PATUS: A Code Generation and Auto-Tuning Framework For Parallel Stencil Computations

Matthias Christen, Olaf Schenk, Helmar Burkhart
Department of Mathematics and Computer Science
University of Basel, Switzerland
{ m.christen | olaf.schenk | helmar.burkhart } @unibas.ch

***Abstract*—PATUS is a code generation and auto-tuning framework for stencil computations targeted at modern multi- and many-core processors, such as multicore CPUs and graphics processing units. Its ultimate goals are to provide a means towards productivity and performance on current and future multi- and many-core platforms. The framework generates the code for a compute kernel from a specification of the stencil operation and a Strategy: a description of the parallelization and optimization methods to be applied. We leverage the auto-tuning methodology to find the optimal hardware architecture-specific and Strategy-specific parameter configuration.**

***Index Terms*—stencil computations; code generation; auto-tuning; high performance computing**

I. Introduction

The class of nearest neighbor computations or *stencil computations* captures a computational pattern which can be encountered in many numerical codes, ranging from simple PDE solvers to complex AMR and multigrid solvers, as well as in discrete simulations modeled by cellular automata, or in image processing filters.

Oftentimes, stencil computations comprise a dominant part of the compute time. Therefore, in order to minimize the time to solution, it is crucial that the stencil kernels make use of the available computing resources as efficiently as possible. However, as microarchitectures are becoming increasingly complex and concurrency is explicitly exposed to the programmer, both parallelization and meticulous architecture- and application-specific tuning is required to elicit the machine's full compute power. This not only requires deeper understanding of the architecture, but is also both a time consuming and error-prone process.

The PATUS framework [1] is a code generation and auto-tuning tool for the class of stencil computations striving for both programmer productivity and performance. Using a small domain specific language (DSL), the user defines the stencil kernel using a C-like syntax. The code generation is driven by a Strategy — the implementation of the parallelization and code optimization method — formulated in another DSL. Strategies are designed to be independent both of the stencil specification and the hardware platform, thus PATUS's productivity and portability aspects are ensured by separating the point-wise computation from the algorithmic implementation. They incorporate domain-specific knowledge that enables optimizing the code beyond the abilities of current general purpose compilers. As the performance of stencil computations typically is limited by the available bandwidth to the memory subsystem because of their low arithmetic intensity, i.e., the low number of floating point operations per transferred data element, it is important to make efficient use of caches or scratch pad memory by optimizing spatial and temporal data locality. Candidates for bandwidth-saving schemes include cache blocking techniques [2], [3] and methods to block across multiple time steps [4], [5], [6], [7], which effectively increase the arithmetic intensity. Complementary hardware-aware programming techniques such as NUMA-aware data initialization, software prefetching, or cache bypassing help to reduce bandwidth usage further.

The framework comes with pre-implemented Strategies from which the user can choose — or, if desired, own Strategies can be implemented. The auto-tuning methodology is the means chosen to enable performance portability: Strategies are typically parameterized, and auto-tuning is used to select an optimal or near-optimal parameter configuration with respect to the chosen stencil kernel, Strategy, and hardware platform.

By adapting the hardware architecture specification and the code generation back-end, PATUS is be able to support future hardware microarchitectures and programming paradigms. Currently we support traditional CPU architectures using OpenMP for parallelization and NVIDIA CUDA-capable GPUs.

Recently, other frameworks specifically dealing with stencil computations have been proposed, although their focus is slightly different from the one of PATUS. In particular, *Panorama* [8], [9] was a research compiler for tiling iterative stencil computations in order to minimize cache misses. The Berkeley stencil auto-tuner [10] seeks to substitute an annotated stencil computation in Fortran95 automatically by an optimized version. The *Pochoir* stencil compiler [11] applies the cache oblivious ideas initially formulated by Frigo and Strumpen [12] to stencil codes with ideally many time steps. Another framework in this spirit is CORALS [13] by Strzodka et al. *Mint* [14] targets NVIDIA GPUs as hardware platforms and translates traditional, but annotated, C code to CUDA C and applies hardware-specific optimizations specifically tailored for stencil computations.

