Auto-tuning Full Applications: A Case Study

ANANTA TIWARI*, JEFFREY K. HOLLINGSWORTH
Department of Computer Science, University of Maryland,
College Park, Maryland 20742, USA

CHUN CHEN, MARY HALL
School of Computing, University of Utah,
Salt Lake City, Utah 84112, USA

CHUNHUA LIAO, DANIEL J. QUINLAN
Center for Applied Scientific Computing,
Lawrence Livermore National Laboratory, Livermore, California 94550, USA

JACQUELINE CHAME
Information Sciences Institute, University of Southern California,
Marina del Ray, California 90292, USA

ABSTRACT

In this paper, we take a concrete step towards materializing our long-term goal of providing a fully automatic end-to-end tuning infrastructure for arbitrary program components and full applications. We describe a general-purpose *offline* auto-tuning framework and apply it to an application benchmark, SMG2000, a semi-coarsening multigrid on structured grids. We show that the proposed system first extracts computationally-intensive loop nests into separate executable functions, a code transformation called outlining. The outlined loop nests are then tuned by the framework and subsequently integrated back into the application. Each loop nest is optimized through a series of composable code transformations, with the transformations parameterized by unbound optimization parameters that are bound during the tuning process. The values for these parameters are selected using a search-based auto-tuner, which performs a parallel heuristic search for the best-performing optimized variants of the outlined loop nests. We show that our system pinpoints a code variant that performs 2.37 times faster than the original loop nest. When the full application is run using the code variant found by the system, the application's performance improves by 27%.

Keywords: Offline Auto-tuning, ROSE, CHiLL, Active Harmony, PERI

*A. V. Willaims Building(4140), Department of Computer Science, University of Maryland, College Park MD 20742, USA

1. Introduction

Currently, the burden of tuning codes largely falls on the shoulders of application programmers. Programmers spend countless hours modifying their codes to exploit performance enhancing architectural features. These features vary between different platforms. As a result, codes tuned for one platform often face performance problems when ported to another platform. Therefore, this process of tuning has to be mostly repeated while moving from one platform to the other.

Auto-tuning software (auto-tuners) help programmers automate this painful and error-prone process of tuning and porting application codes. Domain-specific auto-tuners such as ATLAS [31] for dense linear algebra, OSKI [30] for sparse linear algebra, FFTW [10] and SPIRAL [32] for signal processing, etc. have been very successful in producing highly-optimized architecture-specific codes. This success has sparked a general interest in extending the search-based empirical auto-tuning methodology to arbitrary program components and whole programs. Shifting the focus from empirically tuning a few kernels to tuning whole programs will certainly help avoid the enormous productivity costs associated with tuning and retargeting applications to the next generation exascale systems. However, the shift also comes with its own set of challenges. In this paper, we discuss these challenges and describe a general-purpose offline auto-tuning framework for whole programs. The proposed framework integrates performance analysis tools, compiler frameworks and auto-tuners that are under development within the PERI [1,3] community[a]. These components make significant contributions towards materializing PERI's longer term auto-tuning vision of developing and deploying a fully automated (or as fully automated as practical) tool-chain that can provide an end-to-end tuning for full programs.

We should note that domain-specific specialized libraries, in most cases, are better suited to handle domain-specific computations. Our goal is to provide a general-purpose compiler based framework, which can generate and evaluate different optimizations that can be applied on arbitrary application codes. In the absence of a general-purpose framework, manual exploration of possible optimizations can be prohibitively time consuming and painful for a programmer.

The remainder of this paper is organized as follows: section 2 describes the challenges that auto-tuners face in designing a whole program tuning infrastructure. This section also describes the three main components (ROSE outliner, CHiLL and Active Harmony) of the auto-tuning system presented in this paper. In section 3, we give an overview of the tuning workflow in our framework. Section 4 describes the subject application benchmark. Empirical evaluation of our system using the benchmark is presented in section 5. We discuss related work in section 6. Finally, section 7 provides concluding remarks and future implications of this work.

[a]PERI (Performance Engineering Research Institute) is a DOE SciDAC institute developing tools and techniques to help application teams effectively use leadership class computing systems.

2. Full Application Tuning — Challenges & Enabling Components

In this section, we review some key issues that we faced while designing our system.

(i) Compute intensive loop nests in full applications are often wedged in the middle of large monolithic code sections. Code outlining tools are needed to extract these loop nests to separate standalone functions. These outlined codes can be more easily managed, analyzed and transformed by loop-transformation tools.

