can be easily added on the bus for hardware portions of the system. Figure 4 shows an example of a configuration that can be achieved with Leon-MP. The application has a hardware portion which is implemented as an ASIC and is supported on the high-speed bus as a master. Besides this, there are three processor on the bus. These processors are customized for the software components mapped onto them. One of the processors has floating-point intensive code mapped onto it and hence is enhanced by adding a Floating Point Unit (FPU) coprocessor. Another processor has a local memory attached to it to reduce traffic on the global shared memory. Currently, we are working on adding support for different functional units and local memories for individual processors in Leon-MP, as well as develop a library of hardware components that can be added onto the system.

5. RTKER-MP: A PARAMETRIZED RTOS FOR LEON-MP

The Leon-MP system, like any typical embedded environment, is highly constrained in terms of memory and processing resources. For the type of applications being targeted, individual tasks would typically have timing and other constraints to meet. A light-weight real-time operating system (RTOS) is required for real-time task scheduling and for management of shared resources among the competing tasks. Such an RTOS would be minimal by providing support only for the basic primitive operations. For additional features required such as device support and file systems, custom components would be added on a per application basis.

5.1 Need for Our Own RTOS

In the Srijan framework, there are further opportunities to customize RTOS for an application. Since the partitioning of tasks among processors is static, with no global migration allowed, there is scope for different local scheduling policies in each partition rather than a common global scheduling policy. A partition could possibly consist of several processors. Further, parameters which characterize these scheduling policies, such as time slice, may be customized for each scheduling policy selected. An intelligent partitioning algorithm would, based on the scheduling model, group together related tasks which would perform better under a common scheduling policy on an appropriate computation unit. Also, given the partitioning and architecture information, scheduling estimates can be generated. If constraints are not met with the help of previous output of partitioner, these can be used to refine the partitioning. Refinements would try to generate better schedules. This iterative refinement might be carried out till the constraints are met.

We needed an RTOS that is able to exploit all the above peculiarities of our framework efficiently, besides being suitable for the above refinement process. Flexibility in the choice of components as well as scheduling and migration policies was also required. Due to these considerations we developed RtKer-MP, which is a home-grown multiprocessor RTOS for SMPS.

5.2 Design of RtKer-MP

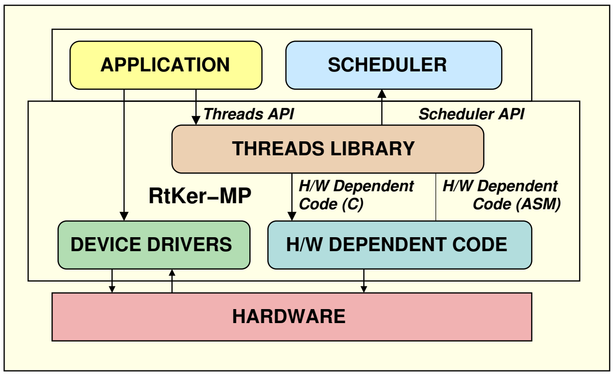

Figure 5: RtKer-MP Organization

Figure 5 shows the organization of RtKer-MP. RtKer-MP is a light-weight kernel with support for threads, semaphores and real-time interrupt handling. The hardware-dependent code consists of booting, context-switch, interrupt handling and C library code. RtKer-MP either provides minimal C library (printf/malloc) for a given architecture, or uses a minimal library from some other source (e.g. newlib). For the Leon-MP system, a minimal C library is available and is used. The next few sections describe the salient features of RtKer-MP and issues in design and implementation.

5.3 Scheduling in RtKer-MP

As figure 5 shows, scheduler is in application space in RtKer-MP. The scheduler code has been modularly separated from the main kernel through a clean and well documented API. This allows the application developer to exploit the peculiarities of an application to write and plug in his/her custom scheduler conforming with the *Scheduler API*, thus optimizing on algorithms and data structures used for scheduling. Or else, the developer can choose from a predefined set of schedulers like fifo, edf, priority, etc. The functions in the *Scheduler API*, along with some other structures, form the scheduler object, which needs to be registered with kernel during initialization. A developer need not know the implementation details of kernel to implement a scheduler, the Scheduler API is all that is required, is provided and is well documented.

