

# Continuous Auto-Tuning

Jeffrey K. Hollingsworth  
hollings@umd.edu  
*University of Maryland*

## Introduction

For exascale systems, continuous auto-tuning will be mandatory. In the past, it was possible to auto-tune (or perhaps even manually tune) a code for a specific machine. At exascale, it will be necessary for auto-tuning to be a continuous and ongoing activity throughout each execution of the application.

This requirement is driven by several factors. First, the sheer number of threads (on the order of one billion) will mean that dynamic load balancing will almost certainly be required. Second, the presence of energy limits (either in the form of total consumption or thermal throttling due to heat dissipation) will mean that the performance of cores will be dynamically varying. This will mean that the exact performance of the hardware will not be known until runtime. Finally, it is likely that hardware failures and faults will be the norm and not the exception. This will also contribute to dynamic changes in the hardware available to run applications.

The applications themselves will also have increasing needs for auto-tuning. For example, Adaptive Mesh Refinement (AMR) techniques naturally have opportunities to expose choices about meshing parameters and frequency of mesh updates.

Limited storage and data migration will likely drive applications to be “super coupled”. Super coupling will mean that not only will multiple physics/chemistry be coupled in a single execution, but previously separate phases of execution such as data staging, analytics, and visualization will be done concurrently with the application execution. These coupled systems will be far less isolated from other activity on the machine or even across the computer center than previous execution models. Thus auto-tuning will need to adapt to cross traffic from other applications too.

Taken together these trends imply that humans will no longer be able to tune such a system since the reaction time will likely be on the order of milli-seconds to seconds and not once per port to a new machine. Thus automation is our only hope.

For continuous auto-tuning to work, it must be a first class system service and have hooks in nearly all aspects of the software stack. Auto-tuning must be pervasive both because the knobs to tune performance are distributed throughout the stack (i.e. OS parameters, message passing configurations, math library choices, and application algorithmic choices) but also because the data required to make informed tuning decisions is also distributed (i.e., CPU hardware counters, network statistics, application data). In addition, it is critical that auto-tuning be coordinated among the components. Left alone, each component of the system could have its own independent auto-tuning system creating a cacophony of separate tuners each pushing and pulling performance in potentially opposing directions.

## Related and Prior Work

In recent years, there has been significant progress made in the area of auto-tuning. However, most of the work has focused on offline tuning using training runs. This approach is highly effective in automating the performance tuning for a new platform, but is not sufficient for the truly dynamic environment of exascale.

There are many research projects working on empirical optimization of linear algebra kernels and domain specific libraries. ATLAS [atlas] uses the technique to generate highly optimized BLAS routines. The OSKI (Optimized Sparse Kernel Interface) [oski] library provides automatically tuned computational kernels for sparse matrices. FFTW [fftw] combines the static models with empirical search to optimize FFTs. SPIRAL [spiral] generates empirically tuned Digital Signal Processing (DSP) libraries.

Several compiler frameworks that enable tunable code transformation have been proposed and developed in recent years. For example, WRaP-IT [wrapIt] is a polyhedral framework that provides a scripting interface to describe code transformations. However, adjusting parameters in this framework may require costly verification process and/or script change. Another approach is to use high-level user-friendly

annotations. Two noticeable examples are Orio [orio] and POET [poet]. Although they are not based on the polyhedral model, they provide sophisticated transformations in their frameworks and some are beyond the capabilities of the polyhedral model. In addition, the Chill framework [chill] supports flexible recipes that have been combined with the Active Harmony tuning framework to support end-to-end auto-tuning [HarmonyChill].

Our recent work with Active Harmony has demonstrated that online auto-tuning (even involving runtime code generation) is not only possible, but can improve application performance in a single execution of the program. This ability to overcome the overhead of online auto-tuning and improve a program in a single execution is a critical demonstration that the idea of continuous auto-tuning is possible and practical [harmonyOnline].

