Exploring Multiprocessor Design and Implementation Issues with In-Class Demonstrations

Naraig Manjikian
Department of Electrical and Computer Engineering
Queen's University, Kingston, Ontario, Canada
naraig.manjikian@queensu.ca

Abstract

This paper outlines the use of in-class demonstrations to aid students in appreciating hardware and software concepts related to multiprocessing in an upper-year undergraduate course on computer system architecture. The demonstrations include visualization of cache coherence behavior, remote login to a multiprocessor server to demonstrate parallel program execution, observation of internal behavior of a single-chip multiprocessor implemented in a programmable logic chip, and the use of parallel ray-tracing software to visualize speedup and load balance on a multicore desktop computer. The heavy use of visualization seeks to ensure student interest and attention.

1 Introduction

Multiprocessor design and implementation issues range from high-level parallel programming to low-level hardware interfacing. Advances in microelectronics technology have led to widespread use of commodity *multicore* chips from Intel and AMD in general-purpose personal computers over the last five years. Starting with dual-core chips and then quad-core chips, the 2010 introduction of hexa-core and octo-core chips reflects the evolution of the technology to implement single-chip multiprocessors.

Given these trends, undergraduate students in electrical and computer engineering should therefore acquire a solid grasp of the relevant hardware and software concepts as they relate to general-purpose computing. Furthermore, the microelectronics industry also produces increasingly larger field-programmable logic chips in which custom multiprocessors and other parallel hardware architectures for embedded applications or specialized digital signal processing can be implemented. Because field-programmable chips and custom multiprocessors are gaining importance for such applications, there is further incentive for students to gain an appreciation of the relevant concepts to prepare for de-

velopments in the future. For general-purpose multiprocessing, students in other engineering disciplines who create or use technical software with significant computational demands may also benefit from some exposure to the fundamental aspects of multiprocessing.

For more than a decade, the course *ELEC 470 Computer System Architecture* at Queen's University has covered hardware/software design and implementation issues for pipelined, superscalar processors with caches and for multiprocessors. This course was introduced into the curriculum by the author and has been instructed by the author in every year except for the period during and immediately after sabbatical leave. The author's nearly quarter-century of experience in the design, implementation, and application of general-purpose and application-specific multiprocessor systems has informed the approach taken for the material and instruction of the course. Although the detailed discussion of multiprocessor concepts occurs primarily in the final two weeks of the course, the foundation is provided by the two main topics in the preceding ten weeks of the course: pipelined processor architecture (three weeks) and cache/memory design (three weeks). The concept of hardware parallelism pervades the entire course. Thus, the final two weeks on multiprocessing do not require additional background, and attention can be given more directly to essential concepts related to multiprocessor architectures with shared bus/memory, parallel programs, and cache coherence protocols and their implementation in controller logic.

This paper describes how ELEC 470 has made routine and effective use of in-class demonstrations related to multiprocessor issues as a means of enhancing student appreciation and understanding of the key concepts. The demonstrations have included simulated execution of parallel programs for dynamic graphical visualization of multiprocessor cache coherence, on-line experimentation through remote login to measure speedup of application software on a multiprocessor server, observa-

```

double a[N], b[N], dot_product = 0;
Lock the_lock;
Barrier the_barrier;
int num_threads;

void ThreadFunction (void)
{
    int start, end, i, my_id = thread_id();
    double my_sum = 0;

    start = my_id * (N / num_threads);
    end = (my_id + 1) * (N / num_threads) - 1;
    for (i = start; i <= end; i++)
        my_sum = my_sum + a[i] * b[i];
    acquire_lock (the_lock);
    dot_product = dot_product + my_sum;
    release_lock (the_lock);
    barrier_sync (the_barrier, num_threads);
}

main ()
{
    num_threads = 4;
    initialize_lock (the_lock);
    initialize_barrier (the_barrier);
    create_threads (ThreadFunction, num_threads);
    printf ("result = %f\n", dot_product);
}

```

Figure 1: Parallel dot product program

tion of low-level hardware interface signals for a custom multiprocessor in a field-programmable logic chip, visualization of parallel numerical algorithm execution on a multicore chip in a personal laptop, and visualization of parallel task distribution and workload balance for execution of ray-tracing software on a multicore desktop computer. Many of the demonstrations have a connection to current industrial practice. Some of the demonstrations also expose students to the research pursued by the instructor as well as by others. To ensure student attention and interest, the demonstrations often involving tangible hardware artifacts and most of them also featuring dynamic visual aspects. The aim is to effectively complement the necessarily more abstract exposition of the relevant topics.