RtKer-MP schedules threads using the scheduler object which has functions like *create_thread*(), *heir_thread*() and *init*(). The scheduler is responsible for maintenance of task queues, task counts, and for updating the information on execution times and deadlines. Also, priority assignment and inheritance is also the responsibility of scheduler. In other words, it is the scheduler which really determines the real-time behaviour of kernel. For instance, if it is a hard real-time system, then the scheduler might internally invoke an admission test on the task before including the task into its queues. Or, it might invoke the recovery task, the pointer to which then needs to be passed by the application coder in the task structure. In this manner, RtKer-MP achieves a clean separation between policy and features.

Since it is an RTOS for multiprocessor systems, there are separate scheduler objects for each CPU. A global data structure contains pointers to the individual scheduler objects. Task partitions consisting of multiple CPUs can be created by having their pointers point to the same scheduler object. Also, associated with each partition is a *sched_lock*, which is required for mutual exclusion in access to partition data structures.

5.4 Important Issues in developing RtKer-MP

While scheduling was an important issue, there were numerous other issues and challenges in developing RtKer-MP. This section describes some of the important ones in detail.

Initialization: During hardware bootup, the boot processor (BP), before freeing the Application Processors, initializes some of the lock locations to non-zero values to indicate that they are not free. Then, the BP jumps to the code location. Application processors (APs), also having been freed, jump to the code location. The first part of the code executed by all the processors makes a call to the initialization function, *rtker_init*. In this function, the BP initializes common structures, interrupts, trap handlers, etc. In the meantime, the APs wait on the lock locations described above, the control being implicitly with BP. After BP completes its exclusive portion, it releases the lock. Then the APs get the lock one by one, do the processor-specific initializations through the function *thread_lib_init*, which initializes, among other things, the scheduler structures.

After all the initializations have been made, BP returns back to the application code. APs, on the other hand, make a call to *suspend_thread*, and the scheduler for each schedules the *idle_thread* for that particular CPU. They idle away, till there is a thread ready to be scheduled on that CPU. The BP, which returns to the application code, then is expected to create all the threads of the application, which can then subsequently be scheduled on some CPU in their designated partition.

Synchronization: As explained in [8], for any kernel to function in a proper way, it has to provide internal locking and protection of its own structures. This would, for example, prevent two processes from modifying the kernel data structures concurrently, something which could lead to inconsistency in those structures. Another situation that could arise in the absence of locking could be allocation of the same memory block to two processes which make simultaneous requests. Such issues are particularly important in the case of multiprocessor systems sharing data structures, since multiple copies of the OS might be trying to access and/or modify these structures concurrently.

In RtKer-MP, we use the technique of fine grained locking to reduce the amount of time locks are held and reduce the critical latency times so as to provide real-time behaviour. We have separate locks for unrelated structures. Thus, for instance, separate locks protect scheduling and semaphore structures. Also, separate locks protect the scheduler objects of different partitions. These locks protect the critical sections of the RTOS such as the sections where the next thread to be scheduled is obtained from the scheduler object, the semaphore queue operations, etc.

Testing and Debugging: The testing and debugging environment was constrained by the absence of a simulator for our multiprocessor platform. The available simulator from Gaisler research is for uniprocessor LEON, and there too, only the binary is available. Development of the simulator is a work that is currently in progress in our group. Our environment, therefore, was limited to the uniprocessor simulator, and simulation of the VHDL code for the hardware as described in section 4.2. Using that simulation approach, one can simulate a code on the architecture. The constraints with this approach are two-fold: (a) the simulation is very slow as compared to any instruction set simulator, and (b) the only information it gives is the instruction sequence executed on the processor. Due to the speed of simulation, one can only simulate small pieces of code on the VHDL model of the processor.

5.5 Customization Opportunities in RtKer-MP

This scheduler API design gives a lot of flexibility in terms of scheduling and task migration policies. As described in [7], there are three categories of scheduling algorithms based on the task migration strategy, i.e. Full migration, Restricted migration, and No migration(partitioned). Our scheduler framework allows implementation of all these three categories of algorithms. For instance, for full migration, all pointers to scheduler objects point to the same structure, allowing a global task queue and a global scheduling policy. Similarly, no migration can be implemented by having separate individual pointers to scheduler object per processor. Thus, a task can be scheduled only on the processor to which it is bound, even if some other processor is idle and this task is ready to be scheduled. Moreover, the pluggable scheduler interface allows for quick exploration of various scheduling policies for a feedback to the estimation systems.

