Progress Report for X-TUNE
Autotuning for Exascale: Self-Tuning Software to Manage Heterogeneity

Lead Institution: University of Utah
Principal Investigator: Mary Hall
Postal Address: 50 S. Central Campus Drive, Salt Lake City, UT 84112
Telephone Number: 801-585-1039
Email: mhall@cs.utah.edu
Collaborating Institutions:
Paul Hovland, Jeff Hammond, Sri Hari Krishna Narayanan, and Stefan Wild
Argonne National Laboratory
Samuel Williams, Leonid Oliker, and Brian Van Straalen
Lawrence Berkeley National Laboratory
Jacqueline Chame, *USC/ISI*

Objectives We will develop a unified autotuning framework that seamlessly integrates programmer-directed and compiler-directed autotuning, so that a programmer and the compiler system can work collaboratively to tune a code, unlike previous systems that place the entire tuning burden on either programmer or compiler. The proposed system will dramatically improve generality and usability of autotuning technology through an integrated, composable collection of tools, including an autotuning compiler framework, a library API, a code transformation framework, compiler decision algorithms and performance models. To maximize productivity impact of autotuning and make it approachable by many users, it should be encapsulated in domain-specific tools developed by expert users and made available to others. To this end, we will demonstrate autotuning on computations from AMR MG, Combustion Co-Design Center, TCE and Nek5000, and will work with DOE to define a small number of other mini-app demonstrations. We will identify opportunities for integration, software reuse and demonstrations with other X-stack projects.

Summary of Activities

The X-TUNE team has made significant progress in its first six months. We conducted research in the following areas:

- Models and Compiler Decision Algorithm
- Autotuning AMR Multigrid
- Optimizing Nekbone
- Optimizing NWChem

In September 2012, PI Hall participated in the X-Stack kickoff PI meeting. In October 2012, Williams presented X-TUNE in a poster session at the Exascale PI meeting. During February and March, 2013, PI Hall participated in videoconferences to design the SET guidelines. In March 2013, Hall, Oliker, Williams and van Straalen all participated in the X-Stack PI meeting. Hall gave a presentation, participated in a panel and provided a demonstration.

