Ubiquity Symposium

The Multicore Transformation

Auto-Tuning Parallel Software: An interview with Thomas Fahringer

by Walter Tichy

Editor's Introduction

In this interview conducted by Ubiquity *editor Walter Tichy, Prof. Thomas Fahringer of the Institute of Computer Science, University of Innsbruck (Austria) discusses the difficulty in predicting the performance of parallel programs, and the subsequent popularity of auto-tuning to automate program optimization.*

Ubiquity Symposium

The Multicore Transformation

Auto-Tuning Parallel Software: An interview with Thomas Fahringer

by Walter Tichy

Most programmers who initially try parallel programming face disappointment: Their parallel code runs much slower than expected, sometimes even slower than the sequential code. Even experienced programmers spend considerable time and effort tuning parallel applications. Auto-tuning is a promising approach that makes that task easier.

***Ubiquity*:** Most programmers do not tune sequential programs. It seems the sequential machine model is fairly predictable. What makes it so hard to predict the performance of parallel programs?

Thomas Fahringer: Although developers of business and server applications may not need to optimize their codes, some programmers of scientific applications actually spend a considerable amount of time tuning sequential programs. A good example is tiling of programs to improve cache behavior. This is non-trivial and very sensitive with respect to the underlying hardware. Experiments have shown that tuning a serial program for cache locality can improve performance of up to an order of magnitude. Often programmers first tune the performance of the serial parts of a program and then focus on the parallel parts.

Predicting the performance of parallel programs is one of the most difficult challenges in parallel computing as it involves unpredictable properties such as how many times a program statement is executed or how long jobs wait in system queues. In-depth knowledge is required as to what data and tasks are processed by which core, and a detailed system and architecture model is needed to consider the costs of basic computations, data accesses, data transfer and synchronization, and the behavior of runtime systems (e.g. scheduling and load balancing). For instance, scheduling is often dependent on runtime situations such as external load or

computational work that is data dependent. Synchronization commonly implies waiting times, which necessitates accurate models for all activities of involved threads and cores. Predictions are often hardware, system, compiler and data dependent. Developing prediction models is a notoriously difficult task. Even automatic approaches, for instance, that are based on machine learning techniques suffer from huge search spaces or use simplifications that limit their applicability.

***Ubiquity*:** Auto-tuning is getting a lot of attention in the parallel programming community. What is it, and what is the objective of tuning?

TF: The idea of auto-tuning is to automatically adapt the execution of a program to a given software and hardware environment to optimize one or more non-functional objectives such as execution time, energy consumption, or computing costs.

Auto-tuning is not to be confused with auto-parallelization. Auto-tuning does not parallelize but builds on an existing parallel program and explores different trade-offs among parallelism, synchronization, load balancing, locality, and other parameters.

Before auto-tuning became popular, many optimization techniques followed a fixed strategy that used a performance model to guide a transformation environment or a runtime library to select specific strategies. Experiments demonstrated that those fixed strategies can often be outperformed by exploring a larger search space of transformations and tunable parameters. Interesting is also when auto-tuning of programs started. Its beginning may go back to the late sixties when researchers such as David Sayre (IBM) and Domenico Ferrari (Berkeley University) explored compiler optimization to enhance locality and other performance aspects of programs. Since then, auto-tuning has become a well-established technique that has been continuously improved to automate program optimization. However, the cost of tuning can be significant, so search heuristics must be used to reduce the search space.

***Ubiquity*:** What are some of the tuning parameters that an auto-tuner manipulates?

TF: This depends on the optimization goals. Most auto-tuners are limited to single optimization objectives such as communication, work distribution, execution time, or energy consumption. These techniques refer to mono-objective auto-tuning. Currently, more and more tuners are being upgraded to deal with dual or multi-objective optimization for handling two or even more optimization objectives. For each objective a specific set of tuning parameters is required. For

instance execution time can be tuned by using tile sizes, scheduling parameters, number of threads or cores, data and work distribution, and others. Energy consumption can be influenced by voltage and frequency settings but also by execution time sensitive parameters. Computing costs as a whole are influenced by execution time parameters, memory size, and data storage allotted. Program transformations are tuning options for all three optimization goals.

A particular problem is to detect and implement those tuning parameters that have a significant impact on a given optimization objective and ignore those with negligible effect. Moreover, the search space may explode due to the large value ranges of these parameters. Even worse, changing the value of one parameter may improve a certain optimization objective at the cost of another.

***Ubiquity*:** Are these tuning parameters numeric in nature, or are there other options? For instance, could a program's control structure change?

