Challenges and Opportunities in Many-Core Computing

With increasing use of computers that employ many independent processing units, commercial and technical-scientific software, as well as general-purpose operating systems, will have to undergo fundamental changes.

By John L. Manferdelli, Naga K. Govindaraju, and Chris Crall

ABSTRACT | In this paper, we present some of the challenges and opportunities in software development based on the current hardware trends and the impact of massive parallelism on both the software and hardware industry. We indicate some of the approaches that can enable software development to effectively exploit the many-core architectures. Some of these include encapsulating domain-specific knowledge in reusable components, such as libraries, integrating concurrency with languages, and supporting explicit declarations to help compilers and operating system schedulers. Tighter interaction between software and underlying hardware is required to build scalable and portable applications with predictable performance and higher power-efficiency. Overall, many-core computing provides us opportunities to enable new application scenarios that support enhanced functionality and a richer experience for the user on commodity hardware.

KEYWORDS | Compilers; many-core computing; operating systems; parallel applications

I. INTRODUCTION

Over the last couple of decades, the computer industry has been witnessing massive improvements in commodity hardware and software technology. These changes have often been foreshadowed by hardware and software technology originating from high performance, scientific, and enterprise computing research. More recently, the demand for realism in games and the entertainment industry has driven improvements in real-time physics, artificial intelligence, and rendering effects, which in turn

have pushed the envelope for software industry standards, such as the Microsoft DirectX application programming interfaces (APIs), and commodity processors in PCs, laptops, and consoles.

Innovation and advancement in scientific and enterprise communities have been fueled by the relentless, exponential improvement in the capability of computer hardware over the last 40 years. The driving force for much of this improvement has been the ability to double the number of microelectronic devices onto a constant area of silicon at a nearly constant cost approximately every two years. This exponential improvement in transistor count every two years is widely referred to as Moore's law.

Virtually every analytical technique from the scientific community has become broadly deployed: operations research, data mining, machine learning, compression and encoding, signal analysis, imaging, mapping, simulation of complex physical and biological systems, and cryptography. These techniques have benefited education, health care, and entertainment. The techniques also enabled the worldwide delivery of cheap, effective, and profitable services from eBay to Google.

In stark contrast to the scientific community, commercial application software programmers have not, until recently, had to grapple with massively concurrent computer hardware. While Moore's law continues to be a reliable predictor of the aggregate computing power that will be available to commercial software, we can expect very little improvement in serial performance of general-purpose CPUs. The increase in performance will come instead from parallel computing. This will have a profound effect on commercial software development. The programming languages, compilers, operating systems (OSs), and software development tools will all evolve, which will in turn have an equally profound effect on computer and computational scientists.
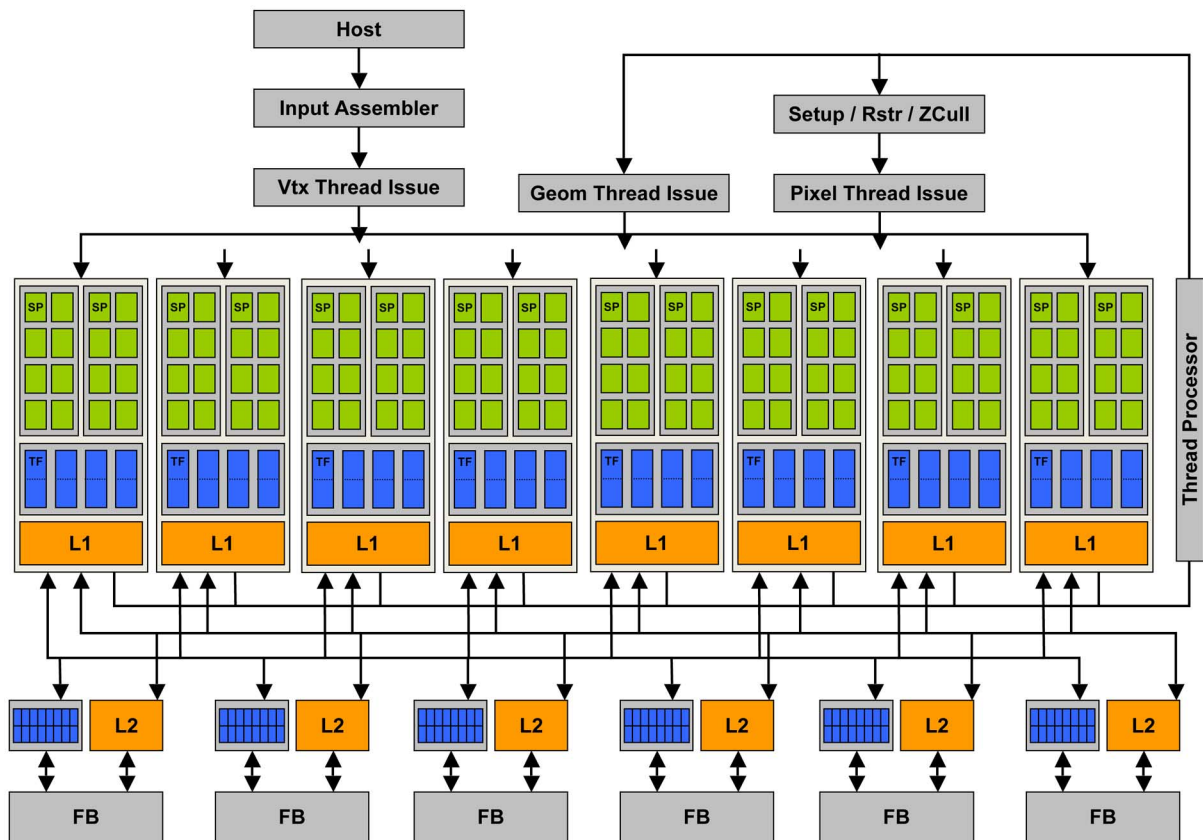
Fig. 1. *The GeForce 8800 GTX GPU with 128 processors is well suited to execute data parallel algorithms.*