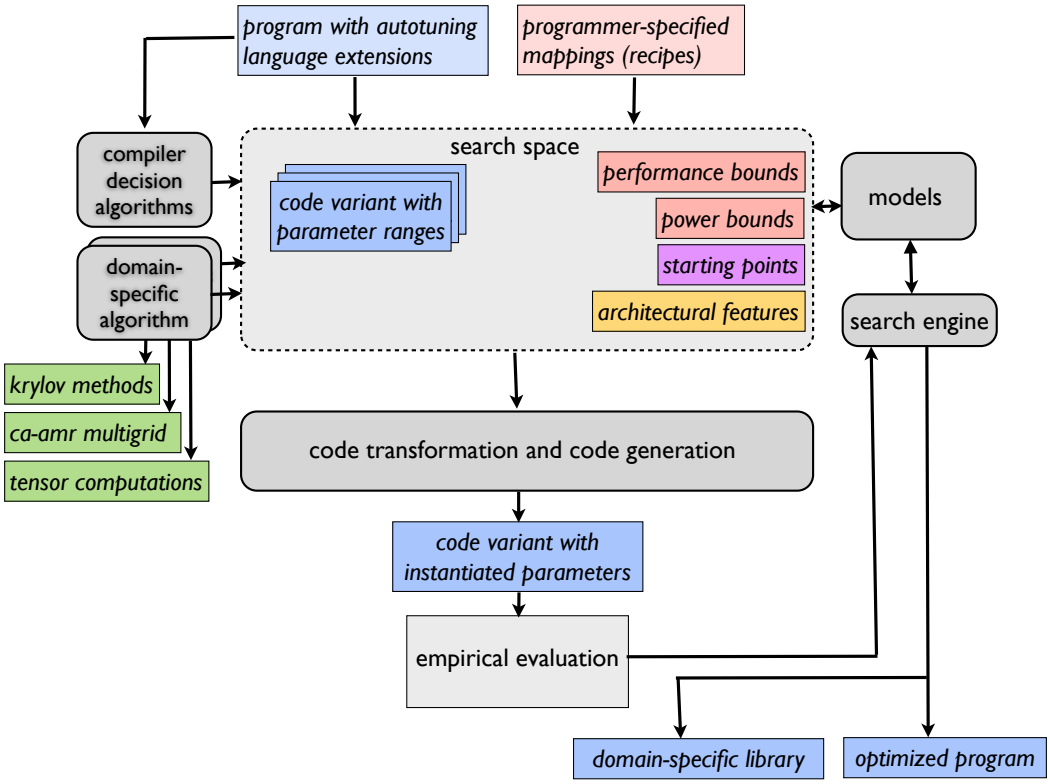


Figure 1.1: Proposed X-TUNE end-to-end autotuning system.

1 X-TUNE Overview

Figure 1.1 depicts the proposed X-TUNE system. Expert developers employ autotuning under their control either by expressing code variants and optimization parameters directly in their application code using a library API or by directing the compiler's optimization and translation process by providing transformation recipes. The code variants may include different algorithms for implementing a computation, or implementations partially customized to different target processors (e.g., in today's systems, this might be OpenMP code for a conventional multicore and CUDA code for a GPU). Using compiler decision algorithms, the compiler may additionally automatically derive a set of variants that reflect different code generation strategies, different data placement in the memory hierarchy, and code customized to different target processors. The compiler decision algorithms are written in a scripting language and designed so that, in addition

to the standard decision algorithms, custom decision algorithms can be implemented by developers of domain-specific tools.

The variants and optimization parameters provided by the programmer and compiler form a combined search space that describes the myriad possible mappings of the software to a variety of hardware platforms, including multiple types of processors in a heterogeneous system. An existing search algorithm navigates this search space using a variety of pruning strategies to efficiently discover the best implementation for a particular anticipated collection of execution contexts. An important aspect of the search process is the development and use of models to prune away uninteresting portions of the search space and provide bounds on the expected behavior of optimization criteria using knowledge of the architecture and analysis of the application software.

Code generation is applied to each point in the search space to be evaluated empirically. The result of this tuning process is either a domain-specific library, which abstracts away how autotuning was used in the application deployment, or an optimized program. In either case, some code selection decisions are deferred until run time when the entire execution context is known.

The proposed system is being constructed from significant extensions to existing software that includes the ROSE compiler infrastructure, CHiLL and CUDA-CHiLL autotuning compiler technology and polyhedral transformation, code generation support integrated into ROSE, PBound modeling software also integrated with ROSE, and ChomboFortran and Tensor Contraction Engine domain-specific tools, as well as application and computation kernels to which autotuning has been applied in a manual way.

2 Activities

2.1 Models and Compiler Decision Algorithms

One of X-TUNE's goals is to automate the generation of optimization strategies, for classes of applications for which compiler models and analysis can derive enough information to assist the automation. As a first step towards this goal we have implemented a compiler decision algorithm that uses models in PBound to identify data reuse and guide the generation of optimization strategies for improving locality in dense matrix computations. In this section we describe the integration of PBound, a compiler decision algorithm and CHiLL.

2.1.1 PBound

PBound models the performance bounds of an application. Given the application's source code and an architectural description, PBound generates parameterized closed-form expressions for the application's computational and data requirements. These are then combined to generate performance bounds. PBound uses data reuse analysis to model the application's cache behavior. Under X-TUNE, we have refined PBound's data reuse model to provide results at a finer granularity and have identified further refinements to be made in the future. We have begun to examine how PBound can be used to guide the empirical search algorithm through parameter selection.

2.1.2 CHiLL

CHiLL is a polyhedral transformation and code generation system that has been designed to support autotuning and programmer-directed compiler transformation. Underlying CHiLL is the use of the ROSE abstract syntax tree, which allows the system to exploit ROSE analyses, frontend capabilities and additional code optimizations. (A thin layer called CUDA-CHiLL takes as input higher-level GPU transformation recipes and generates the corresponding CHiLL commands.) Each compact CHiLL *transformation recipe* represents a parameterized description of a search space of possible implementations that can be automatically generated (that is, a sequence of code transformations and their associated optimization parameters). From the input code and a transformation recipe with instantiated optimization parameters, CHiLL automatically generates a code variant. Therefore, the integrated compiler decision algorithm uses PBound models to guide the automatic generation of CHiLL transformation recipes.