TF: In many cases these parameters are numerical values specifying for instance the number of threads to be used, frequency settings of cores, and tile sizes. However, in principle nothing speaks against non-numeric values such as scheduling directives or transformation sequences. For example, an auto-tuner could explore different OpenMP loop scheduling directives such as static, dynamic or guided. Each of these alternatives can be considered an element of a set of search options, which can be explored by an auto-tuner. Control structures could change based on transformations applied to a program or alternate code-versions provided for exploration.

***Ubiquity*:** Is an application tuned once and for all (perhaps on each platform), or are there other considerations? When does tuning take place?

TF: Unfortunately, there is no tuning that can be frozen. It has been clearly shown that optimization results can substantially change for different input data and hardware. There is no satisfactory solution so far that can deal with varying input data. Usually, auto-tuning is repeated for each target architecture. Tuning has to be adjusted if additional optimizations should be considered or previous optimization goals are dropped.

Auto-tuning may take place during an offline tuning phase, during runtime, or both. Offline tuning is done with a set of benchmarks. The hope is that well-performing settings of tuning parameters determined on the benchmarks work well in production runs. Runtime auto-tuning means that tuning parameter values are set during the actual production run. Clearly, the

advantage of runtime tuning is that all tuning-relevant information is available. But its effectiveness depends on whether the program runs long enough so that the effect of each parameter setting on an optimization objective can be explored.

***Ubiquity*:** How does a programmer prepare an application for tuning? What would be a simple implementation of getting auto-tuning going?

TF: There are different techniques to implement auto-tuners. A simple approach is to pass compiler flags, runtime parameters, and their values ranges when compiling or starting a program. A wrapper program than varies these arguments systematically for each compilation or execution. Also, the programmer can mark parameters in the code that should be tuned and provide their value ranges. This technique is implemented by inserting directives in the code or in the form of a separate file that provides the same information (tunable parameter, location in the program, and value range). Frequently, auto-tuners are provided with information about what transformations should be explored for specific code regions. Here, programmers for instance can indicate scheduling directives, loop unrolling or tiling parameters, or very specific transformation sequences. Constraints on tuning parameters can reduce the search space and also steer auto-tuning. More advanced techniques provide different algorithms or whole libraries which can then be chosen by the auto-tuner.

***Ubiquity*:** Are there language constructs that have built-in parameters?

TF: Yes, there are. A good example is OpenMP, which offers directives ideally suited for auto-tuning, for instance the size parameter for loop scheduling directives. High Performance Fortran is another directive based language with parameters for data distribution, array alignment, and processor numbers that can be used for tuning.

***Ubiquity*:** What would be the ideal solutions?

TF: In the ideal case the programmer does not have to change the code at all. It is left to the auto-tuner with the help of a sophisticated compiler and runtime system to determine the promising tuning parameters and their value ranges. A compiler would then expose these parameters and propose a default value range. During execution of an auto-tunable program,

the runtime system would then try out different settings of the parameters to optimize the program.

Stream programming languages such as XJava offer language constructs for expressing pipelines and master/worker patterns. These constructs provide implicit parameters such as the replication factor of pipeline stages, fusion of successive stages, or the number of workers. These are optimized automatically with an auto-tuner in the runtime system.

***Ubiquity*:** What if a program runs often, but never runs long enough to get to an optimal configuration. Is it possible to tune across many runs?

TF: Auto-tuner can pro-actively tune a code during an offline phase without using production runs. This makes sense for programs that will be executed on a few parallel computers over longer time periods. In this case offline tuning can achieve highly optimized results.

Auto-tuning means exploring the value ranges of tuning parameters and observing those values that improve the optimization objectives. Maintaining the results of historic tuning efforts makes sense as it can shrink the search space for future tuning efforts. Each additional tuning of a program could then reduce the value ranges to be explored and will shorten the time to finish the optimization.

***Ubiquity*:** An important ingredient is obviously the search algorithm. Where would I find a good search algorithm?

TF: There is a plethora of search algorithms for auto-tuning. Unfortunately, there is no single solution that works best for every situation. For mono-objective auto-tuning, random search is suitable if the difference between the best and worst alternative is small. Local search such as hill climbing is an option for solving optimization problems where exhaustive search is impractical. Local search methods iteratively modify the current parameter setting until no further improvement is possible. Local search can be applied to determine numeric parameters as well as sequences of program transformations. However, local search methods suffer from three drawbacks: (1) the computed optimum may depend on the starting point; (2) local search may get trapped in local optima; (3) numerous iterations may be necessary. Nelder-Mead and other simplex-based techniques are non-linear optimizers that converge faster. They extrapolate from a number of points called a simplex. In n-dimensional search space, n+1 points need to be evaluated initially, before the search can start. Some of these points may perform

poorly, which may be undesirable for online tuning. In principle, simplex-based searchers require continuous parameter spaces, but discrete adaptations of Nelder-Mead work satisfactorily.

