Advancing Software Development for a Multiprocessor System-on-Chip

Stephen Bique
Department of Mathematics and Computer Science, Virginia State University
Petersburg, Virgina 23806 USA

ABSTRACT

A low-level language is the right tool to develop applications for some embedded systems. Notwithstanding, a high-level language provides a proper environment to develop the programming tools. The target device is a system-on-chip consisting of an array of processors with only local communication. Applications include typical streaming applications for digital signal processing. We describe the hardware model and stress the advantages of a flexible device. We introduce IDEA, a graphical integrated development environment for an array. A proper foundation for software development is a UML and standard programming abstractions in object-oriented languages.

Keywords: multiprocessor, system-on-chip, digital signal processing, Unified Modeling Language

1. PROGRAM AT THE RIGHT LEVEL

What is the right level of programming? It is the level of the programming language (from machine language or instruction set architecture to a high-level language) that minimizes the programming effort to write a satisfactory implementation starting from a problem specification. As recommended long ago, programming should not be overly constrained by current technology [1]. To minimize the programming effort, a programmer looks for useful tools to map solutions to a programming language or a combination of software and hardware, and seeks to avoid spending time dealing with the details of a programming language implementation or adapting a solution to fit the choices made by hardware designers.

Incredible technological advances continue to bring new and improved devices to the marketplace. A programmer's toolbox includes high-level languages and useful tools such as optimizing compilers, which have proven successful. Often the programmer has a good chance to find the right tool for a given task.

What are the programmer's chances of finding the right tool? Given a little training, a uniprocessor, a dual processor, a simple embedded system or a supercomputer and a common programming task, the programmer will find the right tool with high probability. Parallelizing compilers are useful to translate sequential code to run on a parallel computer.

Despite the successes with programming uniprocessors and the advantages of architecture-independent parallel programming [12], there is no assurance that high-level languages are always the right tools. A common pitfall is expecting to get good performance from high-level languages, despite their well-known advantages in software development [4]. Increasingly, it is feasible to build an embedded system, such as an FPGA-based custom computing machine, capable of delivering unprecedented performance. Low-level programming is generally necessary to achieve peak performance for applications with regular data dependences on a multiprocessor system-on-chip (MPSoC) with only local interconnects.

Tools are urgently needed that help to do the low-level programming, which cannot always be avoided. Today a programmer will find limited support to exploit the potential parallelism for applications with regular data dependences on an MPSoC with only local interconnects and distributed memory. Even if the programmer is handed an optimal solution involving a systolic algorithm, to map that solution efficiently onto a suitable MPSoC, a programmer will not likely be able to employ any high-level language and instead must rely a small set of low-level tools, which are written specifically for that device and have limited functionality.

2. HARDWARE/SOFTWARE CO-DESIGN

Pure software solutions are not practical for some real-time compute-intensive applications. On the other extreme, custom-built hardware that cannot be reconfigured has limited applications. A sound platform lies on a bridge between these two extremes, i.e., a combination of software and general-purpose hardware. The main goals are to

- provide a flexible device,
- enhance software development,
- more easily measure performance, and
- advance the introduction of real-time compute-intensive applications.

Designers use hardware description languages (HDL) to model hardware at the register transfer level and gate level. As hardware becomes more complicated, not only does a hardware engineer need to know more about programming, but also a programmer needs to know more about hardware design. Designing hardware and software separately is increasingly costly and difficult. Hardware/software co-design is the development of the software and hardware in parallel. Co-design encompasses a wide variety of approaches depending on the hardware problem and is key design technology for digital systems [6].

An instruction set architecture (ISA) provides a good vehicle to design concurrently the hardware and compiler. The Unified Modeling Language (UML) is a standard modeling tool. UML structural diagrams are useful to describe models of parallel computers [11]. After the initial exploration of the hardware design, a UML is a proper foundation for software development.

Reconfiguration during execution could be investigated for configurable computing machines (CCMs) using only standard programming abstractions (constructors, destructors) found in object-oriented languages [2]. C++ is a suitable programming language to model the hardware and support dynamic reconfiguration. C++ models

- are useful for high-level behavioral modeling,
- add new capabilities to hardware description languages (HDL) such as VHDL and Verilog, and
- can be translated into HDL models using existing tools [7].

For some other models, existing sets of library routines such as SystemC may be incorporated into a system to enhance interfacing with hardware description languages. However, a complete set of tools should be developed freely without overly adapting the system to fit the requirements of any language or set of library routines.