II. COMPUTER ARCHITECTURE: WHAT HAPPENED?

Power dissipation in clocked digital devices is proportional to the clock frequency, imposing a natural limit on clock rates. While compensating scaling has enabled commercial CPUs to increase clock speed by a factor of 4000 in the last ten years, the ability of manufacturers to dissipate heat has reached a physical limit. Leakage power dissipation gets worse as gates get smaller because gate dielectric thicknesses must proportionately decrease. As a result, a significant increase in clock speed without expensive cooling is not possible. This is the power wall confronting serial performance, and significant clock-frequency increases will not come without heroic measures or breakthroughs in materials technology.

Not only does clock speed appear to be limited but also memory performance improvement increasingly lags behind processor performance improvement. This introduces growing memory latency barrier to computer performance improvements. To attempt to improve the average memory reference time to fetch or write instructions or data, current architectures are increasing

cache sizes. For instance, the current high-end 65 nm Intel Xeon 7000 series CPUs have 8 MB of L2 cache, whereas a generation older high-end 90 nm Intel Xeon processors have 4 MB L2 cache. The cache size has been increased because cache misses are expensive, causing delays of hundreds of (CPU) clock cycles. However, even with the increased cache size, the decrease in memory access latency has not kept pace with processor speeds. The mismatch between memory speed and computation speed presents a memory wall to increase serial performance.

In addition to the performance improvements that have arisen from frequency scaling, hardware engineers have also improved performance by speculative execution of future instructions before the results of current instructions are known. This also requires hardware safeguards to prevent potential errors from out of order execution. Unfortunately, branches must be guessed to decide what instructions to execute simultaneously. This is called instruction-level parallelism (ILP). A big benefit of ILP is that existing serial programs enjoy performance benefits without any modification. But ILP improvements are

difficult to forecast since the speculation success is difficult to predict, and ILP causes a superlinear increase in execution unit complexity and associated power consumption without linear speedup in application performance. For example, if the processor speculates to execute the incorrect branch, then it must throw away this part of the result. In addition, data dependencies may prevent successive instructions from executing in parallel, even if there are no branches. Serial performance acceleration using ILP has stalled because of these effects. This is the ILP wall.