Finally, with reference to earlier related work, the author has described the use of case studies on the architecture and implementation of actual multiprocessor systems in both undergraduate and graduate courses [1]. The author has also described in-class demonstrations with a logic analyzer to observe the detailed hardware behavior of a pipelined processor and its cache/memory system [2]. The current work in this paper goes further with both software and hardware demonstrations related specifically to multiprocessor systems.

2 Example of Parallel Programming

To initiate the discussion of multiprocessor systems in the final portion of the ELEC 470 course, the instructor uses a simple illustrative example of a parallel program for computing a dot product, as shown in Figure 1. The purpose is to introduce thread creation and synchronization, shared (global) and private (stack) data, and other basic concepts. The author’s multiprocessor adaptation of the widely-used SimpleScalar simulator [3] is used in class to demonstrate idealized parallel execution. Students also have the same software from the beginning

Figure 2: Visualization of cache coherence

of the course throughout the preceding discussion on single-processor systems to use for independent learning. When the multiprocessor portion of the course is reached, the instructor merely provides sample parallel programs such as the one in Figure 1.

3 Visualization of Cache Coherence

Shared-memory multiprocessors with caches introduce the problem of maintaining *cache coherence*, i.e., a consistent representation of all data in memory and in one or more caches [4]. Multiple copies of the same data from memory may be present in the caches of different processors, and if one of the processors intends to modify its copy of the shared data, the other processors with copies must be informed. A common approach is to cause the copies in other caches to be *invalidated* so that only one cached copy, in a modified state, exists. A subsequent request by another processor for the same data requires that the cache with the modified copy *intervene* to respond instead of the shared memory.

A simple architectural organization that enables coherence to be maintained in a straightforward manner is to use a common bus for connecting multiple processors (with their individual caches) and the shared memory. A *bus snooping* protocol implemented in controllers for interfacing to the bus allows all bus requests due to cache misses to be observed by all processors so that invalidation and intervention can be performed when neces-

Figure 3: Sun V440 multiprocessor internals

sary. A common protocol is called *MESI* because data is maintained in caches in one of four states: modified (M), shared (S), exclusive-unmodified (E), and invalid (I). For the ELEC 470 course, the MESI protocol is explained in detail, as well as how controller hardware would be designed for inclusion with each processor/cache pair to implement the protocol.

Section 2 mentioned the author's multiprocessor adaptation of the widely-used SimpleScalar simulator. To illustrate how actual parallel program behavior dictates the states maintained for data in each cache, the author's multiprocessor adaptation includes a cache coherence simulator with dynamic visualization [3]. The visualization provides a changing display to represent the simulated contents in all caches at all times. Colors are used to reflect the four states in the MESI protocol. As the execution of memory access instructions is simulated for each processor, the effects of intervention and invalidation for maintaining cache coherence are reflected in the dynamic graphical display.

Initially developed for research purposes, the multiprocessor cache simulation and visualization software has been used for many years in the author's graduate course (ELEC 871) on advanced topics related to shared-memory multiprocessor systems as well as in ELEC 470. The in-class demonstration of the dynamic visualization for cache coherence is a highly effective method of ensuring that students appreciate the connection between actual program behavior, the coherence protocol, and the dynamic effects on cache contents.

Examples of the graphical visualization output are shown in Figure 2 for eight-processor simulations. These depictions are snapshots of the changing display as parallel program execution is simulated. For each simulated processor, the contents of a simulated 8-kbyte primary (level 1) cache and a larger 256-kbyte secondary (level 2) cache are shown. Each pixel represents

Figure 4: Circuit board with Altera Stratix 1S10 chip

a 16-byte block of data from memory. The behavior of the simple dot product program using arrays containing 20,000 elements is shown in Figure 2(a). The behavior of a parallel program for blocked LU decomposition of a 128×128 matrix is shown in Figure 2(b).

4 Remote Login to a Multiprocessor Server to Demonstrate Parallel Execution

Simulated execution of parallel programs is useful for initial introduction to multiprocessor issues. It is important to use actual multiprocessor hardware to make the discussion more concrete. Earlier in-class demonstrations of parallel execution on actual hardware have included remote login to a multiprocessor server, namely a Sun V440 server located in the home building of the author's department. With the aid of photographs, such as the one in Figure 3, students are first exposed to the specific hardware architecture of the target platform before demonstration of parallel program compilation, execution, and speedup assessment.