2.1.3 Locality Decision Algorithm

The main goal of this compiler decision algorithm is to improve locality of dense matrix computations at all levels of a memory hierarchy, based on Chen et al.'s autotuning locality algorithm published in CGO 2005. The algorithm relies on PBound to identify and quantify data reuse, and it uses such information to select transformations that will result into optimized code variants. Such transformations include loop interchange, loop tiling, loop unrolling and scalar replacement. The output of the algorithm is a set of parameterized transformation recipes, each of which is called a *master code variant*. A search engine subsequently instantiates the optimization parameters to produce a transformation recipe with fixed parameter values (a *code variant*), which CHiLL uses to generate the transformed code. Algorithm 1 is a somewhat simplified description of the current implementation.

For each level of the memory hierarchy, the algorithm queries PBound to determine the loop that carries the most reuse, and the corresponding references with reuse. To exploit this reuse the algorithm selects the transformations to be applied to loops that do not carry temporal reuse for these references, according to the memory level: loop tiling for cache levels and unroll-and-jam for the register level. When tiling is applied, the algorithm generates an extra code variant in which the data accessed within a tile is copied to a temporary data structure, to avoid cache conflicts.

The output of the algorithm is a set of variants, where each variant is represented as a loop order, a set of loops to unrolled, a set of loops to be tiled, and a set of data to be copied. For each variant, the algorithm emits the CHiLL commands (transformations) needed to achieve the code variant's loop order, to tile and unroll loops, and to copy tiled data to temporary arrays.

2.1.4 Integration

We have coupled PBound's data reuse model and source code analysis to the decision algorithm, which in turn has been coupled to CHiLL. To keep the decision algorithm as generic as possible we developed input and output interfaces to separate the algorithm from the generation and consumption of output and input respectively. Our input interface links the algorithm with PBound or any tool that models data reuse analysis and compiler that provides information regarding an application's loop nests and references. Our output interface links with CHiLL or any tool that can use the output from the decision algorithm to transform code.

2.2 Autotuning AMR Multigrid

Based on the relationship between computational characteristics and trends in computer architecture, AMR MG codes exhibit a number of performance-optimization challenges within a node including performance bound by bandwidth, a variable and unpredictable degree on fine- and medium-grained parallelism, and complex domain and

Algorithm 1 DERIVEVARIANTS

```
Algorithm DeriveVariants (variant)
Variants ← variant
Loops ← variant.Loops
Refs ← variant.Refs
level ← 0
while Loops ≠ ∅ and level < MEMORY_LEVEL do
    L ← MostProfitableLoops (Loops, Refs)
    newVariants ← ∅
    foreach v ∈ Variants do
        foreach l ∈ L do
            newVariants ← newVariants ∪ GenerateVariant (v, l, level)
    Variants ← newVariants
    Loops ← Loops − L
    level ← level + 1
foreach v ∈ Variants do
    Order(v.ControlLoops)
    Push(v.ControlLoops, v.LoopOrder)
return (Variants)


GenerateVariant(variant, loop, level)
v ← variant
v.Loops ← v.Loops − loop
RefsWTemporalReuse ← MostProfitableRefs(loop, v.Refs)
if RefsWTemporalReuse ≠ ∅ then
    if level == REGISTER_LEVEL then
        foreach l ∈ v.Loops do
            < l^U, U_l > ← Unroll(l)
            v.Loops ← v.Loops −l + l^U
            v.Params (l^U) ← U_l
        Push(loop, v.LoopOrder)
        return (v)
    else
        foreach l ∈ v.Loops do
            if l ∉ v.ControlLoops then
                < l^T, l^C, T_l > ← Tile(l)
                v.Loops ← v.Loops −l + l^C
                v.ControlLoops ← v.ControlLoops +l^C
                Push(l^T, v.LoopOrder)
                v.Params (l^T) ← T_l
        Push(l, v.LoopOrder)
        v ^copy ← GenCopyVariant (v, RefsWTemporalReuse, l, (T_i, i ∈ v.LoopOrder))
        return (v, v ^copy)
```

coarse-fine boundary conditions. Our work addresses these significant challenges, producing performance-portable, tuned code in an automated fashion.