Patterson from Berkeley has a formulaic summary of the serial performance problem: The power wall + the memory wall + the ILP wall = a brick wall for serial performance. Thus, the heroic line of development followed by materials scientists and computer designers to increase serial performance now yields diminishing returns [4]. Computer architects have been forced to turn to parallel architectures to continue to make progress. Parallelism can be exploited by using the additional transistors made possible by Moore's law to add more independent CPUs, data-parallel execution units, additional registers sets for hardware threads, bigger caches,

and more independent memory controllers to increase memory bandwidth. For example, for high-performance graphics workloads, the AMD Radeon 2900 GPU has several memory controllers to provide a 512-bit memory interface. This architecture has a peak memory bandwidth of more than 100 GB/s. Computer architects can also consider incorporating different types of execution units, heterogeneous processors, which dramatically improve certain specific computations. For example, GPUs such as the NVIDIA 8800 GPU in Fig. 1 excel at data parallelism, while streaming execution units can be paired with local memory as in many early Cray machines or the IBM Cell processor. Heterogeneity need not only mean completely different abstract execution unit models but may also include computation engines with the same instruction set architecture but different performance and power consumption characteristics. All of these take advantage of dramatically higher on-chip interconnect data rates.

For the foreseeable future, Moore's law will continue to grant computer architects ever more gates. The challenge is to use them to deliver performance and power characteristics fit for their intended purpose. Fig. 2
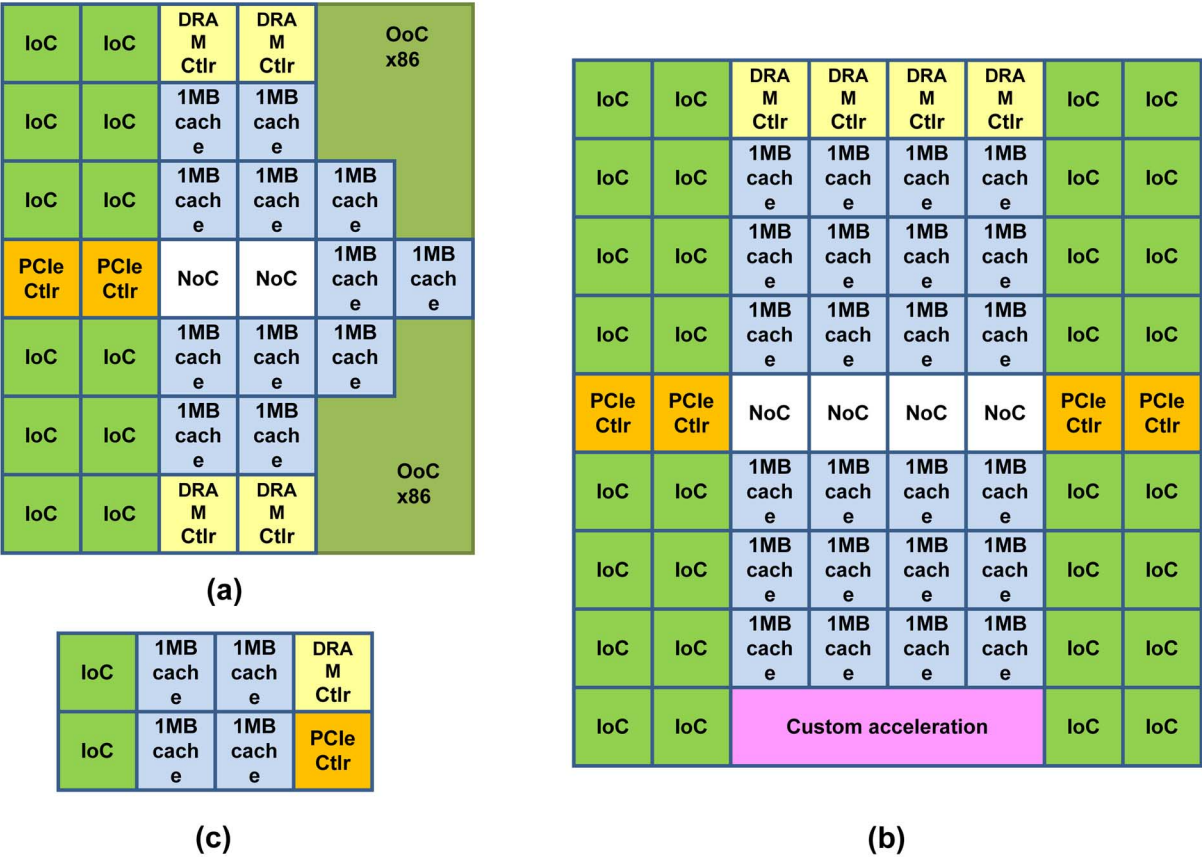