More general approaches, not only limited to stencil computations, consider tiling of perfectly and imperfectly nested loops in the polyhedral model [15]. Loop transformation

and (automatic parallelizing) compiler infrastructures in the polyhedral model include CHiLL [16] and PLuTo [17].

II. A Stencil Specification and a Strategy

Listing 1 shows a PATUS stencil specification. The stencil kernel was taken from a real-world application: a finite difference earthquake simulation code, for which benchmark results for kernels on which most of the compute time is spent are presented in Section IV. The example stencil specification shows the calculation of the three-dimensional velocity field `u1` obtained by applying a discretized differential operator to the stress tensor field `xx`, `xy`, `xz`. In the stencil specification the domain is specified by the `domainsize` keyword, which defines a rectangular domain over which the stencil is iterated. `t_max = 1` tells PATUS that only one time step is to be performed within the generated stencil kernel function. Thus, the C source generated by PATUS can be substituted for the original code.

The actual stencil computation is defined within the `operation`. The arguments to the `operation` are the input and output grids needed for the computation; an additional `const` specifier declares a grid to be constant in time, i.e., as not being written to within the `operation`. Optionally, the grid size can be specified in round brackets to match the size of the array as it was actually allocated. The body of the `operation` contains the localized, point-wise stencil expression; stencil sweeps, i.e., the spatial iterations, are not programmed explicitly.

```
stencil uxx1
{
  domainsize = (nxb .. nxe, nyb .. nye, nzb .. nze);
  t_max = 1;

  operation (
    const float grid d1(-1..nx+2,-1..ny+2,-1..nz+2),
    float grid u1(-1..nx+2, -1..ny+2, -1..nz+2),
    const float grid xx(-1..nx+2,-1..ny+2,-1..nz+2),
    const float grid xy(-1..nx+2,-1..ny+2,-1..nz+2),
    const float grid xz(-1..nx+2,-1..ny+2,-1..nz+2),
    float param dth)
  {
    float c1 = 9./8.;
    float c2 = -1./24.;

    float d = 0.25 * d1[x,y,z] + d1[x,y-1,z] +
      d1[x,y,z-1] + d1[x,y-1,z-1]);
    u1[x,y,z; t+1] = u1[x,y,z; t] + (dth / d) * (
      c1 * (
        xx[x,y,z] - xx[x-1,y,  z  ] +
        xy[x,y,z] - xy[x,  y-1,z  ] +
        xz[x,y,z] - xz[x,  y,  z-1]) +
      c2 * (
        xx[x+1,y,  z  ] - xx[x-2,y,  z  ] +
        xy[x,  y+1,z  ] - xy[x,  y-2,z  ] +
        xz[x,  y,  z+1] - xz[x,  y,  z-2])
    );
  }
}
```

Listing 1. An example stencil specification taken from the earthquake simulation code in Section IV.

The idea of Strategies is to provide a clean mechanism which separates the implementation of parallelization and bandwidth-optimizing methods from the actual stencil computation. In this way, the implementation of the algorithm can be reused for arbitrary stencils.

Listing 2 shows a simple cache blocking Strategy. It iterates over all the time steps in the `t` loop, and within one time step in blocks `v` of size `cb` over the *root domain* `u`, i.e., the entire domain to which to apply the stencil. Both the root domain and the size of the subdomain `v` are given as Strategy parameters. The blocks `v` are executed in parallel by virtue of the `parallel` keyword, which means that the subdomains `v` are dealt out in a cyclic fashion to the worker threads. The parameter `chunk` to the `schedule` keyword defines how many consecutive blocks one thread is given. Then, the stencil is applied for each point in the subdomain `v`.

