Parallel processing is at a critical point of its evolution. After a long period of intense support by government and academia, it slowly moves to derive the bulk of its support from the commercial world. Such a move brings with it a change of emphasis from record breaking performance to price/performance and sustained speed of program execution. The winning architectures are not only fast but also economically sound. As a result, there is a clear trend towards widening the base of parallel processing both in hardware and software. On the hardware side, this means use of off-the-shelf commercially available components (processors, adapters, interconnection switches) which benefit from the rapid pace of technological advancement fueled by the large customer base. Hence, parallel processing is becoming ubiquitous at all levels of computer technology. In microprocessor design, super-scalar techniques - issuing and executing several instructions in the same cycle - are now a standard. Workstations interconnected by a fast network approach the performance of parallel machines.

For a single stream of instructions, the compiler can take advantage of hardware parallelism present at the microprocessor level (multiple pipelined functional units, load/store units, instruction scheduling, speculative execution, etc.) by performing various code transformations (such as software pipelining, speculative multithreading, trace scheduling, hyperblock scheduling, etc.) to improve the instruction level parallelism that is available in the stream of instructions. For a multiprocessor system, the compiler can create parallel threads from a given task which can run mostly independently on the available processors to reduce the execution time.

The increasing importance of parallel processing encourages also standardization of parallel programming languages and tools. Yet, there is no evidence of convergence of the supported programming paradigms to a single model. The compilers must efficiently support different programming paradigms, such as data parallelism, message passing and object-oriented.

Single Program Multiple Data (SPMD) model is adequate for most scientific computation that require application of the same algorithm at many points of the computational domain. The basic compilation problems in this paradigm involve dependence analysis and loop transformation to increase parallelism. When a program is compiled for execution on a distributed memory machine, load balancing and optimization of the communication cost are of paramount importance.

Object oriented approach is being viewed by most people as the future of software engineering. It is expected that features of object orientation, such as, data encapsulation, function overloading and polymorphism improve the software development and maintenance process. However at the same time, object-oriented approach creates unique problems such as difficulty in supporting parallelism and increased stress on the memory subsystem.

The papers in this minitrack address the optimization topics discussed above. The first two papers concentrate on instruction level parallelism. To take advantage of the large number of functional units in today's microprocessors, compiler optimizations are needed to increase the available ILP in an instruction stream. Software pipelining achieves that by scheduling operations from different iterations of a loop for execution in the same processor cycle. The paper, "Improving Software Pipelining With Unroll-and-Jam," by S. Carr, C. Ding and P. Sweany, discusses an unroll-and-jam transformation that can be used to increase the amount of parallelism available in the innermost loop through software pipelining and improve the ratio of the number of memory operations per floating point operations within the nested loop.
The second paper, "Extending List Scheduling to Consider Execution Frequency," by M. Bourke, P. Sweany and S. J. Beatty discusses frequency based list scheduling (FBLS) that extends standard list scheduling by considering execution frequencies within a schedule. This is useful for global instruction scheduling methods that schedule groups of basic blocks, called meta-blocks, as though they were a single block.

In a parallelizing compiler, dependence direction vectors are the basis of many parallelizing transformations. In the paper, "The Janus Test: a Hierarchical Algorithm for Computing Direction and Distance Vectors," author J. Sogno considers a hierarchy of problems which differ from each other only in their data dependence direction vectors. The basic problem is defined without direction constraints and it can be solved through a fast and exact algorithm based on reduction and an integer dual simplex resolution. Then, a hierarchy of problems is solved, each subsequent problem directly built from the previous one.

Compiler optimizations for data parallelism can be enhanced by transforming a program into so-called single assignment form. The paper, "An Efficient Algorithm for the Creation of Single Assignment Form," by P. Pineo, presents an efficient algorithm for transforming unstructured Fortran code to either static single assignment or single assignment form. Experimental results in the paper show the extent of storage enlargement and program length increase incurred by the creation of single assignment code.

Loop parallelization defines the final loop structure and loop bounds of the code and therefore strongly influences the parallel program's performance. The next three papers discuss different aspects of such optimization. In "Practical Loop Generation," Z. S. Chamski and M. F. P. O'Boyle argue that the growing complexity of program transformations used in parallelizing compilers makes loop structure maintenance and reconstruction increasingly expensive. The algorithmic complexity of the associated computations significantly increases the total compilation time. Therefore, the authors propose to develop a reference framework for reasoning about the run-time impact of loop generation techniques, with the ultimate goal of allowing a compiler to select the appropriate technique based on the trade-offs between loop generation complexity and reduction in runtime overhead. The paper describes performance implications of a series of loop generation techniques, such as, Fourier-Motzkin pairwise elimination, Parametric Integer Programming. The compile-time behavior of each technique is also discussed.

The paper, "Run-time Parallelization for Loops," by S.-H. Kao, C.-T. Yang and S.-S. Teng, focuses on a loop transformation based on a run-time inspector/executor approach. The authors developed a general algorithm for constructing an inspector phase applicable to loop scheduling. It can determine at runtime the wavefronts of a loop with indirect array indexing of arbitrary complexity by building def-use table. The experimental results show that the new algorithm can handle data dependence patterns too complex for compile-time analysis, thereby demonstrating usefulness of such run-time loop scheduling.

The next paper, "A Practical Scheduling Scheme for Non-Uniform Parallel Loops on Distributed Memory Parallel Machines," by T. Y. Lee, C. S. Raghavendra and H. Sivaraman presents a global distributed control scheme (GDC) to schedule non-uniform loops on distributed memory architectures. GDC decentralizes scheduling controls among all processors with an attempt to put heavily loaded processors in charge of scheduling tasks. The paper includes the comparative evaluation of GDC and other scheduling schemes implemented on a 512 processor Intel Delta system.

The last two papers in this minitrack explore object-oriented paradigm for parallel processing. In "Application of an Object-Oriented Parallel Run-Time System to a Grand Challenge 3d Multi-Grid Code," C. Baillie, D. Grunwald and S. Vajracharya describe the implementation of a large three dimensional multi-grid physics application (QGMG) under the DUDE object-oriented run-time support system. QGMG provides an opportunity for task parallelism because two multigrids can be solved simultaneously. The data parallelism can be used as well during solution of each multigrid. Every step of the solution must be synchronized on a barrier. To address these challenges, DUDE system does not require each process to wait at a barrier for all other processes to reach it. DUDE also supports an integration of task and data parallelism in a single application.

In the last paper, "Prefetching Strategies for Partitioned Shared Objects," S. B. Hassen investigates a parallel programming model in which partitioned shared objects encapsulate the state and the distribution of data structures in shared-data objects. The paper describes the object model and Partition Dependency Graphs (PDGs). The authors developed a technique for prefetching data in the runtime system that reduces access overhead and increases the overlap between computation and communication at execution time. The paper illustrates the effectiveness of this technique with two applications: Successive Over-Relaxation (SOR) and Fast Fourier Transform (FFT).