As a basis for the initial many-core multigrid studies, we have adopted the compact multigrid benchmark from an SC'12 paper by Williams et al. In the paper, significant manual optimization for locality, prefetching, and SIMD as well as orchestration of parallelism was required in order to get good performance. Under X-TUNE, Utah researchers in collaboration with LBL are extending the CHiLL transformation framework to support these transformations automatically.

Scientists create operators by composing simpler operators. For example a smooth operation may be composed of Laplacian, Helmholtz and a Gauss Seidel Red Black operator, as in our benchmark. For computations on 3D grids, each of these operators is a three deep nested for-loop inside a time step loop, shown below.

```
for (t=0; t<sweeps; t++) {
  for (k...){ for (j...){ for (i...) { // LAPLACIAN OPERATOR which writes to temp[k][j][i]...} }}
  for (k...){ for (j...){ for (i...) { // HELMHOLTZ OPERATOR which reads and writes to temp[k][j][i]...} }}
  for (k...){ for (j...){ for (i...) {
    if ((i+j+k+s+1)%2)//RED BLACK GAUSS SEIDEL OPERATOR reads in temp[k][j][i], writes to phi[k][j][i]... }}
}
```

The first optimization is to fuse the three loops together (at the maximum nesting depth), using dependence analysis and data-flow analysis to verify legality. This version of the fused code can be further improved by substituting the array reference to the temp array with a scalar, to eliminate memory traffic. With this fused version we can generate a wavefront version of the smooth function. In CHiLL, wavefronts are derived using loop skewing and loop permutation. We can reduce communication across sockets by adjusting the ghost zone from the wavefront; this adjustment can be explored through autotuning, which is something we plan to do in future work. Results for these three versions of code along with an unfused version are shown in Figure 2.1.

Recently, we have developed a new transformation to support fusion of residual and restrict functions. This transformation must map from the iteration space of a fine grid into a coarser grid, and reorder memory accesses to reduce memory traffic. We have a prototype implementation, and will provide results in the next report.

Concurrently, LBL researchers have been updating the code and evaluating it on Blue Gene/Q, Kepler, and Xeon Phi. Although these architectures are far more energy efficient than their commodity CPU counterparts, attaining peak performance rquires much more of software (essentially, in order to save energy, their designers have shifted the onus from hardware to software). Discovery of the the performance optimization and parallelization challenges on these architectures will be fed back into CHiLL so that it may automatically exploit them. In this vein with have collaborated with researchers at both Intel and NVIDIA. Third, we are developing a high-order version of the code that performs more flops per unit of data moved but has superior convergence rates. In the past (when flop-limited), these additional flops could impede performance. However today and in the future (data-movement limited) we have compute to spare these high-order methods are thus a better solution for exascale architectures. Finally, we are exploring alternate threading approaches within OpenMP that may better mitigate the dynamic/variable parallel challenges in AMR MG codes like ExaCT's (Combustion