The Strategy argument `cb` has a specifier, `auto`, which means that this parameter will be interfaced with the auto-tuner: it is exposed on the command line of the benchmarking harness so that the auto-tuner can provide values for `cb`= $(c_1, c_2, \ldots, c_d)$, where d is the dimensionality of the stencil, and pick the one for which the best performance is measured.

```
strategy cacheblocking (domain u, auto dim cb,
  auto int chunk)
{
  // iterate over time steps
  for t = 1 .. stencil.t_max
  {
    // iterate over subdomain
    for subdomain v(cb) in u(:; t)
      parallel schedule chunk
    {
      // calculate the stencil for each point
      // in the subdomain
      for point p in v(:; t)
        v[p; t+1] = stencil (v[p; t]);
    }
  }
}
```

Listing 2. A cache blocking Strategy.

The benefit of cache blocking is improved temporal data locality, i.e., more efficient use of the cache, which results in a performance increase. By decomposing the grid into cache size dependent small subdomains it is ensured that data loaded into the cache can be reused before being evicted due to capacity misses.

III. The Patus Framework

PATUS is built from four core components as shown in the high-level overview in Fig. 1: the parsers for the two input files, the stencil definition and the Strategy, the actual code generator, and the auto-tuner. PATUS is written in Java. As parser generator for the stencil specification and Strategy parsers, and also for the parser used to interface with the computer algebra system Maxima [18], which is used as a powerful expression simplifier, Coco/R [19] was used. The Cetus framework ([20], [21]) provides Java classes for the internal representations for both the Strategies and the generated code, i.e., the Strategy parse tree and the abstract syntax tree of the generated code. Cetus also provides the mechanism for unparsing the internal representation of the generated code.

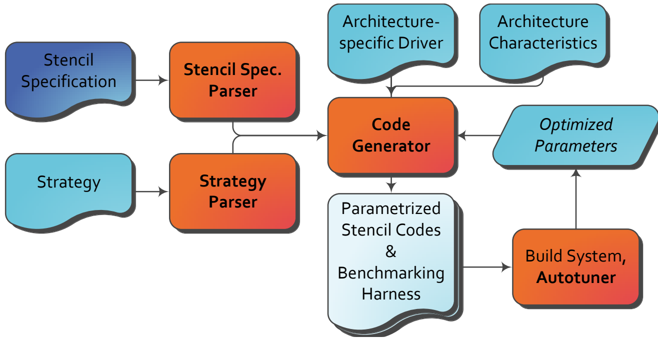

Fig. 1. High-level overview of the software architecture of PATUS. The strategy and the stencil specification are input files, which drive the code generation. The code generator creates a set of parameterized hardware-specific kernels that are executed by the auto-tuner, which determines the optimal parameter set.

The internal representation of the stencil specification consists of the domain size and number-of-iterations attributes and a graph representation of the actual stencil parts described in the stencil `operation`. The Strategy is transformed to an abstract syntax tree that is used as a template by the code generator. To this end, Strategy-specific Cetus-based IR classes were added.

These structures are passed as input to the code generator, along with an additional configuration describing the characteristics of the hardware and the programming model used to program the architecture and specifies the code generation back-end to use. The code generator produces C code for variants of the stencil kernel and also creates an initialization routine that implements a NUMA-aware data initialization based on the parallelization scheme used in the kernel routine.

The objective of the code generator is to translate the Strategy into which the stencil has been substituted, into the final C code. In particular, it transform Strategy loops into C loops and parallelizes them according to the specification in the Strategy, and it unrolls and vectorizes the inner-most loop containing the stencil calculation if desired; it determines which arrays to use based on the Strategy structure and the grids defined in the stencil specification and calculates the indices for the array accesses.