(ii) The number of code variants for a complete application can be enormous. Strategies to judiciously select what transformation techniques to apply to different sections of the application code are needed to keep the tuning time at manageable levels. Our compiler experts work with the application developers to make these decisions. Furthermore, compiler-based auto-tuning requires a code-transformation framework that is able to generate different variants of the source code rapidly during the search by adjusting code transformation parameters. It also demands that the compiler have a clean interface to a separate parameter search engine.

(iii) As the number of tuning parameters increases, the search space becomes high dimensional and exponential in size. Search algorithms that can cope with exponential spaces and deliver results within a few search iterations are needed.

In the next three sections, we describe PERI-components that help us address these issues.

2.1. *Code-outlining via ROSE*

We use the ROSE outliner [17] to extract computational hotspots (kernels) from large scale applications into separate and manageable functions. This process helps reduce the challenging whole program tuning problem into a set of manageable kernel tuning tasks. The outliner is built using the ROSE [23] source-to-source compiler. Compared to other outlining tools [14, 15, 34], the ROSE outliner is designed to handle multiple input languages such as C, C++ , Fortran and OpenMP. Moreover, a set of program analyses, including side-effect and liveness analyses, are leveraged to reduce the performance impact caused by the kernel extraction.

ROSE is an open source compiler infrastructure aimed to allow programmers without expertise in compiler internals to build customized source-to-source program transformation and analysis tools for large-scale applications. With the help from the Edison Design Group (EDG) C++ front-end [9] and the Open Fortran Parser (OFP) [24], ROSE presents a common object-oriented, open source intermediate representation (IR) for C/C++ and Fortran applications. The ROSE IR includes an abstract syntax tree (AST), symbol tables, a control flow graph, etc. A wide range of programming interface functions are developed to support AST query, traversal, construction, consistency checking, file I/O, visualization, and symbol table lookup. As a result, both generic and custom program analysis and transformation can be easily built on top of the ROSE IR. Representative program translations

developed with ROSE are partial redundancy elimination, constant folding, inlining, loop transformations, and automatic parallelization [18]. Finally, a vendor compiler is optionally called to continue the compilation of the generated (transformed) source code, generating a final executable.

The ROSE outliner was initially started as an internal component for the OpenMP implementation [16] of ROSE to generate functions from OpenMP parallel regions. It has since evolved to a standalone kernel extraction tool to support auto-tuning. The outliner uses a preprocessing phase to support outlining code portions with complex control flow due to **return**, **goto** and **break** statements. A control parameter is generated to pass in (for multiple entry points) and out (for multiple exit points) the jump targets of the outlined function. In addition, the ROSE outliner supports passing parameters one by one or wrapping all parameters into a single data structure parameter. This is necessary to work with some APIs (such as PThreads) which allow a function type with only one parameter.

Moreover, we use a novel method, referred to as variable cloning, to reduce the number of pointer dereferences during outlining. This feature was specifically developed for supporting whole program auto-tuning. For a written C/C++ variable that is classically passed as a pointer type and accessed via pointer-dereferencing, we check the actual use of such a variable within the code portion to be outlined. If it is not used by its address, a temporary clone variable (using TYPE clone;) can be introduced to substitute its uses within the outlined function. For the C language, using a variable by address occurs when the address operator is used with the variable (e.g. &X). In C++, associating a variable with a reference type (TYPE & Y = X; or using the variable as a function argument of a reference type) introduces a use by address. The value of a clone variable has to be initialized properly (using clone $= *$ parameter;) before the clone participates in computation, if the original variable is live-in. After the computation, the original variable must be set to the clone's final value (using $*$parameter = clone;), if the original variable is live-out. Therefore, the kernels extracted by the ROSE outliner usually has much less pointer usage and can be more friendly to compiler analyses and facilitate further optimizations (e.g., loop translations). More details on the ROSE outliner can be found in a previously published paper [17].

2.2. *Code Generation via CHiLL*

Once outlined, we generate optimized variants of the code using CHiLL, a polyhedral loop transformation and code generation framework [6, 7, 25, 29]. CHiLL's polyhedral framework mathematically represents loop iteration spaces and array access expressions, facilitating robust composition of iteration space transformations and code generation. It is therefore well-suited for an auto-tuning framework in which optimization parameters will be varied to adjust performance. An external script interface, called a transformation recipe, describes the composable transformation sequence to be applied to the loop nest computation. CHiLL's high-level

transformation recipes enable compiler algorithms or application programmers to use a common interface to describe parameterized code transformations to be applied to a computation. For each code transformation, the parameters control the semantics of the code transformation so that different parameter values lead to different generated code, freeing application developers from managing the significant complexity of writing this code manually.