Fig. 2. *Sample client, server, embedded chips with network-on-chip, in-order cores, PCIe controllers, memory controllers, cache, and-out of-order cores.*

illustrates a few hardware design choices. In Fig. 2(a), a client configuration might consist of two large out-of-order cores (OoCs) incorporating all the ILP of current processors to run existing programs together with many smaller in-order cores (IoCs) for programs that can take advantage of highly parallel algorithms. Why many IoCs rather than correspondingly fewer of the larger OoCs? The reason is that spending gates on out-of-order cores can have poorer performance returns due to limited ILP in typical programs and many of the execution units in OoCs are usually idle for a significant fraction of the program execution time. Alternatively, simple IoCs can utilize the gates better on parallel software that can scale with the number of cores. The Intel Terascale chip [9] and the Xbox 360 CPUs [10] are examples of architectures with IoC CPUs.

The server configuration in Fig. 2(b) incorporates many more IoCs and a custom core (e.g., a cryptography processor). Finally, multicore computers are beneficial not just for raw performance but also for reliability and power management. We believe embedded processors could benefit from an architecture shift as illustrated in Fig. 2(c).

III. PARALLEL SOFTWARE DEVELOPMENT

While these new hardware architectures offer much more computing power, it makes writing software that can fully benefit from the hardware much harder.

In scientific applications, improved performance has historically been achieved by having highly trained specialists modify existing programs to run efficiently as new hardware became available. In fact, even rewriting existing programs in this environment was far too costly, and most organizations focused the specialists on rewriting small portions of the mission critical programs, called kernels. In the good case, the mission-critical applications spent 80% or 90% of their time in these kernels and the kernels represented a few percent of the application code. Thus making a kernel ten times faster could mean a nearly 3–5$\times$ performance improvement. Even so, this rewriting was time consuming, and organizations had to balance the risk of introducing subtle bugs into well-tested programs against the benefit of increased speed at every significant hardware upgrade. All bets were off if the organization did not have the source code for the critical components.

In contrast, commercial vendors have been habituated to a world where all existing programs get faster with each new hardware generation. Software developers could confidently build new innovative software that barely ran on then-current hardware, knowing that it would run quite well on the next-generation machine at the same cost. This will no longer occur for serial codes, but the goal of new software development tools must be to retain this very desirable characteristic as we move into the era of many-core computing. If we are successful, then building your software with the new tools will allow it to execute faster when new hardware adds additional parallel computing power (e.g., an upgrade from a 32- to 128-core system).

In order to benefit from rapidly improving computer performance and to retain the "write once, run faster on new hardware" paradigm, commercial and scientific software must change to new software development and system support mechanisms [1]. Software development systems and supporting software must enable a significant portion of the programming community to construct parallel applications. Several complementary approaches may help us achieve this.

1) *Encapsulate domain-specific knowledge in reusable parallel components.* For most developers, the most effective way to deploy concurrency without needing to disturb the programming model is to encapsulate concurrency together with domain knowledge in common reusable library components. This approach mirrors the use of numerical and signal-processing kernels beloved by computational scientists such as ATLAS [5], LAPACK [6], FFTW [7], and SPIRAL [8]. However, we must move these types of libraries into the world of general-purpose computing. This technique can work very well, although use of multiple such libraries in the same program requires better synchronization and resource-management techniques than are currently available.

2) *Integrate concurrency and coordination into traditional languages.* Current languages have little or no support for expressing or controlling parallelism. Instead, programmers must use libraries or OS facilities. Other language features, like the use of for/while loops and pointer-chasing, obscure potential parallelism from the compiler. To build parallel applications, we need to extend traditional sequential languages with new features to allow programmers to explicitly guide program decomposition into parallel subtasks, as well as provide atomicity and isolation when those subtasks interact with shared data structures. Transactional memory [2] shows promise here and also provides a way towards composing independently developed software components.