6. IMPLEMENTATION PLATFORM

In section 3 we described the high-level design methodology for developing application-specific multiprocessor embedded systems. Subsequently we described Leon-MP and RtKer-MP, the important components of SMPS. Now we describe how the synthesis methodology maps on to SMPS and the role of different components. For this, we first describe the setup where actual prototyping is done. Next we detail the development cycle for this system. The important issue of performance monitoring and how communication is done are also addressed here.

6.1 Prototyping Setup

Our hardware prototyping platform is based on an FPGA board from Alpha Data Parallel Company, ADM-XRC-II [2]. The ADM-XRC-II is a high performance reconfigurable PMC (PCI Mezzanine Card) based on the Xilinx Virtex-II range of Platform FPGAs. Its features include high speed PCI interface, external memory, high density I/O, programmable clocks, temperature monitoring, battery backed encryption and flash boot facilities. It comes with a comprehensive cross platform API, and supports various operating systems. This board contains a Xilinx XC2V3000-FF1152-5 device. The card itself is sitting inside the PC box and is connected to the PCI bus using a PCI to local bus converter. Some pins of the FPGA are available externally for interfacing.

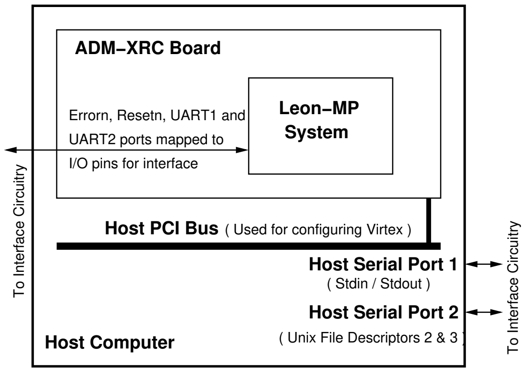

Figure 6: Hardware Layout of Leon-MP

Figure 6 shows a block diagram of ADM-XRC-II board with external connectivity. It also shows which part of the multiprocessor is mapped where. The external hardware interface circuitry contains error and power LEDs along with reset switch and voltage level conversion circuitry. The voltage conversion circuitry is required as the FPGA operates on LVTTL logic (0 to 3.3volts) whereas the RS-232C operates on different range (-12 to +12volts). Multiprocessor UARTs are mapped to host UARTs, which are used for program loading and communication. This board connects the FPGA directly to the local bus and enables its operation. The ADM-XRCII board has high performance PCI and DMA controllers, local bus, RAM banks, user front panel and rear panel connectors. Downloading of bit file is done through the PCI interface by which the board is connected to the host computer.

The interface board takes signals from Leon-MP synthesized on ADM-XRCII board. The error pin of Leon-MP is mapped to the Error LED on the interface board and it glows when the error signal is asserted by Leon. Leon's external Parallel I/O pins are connected to the PCs UARTs through an interface circuitry for the necessary voltage conversion from ADM-XRCII's CMOS/TTL level to the RS-232 compliant levels of the PC UARTs. A MAX-232 chip does the necessary voltage-conversion for ADM-XRCII inputs/outputs to RS232 PC interface. In this manner, UART signals of the testbed as shown in figure 3 are mapped to serial ports of the host. These are used to load application code onto the testbed and to do standard input/output with the running system.

Figure 7: Actual System Prototype

Figure 7 shows a photograph of the hardware setup. The ADM-XRC-II board is visible inside the CPU box of the computer. External wires coming out of this board connect to the external hardware interfacing circuitry, shown along side the power supply. Oscilloscope connected to the probe points can be used to observe the serial data.

6.2 Application development and Loading

As described in section 3, using various exploration, estimation and partitioning tools, the partitioning of tasks between hardware and software, and of software tasks among procesors is decided. Architecture customization to be carried out is also identified. For the hardware portions, ASICs can be developed and added onto the system bus in Leon-MP or chosen from a library if it is available. CPU customization can be carried out in Leon-MP through a GUI configurator as described earlier. If the desired option is not supported, such as a custom FU, user-provided modules can be integrated with the system.

