The Concurrency Challenge

Wen-mei Hwu
University of Illinois at Urbana-Champaign

Timothy G. Mattson
Intel

Kurt Keutzer
University of California, Berkeley

Editor's note:
The evolutionary path of microprocessor design includes both multicore and many-core architectures. Harnessing the most computing throughput from these architectures requires concurrent or parallel execution of instructions. The authors describe the challenges facing the industry as parallel-computing platforms become even more widely available.
—David C. Yeh, Semiconductor Research Corp.

THE SEMICONDUCTOR INDUSTRY has settled on two main trajectories for designing microprocessors. The *multicore trajectory* began with two-core processors, with the number of cores doubling with each semiconductor process generation. A current exemplar is the recent Intel Core 2 Extreme microprocessor with four processor cores, each of which is an out-of-order, multiple-instruction-issue processor supporting the full X86 instruction set. The *many-core trajectory* begins with a large number of far smaller cores and, once again, the number of cores doubles with each generation. A current example is the Nvidia GeForce 8800 GTX graphics processing unit (GPU) with 128 cores, each of which is a heavily multithreaded, single-instruction-issue, in-order processor that shares its control and instruction cache with seven other cores.

Although these processors vary in their microarchitectures, they impose the same challenge: to benefit from the continued increase of computing power according to Moore's law, application software must support concurrent execution by multiple processor cores. Unless software adapts to utilize these parallel systems, the fundamental value proposition behind the semiconductor and computer industries will falter.

In this article, we identify the challenges underlying the current state of the art of parallel programming and propose an application-centric methodology to protect application programmers from these complexities in future parallel-application developments. For these multiple-core processors to be useful, the software industry must adapt and embrace concurrency. Successful programming environments for these processors must be application centric. Concurrent programming environments must be natural and must protect the application programmer from as many hardware idiosyncrasies as possible. Finally, solutions should not be ad hoc but should be derived according to well-established engineering and architectural principles.

Today's major programming models

Microprocessors have long employed concurrency at the hardware level. They have taken one of the four approaches shown in Figure 1 to hide the complexity of parallel execution from programmers.[1] In this figure, the top horizontal line depicts the interface to human, high-level programmers; the middle line depicts the machine-level programming interface to compilers and debuggers.

In Figure 1a, the instruction-set architecture (ISA) and its hardware implementation completely hide the complexity of concurrent execution from the human programmer and the compiler. This model is used by superscalar processors, such as the Pentium 4, where the hardware's execution of instructions in parallel and out of their program order is hidden by a sequential retirement mechanism. The execution state exposed through the debuggers and exception handlers to the human programmers is completely that of sequential execution. Thus, programmers never deal with any complexity or incorrect execution results due to concurrent execution. This is still the programming model for the vast majority of programmers today.

Figure 1b illustrates a model where parallel execution is exposed at the machine programming level to the compilers and object-level tools, but not to the human programmers. The most prominent micropro-

0740-7475/08/$25.00 © 2008 IEEE Copublished by the IEEE CS and the IEEE CASS

cessors based on this model are VLIW (very long instruction word) and Intel Intanium Processor Family processors. These microprocessors use traditional sequential-programming models such as C and C++. A parallelizing compiler identifies the parallel-execution opportunities and performs the required code transformations. Programmers must recompile their source and, in some cases, rewrite their code to use algorithms with more inherent parallelism. Sometimes, they also need to adapt their coding style with constructs that the parallelizing compiler can more effectively analyze and manipulate. Vendors often provide critical math libraries as a domain API implemented at the machine programming level to further enhance their processors' execution efficiency. The complexities of parallel execution are exposed to human programmers whenever they need to use debuggers or when they need to optimize their code because the hardware does not maintain a sequential state. Acceptance of these processors in the general-purpose market has been hampered by the need to recompile source code and the need to deal with parallel execution complexities when debugging and optimizing programs.

