AC 2009-414: EDUCATIONAL GOALS FOR EMBEDDED SYSTEMS IN THE MULTICORE ERA

James Holt, Freescale Semiconductor, Inc.

Jim leads the Multicore Design Evaluation team for Freescale's NMG/NSD division. Jim has 27 years of industry experience focused on distributed systems, microprocessor and SoC architecture, design verification, and optimization. Jim is an IEEE Senior Member, and is a board member for the Multicore Association. He is also chair of the Integrated Systems & Circuits Science area for the Semiconductor Research Corporation (SRC), and chair of the Multicore Resource API Working group for the Multicore Association. Jim earned a Ph.D. in Electrical and Computer Engineering from the University of Texas at Austin, and an MS in Computer Science from Texas State University.

Hongchi Shi, Texas State University, San Marcos

Hongchi Shi is Professor and Chair of the Computer Science Department at Texas State University-San Marcos. Prior to joining Texas State University, he has been an Assistant/Associate/Full Professor of Computer Science and Electrical and Computer Engineering at the University of Missouri. He obtained his BS degree and MS degree in Computer Science and Engineering from Beijing University of Aeronautics and Astronautics in 1983 and 1986, respectively. He obtained his PhD degree in Computer and Information Sciences from the University of Florida in 1994. Hongchi Shi's research interests include parallel and distributed computing, wireless sensor networks, neural networks, and image processing. He has served on many organizing and/or technical program committees of international conferences in his research areas. He is a member of ACM and a senior member of IEEE.

Harold Stern, Texas State University, San Marcos

Harold Stern (BSEE University of Texas, Austin, MSEE and Ph.D., University of Texas, Arlington) is Director of the Ingram School of Engineering at Texas State University – San Marcos. His research interests are signal processing, wireless communication systems, and engineering education. He is co-author of an introductory-level textbook, Communication Systems Analysis and Design.

© American Society for Engineering Education, 2009

Educational Goals for Embedded Systems in the Multicore Era

Abstract

Embedded systems are becoming increasingly sophisticated. For example, today's automobiles have requirements for adaptive engine control to meet emissions and fuel-economy standards, advanced diagnostics for repair, reduction of wiring, new safety features, and new comfort and convenience features. The software required to support this feature set is quite complex and has strict performance requirements, and the hardware must operate well in extreme climate conditions with limited power resources. Yet, in order to keep materials costs low automobile manufacturers must deploy these capabilities using as few microprocessors as possible. For these reasons, the microprocessors used in automotive applications must be both high-performance and energy-efficient. These demands are at cross purposes with traditional high-performance microprocessor design, and the industry has responded with innovative embedded multicore architectures. Instead of throttling up the performance of a single processor core (which is very power intensive), a new breed of microprocessors incorporates multiple low power processor cores onto a single chip. This approach results in a higher throughput system capable of running many concurrent threads of computation in an energy-efficient fashion. These changes in the microprocessor landscape have satisfied the need for high-performance, energy-efficient processors. However, they have left engineers who are experienced only with single-core embedded systems and sequential programming languages at a disadvantage, lacking exposure to critical concepts required to design, program, debug, and optimize this new breed of embedded systems. The problem will surely be exacerbated in the not-too-distant future when (as many experts predict) the multicore era transitions to the manycore era. For these reasons, it is time to consider some new educational goals which will prepare future engineers for the multicore era and beyond. This paper discusses the essential concepts that should be included in an undergraduate computer engineering curriculum that takes embedded multicore systems into account.

Introduction

Many people believe that the incorporation of multiple cores on a single chip began in the high-end workstation and desktop market to mitigate the ever-increasing power demands of traditional performance enhancement techniques such as clock frequency scaling, increased pipeline lengths, and super-scalar instruction issue[1]. But the truth is that many embedded applications were employing multiple cores on a chip long before these high profile chip multiprocessors caught the attention of the public. For example, cell phone chips of the late 1990s commonly incorporated a general purpose applications processor core alongside a digital signal processor core.

A brief examination of the product portfolios of major semiconductor companies will show that multicore chips are becoming part of the mainstream offerings in many embedded application domains including consumer, industrial, networking, medical, military, gaming, and automotive. This change is happening very rapidly, and has caused consternation within the embedded programming community. Many in the industry believe that the concerns stem from two main

issues: (1) lack of tools and infrastructure to enable effective programming, debugging, and optimization of multicore systems, and (2) lack of programmer education on how to design and develop multicore software[2].