3) *Raising semantic level to eliminate explicit sequencing.* Parallelism can be more effectively exploited by avoiding procedural languages and using domain-specific systems based on rules or constraints. Programming styles that are more declarative specify intent rather than sequencing of primitives and thus inherently permit parallel implementations that leverage the concurrency and transaction mechanisms of the system. SQL is

a common example of this. SQL is declarative language that can be adapted by a query optimizer to execute much faster on parallel hardware. Another example is the MapReduce middleware used for data analysis. The programmer uses map and reduce functions to implement algorithms while a runtime is used to execute the program on distributed clusters [11], [13] or multicore CPUs [12]. Similarly, DirectX 10 [15] is another successful domain-specific declarative API for graphics workloads.

To fully exploit parallelism, however, programmers must understand the parallel execution model, develop parallel algorithms, and be equipped with much better tools to develop, test, and automatically tune performance of parallel programs. This requires education as well as software innovation. Compilers, which bridge between intent-oriented features and the underlying execution model of the system, must incorporate idioms to explicitly identify parallel tasks as well as optimization techniques to identify and schedule implicitly parallel tasks that the compiler discovers [3]. Program analysis and testing are difficult tasks in sequential programming and are much harder in parallel programming. We must find mechanisms that contain concurrency and isolate threads and then use these techniques to make testing more robust. We have seen dramatic improvements in static analysis tools that identify software defects, reduce test burden, and improve reliability. These techniques are being extended to incorporate identification of concurrency problems. Debuggers must evolve from the low-level machine model back to a more common and familiar model that allows a developer to reason effectively about correctness. Finally, the need for tools for performance analysis to help identify bottlenecks will become crucial as we face the possibility of orders of magnitude difference between optimized and naïve algorithms.

IV. SYSTEM SOFTWARE ARCHITECTURE

New, many-core computers are more like data-centers-on-a-chip than traditional computers. System software will have to change to manage resources effectively on these systems while decomposing and rationalizing the system software function to provide more reliability and manageability. General-purpose computer operating systems that have not fundamentally changed since system and application software separated with the advent of time shared computers in the 1950s will change as much as development tools.

To understand why, consider the following. Supercomputing applications are typically assigned dedicated system-wide resources for each application run. This allows applications to tune algorithms to available resources. Knowledge of the actual CPU resources and memory available to the application at runtime can be used

to drastically improve a sophisticated application's performance. Database systems do a good job of this now by using facilities to allow the database to control its own resources. In contrast, most commercial operating systems time-multiplex the hardware resources to provide good utilization of expensive resources. Older operating systems also suffer from service, program, and device isolation models, which are no longer appropriate but made perfect sense given earlier assumptions:

1) Many current operating systems manage devices with a uniform device driver model.[1] If all such drivers are in the same address space, this simplifies I/O programming for applications and optimizes performance but creates huge OS kernels with management and security problems.

2) Time-shared operating systems model security under a single authority (the root or administrator accounts) who installs all software that is shared or requires OS modification software and can determine a uniform security and resource-allocation policy across relatively simple user programs. Today's computers operate in multiple trust domains, and different programs need different levels of protection and security policies. There are so many devices, and some are so complex that no single authority can possibly uniformly and safely manage them all. Right now, a buggy device driver used by one program jeopardizes all programs, while highly performant applications using special hardware such as GPUs prefer to manage the device directly without incurring the sometimes catastrophic degradation incurred by context switches in the OS.

3) Homogeneous operating systems are usually designed for one of three modes of operation: high throughput, high reliability, or high real-time guarantees. General purpose OSs fall into the first category, an OS designed to run a central phone switch in a major location falls into the second, and an entertainment or media device falls into the third. It is difficult to design a single scheduler that serves all three environments, but future computers will have applications with all these requirements running simultaneously.

4) Most general-purpose operating system configurations attempt to provide everything any application could want. This approach has dramatically increased OS complexity by decreasing utility and slowing down all application development. For instance, specialized operating systems for relational database systems are more scalable for database workloads and exploit domain knowledge that general-purpose operating systems do not [14].

[1]http://www.project-UDI.org/specs.html.