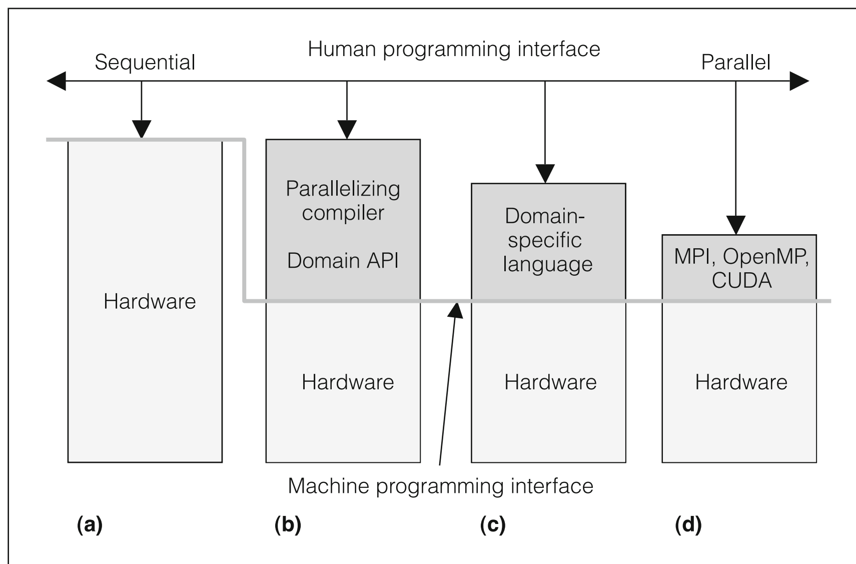

Figure 1. Current programming models of major semiconductor programmable platforms: Hardware hides all complexity of concurrent execution (a). Parallelizing compilers and tools help contain the complexity of concurrent execution (b). Domain-specific languages and environments let programmers express concurrency in ways natural to a particular domain (c). Parallel programming models require programmers to write code for a particular model of concurrent execution (d). (CUDA: Compute Unified Device Architecture; MPI: Message Passing Interface.) (Source: W. Hwu et al., "Implicit Parallel Programming Models for Thousand-Core Microprocessors," ***Proc. 44th Design Automation Conf.*** **(DAC 07), ACM Press, 2007, pp. 754-759 © 2007 ACM Inc. Reprinted by permission.)**

Figure 1c indicates an approach that has been modestly successful in particular application domains such as computer graphics, image processing, and network protocol processing. In this approach, programmers use a language that is naturally adapted to the needs of a particular application domain and makes the expression of parallelism natural; in some cases, it is even more natural than with a traditional sequential language. A good example is the network programming language Click and its parallel extension, NP-Click.[2]

There are three main challenges with this approach. First, programmers resist new languages, but this can be ameliorated by choosing a popular language in the application domain. Second, these programming models are almost always tailored to a particular multiprocessor family. Programmers are reticent to tie their applications so strongly to a single architectural family. Finally, today's increasingly complex systems require assembly of diverse application domains. Integrating a programming environment with a variety of domain-specific languages is an unsolved problem.

All three models have seen some success in their target markets. The approach in Figure 1a is waning because of the power demanded in architectures that use hardware to manage parallelism. The approach in Figure 1b generally fails to extract sufficient amounts of parallelism for multiple-core microprocessors. The successes of Figure 1c have been too narrow to capture widespread interest. Thus, the approach in Figure 1d dominates current practice in programming multiple-core processors. For general-purpose parallel programming, MPI (Message Passing Interface) and OpenMP are the main parallel-programming models. The new CUDA (Compute Unified Device Architecture) language from Nvidia lets programmers write massively threaded parallel programs without dealing with the graphics API functions.[3] These models force application developers to explicitly write parallel code, either by parallelizing individual application

programs or by defining their applications in terms of multiple independent programs that run concurrently on multiple-processor cores. This seriously impacts the costs of developing, verifying, and supporting software products. This drastic shift to explicit parallel programming suggests that the concurrency revolution is primarily a software revolution.[4]

Underlying problems of applications software

Creating applications software is more of an art than an engineering discipline. Compare software engineering to a more established engineering discipline such as civil engineering. Civil engineers build bridges using established practice well-founded in the material sciences and basic physics. Civil engineers create detailed plans of their bridges and have them constructed independently. However, software engineers don't have a well-established practice to follow. By analogy to civil engineering, software engineers build bridges to the best of their understanding and experience. Some of these "bridges" stay up, and other engineers try to understand the principles that allowed them to stay up. Some fall down, and other engineers try to understand the design flaws that caused them to fall down. But the reproducible practices, the underlying architectural principles and fundamental laws, are poorly understood.