Evolutionary computation represents a class of techniques that can be applied to mono-objective auto-tuning. Examples of evolutionary computation are genetic algorithms, genetic programming, evolutionary strategy or other methods that mimic the evolutionary behavior of certain species in nature such as ant colonies. All of these techniques share a common behavior. They work with a set of alternatives, in most cases generated randomly. This set is used to generate new alternatives by reusing the transformations and parameter values of the best configurations found so far. Evolutionary methods have been applied to tuning compiler flags or finding tile sizes and loop unroll factors for optimizing execution time. Evolutionary computation navigates the search space following a stochastic path that explores with a higher probability those areas where the best sequences of transformations and parameter values are located. Evolutionary techniques are popular as they can target almost any optimization problem and complex search spaces, and they have been shown to be robust. However, evolutionary techniques may require many iterations and are therefore best executed off-line, or when the code regions to be tuned are executed frequently. In such cases, auto-tuning can explore a reasonable large search space to find good solutions.

***Ubiquity*:** What if I need to tune several objective functions at once, say performance and energy?

TF: As many auto-tuning objectives may conflict with each other (e.g. energy consumption versus execution time, communication overhead versus computational effort, and resource speed versus economic costs), the result of this complex optimization problem is usually expressed by a set of trade-off solutions called Pareto front. Every solution on this front corresponds to a specific trade-off among different objectives. Each solution fulfills two conditions. Firstly, it cannot be further improved without worsening at least one of the objective functions. Secondly, none of the solutions is better than the others for all the objective functions. More formally a Pareto front consists of those solutions that are not dominated by any other alternative solution. A solution X dominates another solution Y if X outperforms Y regardless of the tradeoff between different objectives. X thus performs better in all objectives compared to Y.

A Pareto font is an essential tool for decision support and preference discovery whose shape can provide new insights and allow users to explore the space of non-dominated solutions with certain properties, possibly revealing performance aspects that are impossible to uncover otherwise. For example, it may happen that by conceding a few percentages in performance, the computing cost halves by using slower and cheaper resources, or by increasing the overall storage capacity by 10 percent, the execution time improves by 40 percent due to a better locality behavior. A Pareto front can expose alternative solutions if the importance of objectives changes, for example by giving economic costs priority over execution time.

An extra step is then required to select a single solution from the Pareto front that can be based on user-preferences.

***Ubiquity*:** The ideal would be if the programmer does not have to think about auto-tuning at all—tuning should be fully automatic. Is this achievable?

TF: For simple tuning problems with a well-defined set of tunable parameters and reasonable value ranges, auto-tuning can be automated. However, in general it is difficult to determine the tunable parameters and their values ranges that actually impact the objectives to be optimized. Auto-tuners are often limited to a pre-defined set of tunable parameter such as MPI runtime parameters, number of threads, scheduling strategy, clock rate, and voltage settings. I am not aware of an auto-tuner that exhaustively explores all possible transformation sequences. It is likely that many transformations that have an impact on the optimization objective are not considered by automatic approaches.

***Ubiquity*:** What are the issues that auto-tuning research must address next?

TF: Future research has to address techniques that avoid repetition of full auto-tuning for changing objectives, input data, and platforms. Furthermore, locating the proper set of tunable parameters and ignoring those that are irrelevant is a major issue. No one really has a systematic approach for that. Furthermore, auto-tuning in most cases relies on real program runs. Complementing auto-tuning with sophisticated performance models that would reduce the number of runs would be a major step forward. With models, one could handle larger search spaces and still find solutions that are close the optimum.

***Ubiquity*:** Prof. Fahringer, thank you very much for this interview.

About the Author

Walter Tichy (walter.tichy@kit.edu) is professor of software engineering at Karlsruhe Institute of Technology (formerly University of Karlsruhe) and a director of the Forschungszentrum Informatik, a technology transfer institute. He is both a Distinguished Scientist and a Fellow of the ACM, and an associate editor of ACM Ubiquity and IEEE Transactions on Software Engineering. He earned M.S. and Ph.D. degrees from Carnegie Mellon University. He received the Intel Award for the Advancement of Parallel Computing, the ACM Sigsoft Impact Paper Award, and the IEEE Most Influential Paper Award, among others.

DOI: 10.1145/2636340