3. HARDWARE MODEL

Although a general-purpose computer is designed for a wide range of applications, the target applications for an embedded system fall into a smaller set. Presumably, hardware design is based on the target applications. Reconfiguration technology offers benefits but complicates design.

The hardware model is a scalable prototype of the target device. Currently, the chosen model is

an array of processors with only local communication. The intended applications are primarily streaming applications for digital signal processing involving systolic computations such as computing matrix products via an efficient parallel algorithm in real-time.

A multiprocessor system-on-chip (MPSoC) is an integrated circuit (IC) that contains multiple instruction-set processors and that implements the functions of a complete electronic system. Heterogeneous processors could be used so that different modules would contain different types of processors as in an application-specific integrated circuit (ASIC). Heterogeneous processors are useful to support task level parallelism. However, the scope of the current investigation does not include general application-specific integrated circuits (ASICs) or general tools for parallel programming, e.g., techniques to transform a sequential program into a parallel program. While employing such techniques may be useful in some way to program the target device, developing such general methods is not highly relevant for the chosen model of computation.

The chosen class of MPSoCs includes ICs consisting of a large array of nearly identical modules with a communication network that matches the structure of the array. Each module is the same except possibly for a couple components. Each component handles data at the level of words. A typical module contains an instruction-set processor and other common objects such as registers. Some modules may contain a memory unit instead of a processor. The memory access protocol could be either first-in, first-out (FIFO) or content-addressable memory (CAM). In any case, each module is similar in terms of the layout of the components.

The communication network consists of global horizontal (rows) and vertical (columns) wires with switches. Just as FPGAs can be seen as the archetypical programmable SoCs, the objects in modules are connected via a scalable switch-based network and the routing approach is both deterministic and static. This means that every connection between objects must be determined before execution of any code involving communication between those objects. The actual route between objects is arbitrarily (not deterministically) decided subject only to the hardware limitations and the requirements of the programming solution. To avoid long propagation delays, reduce energy requirements, and improve reliability, only local objects in nearby modules may be connected. The compiler (or programmer) should check that there only a small number of interconnections exist between connected objects. Since only local connections are allowed, communication protocols or mechanisms such as channels provided by SystemC are irrelevant.

The communication network is comprised of a data and control network. Data (as words) and control (as bits) streams replace the familiar instruction sequence of the von Neumann computer. Data (including addresses) travel over a large number of dedicated wires and control information travels over separate wires. Objects in a module handle data at the level of words and control information at the level of bits.

The design choice for the operation of the MPSoC is both synchronous and asynchronous. Each module operates synchronously in the sense that a clock cycle must be the same for each module to permit systolic (i.e., highly regular and continuous) operation. A simple design strategy is to assume that one clock cycle is the maximum time needed to carry out any operation or to safely delay data or control information. Each module operates asynchronously in the sense of self-synchronization of computations.

Self-synchronization is based on static data flow principles developed by Jack Dennis [3]. Each processor waits for all inputs before processing, automatically becomes active as soon as all inputs are available, and becomes disabled if it is unable to deliver any of its outputs. Automatic synchronization simplifies optimization and verification.

The scope of our work includes the initial exploration of the hardware design. After the initial exploration is complete, a standard way to specify the hardware model is structural diagrams of the UML. Instead of building tools to design such MPSoCs, the task is to build tools based on the UML to study and program such systems.

Although the design is fixed, flexibility of the device is highly advantageous.

4. Flexibility

It is not practical to build an embedded system with more resources or components than needed. Based on knowledge and experience, a hardware engineer will make many decisions to design a board that meets the specifications. Developing software after the board is designed is expensive and development will be restrained by the decisions that the hardware designer has made.

We seek to free programming from the bounds imposed by a particular device or a programming language. This goal can be achieved by constructing a flexible device that has more resources than would be practical to place and route on an IC. Next, we briefly discuss the benefits of this strategy for the chosen hardware model.

A programmer ought to be afforded as much freedom as possible to implement different programming solutions to address difficult programming issues. I/O problems are often ignored. For example, how many I/O ports should be used? A hardware engineer might argue that only a small number of ports should be provided to reduce production costs. Arbitrarily fixing the number of ports or their location will severely limit the programming solutions. An alternative approach is to allow as many I/O ports as appropriate for the chosen hardware model so that a programmer could utilize different I/O ports.