The industry is working to address the first issue through creation of tools, standards, and best practices[3-5]. This infrastructure is critical, and will relieve embedded programmers from the burden of crafting their own. The availability of standard infrastructure will allow programmers to focus on the essential aspects of their application and its architecture, thus saving time and money for companies.

The second issue must be addressed by educational institutions. The key obstacle is that today's computer science and computer engineering curricula are based mainly upon the sequential model of computation, only introducing concurrency and parallelism as specialized or elective topics[6-10]. In the past it was only important for a small percentage of programmers to truly grasp the issues associated with designing, implementing, and debugging parallel and concurrent systems. However, ubiquitous multicore hardware has changed this. The elective-only offering of topics that are relevant to multicore represents a fundamental disconnect with the realities of systems the industry is producing today and will continue to produce for the foreseeable future.

The need for multicore curricular reforms has been recognized by educators, but so far efforts to update curricula at various universities appear to have been limited to topics primarily related to chip multiprocessors[11-14]. This is important knowledge, but it is important that we examine the knowledge required for a broad array of multicore contexts and begin to address all aspects. The main contributions of this paper are to (1) outline the broad body of knowledge necessary for inclusion in embedded multicore curricula, (2) survey published reports on efforts to update curricula to date, and (3) suggest a set of comprehensive strategies to update curricula for embedded multicore systems.

Characteristics of Embedded Multicore Systems

Multicore systems have evolved under pressures from two forces: (1) CMOS scaling is providing the ability to produce ever higher levels of integration of cores, hardware accelerators, and input/output devices into a single cost effective package, and (2) customers want to save component costs and power consumption while achieving increasing performance. These forces have resulted in embedded multicore Systems-on-Chip (SoC) that share many features with multiprocessor and distributed systems, but are unique in some important ways. Important intrinsic and distinguishing characteristics of embedded multicore systems are summarized in Figure 1. This knowledge comprises a superset of what can be gained from studying parallel computing, distributed computing, embedded systems, and chip multiprocessing. With careful planning it can be integrated into current curricula without major disruption.

1. Hardware Characteristics

1.1. *Hardware Support for Virtualization* – processor cores are often modified to support an additional privilege level for a virtualization kernel, adding special instructions, interrupts, and registers; hardware may also support security partitions for trust and reliability

1.2. *Inter-Processor Communications & Synchronization*– processor cores have special interrupts and special assembly instructions to facilitate low latency core-to-core synchronization and simple message passing; boot sequences must be carefully engineered

1.3. *Reliable, Low Latency Interconnect* – the interconnect for embedded multicore is inherently different from traditional parallel interconnect, bus architectures, and Ethernet; tiled architecture and Network-on-Chip interconnect are becoming common, with 3D versions in the near future

1.4. *Cache & Memory Organization* – cache organization may include up to 3 levels on chip; some levels may be distributed or private; caches may have to maintain coherency for all or part of their contents; physical memory is not uniformly accessible by all cores; some cores have limited-size private memory; on-chip SRAM has limited capacity but much lower latency than DDR

1.5. *Power Management* – cores and other hardware units may be clustered on voltage/frequency islands with asynchronous signal boundaries; there may be dedicated hardware management units

1.6. *Debug Hardware* – hardware for multicore debug is becoming very sophisticated including synchronized core breakpoints, as well as tracing of cores, memory, caches, and hardware accelerator units

2. Software Characteristics

2.1. *Software Footprint and Execution Times* – traditional assumptions regarding the system context must be revisited so that unnecessary overheads can be eliminated -- given reliable interconnect, low latency interconnect, on-chip fast memories, and limited memory capacities

2.2. *Software Support for Virtualization* – interactions of hypervisor kernels and guest operating systems are complex and difficult to code and verify; software must execute more complex boot sequencing

2.3. *Software Support for Heterogeneity* – programmers, operating systems, compilers, debuggers, and software libraries must interact with multiple instruction set architectures, non-uniform memory architectures, and application specific hardware acceleration units