Thus, we come to the problem of concurrent software engineering with a weak engineering foundation in sequential software engineering. Any long-term solution to this problem will need to establish reproducible practices and software architectures as opposed to programs that are hand-crafted point solutions. Parallel programming is complicated,[5] and most programmers are not prepared to deal with that complexity. As a result, after 30 years of hard work in the high-performance computing community, only a tiny fraction of programmers write parallel software. The complexities of parallel programming impact every phase of supporting a software product, from code development and verification to user support and maintenance.

By reviewing some of the most pertinent challenges of parallel programming, we can determine the new engineering principles and software architectures required for addressing these challenges.

Understanding the system

A program implies a control flow, data structures, and memory management. These combine to create a huge number of states that a software system can occupy at any point in time. This is why software systems tend to be far more difficult to debug and verify than hardware systems. Concurrency multiplies this with an additional set of issues arising from the multiple threads of control that are active and making progress at any given time. Understanding the behavior of the entire system against the backdrop of so many states is difficult. We have few conceptual tools beyond Unified Modeling Language (UML) sequence diagrams to help us. We are largely left to puzzle through complex interactions in our heads or on a whiteboard.

Discovering concurrency

To write a parallel application program, developers must identify and expose the tasks within a system that can execute concurrently. Once these tasks are available, their granularity, their ordering constraints, and how they interact with the program's data structures must be managed. Typically, the burden of discovering and managing concurrency falls on programmers, forcing them to think carefully about the entire application's architecture and their algorithms' concurrent behavior.

Multithreaded reading

Programmers write code and then read it, building an execution model in their minds. How they read a program affects how they understand a program and how they write code in the first place. In a serial program, there is a single thread of control working through their models. The current generation of programmers have been trained to think in these terms and read their programs with this single-threaded approach. A concurrent program, however, requires a multithreaded reading. At any point in the code, programmers must understand the state of the data structures and the flow of control in terms of every semantically allowed interleaving of statements. With current programming tools, this interleaving is not captured in the program's text. A programmer must imagine the interplay of parallel activities and ensure that they correspond to the given functional specification.

Nondeterministic execution

The system state in a concurrent program depends on the way execution overlaps across processor cores. Because the detailed scheduling of threads is con-

trolled by the system and not the programmer, this leads to a fundamental nondeterminism in program execution. Debugging and maintaining a program in the face of nondeterminism is extremely difficult.

Interactions between threads

Threads in all but the most trivial parallel programs interact. They pass messages, synchronize, update shared data structures, and compete for system resources. Reasoning about interactions between threads is unique to parallel programming and a topic about which most programmers have little or no training.

Testing and verification

Applications include test suites to verify correctness as they are optimized, ported to new platforms, or extended with new features. Developers face two major challenges when moving test suites to a parallel platform. First, a parallel test suite must exercise a range of thread counts and a more diverse range of hardware configurations. Second, the tests must deal with rounding errors that change as the number and scheduling of threads modify the order of associated operations. Mathematically, every semantically allowed ordering of statements is equally valid, so it is incorrect to pick one arbitrary ordering (such as the serial order) and insist on bitwise conformity to that result. Addressing these two issues may require major changes to the testing infrastructure, which can be a formidable task if the test suite's original developers are no longer available to help.

Distributed development teams

Modern software development is a team effort. In most cases, these teams are distributed across multiple time zones and organizations. Changes in data structures and high-level algorithms often cause subtle problems in modules owned by other team members. Managing those distributed teams of programmers is difficult for sequential and concurrent programming. But concurrent programs add complexities of their own. First, concurrent modules do not compose very well. With the commonly available parallel-programming environments, it's not always possible to use parallel components to build large parallel systems and expect correct and efficient execution. Program components that behave reasonably well in isolation can result in serious problems when they are combined into a concurrent program.[4] This makes traditional approaches based on components created and tested in isolation and later assembled into a larger system untenable.