Figure 8 illustrates the software development flow for the prototyping system. The multi-threaded application code, compliant with RtKer-MP API, and written in the C programming language is compiled with the help of the LECCS

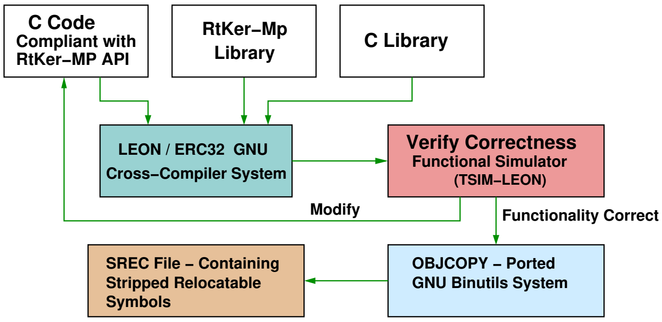

Figure 8: Software Development Flow

[14] toolkit which is a set of cross-compilation utilities for Leon Processor. The application code is responsible for calling the RTOS initialization functions with proper parameters, create the appropriate partitions and register corresponding scheduler structures with the kernel. LECCS allows cross-compilation of C and C++ applications for LEON. The compiled code is linked against the RtKer-MP library for implementations of the RTOS API functions. It is further linked with the Rtems Library to obtain the binary code. Verification is done to establish the functional correctness of this application. This could be done with the help of a functional simulator such as TSIM [14] or a multiprocessor simulator.

Having verified the application, the binary is converted to the S-Record format next. This format contains the Stripped Relocatable Symbols, with the help of *objcopy* binary utility, and is ready to be loaded onto the actual hardware. Motorola S-Records are an industry-standard format for transmitting binary files to target systems. S-Record follow the standard

$$S < type >< length >< address >< data \ldots >< checksum >$$

for each line in the file. In this form, the application is ready to be loaded onto the multiprocessor system. Once the bit file containing the synthesized Leon-MP is loaded onto the ADM-XRC card, the system boots up as described in section 4.2. After booting up, Leon-MP waits for this SREC file to be downloaded through the UART connection. Once this file is downloaded, the RTOS booting follows as detailed in section 5.4 and normal execution of application begins.

6.3 Monitoring and Communication

As described earlier, Leon's external Parallel I/O pins are connected to the host PC's UARTs. In this manner, communication can be achieved between the host PC and the system prototype running on FPGA. This can be used for I/O as well as performance monitoring. For our purposes, this feature is exploited by the C library, which maps the Stdin/Stdout I/O descriptors of applications running on the prototype to the UARTS of host PC. A debug monior, Dsumon, can also be attached to one of the processors and can be used for remote debugging. One of the future directions is integrating the network interface with Leon-MP, which would remove the communication bottleneck enforced by the serial connection.

7. RESULTS

Snoopy Policy	H/W Mul-Div	Bus Arbit.	LUT (%)	FF (%)	Clk. (ns)
Inval.	No	Prior.	29.7	10.3	21.8
Inval.	No	RR	30.1	10.3	20.8
Inval.	Yes	Prior.	39.0	11.5	22.0
Inval.	Yes	RR	39.9	11.5	21.8
Update	No	Prior.	29.9	10.3	22.9
Update	No	RR	29.8	10.3	22.8
Update	Yes	Prior.	39.9	11.5	21.4
Update	Yes	RR	40.2	11.5	21.4

Table 1: Synthesis Results for Two Processor Configuration

Snoopy Policy	Cache Size	Bus Arbit.	LUT (%)	FF (%)	Clk. (ns)
Inval.	1K+1K	RR	55.3	17.5	22.4
Inval.	2K+2K	RR	57.4	18.1	23.3
Updt.	1K+1K	RR	56.2	17.5	22.8
Updt.	2K+2K	RR	57.7	18.0	24.0

Table 2: Synthesis Results for Four Processor Configuration

Tables 1 and 2 show the synthesis results. The column *Snoopy Policy* shows the cache coherence policy implemented, which could be either Update or Invalidate. The column *H/W Mul-Div*, shows whether a hardware multiple and divider have been synthesized or not. Column *Bus Arbitration* shows the bus arbitration mechanism, which could be either round robin or priority based. Columns *LUT* and *FF* show the LUTs and Flip-flops used respectively. Column *Clk.* shows the achieved clock period. All these options are configurable using a Graphical User Interface (GUI). For four processors and beyond a simple static priority based bus arbitration does not give the bus access to processors two and three, so the only available option is the round robin bus arbitration. From the results it is evident that the clock period does not vary much with an increase in the number of processors. This is due to the fact that the critical path is not through the AHB bus.