5) Most operating systems, again to simplify programming, have a chore-scheduling model in which each independent thread of execution is scheduled by the OS. This means that every chore switch incurs an expensive context switch into the kernel. The OS scheduler, which knows nothing about the individual application, must guess what's best to do next. Historically, operating systems may have given their applications 1 ms to run before interrupting and rescheduling. Switching to another thread might have taken 100 instructions. On a 1 MIP machine, this means a thread can run about 1000 instructions of useful work, so system overhead is a very acceptable 10%. On a very fast machine, a millisecond accounts for a few million instructions. It is very hard to write general-purpose programs where this quantum of instructions yields a highly concurrent duty cycle. This forces programs with high concurrency to structure themselves into bigger, less parallel subtasks or suffer catastrophic performance. One solution is to let a runtime, linked into the application, handle the vast majority of chore switches without OS intervention. This kind of runtime can have very detailed knowledge of the actual hardware configuration and make resource and scheduling decisions appropriately. Examples of these include DirectX or OpenGL runtimes used to run graphics programs.

Many-core operating systems (see Fig. 3) will incorporate a hypervisor, a small and very reliable component that hosts many different operating systems or copies of the same OS with different performance or security characteristics. Hypervisors perform the relatively slowly changing space sharing of resources. For example, a hypervisor might simultaneously dedicate a core for long periods of time to a multimedia OS partition/application combination and assign I/O devices to it; host an older version of an OS for compatibility; host a tightly controlled corporate partition; host a game partition requiring strong performance guarantees; and host a loosely controlled partition for Web browsing, all on the same hardware [17]–[20]. Each partition can be sure of both performance and security isolation, and one partition need not incur the performance, reliability, or security characteristics of another.

A many-core system stack that includes hypervisors, OS kernel, and user-mode runtime must effectively *assign resources securely and host concurrent operating environments*. Machine-wide and OS health (root-kit detection, OS stack), power management, and coarse hardware resource allocation can be managed centrally while insulating partitions from harmful effects of other partitions. As with other software decomposition strategies, this simplifies software construction. Coarse partitioning also provides a good way to get coarse parallelism. Applications running concurrently in separate trust domains need the benefits of either rich operating environments or specialized environments that provide
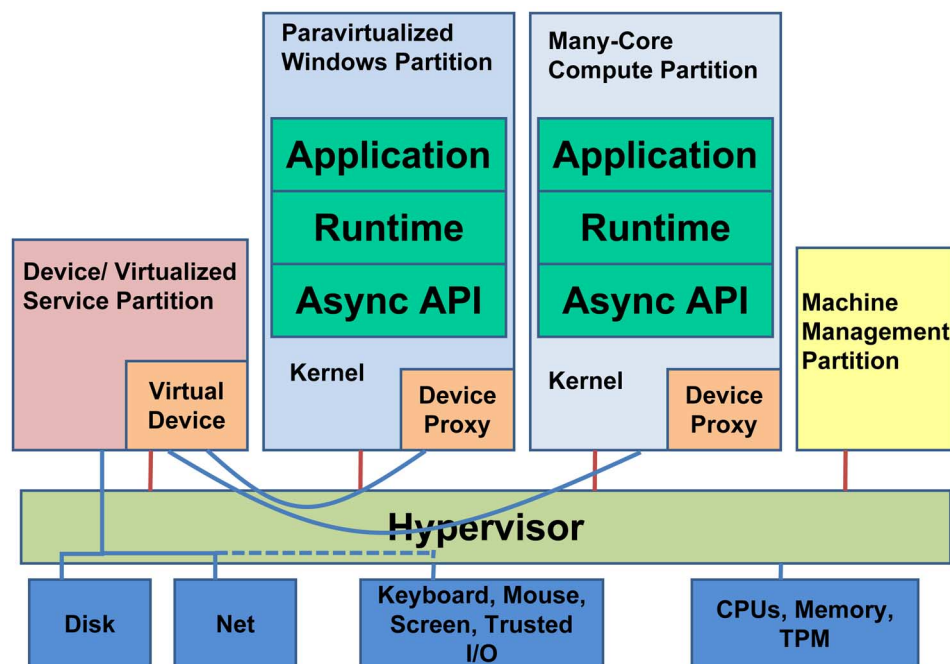

Fig. 3. *Many-core system architecture.*

specific guarantees (such as real-time scheduling). This also provides a vehicle to stage new facilities while retaining unmodified legacy environments. Each OS partition can exercise finer resource control over its resources in conjunction with its application mix. Within a process, the application and supporting runtime in conjunction with the OS can exert very fine grain control over resources. Furthermore, the OS must include a better asynchronous system API and lightweight native threads (see [16]). Finally, the system stack must manage heterogeneous hardware: general-purpose cores, GPUs, vector units, and special cores such as encryption or compression.