Along with an implementation for the stencil kernel, the code generator also creates a benchmarking harness from an architecture- and programming model-specific template into which the dynamic memory allocations, the grid initializations, and the kernel invocation are substituted. The benchmarking harness expects the problem-specific parameters related to the domain size (specified in the stencil specification), the Strategy-specific `auto` parameters, as well as internal code generation parameters (currently loop unrolling factors) to be provided to the benchmarking executable as command line arguments.

Based on a range specification for the parameters and optional constraints (e.g., to assert that the number of `v`-blocks in Listing 2 is at least the number of running threads), the auto-tuner runs the benchmark executable repeatedly with varied parameter configurations. Cetus's expression simplifier is used to determine whether the constraints are satisfied for a specific parameter configuration. The exploration of the search space is driven by a derivative-free search method. A selection of methods, including exhaustive search, a greedy search searching along coordinate axes and fixing the best value before progressing to the next axis, general combined elimination [22], simplex search, and a genetic algorithm, has been implemented. The auto-tuner can be readily supplemented with additional search methods in a modular fashion.

TABLE I
SUMMARY OF THE AWP KERNELS WHICH WERE USED IN THE PERFORMANCE BENCHMARKS.

Name	Description	Flops	Arith. Int.
uxx1	Velocity in x-dimension	20	0.70 Flop/B
xy1	Stress tensor component σ_{xy}	16	0.65 Flop/B
xyz1	Stress tensor components σ_{xx}, σ_{yy}, σ_{zz}	90	1.58 Flop/B
xyzq	Visc. stress tensor cmps. σ_{xx}, σ_{yy}, σ_{zz}	129	1.22 Flop/B

IV. SELECTED PERFORMANCE RESULTS

In this section, we present performance results obtained from stencil kernels taken from a real-world application, The Anelastic Wave Propagation code AWP-ODC of the Southern California Earthquake Center (SCEC), which was developed by Olsen, Day, Cui, and Dalguer [23]. This scientific modeling code is a finite difference code implemented in Fortran+MPI for simulating both dynamic rupture and earthquake wave propagation. It has been used to conduct numerous significant simulations at the SCEC.

The four stencil kernels, in which most of the compute time is spent, are summarized in Table I. The "Name" column shows the names of the original Fortran routines, which were translated into the PATUS stencil specification DSL and for which the performance benchmarks were carried out. We only present the kernel-specific results, not results from entire application benchmarks. The "Flops" column shows the number of floating points operations per stencil evaluation, and the "Arith. Int." column gives the arithmetic intensity numbers in Flops per transferred Byte. The benchmarks were carried out on a $188 \times 188 \times 152$ domain, and the arithmetic intensities account for the boundary and write allocate traffic incurred.

The benchmarks were done on two architectures, a AMD Opteron "Magny Cours" and an NVIDIA C2050 Fermi GPU. The former is a dual-socket platform with two dies per socket and 6 cores per die. Thus, the total of 24 cores are distributed among 4 NUMA domains. We used the GNU gfortran/gcc 4.5.2 compilers with the `-O3` optimization flag, for both the reference codes and the generated PATUS codes. The GPU features 448 streaming processors packaged into 14 streaming multiprocessors. We used CUDA 4.0 and NVIDIA's nvcc compiler.

The performance benchmarks were done in single precision, as the original application uses that precision mode. The orange markers show the maximum attainable performance

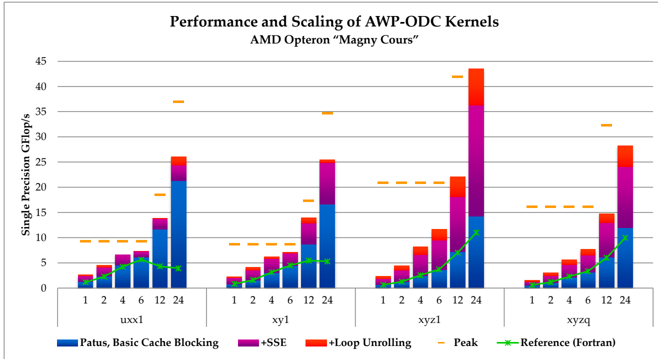

Fig. 2. AWP kernel benchmarks on the AMD Opteron “Magny Cours.”