Optimization and porting

The purpose of parallel programming is increased performance. Hence, it is fundamentally a program optimization problem. For sequential programming, the architectural parameters involved in performance optimizations are well-understood in terms of commonly accepted hardware models. The optimization problem is difficult, but the common models with a well-known hierarchy of latencies—from registers to cache, to memory—support a mature optimization technology folded into the compilers. For parallel programming, the state of affairs is far less mature. The memory hierarchies are radically different between platforms. How cores are connected on a single processor and how many-core processors are connected into a larger platform vary widely. The optimization problem is consequently dramatically more complicated.

Current tools for concurrent programming put the burden of optimization largely on the programmer. To optimize a parallel program today, a programmer must understand the capabilities and constraints of the specific hardware and perform optimizations, determining their effects with tedious and error-prone experimentation. Each time the software is moved between platforms, this experimentation must be repeated. To a generation of programmers used to leaving optimization to the compiler, this presents a huge hurdle to the adoption of parallel programming.

Features versus performance

Parallel programming has been largely an activity in the high-performance computing (HPC) community, where the culture is to pursue performance at all costs. But for software vendors working outside HPC, performance takes a back seat to adding and supporting new features. Consumers purchase software on the basis of the available features and how well these features are implemented, not execution speed. New concurrent-programming tool chains for software vendors must keep this perspective in mind.

Economic considerations

Historically, the complexities of parallel programming have limited its commercial adoption. Understanding this history is critical if we want to succeed

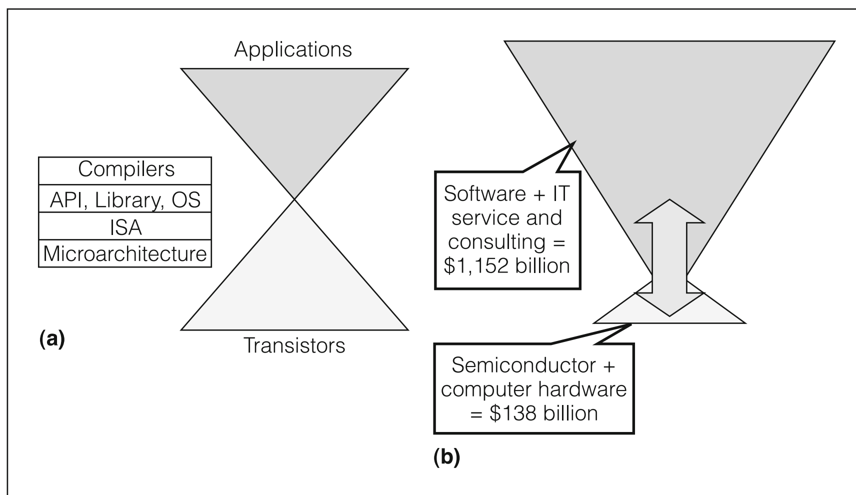

Figure 2. Magnification of value and complexity exposed to the programmers: classic view of hardware and software interface (a), weighted view according to associated revenues (b). (ISA: instruction-set architecture.) (Source: W. Hwu et al., "Implicit Parallel Programming Models for Thousand-Core Microprocessors," *Proc. 44th Design Automation Conf.* (DAC 07), ACM Press, 2007, pp. 754-759 © 2007 ACM Inc. Reprinted by permission.)

this time around. Figure 2a illustrates the classical interface between hardware and software.[1,6] The top triangle illustrates the design space to explore, to find a particular application solution. The bottom triangle illustrates the design space that must be explored to find the right target hardware architecture to host the application.

Implicit in Figure 2a is the misleading notion that the hardware search space is as important as the application search space. This is not the case. The triangles in Figure 2b are sized according to the relative size of the two industries. According to the US Department of Commerce, the 2005 revenue of the semiconductor industry was $138 billion, whereas the revenue of the applications industry was $1,152 billion. In economic terms, it is the hardware architectures that should serve the needs of programmers, who in turn serve the needs of the application industry. This suggests that to meet the concurrency challenge, future hardware solutions must be designed from an application- and software-centric point of view.

An additional consideration is the increasing influence of consumer and media applications. The consumer electronics market often facilitates a winner-take-all dynamic, resulting in reduced diversity. Ultimately, this could reduce the number of platforms that programmers need to support.