The transformation recipe also provides an interface to the external auto-tuning search engine to instantiate the parameter values to find the best-performing implementation. An example recipe is shown in Table 1 in the next section. Besides making it easy to interface with the code-generation utility, these code transformation recipes offer an additional advantage. *Unlike traditional compiler optimizations which must be coded into the compiler, these recipes can be evolved and reused over time.* A recipe library, created by compiler experts and developers based on their experience working with real codes, can then be consulted by auto-tuners to tune arbitrary loop nests.

To illustrate the usability of CHiLL, to optimize SMG2000 (in Table 1) we use three common loop transformations: permutation, tiling and unrolling. Permutation reorders the loop nest so that the data in the computation is being accessed in an order that more closely matches the organization of the data in memory, and facilitates subsequent optimizations. Tiling partitions the loop nest's iteration spaces into small blocks and then iterates through those blocks in sequence. It maintains a small data footprint for the sub-loop nests for cache optimization or to partition the computation across parallel threads. In CHiLL, the only required parameters to the tile transformation are the sub-loop nest of interest, its tile size, and the position of the tile controlling loop created by the transformation. The parameters are the same even for an imperfect loop nest, or loop nest with complex loop bounds. Even though the generated code can be quite complex, the tiling algorithm implemented in CHiLL handles those differences seamlessly. Further, unrolling in CHiLL implements unroll-and-jam, which explicitly duplicates even an outer loop in a nest and fuses together the copies of the inner loop bodies that are created by the transformation. The result of unroll-and-jam is a long sequence of straightline code in the innermost loop body. Through simplifying control flow, exposing independent computations to the scheduler, and reused data to the register allocator, unroll-and-jam improves instruction-level parallelism and register utilization. The only required parameters to unrolling are the sub-loop nest of interest and the unroll factor for the specified loop. A special feature of CHiLL's unroll-and-jam transformation is that it manages the complexity of unrolling triangular loops, where the number of iterations of the inner loop varies across iterations of the outer loop.

2.3. *Search-based Auto-tuning via Active Harmony*

We use the Active Harmony system [28, 29], which is a search-based auto-tuning framework, to drive the overall tuning workflow. Active Harmony allows applica-

tion programmers, library writers, and compilers to describe and export a set of performance related tunable parameters. These parameters define a tuning search-space. More often than not, this search-space is high-dimensional and exponential in size and thus, cannot be explored manually. Our system monitors the program performance and makes adaptation decisions. The decisions are made by a central controller (henceforth referenced as the Active Harmony server) using a parallel search algorithm — Parallel Rank Ordering (PRO) algorithm [27]. PRO leverages parallel architectures to simultaneously search across a set of optimization parameter values. Different nodes of a parallel system evaluate different configurations at each timestep.

The success of Active Harmony is largely driven by how well the *search* algorithm navigates the parameter space. The advantages of the parallel search algorithm that the auto-tuner uses has been demonstrated in earlier results [29]. Those results showed the benefits of using PRO to navigate compiler-generated search spaces. For well-defined benchmark kernels such as matrix multiplication, the search algorithm was able to find code variants that delivered significant improvements over the optimizations offered by native compilers.

We combine Active Harmony's parallel search backend with ROSE outliner and CHiLL compiler transformation framework to do empirical optimization — a systematic *search* over a collection of automatically generated code variants. As we discussed in section 2.2, each code transformation exposes its own set of parameters (for example, loop-unrolling exposes loop-unroll factors). Parameter configurations for loop-transformations serve as points in the search space and the objective function values[b] associated with the points are gathered by running the code variants generated by CHiLL on the target architecture.

3. Overall Workflow

Figure 1 shows the overall workflow of our system. The tuning process starts by first using application profiling tools (such as HPCToolkit [2], TAU [26], etc.) to identify computationally intensive loop nests (not shown in the figure). The ROSE outliner extracts the kernels to separate and independently compilable C source files. Code-outlining is a one-time process — outlined kernels can be reused in subsequent auto-tuning runs.