V. NEW APPLICATIONS

Can people use this much computing power? Yes. Although the ultimate application mix is hard to forecast, we can certainly envision applications and facilities that can use this kind of computing power.

It is uncontroversial that servers (including home servers) will benefit from many-core computing. Powerful servers will also boost the need for powerful clients. With cheap and ubiquitous sensors and natural language processing, we can anticipate human/computer interfaces that are environment aware, utilize a variety of input and output modes (i.e., vision, speech, gesture, object recognition), learn user behaviors, and offer suggestions or possibly automatically manage some tasks for users.

Better data mining and modeling will provide business intelligence and targeted customer service. Automated medical imaging, diagnosis, and well-being monitoring will be commonplace. High-level tools like MATLAB or Excel when designed for parallelism will take advantage of increased power and delegate processing across the network.

With terabyte disks, many-core systems will make superb media library, capture, edit, and playback systems. Film fans will be able to purchase, download, and view protected feature films on opening day. Most printed material and other media can be replaced with electronic versions, accessed via a broadband connection, with vastly improved search and cross-reference capabilities. These machines can make virtual reality and realistic games, well, real. Not only entertainment but also education will benefit as realistic simulation and training experiences are built.

Today's corporate servers will shrink to a few racks and become highly resilient to failure. State check pointing and load balancing will improve performance and reliability. Damage from catastrophic failures is limited to a few seconds of downtime and rollback. Provisioning, deploying, and administering these servers and applications are simplified and automated.

Massively parallel computational grids built of commodity hardware already solve scientific problems like computational chemistry, protein folding, and drug design. Supercomputers already analyze nuclear events and water tables and predict the climate and the economy. The power of these systems and the reach of these techniques will vastly improve with new hardware, and scientists will have supercomputers under their desks. Moreover, scientific, financial, and medical supercomputing are no longer small market opportunities. More than 10% of servers are used in scientific applications.

Classic computational techniques known in the scientific community as the seven dwarves,[2] including equation solvers and adaptive mesh modeling, will help explore regimes that will change our lives [4]. Already, many world-class scientists are using advanced computational techniques to explore potential cures for AIDS and cancer, model hydrologic activity in agriculturally sensitive regions, perform seismic modeling, and run virtual laboratories for advanced physics. As in the past, use by scientists will help illuminate the path for the rest of us.

VI. CONCLUSION

Programmable systems are playing an increasingly large part in our lives and, in many ways, provide a worldwide paradigm shift comparable in scope and benefit to the appearance of cheap, mass market printing. Many-core computers signal a shift in computer science, computational science, and classical commercial software that marry the past advances of many knowledge workers as well as provide avenues for qualitatively new advances. ∎

[2]So named by Collela of LBL, these include the important computational kernels for modeling and analysis.

REFERENCES

[1] J. Larus and H. Sutter, "Software and the concurrency revolution," *ACM Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005.

[2] J. R. Larus and R. Rajwar, *Transactional Memory.* San Rafael, CA: Morgan & Claypool, 2006.

[3] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures.* Amsterdam, The Netherlands: Elsevier, 2002.

[4] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The landscape of parallel computing research: A view from Berkeley," UCB/EECS-2006-183, 2006.

[5] R. C. Whalley, A. Petitet, and J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Comput.*, vol. 27, no. 1–2, pp. 3–35, 2001.

[6] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche, "Self adapting software for numerical linear algebra and LAPACK for clusters," *Parallel Comput.*, pp. 1723–1743, 2003.

[7] M. Frigo, "A fast Fourier transform compiler," in *Proc. PLDI 1999*, 1999.

[8] M. Puschel, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veleso, and R. W. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing algorithms," *Int. J. High Perform. Comput. Applicat.*, vol. 18, no. 1, pp. 21–45, 2004.

[9] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. S. Vaidya, "Integration challenges and tradeoffs for terascale architectures," *Intel Technol. J.*, vol. 11, no. 3, 2007.