Addressing the challenge

From these observations, we define three key principles that underlie our approach to the concurrency challenge:

- We must approach the concurrency challenge in an application-driven and software-centric manner.
- The best way to deal with concurrency problems is to obviate them by using application development frameworks, parallel libraries, and tools. Compilers and autotuners should do the heavy lifting so that programmers can quickly, or even automatically, derive high-performance solutions to new problems.
- When application programmers must face the concurrency challenge directly, we should help them employ a sound engineering approach—so that they can create a design in terms of well-understood software architectures with proven, replicable solutions appropriate to that architecture. Programming models play an important role, but they add value only to the extent that they support the chosen software architecture.

We strive to shield the vast majority of application programmers from the complexity of parallel programming. We advocate an application development model where programmers classify their problems within an application-oriented taxonomy that leads them naturally to known solution strategies. They then turn these strategies into code by using frameworks or by adapting program skeletons that are designed and implemented by expert parallel programmers. As these application codes move through the tool chain, compilers and autotuning tools optimize the code for high-performance, parallel execution.

Application frameworks

Application programmers should address their parallel-software problems in familiar terms. Fortunately, many applications—games, media, signal processing, and financial analytics—are naturally concurrent. In many cases, application developers are already familiar with the concurrency in their problems; they

just need technologies to express that concurrency in ways natural to their application domain.

Consider a photo-retrieval application. An application developer solving a face-recognition problem would use an image-processing application framework that supports primitives for feature extraction and classification. The application developer selects the primitives and specifies the interactions between them to form a solution to the face-recognition problem. Feature extraction and classification algorithms contain ample opportunities for data parallelism, but there is no reason to trouble the application developer with these details. The framework engineers that develop and support these primitives will manage any complications due to the concurrency.

Linking applications and programming

We have been developing a methodology to systematically link application developers to the appropriate concurrent software frameworks. We identified a set of recurrent application solution strategies that capture a developer's strategy for a particular solution, yet are general enough to permit a variety of implementations specific to the needs of a particular problem and target architecture. We call these strategies *dwarfs*.[7] Conceptually, they form a platform (the arrow in Figure 2b) that provides a high-level link between applications and hardware.

We have identified 13 dwarfs: dense linear algebra, sparse linear algebra, spectral methods, *n*-body methods, structured grids, unstructured grids, MapReduce, combinational logic, graph traversals, dynamic programming, backtrack branch and bound, graphical models, and finite-state machines.

In the face-recognition problem, the MapReduce dwarf supports image-feature extraction well. An engineer working on image-feature extraction would choose MapReduce as the basic solution strategy. A collection of different MapReduce implementations would be encapsulated into a single MapReduce programming framework for feature extraction.[8] Likewise, developers working in quantum chemistry are well-versed in dense linear algebra and understand how their problems are decomposed into dense linear-algebra operations, just as programmers working in game physics understand their problem as a composition of *n*-body and structured-grid solutions.

The ParLab group at the University of California, Berkeley has worked with application communities such as HPC, computational music, and visual computing to grow the well-known list of seven dwarfs of high-performance computing to 13.[7] Is this list of 13 dwarfs complete? Probably not, but there's no barrier to adding more.

Programming support

Although dwarfs provide a critical link between application-level problems and their solution strategies, translating a solution strategy described by dwarfs into working parallel code requires many supporting steps. Fortunately, we can ease this task with programming frameworks and skeletons, which raise the abstraction level and make the programming process considerably easier. In the examples we have discussed so far, a generic MapReduce programming framework could easily be tailored to implement feature extraction, and a broadly applicable dense-linear-algebra framework could serve to craft quantum chemistry solutions. A higher-level framework could define how these solutions are assembled into an entire application.

We envision that parallel applications (such as a media-search engine) will be expressed in application frameworks (such as an image-manipulation framework) and a library of primitives (such as image retrieval). Each primitive will be implemented with one or more well-documented solution strategies (such as the MapReduce dwarf) and corresponding skeletons that the programmer can adapt to implement a particular application. A rich set of frameworks will be developed and documented to cover the needs of applications. These programming frameworks will likely be built on languages such as C or C++ with OpenMP and MPI.

Although our long-term goal is to raise the abstraction level and spare programmers from the details of managing concurrency, we realize we cannot provide complete coverage with frameworks. Some application programmers will inevitably need to program at a low level, and hence, parallel programming models will continue to be an important part of programming systems.