Although the discussion of the target platform prior to the in-class demonstration with remote login session is useful, its physical absence has limited its effectiveness somewhat. Nonetheless, because the architecture of this multiprocessor implements shared memory in modular fashion and because the system has been engineered for high availability with redundant power supplies and fans as well as hot-swappable disk drives, it has served as an interesting case study for students.

5 Observing Hardware Signals in a Custom Single-Chip Multiprocessor

To provide a more tangible hardware aspect in class, the instructor has demonstrated the operation of a custom single-chip multiprocessor developed for research purposes in field-programmable logic chips. An example of a circuit board with such a chip from Altera Corp. is

Figure 5: Multiprocessor implemented in Altera chip

Address	Instruction
0x00000000	sw \$0, <data_address>
0x00000004	j 0x00000000

Figure 6: Test program for multiprocessor

shown in Figure 4. The in-class activity involves demonstrating hardware operation with special software (stemming from the author's research activities) that executes on a host computer connected to the board with a cable. The software communicates with custom hardware in the chip to capture and display the behavior of the signals in the multiprocessor, such as address/data information appearing on the shared bus.

Students are exposed to field-programmable logic chips in their second-year laboratory sessions related to digital logic. Computer engineering students have additional experience with such chips in a core third-year course in digital systems engineering. Hence, more immediate attention can be given to the system implemented within the chip. Figure 5 depicts what is implemented within the Altera chip. The author's custom five-stage pipeline processor design with a bus-snooping controller for a coherent data cache is the basis for the implementation. Students in ELEC 470 are quite familiar with pipelined processor, bus, and memory design from the earlier weeks in the course. Here, they can focus on a multiprocessor arrangement of the same elements and the concept of cache coherence.

For a focused demonstration of the operation of the multiprocessor in Figure 5 and the interactions stemming from the bus-snooping hardware that enforces cache coherence, the extremely simple test program shown in Figure 6 has been used. Each processor executes the same code after coming out of reset: write a constant value (zero) with the store-word (sw) instruction to the same address in memory, then repeat with a jump (j) instruction to the beginning of the program. With the controllers for the processors observing

all bus activity, invalidations and interventions will be performed in accordance with the cache coherence protocol.

The special software that allows the internal hardware behavior to be captured and displayed uses a waveform output to depict the behavior, as shown in Figure 7. Additional annotations reflect the material that is provided to students to supplement what is shown in class. Bus arbitration and usage have been discussed prior to the multiprocessor portion of the course, hence the students can focus their attention on the issues related to multiprocessing and cache coherence.

6 Visualization of Parallel Numerical Algorithm Execution

Section 4 described remote login to a multiprocessor server for in-class demonstration. After acquiring a multicore laptop when they became widely available, however, the author sought to provide a useful demonstration with hardware present in class, but using visualization to aid students in understanding execution behavior. As explained in Section 2, a dot product example is used as the basis of the introduction to parallel programming. Building on the dot product example, matrix multiplication is the logic progression for a larger parallel program. The author has developed a demonstration involving two processors executing independent parts of the overall matrix-multiplication program to achieve parallel speedup. The progress of each processor is shown graphically on the screen as the execution proceeds.

Figure 8 shows a snapshot of the output during execution of the author's parallel matrix-multiplication demonstration program. The result matrix on the right side of the figure shows portions completed by the two processors. The two other matrices on the left show the current row and column being processed by each processor. When the program executes, the row/column lines rapidly sweep across the boxes representing these two matrices, while the pixels appearing at a slower rate in the result matrix reflect the completed computations.

7 Parallel Ray-Tracing Software to Visualize Speedup and Load Balance

The final example of in-class demonstration developed by the author involves an adaptation of software called *tachyon* generated by another researcher, J. Stone, for ray-tracing in visual images [5]. The *tachyon* software was written to exploit multiple processors for faster rendering of images with simulated motion of the point of view. A snapshot of sample output from this software is shown in Figure 9. The full output is a sequence of images, rendered as quickly as possible with a specified number of processors, as the point of view is moved

P0 snoops P1's request. Snoop hit in dup. tags is followed by main cache access to retrieve data and update state in both dup. and main tags.