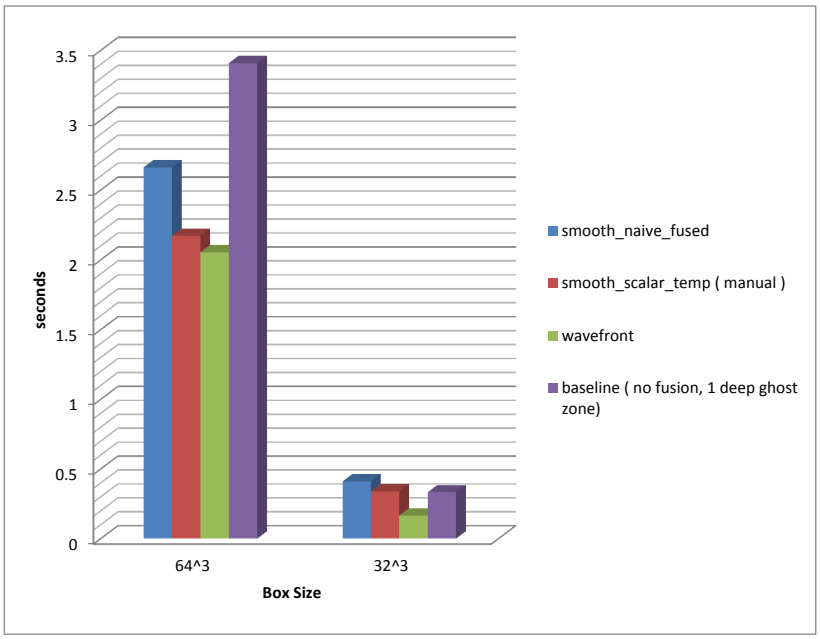


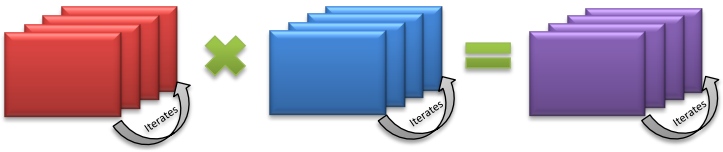
Figure 2.1: Performance results for optimizing smooth operation.

Co-Design Center's) Low-Mach Code (LMC). If different architectures require different approaches to parallelism, we hope to discover which works best and encode it in the compiler. Additionally, we have passed this code to members in DEGAS who are interested in quickly evaluating Habanero C's "async" construct in the context of MG without having to rewrite a 100K-line fortran (+700K lines of C++) code like LMC.

2.3 Optimizing Nekbone

Nekbone solves a Poisson Equation using conjugate gradient (CG) iteration with no preconditioner, and is a compact application representing nek5000, which is part of the CESAR Co-Design Center. The focus of this work is to accelerate the batched matrix-matrix multiplication that is performed in the CG and implement it on the GPU. Basically, CG iterates over multiple matrices that are clustered and applies matrix-matrix multiplication. To show how expensive this function is, Nekbone was profiled with HPCToolkit. Even though this application uses a highly optimized Matrix-Matrix Multiplication kernel, we found that 66% of the computational time is spent doing these operations.

To accelerate the process, these matrix multiplications were moved to a GPU. The idea is that the GPU batch all the matrices and perform the multiplication in one count. Because the matrices are small (10x10), it is possible to pair each block of the computational grid with each operation. This allows performing the multiplications in



(a) Original separate implementation.

(b) Batched implementation.

Figure 2.2: Matrix Multiplication in Nekbone.

parallel.

The GPU functions (kernels) were generated using CUDA-CHiLL and run on a Tesla C2050 (Fermi) GPU. Using CUDA-CHiLL makes it possible to apply different optimizations (e.g. unrolling and tiling) for performance improvement. In this case, no loop was unrolled but the reused data was placed in the device shared memory. Finally, asynchronous calls were used to hide latency of data copy, and at the same time the kernels use concurrent launch to reduce wait time between them.

The preliminary results show that, for the test case, it is possible to achieve a 1.97x speedup using the GPU. The next step of this problem is to identify other parts in the code that share the same properties. Once these functions are identified, we are going to apply a similar strategy in order to achieve higher performance.

2.4 Optimizing NWChem

NWChem is a computational chemistry software suite that includes a wide variety of functionality, including molecular dynamics, density-functional and wavefunction methods. The focus of our effort is the optimization of the coupled-cluster methods in the Tensor Contraction Engine (TCE) module of NWChem, as this is one of the most widely used modules as well as the one most frequently run at large-scale on supercomputers, as coupled-cluster methods are both floating-point and memory intensive methods. The kernels of coupled-cluster theory fall into three basic categories: (1) dense matrix multiplication operations performed by BLAS (i.e. DGEMM), (2) multidimensional array permutation operations, and (3) loop-based multidimensional array contractions. We focus on 3, since 1 is already available in vendor- or expert-optimized libraries and 2 are bandwidth-limited, meaning that there is little opportunity to make them run faster via tuning. The kernels in 3 are sometimes obtained by fusing 1 and 2, which has the disadvantage of not using optimized BLAS but the advantage of an overall reduction in memory traffic. If 3 can be optimized to be as efficient as DGEMM, the overall performance benefit will be substantial.