Current parallel-programming models such as MPI, OpenMP, and the Partitioned Global Address Space (PGAS) languages have all been shown to work for both expert parallel programmers and application programmers. We expect that as many-core systems evolve, these programming models will evolve as well, not only to better match the hardware but also to better support the frameworks we envision.

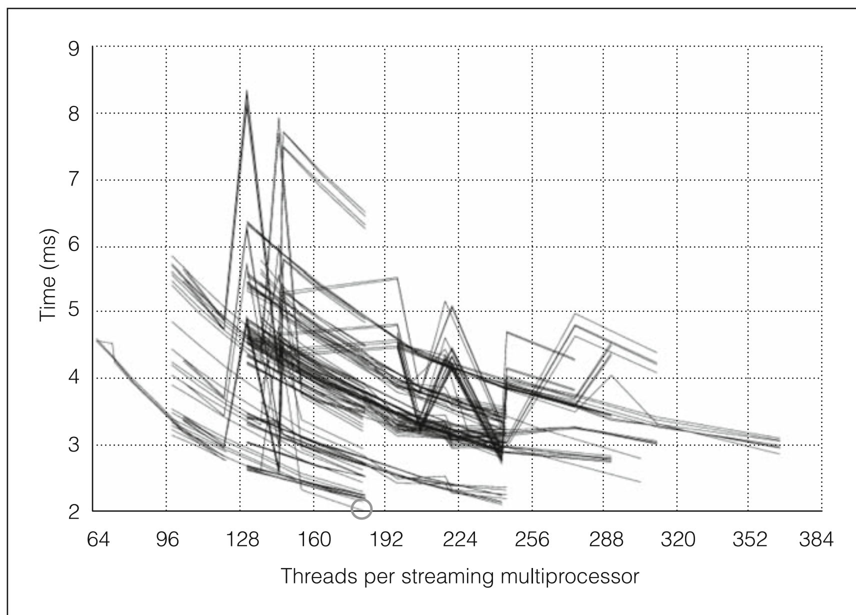

Figure 3. Performance profiles for different optimization parameters for a single SAD (sum of absolute differences between elements of two image areas) program. (Source: S. Ryoo et al., "Program Optimization Space Pruning for a Multithreaded GPU," ***Proc. 6th Ann. IEEE/ACM Int'l Symp. Code Generation and Optimization*** **(CGO 08), ACM Press, 2008, pp. 195-204 © ACM Inc. Reprinted by permission.)**

For example, the next release of OpenMP (OpenMP 3.0) will include a flexible task queue mechanism.[9] This greatly expands the range of algorithms (and dwarfs) that can be addressed with OpenMP. We anticipate a healthy interplay between frameworks and programming models, so that they can evolve together to better meet the needs of application programmers.

Code-optimization space exploration

Both frameworks and programming models will require lower-level support to create efficient code. For example, for a given skeleton and user-supplied fill-in code, there are multiple ways to exploit concurrency; the best choice is often determined by the underlying hardware organization.

Ryoo et al., working with the CUDA programming model and an Nvidia 8800 GPU, found that different plausible rearrangements of code resulted in significant performance differences.[10] Different allocations of resources among the large number of concurrent threads led to nonlinear performance results as several application kernels were optimized. Even a simple computational kernel had multiple dimensions of optimization based on, for example, loop tiling, thread work assignment, loop unrolling, or data prefetch.

Figure 3 shows the execution time of different versions of SAD (sum of absolute differences between elements of two image areas).[10] Each curve shows the effect of varying the thread granularity while keeping other optimization parameters fixed. There is a 4× spread across these different choices of optimization parameters. Taking these factors into account is critical to the success of future programming frameworks. However, it is unreasonable to expect a human programmer to explore such a complex optimization space.