[10] J. Andrews and N. Baker, "Xbox 360 system architecture," *IEEE Micro*, vol. 26, no. 2, pp. 25–37, 2006.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. OSDI*, 2004, pp. 137–150.

[12] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. HPCA 2007*.

[13] H. Yang, A. Dasdan, R. Hsiao, and D. S. Parker, "Map-reduce-merge: Simplified relational data processing on large clusters," in *Proc. ACM SIGMOD 2007*.

[14] J. Gray, *Notes on Data Base Operating Systems*, ser. Lecture Notes in Computer Science.   London, U.K.: Springer-Verlag, 1978, vol. 60, pp. 393–481.

[15] D. Blythe, "The Direct3D 10 system," in *Proc. ACM SIGGRAPH 2006*, 2006.

[16] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A trusted open platform," *IEEE Computer*, vol. 36, no. 7, pp. 55–62, 2003.

[17] R. J. Adair, R. U. Bayles, L. W. Comeau, and R. J. Creasy, "A virtual machine system for the 360/40," IBM Cambridge Scientific Center, Cambridge, MA, Rep. 320-2007, May 1966.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. ACM SOSP*, 2003, pp. 19–22.

[19] J. S. Hall and P. T. Robinson, "Virtualizing the VAX architecture," in *Proc. ISCA*, 1991, pp. 380–389.

[20] E. C. Hendricks and T. C. Hartmann, "Evolution of a virtual machine subsystem," *IBM Syst. J.*, pp. 111–142, 1979.

ABOUT THE AUTHORS

John L. Manferdelli received the Bachelor's degree in physics from Cooper Union for the Advancement of Science and Art, New York, and the Ph.D. degree in mathematics from the University of California, Berkeley.

He has been a Senior Researcher, Software Architect, Product Unit Manager, General Manager, and most recently a Distinguished Engineer with Microsoft. His contributions include the development of the next-generation secure computing base technologies and the rights management capabilities currently integrated into Windows, for which he was the original architect. He also was with Microsoft Research and the SQL Server Group. He joined Microsoft in February 1995 when it acquired his company, Natural Language Inc., based in Berkeley. He is currently General Manager of incubation in the CTO office. At Natural Language, he was the Founder and, at various times, Vice President of R&D and CEO. Other positions he has held include Staff Engineer with TRW Inc., Computer Scientist and Mathematician with Lawrence Livermore National Laboratory, and Principal Investigator with Bell Labs. He was also an Adjunct Associate Professor at Stevens Institute of Technology and is an Affiliate Faculty Member at the University of Washington.

Naga K. Govindaraju received the B.Tech. degree in computer science from the Indian Institute of Technology, Bombay, in 2001 and the M.S. and Ph.D. degrees in computer science from the University of North Carolina at Chapel Hill in 2003 and 2004, respectively.

He is a Senior Researcher in the Many-core Technology Incubation Group, Microsoft Corporation. Before joining Microsoft, he was a Research Assistant Professor in the Department of Computer Science, University of North Carolina at Chapel Hill. His research focuses on the design of efficient parallel algorithms to solve several computational problems in computer graphics, databases, and high-performance computing. He has published more than 40 peer-reviewed articles in major graphics, database, and HPC journals and conferences such as *ACM SIGGRAPH, ACM SIGMOD,* and *ACM SuperComputing.* He has organized and presented tutorials at ACM SIGGRAPH, ACM SuperComputing, Eurographics, VLDB, and IEEE ICDE. He has also served on the program committees of many conferences.

Dr. Govindaraju received the IEEE VR PRESENCE best paper award in 2005 and the Indy PennySort award in 2006 for designing the world's best reported performance/price sorting algorithm for large data-management systems.

Chris Crall received the B.S. and M.S. degrees in computer science from Iowa State University, Ames, with an emphasis in operating systems and networking.

He is a Group Program Manager with the Incubation Team, Microsoft, under the Chief Research and Strategy Officer, where he is responsible for a team working on many-core technologies. After graduation he spent 14 years in product development with Hewlett-Packard. He started as a Developer on TCP/IP networking, OSI networking and network management, and later as an Architect for Internet security products at HP. After HP, he joined Commerce One, an electronic commerce company, as a Security Architect developing e-commerce portal products. For the last five years, he has been a Program Manager with Microsoft working on Windows security as well as many-core incubation efforts.