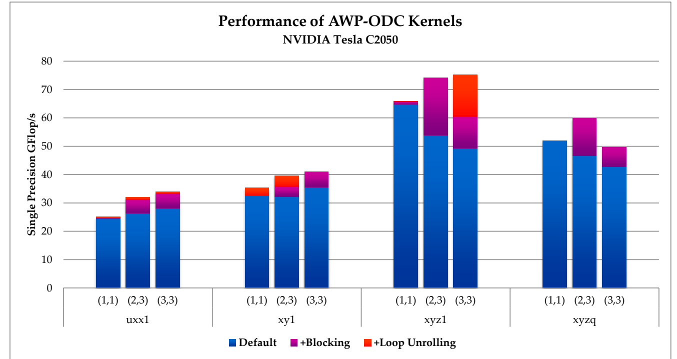

Fig. 3. AWP kernel benchmarks on the NVIDIA Tesla C2050 Fermi GPU.

calculated from the measured sustained bandwidth (53.1 GB/s on AMD Opteron “Magny Cours” when all 4 NUMA domains are used) and the arithmetic intensity of the stencil kernels. On one die (6 threads) the “uxx1” and “xy1” kernels reach around 80% of the peak and around 70% if all 24 threads are used. The theoretical maximum is quite high for the “xyz*” kernels, and the reason why only a fraction (around 40%–50%) of the maximum was achieved lies in the arithmetic operations: the kernels contain many divisions (18 in both cases), which are notorious for incurring pipeline stalls.

The green line shows the performance of the reference Fortran code, which was parallelized by inserting an OpenMP sentinel above the outer most spatial loop. No NUMA optimization was done, which is evident from the scaling behavior of the reference “uxx1” and “xy1” kernels. The arithmetic intensities of the “xyz*” kernels are higher; hence the NUMA effect is mitigated to some extent by the relatively high number of floating point operations, and the performance can increase further when going to 2 and 4 NUMA domains (12 and 24 threads, respectively).

The blue bars show the performances of auto-tuned blocked codes, including the NUMA optimization, and relying on the compiler to do the vectorization. With the NUMA optimization enabled, the performance scales almost linearly up to 24 threads. Fig. 2 shows that explicit use of SSE intrinsics and padding for optimal vector alignment results in a significant performance boost, in particular for the “xyz1” kernel, where explicit vectorization gave a performance increase of 150%. Activating and tuning for loop unrolling gave another slight gain in performance.

Overall, PATUS achieved speedups between $2.8\times$ and $6.6\times$ with respect to the original Fortran code using 24 threads on the AMD Opteron “Magny Cours” with the NUMA optimization enabled and a cache blocking Strategy (the one shown in Listing 2).

The GPU performance results are shown in Fig. 3. GPU support in PATUS is still work in progress, thus the performance numbers shown in the figure are to be treated as a result of a proof of concept. Again, the results are for single precision stencils, and the three indexing modes were used: one-dimensional thread blocks and grids, three-dimensional thread blocks and a two-dimensional grid, and both three-dimensional thread blocks and grids, supported as of CUDA 4.0 on Fermi GPUs. For both the “uxx1” and “xy1” kernels, the default thread block size of $16 \times 4 \times 4$ threads was an adequate choice, and tuning the thread block sizes only increased the performance slightly. In both cases, the fully 3D indexing mode outperformed the $(2, 3)$-dimensional indexing by a tight margin due to the simplified index calculation code. We hope to be able to increase the performance in the future by making use of the GPU-specific hardware features such as its shared memory.

