Preface

This issue of the *IBM Journal of Research and Development* focuses on exploratory systems. As the name suggests, exploratory systems cover a wide range of cutting-edge technologies that will make future systems better: faster, lower-power, more reliable, more expandable, easier to use, easier to design. The systems in question span a wide range from small embedded devices such as those found in cell phones, washing machines, and animal-tracking collars to installations with thousands of processors and petabytes of storage for handling large databases or for large-scale simulation.

The first set of papers in this issue explore how a wide range of systems can be supported with a few simple building blocks. The paper by Agerwala and Gupta surveys the challenges in constructing a large, robust system from a large number of relatively simple processors. Difficulties range from the fact that an individual component in a very large system can be expected to fail every few days to employing enough parallelism via algorithms, compilers, and runtime optimizers to utilize thousands of processors. The paper uses examples from the Blue Gene® project to illustrate these challenges.

In a similar vein, Wilcke et al. describe how a data center can be composed of large numbers of atomic elements, i.e., "intelligent bricks" of storage. In addition to defining the properties required in individual bricks to allow efficient construction of large systems, the paper also shows how bricks permit high-density storage—not areal bit density on an individual disk platter but volumetric efficiency (i.e., the number of bytes that can be stored in a cubic meter). The question then arises, Can a dense system of bricks be efficiently cooled? The answer is "yes," as the paper shows in part by presenting measurements and experience with a prototype model. In a companion paper, Fleiner et al. examine the reliability of brick systems. Even if one or more bricks fail, it is important that the system not lose data, continue running with no immediate human intervention, and suffer little or no loss in performance. Through Monte Carlo simulations, the authors show that brick-based systems score very well on all of these metrics.

Given this foundation of large systems built from simple building blocks, the next set of papers describe how large systems can be efficiently managed and used. Bacon and Shen tackle the memory wall problem—the increasing number of processor cycles required to retrieve data from memory. They do so by taking advantage of the growing numbers of threads and processors in large systems. The observation that a second thread can run while a first waits for data from memory is not new. However, exploitation of this observation has been limited by the difficulty in finding independent threads within one program or by having a large enough collection of simultaneously running independent programs. To overcome these difficulties, Bacon and Shen introduce the high-level language constructs of braids and fibers, which enable adaptive responses to memory latencies. They show that a small amount of new architecture and microarchitecture efficiently supports braids and fibers and significantly helps mitigate the memory wall problem.

Dongarra et al. discuss a three-pronged approach employed at the University of Tennessee for efficiently mapping numerical problems to large systems: 1) choosing the right algorithm; 2) tuning kernels to run well on particular processors on the basis of latency and bandwidth to cache, number of processor execution units, etc.; and 3) managing the system, e.g., assigning computations (threads) to processors, especially so as to adjust to the changing state of the entire system—whether these changes be idle processors or failed processors. In their paper, Caşcaval et al. discuss alternative means of handling the second and third problems. In particular, they use the performance-monitor counters present in most modern processors to detect sub-optimal system performance and adjust execution to help overcome such problems. A particularly effective example of such tuning is adjustment of the page size. When the processor counters report many misses in the TLB (translation lookaside buffer), the data may be spread over more pages of memory than the processor can efficiently track. To overcome this problem, this paper finds that it is often helpful to increase the amount of data stored in specific pages. Many modern processors and operating systems provide efficient mechanisms to make such changes, for example by allowing the page size for a problematic set of data to be changed from 4 kilobytes to 16 megabytes.

Even once the techniques proposed in the papers above allow applications to effectively use large systems, there remains a need for people to find and use these applications and systems. A promising direction in this regard is grid systems, whereby a user "plugs in" to a large array of machines available on a network and uses their combined computational power to solve a problem. Kandaswamy et al. describe a novel web service which allows scientific applications to be used, unmodified, by other users on the network. In addition, multiple scientific applications can be composed in this framework, so as to allow new meta-level applications to be built.

The previous set of papers all assume that correct, debugged programs are being run. However, it often requires a great deal of time and effort for a program even to approach this state. Some of the most common bugs relate to improper management of memory (at least in C

and C++ programs). For example, memory may be allocated, but not freed when it is no longer needed, or pointers may point to memory that has been (erroneously) freed. Shetty et al. propose to use some of the processors and threads available on larger systems to automatically detect such problems and report them to the programmer. Thus, some resources are used for reliability (program correctness) instead of performance.