Application developers make simple modifications to the driver code that we provide as part of the Active Harmony software release package. These changes are made to export application-specific tuning options to the Active Harmony server. The driver, which can be run on the login nodes of a parallel machine, connects to a given Active Harmony server and requests candidate parameter configurations. The driver then invokes CHiLL to generate variants of the outlined kernel based on the

[b]The objective function values associated with points in the search space can be any desired metric of performance (for example - time per timestep, MFLOPS, cache utilization etc.). In this paper, the metrics is always time per representative execution (i.e. time of a few typical timesteps).

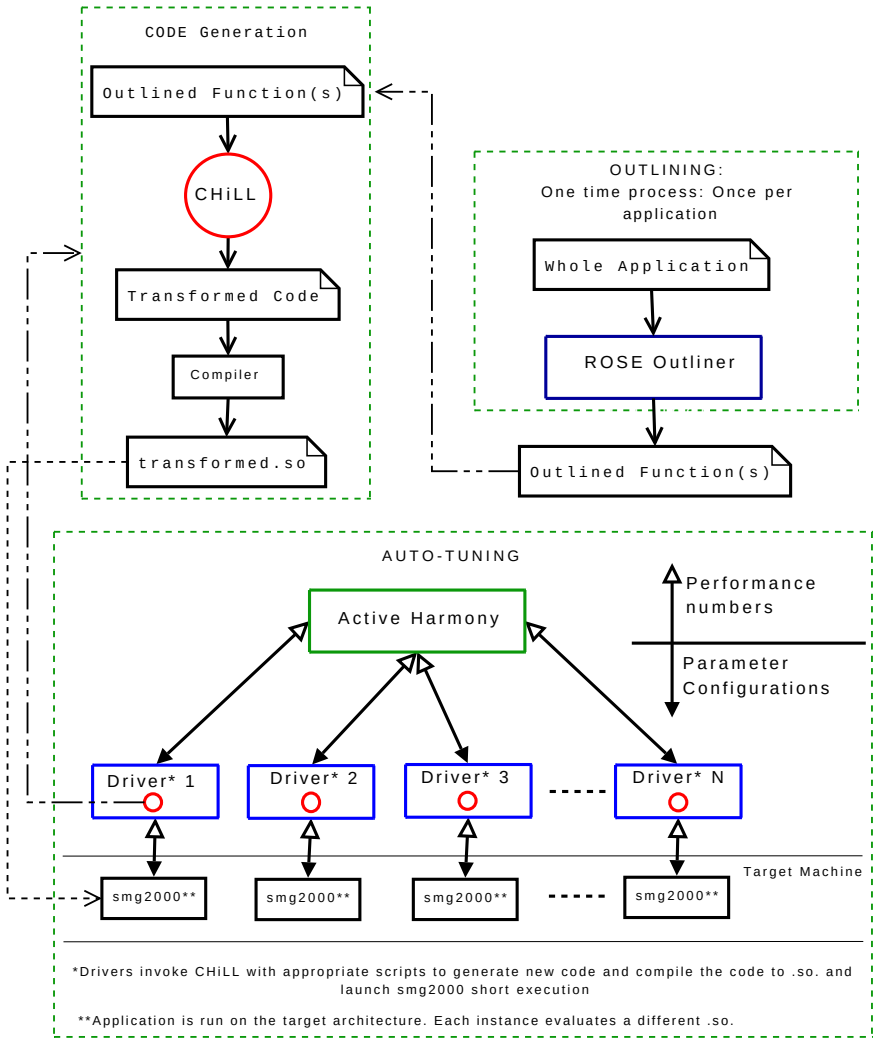

Fig. 1. Overall workflow: SMG2000 Tuning

code transformation parameters supplied by the Active Harmony server. The code generated on-demand is compiled into a shared library. Once the new code is ready, the application is run on the target machine. The application dynamically loads the transformed kernel by using the dlopen/dlsym mechanism. Once the execution is complete, the driver collects performance measurement and sends them to the Active Harmony server. The process continues for a specified number of iterations or until the search algorithm converges to a point in the search space. For parallel search algorithm, we run multiple copies of the driver. The number of copies is determined by the number of tunable parameters and the simplex size (which is, in turn, determined by the available resources). The use of the shared library mecha-

nism helps to keep the tuning time short because only the outlined and transformed code has to be recompiled between successive search steps.