2.4. *Software Analysis and Implementation* – programmers must produce designs that accommodate and exploit the presence of heterogeneous cores, operating systems, memory architectures, and hardware acceleration, and must use complex and varied inter-process communications, synchronization, and shared resource management facilities, especially in the absence of an SMP operating system or physical shared memory

2.5. *Software Debug and Optimization* – programmers must understand non-deterministic intrusive debug techniques versus deterministic non-intrusive debug, the use of performance counters, and may have to compensate for the lack of a single source debugger in heterogeneous environments

Figure 1 - Embedded Multicore Systems Characteristics

Early Efforts to Integrate Multicore Concepts

Early efforts to update curricula to prepare students for the multicore era generally fall into one of two approaches. The first approach is to bolster the existing curriculum with one or more classes specifically focusing on multicore topics. The second approach is to integrate multicore topics as course modules in existing course offerings. Faculty taking the first approach report that they do so to avoid the difficulty of redesigning an entire curriculum, whereas those taking the second approach believe that it is ultimately more important to introduce these concepts early and revisit them in multiple courses.

Faculty at Shenkar College of Engineering and Design have added a mandatory course for third year B.Sc. students in the software engineering track[15]. The preliminary part of the class teaches the why concurrency and parallel computing are important, and how to reason about concurrency. This is followed by traditional material on parallel architectures, including Flynn's taxonomy, interconnection networks, cache coherency, and chip multiprocessors. The course then moves into parallel software topics including design , performance and metrics, parallel algorithms, implementation using OpenMP and MPI[16, 17], and finally operating systems support for parallelism. While this is a very useful course, it occurs very late in the curriculum, and is targeted mainly at scientific computing. Multicore-specific topics are only introduced late in the semester, and the course material is limited to distinguishing chip-multiprocessing concerns from high-performance parallel computing concerns. Unfortunately, late introduction and insufficient breadth limit the effectiveness of this approach for imparting embedded multicore knowledge.

Faculty at several Chinese Universities have taken a more extensive approach to adding multicore courses to their curricula[11], creating specific elective courses including Multicore SoC Design Technology, Parallel and Multicore Architectures, Multicore Operating Systems, Multi-threaded and Multicore Programming, Compilers and Tools for Multicore, and Performance Assessment for Multicore. This is an important set of courses covering multicore topics not included in other mandatory classes in the curriculum. Yet, the approach does not instill basic multicore knowledge into all students following the curriculum, and it does not guarantee that those students who do choose to take multicore electives will have a broad set of knowledge about multicore.

For these reasons, other Chinese universities have begun to integrate multicore knowledge into their standard curriculum, introducing the topics alongside other mandatory topics over the duration of a students tenure in the curriculum[11]. Courses that were modified include Computer Architecture, Advanced Computer Architecture, Advanced Microcomputers, Assembly Language Programming, Operating Systems, Advanced Operating Systems, Distributed Computing, Parallel Computing, and Embedded Systems. This approach has the advantage of exposing students to concepts of concurrency and parallelism in a broad fashion, although the topics are not introduced very early in the curriculum which may not be as effective at teaching students to reason about concurrency.

At the University of Wisconsin – Eau Claire the Computer Science faculty have begun to introduce concurrency very early in their curriculum[12]. They have chosen to integrate these concepts into traditional material throughout their courses, beginning with their Foundations of Computer Science course. Their strategy involves giving students practice with the concepts behind parallel programming using Java rather than any specialized parallel programming language. They first introduce students to a four step methodology for analyzing, designing, and implementing parallel software: (1) decomposition, (2) assignment, (3) orchestration, and (4) mapping. The methodology is first introduced using embarrassingly parallel problems, and as students progress through their coursework they are given increasingly more complex assignments which require this methodology. For example, first year students are assigned a simple image convolution problem (for which the algorithm is completely independent per pixel of the image). In later courses they tackle image mosaics (which requires synchronization and barriers), study concurrency limitations in algorithms (Amdahl's law), and finally operating