The only true measure of performance is the total time needed to solve a problem. A fast solution that only solves part of a problem may not yield the best performance after taking into account the total time to perform the remaining parts. A more flexible device will allow the investigation of more ways to combine different parts of a solution.

One way to deal with resource conflicts is to provide extra resources. Use as many resources as needed to develop a working implementation. Develop useful tools to visually inspect and modify an implementation to reduce the number of resources used. In this way, it is possible to develop an implementation for any desired configuration. In other words, allow the number of any available resource to be specified but treat the number more as a goal rather than a prerequisite.

Routing refers to the process of defining connections between objects. Based on programming experience, routing is a time consuming task for the chosen hardware model. To achieve optimal performance for suitable applications, precise timing requirements must be met. Every connection must meet both the requirements imposed by both the hardware and the desired programming solution. The programming effort to satisfy such requirements varies inversely to the number of available resources.

In particular, using regular structures helps to achieve systolic flow. To reduce the programming effort, provide useful tools to map such regular structures to the hardware. It may be impossible to use the same repeated structures or even find any solution due to resource conflicts when the number of dataflow paths is small.

The same tools can be used to program different devices. Reusing the same tools will reduce software development costs and training expenses. New tools and enhancements will be more beneficial because the software is more widely used.

Although it will be necessary to make some choices and impose some limitations, unnecessary details and arbitrary restrictions on the physical design should be avoided. For example, allow the user to specify various parameters, such as the size of the array, instead of fixing them. Programmers can ignore many details that the hardware designer must add to the device, e.g., a programmer needs to know information at the I/O level, not necessarily at to the gate or switch level [8].

A processor in a module could be a general-purpose reduced instruction set computing (RISC) processor, an application-specific processor, a digital signal processor (DSP) or a configurable extensible processor that uses firmware [10]. Designing a processor is a task for a hardware engineer. Systolic algorithms

typically involve only a small number of instructions. Depending upon the application area, the programmer may customize the instruction set to include any small set of instructions that are needed. Reconfiguration data may be stored locally using for example extra sets of registers.

Ultimately, we seek to advance the introduction of new devices. Programming studies are needed to justify the construction of new embedded systems. The primary purpose of our work is to enhance such studies.

5. Graphical IDE

A graphical programming development environment (PDE) enhances embedded programming especially if the IC has a large area. Such a graphical integrated development environment for an array (IDEA) will include a comprehensive set of tools and permit flexibility of the device via parameterization. IDEA will be used by programmers to more quickly develop time-varying compute-intensive applications, starting from configuration of the device to testing via simulation.

A graphical PDE helps to visualize and design a program. CAD software is helpful to make at least some of the low-level definitions instead of writing every definition in a low-level language. A programmer should be able to use templates and the mouse to configure each element (drag and drop, etc.). An experienced programmer should find that the tools are relatively easy to learn and easy to use.

IDEA will be developed without dependence on a particular hardware system. This means it will be necessary to provide a clock-accurate simulator but it also means the system will be simpler and more portable without dependence on a particular machine language. This approach allows the programmer to focus on the essential computations and avoid unnecessary details of resource management.

The foundation is structural diagrams to specify the hardware model. All tools will be written in C++ or java. Using constructors and destructors will permit reconfiguration of objects. The

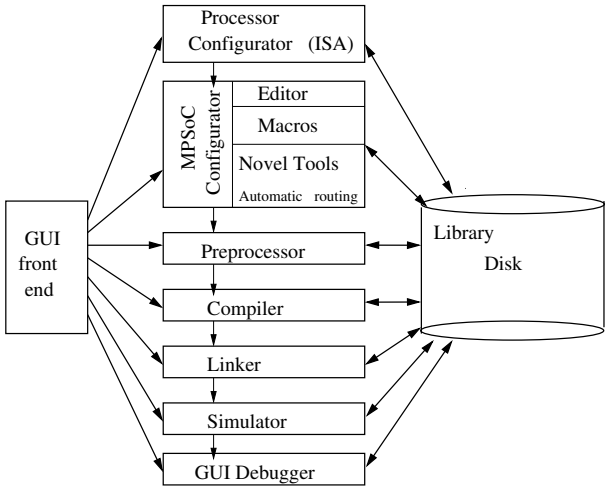

Figure 1

programming development environment (PDE) is depicted below:

Besides basic tools such as a compiler, novel tools to simplify low-level programming are needed. Tools are needed, not to eliminate low-level programming, which is necessary, but to simplify the task by alleviating the tedium. For instance, an automated routing tool is useful to make intermediate connections between objects. In addition, tools based on macros would be useful to transform definitions from one region to other regions of the device.