4. Subject Application: SMG2000

We consider the SMG2000 [4] benchmark as a subject application. SMG2000 is a parallel semi-coarsening multigrid solver for the linear systems arising from finite difference, finite volume, or finite element discretizations of the diffusion equation on logically rectangular grids (equation 1).

$$\bigtriangledown \cdot (D \bigtriangledown u) + \sigma u = f \tag{1}$$

The code solves both 2D and 3D problems with discretization stencils of up to nine points in 2D and up to twenty seven points in 3D. SMG2000 was developed at Lawrence Livermore National Labs (LLNL) of Department of Energy (DOE). SMG2000 was picked because it is a representative numerical computation application used at LLNL. This shows that the work presented in this paper is relevant to DOE and can actually make impact on real lab applications.

The most time-consuming kernel (approximately 55% of the execution time on the target system used in this paper) in the SMG2000 benchmark is shown in Table 1. The kernel consists of sparse matrix vector multiplication expressed in four-deep loop nest[c]. The kernel performs a stencil computation by sweeping the same array data (accessed using the inner `i`, `j`, and `k` indices) multiple times for each stencil element (the outermost `s` index). Thus, the kernel lacks data reuse and causes excessive cache misses [17].

To minimize the time required for tuning, offline auto-tuners often use "representative short application executions". We use this methodology in the work presented here. In this technique, the application being tuned is run with a meaningful input data for a short period of time and tuning modifications are made between successive short executions [8]. Recall that the objective function values associated with different parameter configurations are derived by running the application on the target machine. Therefore, representative short runs help reduce the overall time required for offline auto-tuning. SMG2000 execution is divided into three distinct phases — `initialization`, `setup` and `solve`. All three phases make several calls to the outlined function — the function being tuned. We disable the third phase and record the total time spent in just the outlined kernel in the first two phases. This timing measurement is used by Active Harmony to drive the search process. Since the first two phases make significant number of calls to the outlined function, the measured timings are still representative of the overall time spent in the outlined kernel in full application execution.

[c]The outlined kernel shown is a simplified version. Actual code is less clean.

5. Empirical Results

The auto-tuning experiments were performed on a 64-node Linux cluster. Each node is equipped with dual-core Intel Xeon 2.66 GHz (SSE2) processor. L1- and L2-cache sizes are 128 KB and 4096 KB respectively. Active Harmony uses the Parallel Rank Ordering (PRO) algorithm to navigate the search space. Short executions of SMG2000 are done in parallel on the target machine, with each execution instance using a different code variant. Transformation parameters are adjusted and corresponding new code variants are generated between successive runs of SMG2000. The search uses a 24-point simplex, which means up to 23 new code variants are evaluated in parallel at each search-step.

Table 1. SMG2000 Optimization: Original Code, CHiLL Recipe, Search Space Constraints

Original	*Recipe*	*Constraints*

`for (s = 0; s < stencil_size; s++)`
`for (k = 0; k < hypre_mz; k++)`
`for (j = 0; j < hypre_my; j++)`
`for (i = 0; i < hypre_mx; i++)`
`rp[((ri+i)+(j*hypre_sy3))+(k*hypre_sz3)]-=`
`((Ap[((i+(j*hypre_sy1))+ (k*hypre_sz1))+`
`(((A->data_indices)[i])[s])])*`
`(xp[((i+(j*hypre_sy2))+(k*hypre_sz2))+`
`((*dxp_s)[s])]));`
	`permute([2,3,1,4])`
	`tile(0,4,TI)`
	`tile(0,3,TJ)`
	`tile(0,3,TK)`
	`unroll(0,6,US)`
	`unroll(0,7,UI)`
		$0 \leq TI \leq 122$
		$0 \leq TJ \leq 122$
		$0 \leq TK \leq 122$
		$0 \leq UI \leq 16$
		$0 \leq US \leq 10$
		$comp \in \{gcc, icc\}$

The optimization strategy (expressed in terms of a CHiLL-recipe) and constraints on transformation parameters are provided in Table 1. The recipe tiles the `i`, `j` and `k` loops (with `TI`, `TJ` and `TK` tiling factors) to improve data reuse in caches. The stencil loop and the innermost loop are unrolled (with `US` and `UI` unrolling factors) to improve reuse in registers. The search-space is six-dimensional and includes a parameter that chooses between two compilers to compile the transformed kernel — `gcc` and `icc`.