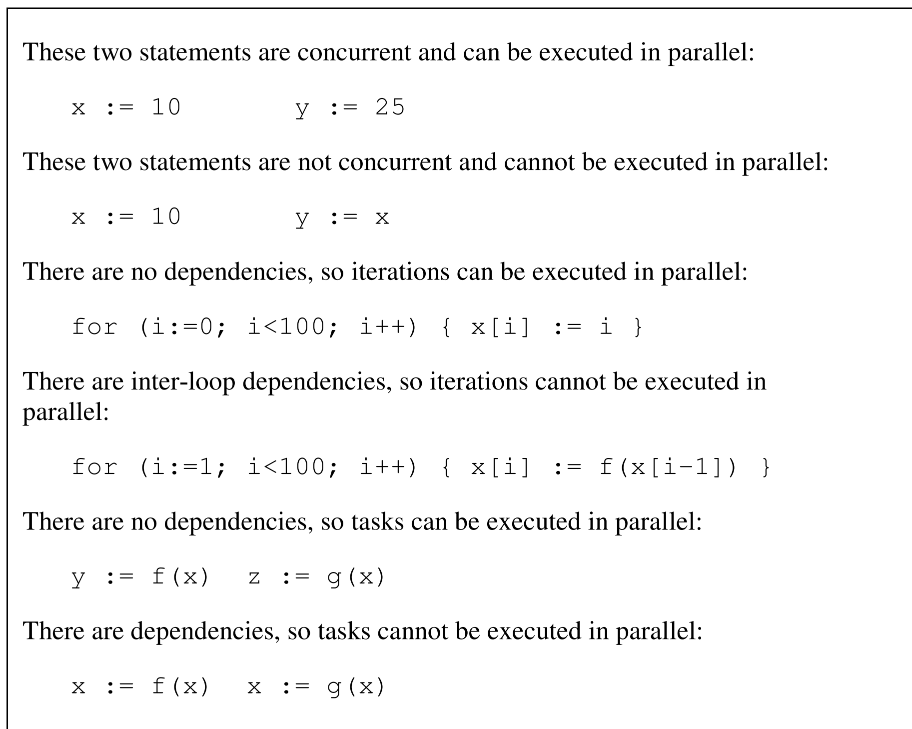
systems topics. This is a promising strategy, but when considered in the context of embedded multicore there are still some important computer architecture topics missing including memory hierarchy, coherency, and interconnect. Furthermore, some very important embedded multicore software considerations are missing, especially virtualization, compiler support for multicore and multithreading, and performance optimizations which take into account cache coherency and data layout.

Using an even more comprehensive approach, the Computer Science and Computer Engineering faculty at Georgia Institute of Technology have embraced the idea of adding new modules throughout their curriculum[13]. This was done using a "rolling introduction" approach. They began by introducing new modules into upper division courses including Systems and Architectures, Advanced Operating Systems, Parallel Computing, Distributed Computing, and Embedded and Real-time Systems. This was followed by changes to junior-level courses on Computer Architecture, Operating Systems, Processor Design, Compilers and Languages, and Applications, and then finally by changes to introductory courses. To support this further, the faculty also added a new graduate course titled Multicore Systems, which is coupled with industry guest lectures. Each course module introduced in the undergraduate curriculum is one to three weeks long, and focuses on new architectural enhancements, techniques for exploiting parallelism at the instruction and thread level, correctness issues, performance analysis and debugging, and application design and development. Perhaps the only thing missing in this approach is an undergraduate upper division integrative course which would tie all the topics together and add some depth on multicore embedded issues.

Recommendations for Embedded Multicore Curricula

The differences between distributed computing, multiprocessing, traditional embedded systems, and embedded multicore are significant, but it should be clear from examining Figure 1 that embedded multicore computing intersects many concepts from these other areas. The information outlined in Figure 1 represents a deep body of knowledge that could be an end goal for an embedded multicore curriculum in computer engineering. This new curriculum could be built by the following steps: (1) making concurrency part of the entire curriculum, (2) adding specific embedded multicore course modules to existing courses, and (3) introducing an upper division embedded multicore systems elective as an integrative course.

Reasoning about concurrency can and should be introduced as early as possible in the curriculum. For example, it does not take very much programming background to have meaningful and lively discussions, homework assignments, and programming labs built around the simple concepts depicted in Figure 2.



