Multicore architectures, Part 5 - Programming challenges

Frank Schirrmeister, Cadence Design Systems - July 30, 2013

Editor's Note: Multicore architectures find use across a diverse range of applications thanks to their performance and efficiency. By combining several general-purpose MCU cores -- or MCU cores and specialized cores such as DSPs -- IC manufacturers can deliver devices well tuned to specific application requirements. Real World Multicore Embedded Systems brings together specialists offering the latest thinking on each facet of a multicore architecture. This excerpt offers an in-depth review and discussion on the nature of multicore architectures themselves. In Part 1, the author led us through a thoughtful review of the key motivations behind those architectures. Part 2 offered a review of the key characteristics of multicore architectures. Part 3 discussed how processors communicate with each other and with memory. Part 4 examined how hardware architectures are tuned to the application. This installment describes multicore programming challenges.

Adapted from "Real World Multicore Embedded Systems, 1st Edition", B Moyer, Editor (Newnes)

Programming challenges
Programming challenges are biggest when the application and the implementation architecture are orthogonal. Application-specific architectures offer little opportunity for software to be a contributing factor in overall system performance and optimization (once the architecture is decided on). With a fixed software layout in an application-specific architecture, software issues become one of validation.

Programming challenges can be severe when compile-time scheduling of tasks is used instead of run-time scheduling. When build-time configuration is needed for allocation of tasks to processors, automation of optimization becomes very desirable in complex systems. If the apportioning is performed when the application is decomposed or functions are coded, users need analysis tools to guide their decisions. If the scheduling is done at run time, then the operating system or scheduler assigns tasks to processors dynamically and may move tasks around to balance the load. The programming challenge is then focused on finding and expressing enough parallelism in the application.

The MPSoC user wants to develop or optimize an application for an MPSoC. However, in order to ensure the feasibility of meeting performance, constraints must be checked and the power consumption optimized, which requires the exploration of various options of partitioning the parallel software.
The MPSoC designer wants to ensure that selected applications can be run at the required performance and efficiency. This involves programming the computation-intensive parts of the applications and verification of MPSoC performance over a range of representative applications.

In each scenario, both the MPSoC user and designer need MPSoC simulation, debug and analysis. In addition, parallel programming techniques for the software need to be available in combination with efficient automation of mapping parallel software to parallel hardware. The degree of integration and the execution speed both determine the productivity achieved by users.

**Application characteristics**
The characteristics of the actual application running on the multicore processor influence the specific types of programming challenges. There are several issues of interest:

- Applications can be homogeneous or heterogeneous. Data parallel applications can have multiple channels of audio/video or multiple network streams. Task parallel applications can be pipelined to meet performance (video codec, software radio) or can have a mix of task types, like audio and video, radio and control, or packet routing and encryption and traffic shaping and filtering. Data parallel, homogeneous applications are likely to have a simple fixed layout, in which all CPUs do the same task or are scheduled at run time. The programming challenge is, in this case, limited to finding suitable parallelization in the application, as the actual mapping to the hardware architecture is straightforward.

- The computational demand of applications determines whether an application can be scheduled at compile time or needs to be scheduled at run time. Totally irregular applications will likely use run-time scheduling. Fixed demands normally imply a static layout and schedule, in which case parallelization of the application and the decision as to which tasks to run on which processor are done at compile time. Fixed computational demand is required for software radio and encryption algorithms; audio/video codecs are irregular but bounded (which means designers can work with worst cases). General-purpose computing with web browsers and user interfaces is completely irregular.

- The way data flows through the system determines its dependency on external memory performance and cache efficiency. When data flow is completely managed by the application using DMA engines, specific communication using FIFOs, or local memories then the influence of caches on performance diminishes. If, however, the application uses only one large shared memory, then the cache serves effectively as a run-time scheduler for data and will impact the performance of inefficiently written software. In cases where data flow is totally irregular, often the only real option for a designer is to use a cache and let the system optimize at run time. If data flow is predetermined and fixed, like in some DSP applications, then the data flow can be statically allocated to FIFOs and local memories and scheduled. Often applications require a mixture: data streaming through the system with fixed rates (uncompressed multimedia streams, packets, wireless data) combined with fixed topologies and variable but bounded rates (compressed data) and variable irregular movement (system management functions, search algorithms). The question of how much data movement is optimized at design time and how much is decided on at run time, relying on caches and bus arbiters, has a profound impact on system performance and how to do the programming.

**MPSoC analysis, debug and verification**
In the majority of projects today, once the real hardware is available, the verification of software is finished by connecting single-core-focused debuggers via JTAG to development boards. Sometimes prototype boards are used in which FPGAs represent the ASIC or ASSP currently under development. More recently, designers have been able to use virtual prototypes utilizing simulation of the processor and its peripherals either in software or using dedicated hardware accelerators. All
Software verification on real hardware is only available late in the design flow and offers limited ability to “see” into the hardware. This approach does not normally take into account turnaround time in cases when defects are found that can only be fixed with a hardware change.