The search converges in 20 steps. The search-evolution (performance of the best-point at each search-step) is shown in Figure 2. The y-axis shows the total time spent in the outlined kernel(in seconds) per short representative SMG2000 execution. The x-axis shows the PRO search steps. The configuration that PRO converges to is: `TI`=122, `TJ`=106, `TK`=56, `UI`=8, `US`=3, `comp`=`gcc` [d]. The performance improvement is $2.37X$ of the time for the outlined kernel. We then use the code variant associated with this parameter configuration to do a full run of SMG2000 (with input parameters `-n 120 120 120 -d 3`). The results from full SMG2000 run are summarized in Table 2. Full application execution improves by 27.2% (average of five full application executions).

[d]This was gcc version 4.1.2 and icc version 10.0.026, where icc has been known to have poor performance.

Most scientific applications exhibit similar characteristics to SMG2000 in that the majority of the computational time is spent in a few core loops. In this paper, we showed that our auto-tuning system can extract such computationally intensive loops into separate functions. The outlined loop nests are then tuned by our system and subsequently integrated back into the application. Thus, our system can be easily applied to other scientific applications.

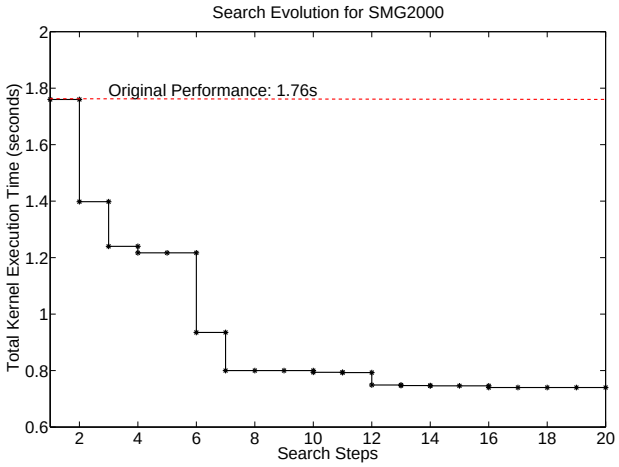

Fig. 2. Search Evolution for Offline SMG2000 Tuning

Table 2. SMG2000 Full Run Performance

Auto-tuned	Original	Improvement
$49.86s$	$68.52s$	27.2%

6. Related Work

There are many research projects working on empirical optimization of linear algebra kernels and domain specific libraries. ATLAS [31] uses the technique to generate highly optimized BLAS routines. The OSKI (Optimized Sparse Kernel Interface) [30] library provides automatically tuned computational kernels for sparse matrices. FFTW [10] combines the static models with empirical search to optimize FFTs. SPIRAL [32] generates empirically tuned Digital Signal Processing (DSP) libraries. Chandramowlishwaran et al [5] look into single-node performance optimization, tuning and analysis of the fast multipole method (FMM) on a diverse set of multi-core systems. They consider numerous performance enhancing strategies for FMM — SIMD vectorization and scheduling, numerical approximation, data structure transformations, OpenMP-based parallelization, etc. Rather than focusing on one particular domain, our framework aims at providing a general-purpose

compiler based approach for tuning arbitrary application codes.

Several compiler frameworks that support flexible and tunable code transformation have been put forth in the literature. WRaP-IT [11] 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. The work on iterative compilation [20, 21] searches one- or multi-dimensional scheduling represented in the polyhedral model using heuristics or genetic algorithms. This approach is limited in that scheduling can not express tiling or unroll-and-jam directly. Another approach is to use high-level user-friendly annotations. Two noticeable examples are Orio [12] and POET [33]. 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.

Different projects have considered search techniques to explore compiler generated parameter spaces. Kisuki et al [13] address the problem of selecting tile sizes and unroll factors simultaneously. Different search algorithms are used to search the parameter space - Genetic algorithms, Simulated Annealing, Pyramid search, Window search and Random search. Qasem et al [22] use a modified version of pattern-based direct search algorithm to explore the same search space. Kisuki et al report converging to a solution in hundreds of iterations. By effectively utilizing the parallel infrastructure, we converge to solutions in a few tens of iterations.

7. Conclusion

In this paper, we combined three tools from the PERI research project – the ROSE outliner, the CHiLL transformation and code generation framework, and Active Harmony – to create a general-purpose offline auto-tuner that can handle arbitrary program components and full applications. We demonstrated the benefits of the system on a real scientific application benchmark — SMG2000. We showed how these three components complement each other and work together to create an integrated framework that supports automatic compilation and parallel search and finds code variants that perform 2.37 times faster than the original code.