All of the papers thus far have dealt with the use of large systems and the construction of large systems from simple components. For a system to be effective, it is important that these "simple" components be designed well. The final set of papers deal with issues in this area, in particular the circuits with which microprocessors are built, the tools used to design them, and some important aspects of those designs.

In their paper, Belluomini et al. describe limited switch dynamic logic circuits (LSDL), a new static–dynamic hybrid circuit family with many desirable characteristics—in particular, low power, the ability to operate at high frequency, and simplified design, in the sense that evaluation of each LSDL circuit is triggered by a rising clock edge. As proof of concept, the paper reports on test implementations, including a 90-nm multiplier that can run at up to 8 GHz.

Good circuits are but one element of a good processor design. The higher-level structures in the processor pipeline must also perform well, executing multiple instructions in parallel with as little power as possible, but at high frequency and without an excessive number of stages in the pipeline. The load-store unit is very important in this regard, particularly in enabling parallel instruction execution, since much computation depends on data values loaded from cache or memory. Baugh and Zilles present a new technique for managing the queue of stores waiting to go to memory. Traditionally such queues have been structured around the order in which store instructions occur in a program. As a result, load instructions must search every element of the store queue to see whether the most recent value from the location being loaded resides in the store queue instead of in cache or in memory. Baugh and Zilles describe a novel method of organizing the store queue by store address instead of program order. As a result, loads can efficiently (and with less power) search only a small subset of the store queue, for which the address of a store may match the address being loaded.

The paper by Cheng and Tyson focuses on reducing power consumption. To achieve this goal, they restrict the set of instructions executed by a processor to the set of instructions needed to efficiently execute a particular (embedded) application. However, without going further, this approach would likely restrict the use of such a processor to a limited domain. The lower volumes and resultant higher costs of designing a processor for a limited domain would then serve to limit the appeal of this approach. To overcome this problem, Cheng and Tyson propose the use of programmable decoders, which can be tuned in the field for each domain or application. As a result, each domain executes its own limited set of instructions, thus saving power. Each domain or application executes a different set of instructions, thus maintaining flexibility.

Even with good circuits and good microarchitecture, there are often difficulties in fitting the two together. The microarchitecture breaks computation into pipeline stages and assumes that the circuits can complete the work in each stage in one cycle. If such assumptions are incorrect, major design changes may be required late in the design cycle. Even if these problems are avoided or overcome, there is an additional problem of floorplanning, i.e., deciding where on the chip to place the different parts of the design. Signals may take a full cycle or more to cross a chip. As a result, "bubbles" in the pipeline may be required to allow transmission of results from one stage to another. Such bubbles may not have been envisaged in the original microarchitecture, again resulting in major design changes late in the design cycle. Carter and Hussain address these and other problems with Justice, a simulation model that includes all of the elements just described, thus allowing these problems to be detected and addressed early in the design cycle.

Feedback cycles in processor design are not limited to those involving circuits, floorplanning, and microarchitecture. There are similarly important cycles involving microarchitecture, compilers, and operating systems. For example, how much does a particular microarchitectural feature speed up a particular program? How much does a particular compiler optimization improve performance? Local compiler scheduling of instructions may be less important to good performance in an out-of-order superscalar design than in a very long instruction word (VLIW) design. However, a VLIW design with compiler scheduling may require less power. In the final paper, Peterson et al. describe the Mambo full-system simulator, which allows fast, detailed simulation of this latter set of interactions and tradeoffs among microarchitecture, compilers, and operating systems. The paper provides particular insight into how the Mambo capabilities were used in the design of the Cell Broadband Engine™ and in the IBM PERCS project.

Preliminary versions of several papers in this issue were presented at the 2004 $P=ac^2$ Conference at the IBM Thomas J. Watson Research Center, September 28–30,

2004. In addition, we are grateful to Siddhartha Chatterjee for many helpful suggestions and ideas, and to Tilak Agerwala for his encouragement and support for this special issue of the *IBM Journal of Research and Development*.

Erik Altman
Architecture and Performance
IBM Research Division

Sumedh Sathaye
Microprocessor Architect
IBM Systems and Technology Group

Guest Editors