International Journal of Parallel Programming, Vol. 28, No. 4, 2000

# Introduction

Parallel programming has been around for at least three decades and has remained a difficult field. The biggest challenge arises when the main purpose of parallelism is to improve performance. It is notoriously difficult to make parallel programs operate correctly and efficiently. Although progress has been made on the semantics and verification of parallel programs in certain domains, no practical, general technique is known for the creation of reliable, portable parallel application code.

One approach towards this goal is to unburden the programmer from the difficult task of handling parallelism and delegate this to the compiler or the machine architecture. The research area that gives the programmer and compiler the control over coarse and medium-grained parallelism is *parallelizing compilation*, the research area that lets the compiler and the machine automatically extract very fine-grained parallelism is *instruction-level parallelism (ILP)*.

This and the following issue of the *International Journal of Parallel Programming* contain work presented at the Dagstuhl-Seminar on *Instruction-Level Parallelism and Parallelizing Compilation*, held at Schloss Dagstuhl on April 18–23, 1999. The aim of the seminar was to bring together these two research areas, which have developed side-by-side, but with little exchange of results. Both areas deal with similar issues: dependence analysis, synchronous vs. asynchronous parallelism, static vs. dynamic parallelization, and speculative execution. But the different levels of abstraction at which the parallelization takes place call for different techniques and impose different optimization criteria.

At the instruction level, the parallelism is transparent to the programmer, since it is automatically extracted from program parts that are atomic at the level of the programming language. The emphasis is on driving the parallelization process by the availability of architectural resources. Compile-time parallelization has been targeted at very long instruction word (VLIW) architectures and run-time parallelization at superscalar



0885-7458/00/0800-0321$18.00/0 

architectures. Heuristics are being applied to achieve good but, in general, suboptimal performance.

In parallelizing compilation, parallelism visible at the level of the programming language must be exposed. The programmer usually aids the parallelization process with program annotations, or by representing the program in a certain syntactic form. The emphasis has been on compile-time parallelization methods. One can apply either heuristics or an optimizing algorithm to search for best performance. Resource limitations can be taken into account during the search, or they can be imposed in a later step, e.g., through tiling or partitioning the computation domain.

The present issue is devoted to ILP and embedded systems, specifically, to the following topics.

## Compilation for Embedded Systems

Embedded software has traditionally tended to use low-level languages and hand-crafted techniques for optimizing execution time and memory usage. Given the scope for exploiting parallelism in embedded software, especially in multimedia applications, and the emergence of ILP processors, such as VLIW machines, there is a growing body of work investigating the automatic parallelization of high-level programs aimed at ILP targets. The challenge, of course, is to match the performance, under tight resource constraints, of traditional hand-crafted techniques.

Rohou, Bodin, Eisenbeis, and Seznec adopt a global constraint-driven strategy for guiding program optimization. Global criteria such as overall speed, code size and instruction cache behavior guide the decisions for code optimizations which, in turn, inform the choice and sequence of optimization strategies at the local levels. The authors present results for a linear optimization strategy which takes into account the code size and execution cost of loops in embedded programs, minimizing one while constraining the other.

## Sub-Word Parallelism

The traditional instruction sets of processors have been extended to exploit sub-word parallelism efficiently. In these multimedia extensions, a number of short data elements are packed in a single register and data-parallel operations are executed on them. Examples include the Visual Instruction Set for the UltraSPARC processor, the AltiVec for the PowerPC, the MMX extension for the Pentium processor, and the MAX-2 instruction set of the PA-RISC processor. At present, there is little or no compiler

support to exploit sub-word parallelism—the user is expected to handcode large parts of the application in assembly languages.

Krall and Lelait of the University of Vienna use the technique of vectorization by unrolling to automate this process. Data dependence analysis and dynamic run-time checking are used to handle unaligned memory accesses. Sreraman and Govindarajan of the Indian Institute of Sciences, Bangalore, use standard vectorization techniques on loops which are tailored for short vector lengths.

## Microarchitecture of Embedded Systems

The microarchitecture of future processors has been the subject of intense debate and is the topic of the paper by Corporaal, Janssen, and Arnold of the Technical University of Delft. They recognize that future architectures have to contend with varying workloads and longer communication and memory latencies and observe that communication is of primary importance in the design of future microarchitectures. In the transport-triggered architecture that they advocate, communication between functional units, and with the register files, is programmed explicitly; the computation is now a side-effect, triggered by the communication. All communication inside the microarchitecture is visible to the compiler, which leads to a number of communication-level optimizations that the compiler can perform to enhance performance. Embedded programs can be analyzed and implemented on a transport-triggered architecture with an optimal number of functional units and communication patterns.

The following issue will be devoted to parallelizing compilation.

D. K. Arvind (Edinburgh)
K. Ebcioglu (Yorktown Heights)
C. Lengauer (Passau)
R. Schreiber (Palo Alto)
*Guest Editors*