We envision that parameterized programming and automatic parameter space explorations guided by performance models (such as autotuners) will be sufficient to achieve high levels of performance with reasonable programmer effort.[10] In particular, autotuners combined with frameworks should help nonexpert programmers achieve optimal performance. Related work has shown that efficient code for multicore microprocessors from Intel and AMD can be derived using this methodology.[11]

To be successful, the programming frameworks we describe must be retargetable to a range of multiple-core architectures and produce high-performance code. To address these often conflicting goals, we envision that frameworks and parallel-programming models will be supported by powerful parallel-compilation technology and autotuning tools to help automatically optimize a solution for a particular platform. This will require frameworks that are parameterized and annotated to describe the computation's fundamental properties. A new generation of parallel compilers and autotuning tools will use these parameterizations and annotations to optimize software well beyond the capabilities from current-generation compilers and tools.

We understand those who question whether our proposed application-centric approach—even if it works well in particular application domains—will ever cover the full range of future applications. We answer these skeptics in two ways. First, we believe that a consumer-oriented, winner-take-all market

dynamic will work to our advantage. Just as consolidation reduced the number of players in consumer electronics, over time the same effect will occur for consumer-oriented application software. We've seen this already, for example, in the dominance of Microsoft products in the office-productivity domain. The same effect will happen over time in other application domains. For example, videogame software suppliers (such as Electronic Arts) are playing a significant role in the software industry, as are their console makers (such as Sony) and their semiconductor suppliers (such as Nvidia). They are all influential in their respective industries. Over time, these roles might evolve into positions of dominance and the subsequent consolidation will make it easier for a few application frameworks to support an entire industry.

Second, we don't need 100% application coverage to have a major impact. There will always be classes of applications that will not decompose into well-understood problems (such as dwarfs) with well-understood parallel solutions. In most cases, the same features that make these applications hard to decompose into dwarfs will also make them hard to parallelize. Hence, there will always be a need for research on new methodologies, compiler techniques, and tools to address these pathologically difficult applications. ■

Acknowledgments

We acknowledge the support of the Gigascale Systems Research Center (GSRC).

■ References

1. W. Hwu et al., "Implicit Parallel Programming Models for Thousand-Core Microprocessors," *Proc. 44th Design Automation Conf.* (DAC 07), ACM Press, 2007, pp. 754-759; http://doi.acm.org/10.1145/1278480.1278669 © ACM Inc. Reprinted by permission.
2. N. Shah et al., "NP-Click: A Productive Software Development Approach for Network Processors," *IEEE Micro*, vol. 24, no. 5, Sept./Oct. 2004, pp. 45-54.
3. NVIDIA *CUDA Compute Unified Device Architecture Programming Guide,* v1.0, Nvidia Corp., June 2007, http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.
4. H. Sutter and J. Larus, "Software and the Concurrency Revolution," *ACM Queue*, vol. 3, no. 7, Sept. 2005, pp. 54-62.
5. T.G. Mattson, B.A. Sanders, and B.L. Massingill, *Patterns of Parallel Programming*, Addison-Wesley Professional, 2004.
6. K. Keutzer et al., "System Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Trans. Computer-Aided Design*, vol. 19, no. 12, Dec. 2000, pp. 1523-1543.
7. K. Asanovic et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, tech. report UCB/EECS-2006-183, EECS Dept., Univ. of California, Berkeley, 2006.
8. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. 6th Symp. Operating System Design and Implementation* (OSDI 04), Usenix, 2004, pp. 137-150.
9. "OpenMP Application Program Interface," OpenMP Architecture Review Board, May 2005, http://www.openmp.org.
10. S. Ryoo et al., "Program Optimization Space Pruning for a Multithreaded GPU," *Proc. 6th Ann. IEEE/ACM Int'l Symp. Code Generation and Optimization* (CGO 08), ACM Press, 2008, pp. 195-204, http://doi.acm.org/10.1145/1356058.1356084 © ACM Inc. Reprinted by permission.
11. S. Williams et al., "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," *Proc. ACM/IEEE Conf. Supercomputing* (SC 07), ACM Press, 2007, New York, article 38.

Wen-mei Hwu is a professor and the Sanders-AMD Endowed Chair of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His research interests include architecture and compilation for parallel-computing systems. He has a BS in electrical engineering from National Taiwan University, and a PhD in computer science from the University of California, Berkeley. He is a Fellow of both the IEEE and the ACM.