Figure 7: Waveforms from observation of signals for multiprocessor implemented in Altera chip

Figure 8: Parallel matrix multiplication snapshot

Figure 9: Graphical display from ray-tracing software through a scripted sequence of positions in space.

The *tachyon* software is large, exceeding 10,000 lines of code. The author has studied the software and made various modifications. One key change is to display the

Figure 10: System monitor for processor utilization

progress of rendering as execution proceeds, rather than displaying only once after each image is fully rendered. Another change is to distribute the workload among the processors in a blocked manner, rather than an interleaved manner, to highlight issues related to load balance, as will be explained shortly. For convenience, the author has used Ubuntu Linux as the platform for this software, where the standard `pthread`s library is employed to support parallel execution. The in-class demonstration with this software involves the author bringing a multicore desktop computer to the lecture. For the purposes of the demonstration, the desktop computer is connected directly to the digital projector for display on the screen, instead of the usual laptop used for lecture presentations.

For demonstrations using a quad-core desktop computer, the author emphasizes the use of all four processor by the ray-tracing software by showing the output of the system monitor program that tracks the utilization of the processors. Sample output is shown in Figure 10. The point at which the ray-tracing software begins executing is clearly evident when the utilization increases suddenly from near 0%. All four processors are utilized

Figure 11: Visualization of workload imbalance

heavily as the software executes in parallel.

The software is first demonstrated in its original form, where the intent is to have fast rendering of a sequence of images as the simulated point of view is moved. The differences in rendering speed are clearly evident when the software is executed on a single processor, then two processors, and finally four processors.

Subsequently, the author demonstrates the execution of the modified version of the software that updates the graphical display multiple times during the rendering process, rather than only once at the end. Also the distribution of the workload is done in a blocked manner. A snapshot of the display (with explanatory annotations added for this paper) is shown in Figure 11. There are regions of the current image assigned to each processor, and it is evident what portion has been rendered by each processor and what portion remains to be rendered. The actual demonstration is a rapid succession of images where these regions change dynamically as the processors complete their work for each image.

Students are guided through a discussion that seeks to understand the significance of the behavior exemplified in this snapshot. Because there is a portion with no image content, the processor assigned to that portion completes its computations more quickly, as evident from the progress that is visible in Figure 11. As a consequence of this *load imbalance*, the overall execution of the parallel version of the software is less efficient because the processor that finishes more quickly waits for the others to complete their workload. It is for this reason that the *original* software interleaves the workload among the processors by rows of the image. Thus, all

processors have portions of the image that are processed more quickly for a more even distribution of the workload. By explicitly using a non-interleaved distribution and by visualizing the rendering as it progresses rather than only at the end, the author is ensuring that students observing the demonstration gather more insight into parallel program behavior on an actual multiprocessor system.

8 Conclusion

With the widespread availability of computers with multicore chips, the in-class demonstrations outlined in this paper seek to enhance student appreciation and understanding of important multiprocessor-related concepts. Visualization of cache coherence behavior through simulation exposes aspects that are not evident when executing parallel programs on actual hardware. Exposure to custom single-chip multiprocessor implementation in a programmable logic chip provides an indication of what is feasible with current technology. Observing the behavior of sophisticated ray-tracing software executing in parallel on a multicore desktop computer offers useful insight, particularly when the software has been modified to expose more of the behavior and other relevant issues. Although additional effort is necessary to prepare and effectively employ such demonstrations, students are generally receptive, largely due to the increased availability of multiprocessors in recent years.

References

- [1] N. Manjikian, "Case Studies in Design and Implementation for Computer Engineering Education", *Proceedings of the 12th Canadian Conference On Engineering Education*, Victoria, BC, August 2001, pp. 322-331.
- [2] N. Manjikian, "In-lecture hardware demonstrations with a logic analyzer to illustrate pipelined execution and cache/memory behavior," *Proceedings of the 2000 Workshop on Computer Architecture Education*, Vancouver, BC, June 2000.
- [3] N. Manjikian, "Multiprocessor Enhancements of the SimpleScalar Tool Set," *ACM SIGARCH Computer Architecture News*, Vol. 29, No. 1, March 2001, pp. 8-15.
- [4] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th edition, Morgan Kaufmann, 2008.
- [5] J. E. Stone, *An Efficient Library for Parallel Ray Tracing and Animation*, M.Sc. thesis, Computer Science, University of Missouri-Rolla, 1998. The thesis, software, and other documentation are available at <http://jedi.ks.uiuc.edu/~johns/raytracer>.