These two statements are concurrent and can be executed in parallel:

```
x := 10        y := 25
```

These two statements are not concurrent and cannot be executed in parallel:

```
x := 10        y := x
```

There are no dependencies, so iterations can be executed in parallel:

```
for (i:=0; i<100; i++) { x[i] := i }
```

There are inter-loop dependencies, so iterations cannot be executed in parallel:

```
for (i:=1; i<100; i++) { x[i] := f(x[i-1]) }
```

There are no dependencies, so tasks can be executed in parallel:

```
y := f(x)   z := g(x)
```

There are dependencies, so tasks cannot be executed in parallel:

```
x := f(x)   x := g(x)
```

Figure 2 - Reasoning About Concurrency

Over the duration of an entire curriculum these six basic concepts have profound implications for many domains that engineers and computer scientists might encounter including parallel algorithms, distributed systems, and multiprocessing. These implications also hold for many aspects of embedded multicore systems including hardware design (memory coherency, interconnect, cache organization, etc.), compilers (extracting loop parallelism, using speculative memory, etc.), operating systems (scheduling algorithms, cpu affinity, etc.), software analysis and design (finding and exploiting data parallelism and task parallelism), software implementation (choosing threads, OpenMP, MPI, etc.), debugging (identifying race conditions and deadlocks), and performance analysis (determining load balance and data layout within cache).

The fundamental concurrency concepts in Figure 2 are so widely applicable in today's computing landscape that they should be introduced early and often. The best way to do this is to modify existing modules in the curriculum to include aspects of concurrency alongside traditional material. But this is not enough to cover the entire breadth of *embedded Multicore* knowledge required (see Figure 1). To address this deficiency, the fundamental concurrency concepts of Figure 2 can be augmented with additional modifications to important core courses in the Computer Engineering and Computer Science Curricula. Table 1 outlines these further recommended embedded multicore curriculum changes using the nomenclature from the CE2004 report[10].

Multicore Key Knowledge	Multicore Curriculum Recommendations
1.1 Hardware Support for Virtualization	Not currently covered in CE2004; add *elective* modules to CE-CAO (Computer Architecture), CE-CSE (Computer Systems Engineering), and CE-ESY (Embedded Systems)
1.2 Inter-Processor Communications & Synchronization	Augment existing CE2004 *core* modules in CE-CAO (Computer Architecture), CE-CSE (Computer Systems Engineering), and CE-OPS (Operating Systems)
1.3 Reliable, Low-latency Interconnect	Augment existing CE2004 *core* modules in CE-CAO (Computer Architecture), CE-CSE (Computer Systems Engineering), and CE-ESY (Embedded Systems)
1.4 Cache & Memory Organization	Augment existing CE2004 *core* modules in CE-CAO (Computer Architecture), CE-CSE (Computer Systems Engineering), and CE-ESY (Embedded Systems)
1.5 Power Management	Not currently covered in CE2004; add *elective* modules to CE-CAO (Computer Architecture), CE-CSE (Computer Systems Engineering), CE-OPS (Operating Systems), and CE-ESY (Embedded Systems)
1.6 Debug Hardware	Not currently covered in CE2004; add *core* modules to CE-CAO (Computer Architecture), CE-CSE (Computer Systems Engineering), and CE-ESY (Embedded Systems)
2.1 Software Footprint & Execution Times	Augment existing CE2004 *core* modules in CE-ALG (Algorithms), CE-OPS (Operating Systems), and CE-ESY (Embedded Systems)
2.2 Software Support for Virtualization	Not currently covered in CE2004; add *elective* module to CE-OPS (Operating Systems)
2.3 Software Support for Heterogeneity	Partially covered in CE2004 CE-ESY (Embedded Systems) *core* modules; add *elective* modules to CE-PRF (Programming Fundamentals), CE-CSE (Software Engineering), CE-ESY, CE-OPS (Operating Systems)
2.4 Software Analysis & Implementation	Not sufficiently covered in CE2004, integrate alongside existing serial/single-cpu material in existing *core* modules of CE-ALG (Algorithms), CE-PRF (Programming Fundamentals), CE-CSE (Software Engineering); differentiate parallel and distributed programming from multicore programming; instill proper analysis methods; add more in-depth *elective* modules to CE-ALG and CE-PRF
2.5 Software Debug & Optimization	Not covered in CE2004, add *core* module on debug & optimization to CE-ESY (Embedded Systems), add *core* module on debugging deadlocks and race conditions to CE-PRF (Programming Fundamentals)

Table 1 - Summary of Recommended Embedded Multicore Curriculum Changes

Finally, an elective upper division course could integrate together all of the material from Figure 1, Figure 2, and Table 1 by having the students do an advanced design and implementation project as a team. In the cases where topics recommended as *elective* modules in Table 1 are unimplemented in the rest of the curriculum (e.g., *virtualization*, *power management*, *software support for heterogeneity*, and *advanced debug and optimization*), this upper division course could dedicate *core* modules to ensure coverage of those topics.