Kurt Keutzer is a professor of electrical engineering and computer science at the University of California, Berkeley and a principal investigator in UC Berkeley's Universal Parallel Computing Research Center. His research focuses on the design and programming of ICs. He has a BS in mathematics from Maharishi International Univer-

sity, and an MS and a PhD in computer science from Indiana University, Bloomington. He is a Fellow of the IEEE and a member of the ACM.

Timothy G. Mattson is a principal engineer in the Applications Research Laboratory at Intel. He research interests focus on performance modeling for future multicore microprocessors and how different programming models map onto these systems. He has a BS in chemistry from the University of California, Riverside; an MS in chemistry from the university of California, Santa Cruz; and a PhD in theoretical chemistry from the University of California, Santa Cruz. He is a member of the American Association for the Advancement of Science (AAAS).

■ Direct questions and comments about this article to Wen-mei Hwu, Univ. of Illinois at Urbana-Champaign, Coordinated Science Lab, 1308 W. Main St., Urbana, IL 61801; w-hwu@uiuc.edu.

For further information about this or any other computing topic, please visit our Digital Library at http://www.computer.org/csdl.

Lower nonmember rate of $29 for *S&P* magazine!

IEEE Security & Privacy magazine is the premier magazine for security professionals. Each issue is packed with information about cybercrime, security & policy, privacy and legal issues, and intellectual property protection.

Top security professionals in the field share information you can rely on:

- Silver Bullet podcasts and interviews
- Intellectual Property Protection and Piracy
- Designing for Infrastructure Security
- Privacy Issues
- Legal Issues and Cybercrime
- Digital Rights Management
- The Security Profession

Subscribe now!

www.computer.org/services/nonmem/spbnr

The GSRC: Bridging Academia and Industry

Richard Oehler, AMD

The problem of finding parallelism in application code and exploiting it through automatic tools has long been recognized as the Holy Grail of high-performance computing (HPC). In the late 1970s, many academic institutions and government laboratories spent considerable time and energy investigating issues and challenges in this area. Although they found some success in parallelizing Fortran compilers and compiler front ends, and a little more success in parallelizing high-performance math libraries, the general problem still remains. There are no general-purpose tools that can parallelize run-of-the-mill commercial code, and writing (or rewriting) commercial code to exploit parallelization is very difficult and prone to error.

By the early 1990s, it became clear that the introduction of chip-level multiprocessing (CMP) would put renewed emphasis on multithreading and multiprocessing capabilities. By the early 2000s, it was also clear that CPU single-thread performance scaling was at an end, owing to ever-increasing power and heat issues and complexity issues. No longer could the processor chip providers compete only in terms of frequency (the fastest single thread) to achieve performance leadership. They would need to rely on total parallel processing, including multithreading performance, to win the performance race.

This new direction to achieve application performance leadership has brought the underlying unsolved parallelization problem to the top of the list of problems that industry must solve if we are to continue scaling commercial application performance. Many researchers in academia and industry have begun to work on this problem. The interest in solving this problem is far broader and recognized as critical more than ever before.

To solve this problem, the industry needs coordinated activity by academia and industry. Several organizations have such charters, and at least one has embraced concurrency as a long-term research agenda. In 2006, the Gigascale Systems Research Center (GSRC) refocused its agenda to include four themes and a driver. The one most interesting to me was Design Technologies for Concurrent Systems, led by Wen-mei Hwu. "The Concurrency Challenge" article in this issue of *IEEE Design & Test* describes the multithreading problem, discusses how the solution space is being analyzed, and predicts where breakthroughs are likely to occur.

From my perspective, success in this endeavor requires major industry participation. Participation is required not only among processor chip or hardware system developers; it must also include operating system and tool providers and, most important, the application writers and developers who will engage and in many ways drive toward acceptable solutions.

At AMD, we must be prepared to try new low-overhead approaches to solve the major hardware issues of synchronization and control over threads, concurrency and locking protocols, debugging, and performance-measuring assists. We need to provide the vehicles on which industry and academia can try new software approaches to concurrency. Finally, we must be prepared to share our experiences so that both industry and academia can reach a common understanding of the necessary hardware and software components that will solve the multithreading and concurrency problems.

Richard Oehler *is Server CTO and Corporate Fellow at AMD. Contact him at rich.oehler@amd.com.*

 0740-7475/08/$25.00 © 2008 IEEE