8. CONCLUSIONS AND FUTURE WORK

We have successfully created a multiprocessor system which can be used to prototype and validate application-specific heterogeneous shared memory shared address space multiprocessor Embedded Systems. Numerous capabilities are provided for architecture customization and many other possibilities that are not provided can also be developed with reasonable effort. We have synthesized and evaluated different configurations which prove that the whole setup is highly retargetable. The current setup is being used in our group for multiprocessor application development. We have already developed a multiprocessor RTOS using the lock location feature of this multiprocessor system. The complete VHDL model along with this home grown RTOS will be available from the group website towards the end of November, 2004.

This work has many possible extensions. Support for different types of co-processors can be added for the Leon Processor so that exploration space can be increased. Additional bus arbitration mechanisms can be added to support a wider design space. A trade off between write-back cache coherency protocols and the simple write-through implementation could be evaluated as a possible alternative. Our current work-in-progress is in the direction of adding network connectivity to the multiprocessor for better data throughput.

9. REFERENCES

[1] Srijan: A Methodology for Synthesis of ASIP based Multiprocessor SoCs. Technical report, Department of Computer Science and Engineering, Indian Institute of Technology Delhi.

[2] Alpha Data. ADM-XRC-II Xilinx Virtex-II PMC. http://www.alpha-data.com/adm-xrc-ii.html.

[3] ARM Limited. MPCore: A synthesizable multiprocessor core. http://www.arm.com.

[4] H. Aydin and Q. Yang. Energy-Aware Partitioning for Multiprocessor Real-Time Systems. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS'03)*, Apr. 2003.

[5] A. Baghdadi, D. Lyonnard, N. E. Zergainoh, and A. A. Jerraya. An Efficient Architecture Model for Systematic Design of Application-Specific Multiprocessor SoC. In *Proc. DATE*, 2001.

[6] L. Barroso, S. Iman, J. Jeong, K. ner, K. Ramamurthy, and M. Dubois. RPM: A Rapid Prototyping Engine for Multiprocessor Systems. *IEEE Computer*, 28(2):260–34, Feb. 1995.

[7] J. Carpenter, P. H. S. Funk, A. Srinivasan, J. Anderson, and S. Baruah. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis.* CRC Press, 2004.

[8] . A. Cox. An Implementation Of Multiprocessor Linux. http://parallel.rz.uni-mannheim.de/Linux/smp/smp.html.

[9] CRADLE Technologies. MDSP: Cradle multiprocessor DSP platform. http://www.cradle.com.

[10] E. A. de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. International Symposium on System Synthesis (ISSS-2002)*, Oct. 2002.

[11] B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Automatic Synthesis of System on Chip Multiprocessor Architectures for Process Networks. In *Proc. Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2004), Stockholm, Sweden*, Sept. 2004.

[12] Freescale Semiconductor. MPC8641D: Freescale dual powerPC core. http://www.freescale.com.

[13] J. Gaisler. LEON: A Sparc V8 Compliant Soft Uniprocessor Core. http://www.gaisler.com/leon.html.

[14] J. Gaisler. The LEON/ERC32 GNU Cross-Compiler System. http://www.gaisler.com/leon.html.

[15] R. Janka and L. Willis. Early System-Level Design Exploration of Large DSP Systems Targeted for Real-Time Embedded COTS Multiprocessors. In *Prof. Intl. Conf. Signal Processing Applications and Technology and DSP World Workshops (DSP WorldICSPAT)*, 1999.

[16] R. S. Janka, L. M. Wills, and L. B. Baumstark. Virtual Benchmarking and Model Continuity in Prototyping Embedded Multiprocessor Signal Processing

Systems. *IEEE Transactions on Software Engineering*, 28(9):832–846, Sept. 2002.

[17] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP Congress 74*. North Holland Publishing Co, 1974.

[18] D. Lewis, D. Galloway, M. Ierssel, J. Rose, and P. Chow. The transmogrifier-2: A 1-million gate rapid prototyping system, 1997.

[19] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya. Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip. In *Proc. DAC*, 2001.

[20] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu. Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management. In *Proc. of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 48–53. ACM, Sept. 2004.

[21] M. Ruppa, A. Burgb, and E. Beckc. Rapid prototyping for wireless designs: the five-ones approach. *Signal Processing*, 83:1427–1444, 2003.

[22] D. Shin and J. Kim. Power-Aware Scheduling of Conditional Task Graphs in Real-Time Multiprocessor Systems. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED-2003)*, pages 408–413, Aug. 2003.