While there are more than 100 different types of tensor contraction kernels present in the TCE module, we focus on the ones associated with the perturbative triples corrections used in the widely used CCSD(T) method. These kernels were profiled and found to consume approximately 90% of the total time for CCSD(T) calculation, at least at workstation scale (they will still be dominant on hundreds of thousands of cores since they are the leading order floating-point cost). These kernels were factorized out of NWChem and a standalone driver was developed.

Preliminary experiments with the CCSD(T) kernels have focused on OpenMP (none of the TCE codes use threads except via BLAS) and compiler-based vectorization on Intel Sandy Bridge (AVX) and Intel Knights Corner (KNC) architectures. The hand-tuned kernels will serve as a reference point to which the autotuned kernels can be compared.

In Figure 2.3 we see that the thread-scaling of kernels is quite good with straightforward use of OpenMP directives using the Intel 13 compiler. However, the absolute performance of these kernels is lower than what it should be, since these tensor contraction kernels have some of the characteristics of DGEMM and thus should run at a significant fraction of peak when cache optimizations are performed. The current peak performance corresponding to the figure data is 32.5%, 18.15% and 13.37% of usable peak on Intel Westmere (SSE), Sandy Bridge (AVX) and Knights Corner processors. Usable peak performance is determined with a DGEMM call for $m = n = k = 3200$ and $\alpha = \beta = 1.0$. Optimizing for memory locality and cache reuse is far more architecture-specific than threading, hence it is more programmer-intensive to do by hand. Clearly, automated methods for exploring this design space will be essential to realizing the full potential of modern architectures such as Intel Sandy Bridge and Knights Corner.

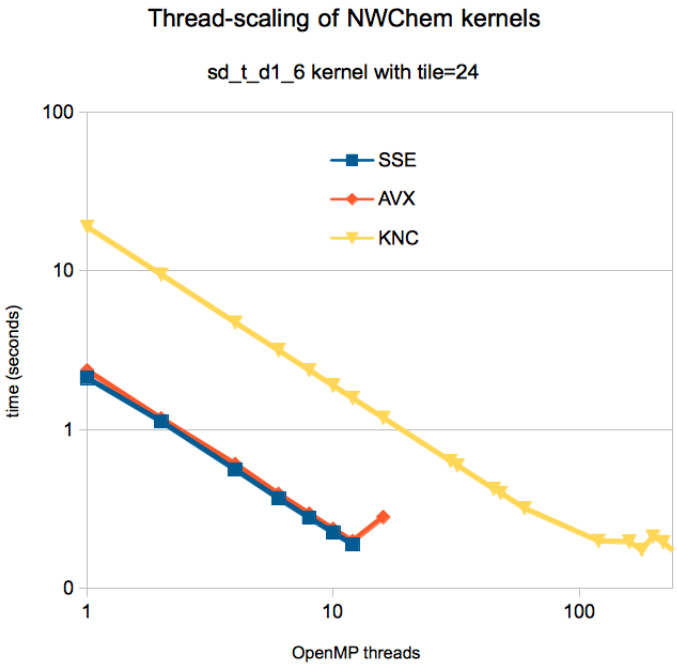


Figure 2.3: Thread-scaling of the NWChem TCE kernel `sd_t_d1_6` with tilesize=24.

3 Plans

Overall, the plan for X-TUNE is to continue the approach of the first six months. Whenever possible, we will leverage existing manually-tuned code, or work with an expert developer. In this way, we ensure that the optimization strategies built into our tools closely approximate the state of the art for current and future architectures. We will work towards automation of these techniques. Already we are developing new optimizations not currently exploited by the compiler community as an outgrowth of working with expert programmers. A nice feature of autotuning technology is that it permits exploration of multiple distinct optimization strategies, which means that we can incorporate a collection of different approaches to find the best optimization result. As the technology matures, we will put our tools in the hands of users to explore how to best meet users' needs.