Prototype boards are available earlier than the real hardware, but they require the design team to maintain several code bases of the design - one for the FPGAs used as a prototype and one for the real ASIC/ASSP used later. This approach also makes it difficult to achieve proper visibility into the hardware design to enable efficient debug.

Virtual prototypes, either in software or using hardware acceleration, are available earliest in the design flow and offer the best visibility into the design, but they often represent an abstraction and, as such, are not “the real thing”. This approach runs the risk that either defects are found that do not exist in the real implementation, or defects of the real implementation are not found because the more abstract representation did not allow it. Within this category there are significant differences between the time when the virtual prototypes become available and their speed. Often, abstract software processor models can be available long before RTL is verified, and they can be reasonably fast (of the order of 10 s of MIPS). However, users typically pay for this advantage by having to sacrifice some accuracy of the model. When cycle accuracy is required, models typically are available not long before the RTL, in which case hardware-assisted methods such as emulation become a feasible alternative.

**Shortcomings and solutions**

A fundamental shortcoming is that many solutions are single-core focused. The most pressing issues in systems running parallel software on parallel hardware require new techniques of analysis and debug. Users face issues of both functional correctness and performance. Data races, stalls, deadlocks, false sharing and memory corruption are what keep designers of multicore software awake at night (see also [2]). Multicore debug is covered in more detail in its own chapter later in this book.

**MPSoC parallel programming**

The choice, adoption and standardization of the right programming models will be a key trigger to the move of MPSoCs into mainstream computing. There are various advantages to using high-level MPSoC programming models that hide hardware complexity and enhance code longevity and portability.

Several programming models have been analyzed in projects under the MESCAL research program, including some dedicated to the INTEL IXP family of network processors and some as a subset of the MPI (message passing interface) standard. Other programming models focused on high-performance computing are OpenMP and HPF.

In the SoC world, ST microelectronics research is reporting on a project called MultiFlex to align more with the POSIX standard and CORBA (which has also been standardized on by the US DoD for future radios; JTRS, see [3]). Philips has been presenting an abstract task-level interface named TTL, following earlier work on YAPI. Another DSP-focused programming model is called StreamIt; even SystemC, with its concepts of channels and ports, could be viewed as a software programming model, but its adoption for software design is open to question.

Each of the different programming models offers specific advantages, often within specific application domains. In addition, the target architectures may affect the choice of programming model - for example, in non-uniform memory architectures (NUMA). This means there is a trade-off
between abstraction and performance: e.g., CORBA has significant overhead and may not be suitable for high performance.

Figure 3-18 outlines graphically one possible approach in which parallel tasks - the units of work in an application - communicate with each other via channels and talk to channels via ports. Various communication modes like blocking and non-blocking can be supported, and communication can be implemented in various ways depending on the platform.

![Diagram](image)

**Figure 3-18. Tasks, ports and channels in a programming model.**

**Parallel software and MPSoCs**

Based on the use of models described above, one key aspect for both MPSoC designers and MPSoC users is the ability to rapidly program a variety of different combinations of parallel software that run on parallel hardware in an automated fashion. For automation to be possible, it is essential that the descriptions of the application functionality and the hardware topology be independent of each other and that a user have the ability to define different combinations using a mapping of parallel software to parallel hardware.

This requires a description of the software architecture in combination with the parallel programming models mentioned above. If a mechanism such as that shown in Figure 3-16 is used, in which the communication structures are separated from the tasks, then a coordination language to describe the topology is required.

In addition, a description of the hardware architecture topology is required which then allows a mapping to define which elements of the software are to be executed on which resources in the hardware and which hardware/software communication mechanisms are to be used for communication between software elements.

In the hardware world, the topology of architectures can be elegantly defined using XML-based descriptions as defined in SPIRIT. In the software world, several techniques exist to express the topology of software architectures, such as those defined in UML.

Figure 3-19 illustrates this relationship. In the upper left, the topology of a video scaler application with 13 processes is shown, which are communicating via 33 channels. The lower right shows an MPSoC topology with four processors and shared memory. If they are kept independent, then different design experiments can be set up, mapping between processes in the application and the processors executing them.
Summary
The processor power crisis can be addressed by switching to more parallel hardware with multiple processors. Several variants of multi-processor architectures exist and are determined mostly by the type of processing units they use, the methods used for communication and their memory architecture.

The inevitable switch to multicore designs will cause a fundamental shift in design methodologies. However, the effects this switch will have on software programming and the ability of MPSoC designers and users to interact efficiently using design automation are not yet well understood and are likely to spawn a new generation of system design automation tools.

References


Adapted from "Real World Multicore Embedded Systems, 1st Edition," B Moyer, Editor (Newnes)

Frank Schirrmeister is Senior Director for Product Management of the System Development Suite at Cadence Design Systems. He has over 20 years of experience in IP and semiconductor design, embedded software development, hardware/software co-development and electronic design automation. He holds a MSEE (Dipl.-Ing.) from the Technical University of Berlin, Germany.