## Discussion

*Challenges Addressed:* Successful research into continuous auto-tuning will help meet several identified challenges of exascale computation. First, and foremost it provides a way to get applications to perform well on exascale system and allow the science for which these system are designed to be accomplished. In particular, it will help to mitigate the programmer impact of power constraints and reliability issues by letting programmers express options and constraints to the runtime system which will adapt the code to the changing conditions.

*Maturity:* Auto-tuning for libraries, loops, and specific parts of runtime libraries has already proven successful as described in the related work section. Likewise, initial results have shown the potential for sophisticated online adaptation, but substantial research is required to scale this up.

*Uniqueness:* The scale and truly dynamic nature of exascale means full-scale continuous auto-tuning will not come to fruition in other research anytime soon. Auto-tuning work will continue to be done for smaller scale systems. However, many problems unique to exascale must be addressed if this technology will be ready in time. In particular, fundamental work in high dimensional optimization algorithms are required; coordination of distributed auto-tuning will be also required given the large number of nodes and cores in an exascale system. There is also substantial work to ensure that various components of the exascale stack expose options for auto-tuners.

*Novelty:* No one is doing auto-tuning at the scale proposed here. Likewise, the efficient continuous reaction to an evolving computational environment is unique.

*Applicability:* If this research direction is successful, it will likely have application beyond exascale computation. First, it will likely become widely used at smaller scale HPC. In addition, the general approach to optimization is likely to be useful to the commercial sector in the large-scale datacenter environment.

*Effort:* To successfully conduct this research, a core team of about a half dozen researchers is required for 3-4 years. This team could be either at a single location or draw on expertise of several sites. A critical part of this effort is to include a model of augmented co-funding of compiler, runtime library, and numerical libraries teams with auto-tuners to enable all components of the software stack to take advantage of the framework. The second effort is critical to provide a full-scale exascale software stack that is auto-tune ready.

## Conclusion

A substantial research effort in continuous auto-tuning for exascale is required. This effort will transform the programming and execution of applications. While moving to a world of continuous auto-tuning represents a major departure from the current model of tune once - run many times, the needs of exascale demand such a new paradigm. While there are many technical hurdles to overcome to achieve this vision, recent successes with off-line auto-tuning and early experiments with online auto-tuning help to mitigate the risks of this new approach.

## References

- [atlas] R. C. Whaley and J. J. Dongarra. “Automatically tuned linear algebra software.” In Proceedings of the 1998 ACM/IEEE conference on Supercomputing, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.
- [chill] C. Chen, J. Chame, M. Hall, J. Shin, and G. Rudy. “Loop Transformation Recipes for Code Generation and Auto-Tuning.” In 22nd International Workshop on Languages and Compilers for Parallel Computing, 2009.
- [fftw] M. Frigo and S. Johnson. “The Design and Implementation of FFTW3.” Proceedings of the IEEE, 93(2):216–231, 2005.
- [harmonyChill ] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. “A Scalable Auto-Tuning Framework for Compiler Optimization.” In 23rd IEEE International Parallel & Distributed Processing Symposium, Rome, Italy, May 2009.
- [harmonyOnline] A. T., J. K. Hollingsworth, “Online Adaptive Code Generation and Tuning” IPDPS, May 2011.
- [orio] A. Hartono, B. Norris, and P. Sadayappan. “Annotation-based empirical performance tuning using Orio.” In Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, pages 1 – 11, May 2009.
- [oski] R. Vuduc, J. W. Demmel, and K. A. Yelick. “Oski: A library of automatically tuned sparse matrix kernels.” Journal of Physics: Conference Series, 16:521–530, June 2005.
- [poet] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. “POET: Parameterized Optimizations for Empirical Tuning. Parallel and Distributed Processing Symposium”, IPDPS 2007. IEEE International, pages 1–8, March 2007.
- [spiral] J. Xiong, J. Johnson, R. Johnson, and D. Padua. “SPL: a language and compiler for DSP algorithms.” In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, PLDI '01, pages 298–308, New York, NY, USA, 2001. ACM.
- [wrapIt] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parelo, M. Sigler, and O. Temam. “Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies.” Int. Journal Parallel Programming, 34:261–317, June 2006.