A programmer should be able to visualize a simulation to observe the changes in state during every cycle. A GUI debugger will be useful to visually fix programming errors in the usual edit-compile-test cycle. In addition, a programmer should be able to visually inspect the routing to improve an implementation by reducing the number of resources used. For example, if extra buses are needed in only a couple modules, try to reroute so that those extra buses are not needed.

Ideally, there exists a good match between the program and hardware structures. A sound starting point for digital signal processing is a signal flow graph (SFG) that shows both the data flow and the timing requirements [9]. A typical SFG exhibits regular structure. An array also possesses a regular structure. As an array has corners and edges, typically some adjustments are required around them. A programmer might

manually create templates by explicitly mapping a repeated "slice" of a SFG to one of more possible configurations for the device. Novel tools should be developed to automatically map a SFG to an array using such specified templates.

To compare different implementations and ultimately justify the introduction of a new device, performance metrics are useful. Such metrics can be defined and calculated in different ways. Yet metrics should not be an afterthought. Instead, metrics should be automatically computed.

6. Future Work

We proposed a methodology for a particular hardware model. A similar approach may be applied to other models. After the initial exploration of the hardware model, design a flexible prototype device and provide a UML description. Build a graphical integrated development environment to enhance software development. Write the tools in an object-oriented language such as C++ and employ standard programming abstractions (constructors, destructors).

We expect to be able to build more effective tools by taking advantage of the peculiarities of a fixed model and typical patterns of computation using that model. In addition to the essential tools to program a device, add novel tools to the PDE to simplify low-level programming. Study relevant algorithms and heuristics to build more effective tools. For a particular embedded system, the peculiarities of the system may allow a special algorithm or heuristic to yield better performance than can be expected in general.

Ultimately, the goal is demonstrate faster applications on more efficient devices. Case studies are needed to verify correctness, assess usability, and improve the tools. Future work will include studying different ways to implement dynamic reconfiguration and temporal partitioning. Large problems could be implemented on a device with insufficient resources via reconfiguration so that parallel solutions are combined sequentially [5]. An alternative approach is to use time division multiplexing, which utilizes each resource to combine more than one value.

Parallel computers remain most promising to meet the extreme computing demands beyond the limits of uniprocessors. Programming parallel computers poses unique challenges. A promising way to meet these challenges is to enhance software development for MPSoCs.

References

[1] J. Backus, *Can programming be liberated from the von neumann style? a functional style and its algebra of programs*, Communications of the ACM **21** (1978), no. 8, 613–641.

[2] P. Bellows and B. Hutchings, *Jhdl - an hdl for reconfigurable systems*, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98) (Washington, DC, USA) (Kenneth L. Pocek and Jeffrey Arnold, eds.), IEEE Computer Society Press, 1998, pp. 175–184.

[3] J.B. Dennis, *Dataflow supercomputers*, IEEE Computer Magazine **13** (1980), no. 11, 48–56.

[4] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach*, third ed., Morgan Kaufmann Publishers, 2003.

[5] Bingfeng Mei, Serge Vernalde, Hugo De Man, and Rudy Lauwereins, *Design and optimization of dynamically reconfigurable embedded systems*, 1st Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA), CSREA Press, 2001, pp. 78–84.

[6] G. De Micheli and M. Sami, *Hardware-software codesign*, 1996.

[7] D. Mitchell, *Interfacing vhdl and verilog designs to c++ models*, ECN Magazine (2002), 89.

[8] ———, *Modeling with c++*, Integrated Communications Design (iCD) (2002), 5 pages.

[9] Peter Pirsch, *Architectures for digital signal processing*, John Wiley & Sons, Inc., New York, NY, USA, 1998.

[10] Chris Rowen, *Performance and flexibility for multiple-processor soc design*, Multiprocessor System-on-Chips (Ahmed A. Jerraya and Wayne Wolf, eds.), Morgan Kaufmann Publishers, 2005, pp. 113–151.

[11] M. Scherger, J. Potter, and J. Baker, *On using the uml to describe the masc model of parallel computation.*, PDPTA (Hamid R. Arabnia, ed.), CSREA Press, 2000.

[12] D.B. Skillicorn, *Architecture-independent parallel computation*, IEEE Computer **23** (1990), no. 12, 38–51.