Conclusions

Embedded multicore computing requires a broad and deep understanding of the interaction between software and hardware which may be impossible to gain in a single course. The complexities of these topics have often been the justification for why parallelism is not introduced until late in the curriculum, but the ubiquitous nature of multicore systems

overshadows this concern and it will be to the students' advantage to introduce these concepts early and relate them to many areas of computer science and engineering in an ongoing fashion. The best way to modify the curriculum is to introduce concepts of parallelism and concurrency as early as possible, to add modules to each of the CE/CS core classes in Table 1, and to add an upper division integrative class on embedded multicore for those students who want to tie it all together in an advanced class.

References

1. D. Geer, *Chip Makers Turn to Multicore Processors*, IEEE Computer **38** (2005), no. 5, pp. 11-13.
2. M. Creeger, *Multicore CPUs for the Masses*, ACM Queue **3** (2005), no. 7, pp. 63-64.
3. The Multicore Association. *The Multicore Association Roadmap*. Available from: http://www.multicore-association.org/home.php.
4. The Multicore Association. *Multicore Communications API Specification V1.065*. Available from: http://www.multicore-association.org/workgroup/comapi.php.
5. M. Domeika, *Software Development for Embedded Multi-core Systems: A Practical Guide Using Embedded Intel(r) Architecture*, ed. Newnes. 2008, Boston, MA.
6. M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin and D. I. August, *Revisiting the Sequential Programming Model for Multi-core*, IEEE Micro **28** (2008), no. 1, pp. 12-20.
7. W.-m. W. Hwu, K. Keutzer and T. G. Mattson, *The Concurrency Challenge*, IEEE Design and Test of Computers **25** (2008), no. 4, pp. 312-320.
8. E. A. Lee, *The Problem with Threads*, IEEE Computer **39** (2006), no. 5, pp. 33-42.
9. C. Chang, P. Denning, J. H. Cross, G. Engel, R. Sloan, D. Carver, R. Eckhouse, W. King, F. Lau, S. Mengel, P. K. Srimani, E. Roberts, R. Shackelford, R. Austing, C. F. Cover, G. Davies, A. McGettrick, G. M. Schneider and U. Wolz. *Computing Curricula 2001 Computer Science*. 2001. Available from: http://www.acm.org/education/curricula.html.
10. D. Soldan, J. L. A. Hughes, J. Impagliazzo, A. McGettrick, V. Nelson, P. K. Srimani and M. D. Theys. *Computer Engineering 2004 Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering*. 2004 December 2004. Available from: http://www.acm.org/education/curricula.html.
11. T. Chen, Q. Shi, J. Wang and N. Bao, *Multicore Challenge in Pervasive Computing Education*, The 3rd International Conference on Grid and Pervasive Computing, 2008, pp. 310-315.
12. D. J. Ernst and D. E. Stevenson, *Concurrent CS: Preparing Students for a Multicore World*, Annual Joint Conference Integrating Technology into Computer Science Education 2008, pp. 230-234.
13. A. Gavrilovska, H.-H. Lee, K. Schwan, S. Yalamanchili and M. Wof, "Multi-core Curriculum Development at Georgia Tech: Experience and Future Steps," *I/UCRC Workshop on Experimental Research in Computer Systems* 2006.
14. Q. Shi, T. Chen, H. Wei, J. Wang and N. Bao, *Online Programming Experience Platform for Multicore Curriculum*, International Conference on Computer Science and Software Engineering 2008, pp. 785-788.
15. A. Marowka, *Think Parallel: Teaching Parallel Programming Today*, IEEE Distributed Systems Online **9** (2008), no. 8, pp.
16. OpenMP.org. *The OpenMP API specification for parallel programming* Available from: http://openmp.org/wp/.
17. The MPI Forum. *MPI v2.1*. Available from: http://www.mpi-forum.org/.