In the future, we plan to extend this work to production-sized applications. Currently, we are working with the PFLOTRAN [19] developers to apply our system to tune their code. We also plan to extend this work to provide *online* tuning for large-scale scientific applications.

8. Acknowledgment

This work was sponsored in part by the Performance Engineering Research Institute, which is supported by the SciDAC Program of the U.S. Department of Energy.

References

[1] *SciDAC Performance Engineering Research Institute*, 2010. `http://www.peri-scidac.org/perci/`.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: tools for performance analysis of optimized parallel programs. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, 2010.

[3] D. H. Bailey, R. Lucas, P. Hovland, B. Norris, K. Yelick, D. Gunter, B. de Supinski, D. Quinlan, P. Worley, J. Vetter, P. Roth, J. Mellor-Crummey, A. Snavely, J. Hollingsworth, D. Reed, R. Fowler, Y. Zhang, M. Hall, J. Chame, J. Dongarra, and S. Moore. Performance engineering: Understanding and improving the performance of large-scale codes. *CTWatch Quarterly*, 3(4), 2007.

[4] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening Multigrid on Distributed Memory Machines. *SIAM J. Sci. Comput.*, 21(5):1823–1834, 2000.

[5] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and Tuning the Fast Multipole Method for State-of-the-art Multicore Architectures. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, 2010.

[6] C. Chen. *Model-Guided Empirical Optimization for Memory Hierarchy*. PhD thesis, University of Southern California, 2007.

[7] 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.

[8] I. Chung and J. K. Hollingsworth. A case study using automatic performance tuning for large-scale scientific programs. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on High Performance Distributed Computing*, pages 45–56, 2006.

[9] Edison Design Group. C++ Front End. `http://www.edg.com`.

[10] M. Frigo and S. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216 –231, 2005.

[11] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34:261–317, June 2006.

[12] 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.

[13] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[14] A. Lakhotia and J. Deprez. Restructuring programs by tucking statements into functions. *Special Issue on Program Slicing*, 40:677–689, 1998.

[15] Y. Lee and M. W. Hall. A Code Isolator: Isolating Code Fragments from Large Programs. In R. Eigenmann, Z. Li, and S. P. Midkiff, editors, *LCPC*, volume 3602 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2004.

[16] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, editors, *IWOMP*, volume 6132 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 2010.

[17] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas. Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization. In *LCPC '09: International Workshop on Languages and Compilers for Parallel Computing*, Newark, Delaware, 2009.

[18] C. Liao, D. J. Quinlan, J. Willcock, and T. Panas. Semantic-aware Automatic Paral-

lelization of Modern Applications Using High-Level Abstractions. *International Journal of Parallel Programming*, 38(5-6):361–378, 2010.

[19] R. T. Mills, C. Lu, P. C. Lichtner, and G. E. Hammond. Simulating subsurface flow and transport on ultrascale computers using PFLOTRAN. *Journal of Physics: Conference Series*, 78(1):012051, 2007.

[20] L. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 90–100, New York, NY, USA, 2008. ACM.

[21] L. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 144–156, Washington, DC, USA, 2007. IEEE Computer Society.

[22] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *J. Supercomput.*, 36(2):183–196, 2006.

[23] D. J. Quinlan et al. ROSE compiler project. `http://www.rosecompiler.org/`.

[24] C. Rasmussen et al. Open Fortran Parser. `http://fortran-parser.sourceforge.net/`.

[25] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame. A Programming Language Interface to Describe Transformations and Code Generation. In *23rd International Workshop on Languages and Compilers for Parallel Computing*, 2010.

[26] S. S. Shende and A. D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[27] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel Parameter Tuning for Applications with Performance Variability. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 57, Washington, DC, USA, 2005. IEEE Computer Society.

[28] C. Tapus, I. Chung, and J. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. In *Supercomputing, ACM/IEEE 2002 Conference*, page 44, 2002.

[29] 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.

[30] 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.

[31] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[32] 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.

[33] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized Optimizations for Empirical Tuning. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[34] P. Zhao and J. N. Amaral. Ablego: a function outlining and partial inlining framework: Research articles. *Softw. Pract. Exper.*, 37(5):465–491, 2007.