V. CONCLUSIONS

In this paper, we presented PATUS, a code generation and auto-tuning framework for general stencil computations. It is thought of as both a productivity tool and a tool for experimenting with parallelization and optimization strategies, such as bandwidth-saving algorithms: it is for both programmers in need of an efficient implementation of a stencil kernel for a given hardware architecture, but who do not want to care about hardware-specific tuning, and for domain experts who want to experiment. The modular architecture of the system allows to add new components, such as back-ends for other and future hardware.

We have shown that the approach works for both modern multi- and many-core architectures, and the performance numbers demonstrate the potential of leveraging non-trivial strategies and the auto-tuning methodology.

In its current state, the framework still has limitations (restriction to shared memory architectures, no special boundary treatment, lacking support for temporal blocking schemes), which we intend to overcome in the future.

PATUS is open source software and licensed under the GNU Lesser GPL. It can be obtained from `http://code.google.com/p/patus/`.

REFERENCES

[1] M. Christen, O. Schenk, and H. Burkhart, “PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures,” in *25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011)*, 2011.

[2] Y. Song and Z. Li, "New Tiling Techniques to Improve Cache Temporal Locality," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.
[3] G. Rivera and C. Tseng, "Tiling optimizations for 3D scientific computations," in *Supercomputing, ACM/IEEE 2000 Conference*, 2000.
[4] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors," *SIAM Review*, vol. 51, no. 1, pp. 129–159, 2009.
[5] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart, "Parallel Data-Locality Aware Stencil Computations on Modern Micro-Architectures," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 2009, pp. 1 –10.
[6] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *COMPSAC (1)*, 2009, pp. 579–586.
[7] J. Meng and K. Skadron, "A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations," *International Journal of Parallel Programming*, vol. 39, pp. 115–142, 2011, 10.1007/s10766-010-0142-5. [Online]. Available: http://dx.doi.org/10.1007/s10766-010-0142-5
[8] Z. Li and Y. Song, "Automatic tiling of iterative stencil loops," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 6, pp. 975–1028, 2004.
[9] Y. Song and Z. Li, "A compiler framework for tiling imperfectly-nested loops," in *Proc. of 12th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, 1999, pp. 185–200.
[10] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An Auto-tuning Framework For Parallel Multicore Stencil Computations," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, April 2010, pp. 1 –12.
[11] Y. Tang, R. Chowdhury, B. Kuszmaul, C. Luk, and C. Leiserson, "The Pochoir Stencil Compiler," in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989508
[12] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 361–366.
[13] R. Strzodka, M. Shaheen, and D. Pajak, "Time skewing made simple," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 295–296. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941596
[14] D. Unat, X. Cai, and S. Baden, "Mint: realizing CUDA performance in 3D stencil methods with annotated C," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 214–224. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995932
[15] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. Strout, "Parameterized Tiled Loops for Free," *SIGPLAN Not.*, vol. 42, pp. 405–414, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250780
[16] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. Khan, "Loop Transformation Recipes for Code Generation and Auto-Tuning," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, G. Gao, L. Pollock, J. Cavazos, and X. Li, Eds., vol. 5898. Springer Berlin / Heidelberg, 2010, pp. 50–64.
[17] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," in *Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*, 2008.
[18] L. Beshenov, "Maxima, a Computer Algebra System," http://maxima.sourceforge.net/, accessed July 2011.
[19] H. Mössenböck, M. Löberbauer, and A. Wöß, "The Compiler Generator Coco/R," http://www.ssw.uni-linz.ac.at/coco, accessed July 2011.
[20] T. C. Team, "Cetus – A Source-to-Source Compiler Infrastructure for C Programs," accessed July 2011. [Online]. Available: http://cetus.ecn.purdue.edu/
[21] H. Bae, L. Bachega, C. Dave, S. Lee, S. Lee, S. Min, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," in *Proceedings of the 14th Int'l Workshop on Compilers for Parallel Computing*, 2009.
[22] Z. Pan and R. Eigenmann, "PEAK – A Fast and Effective Performance Tuning System via Compiler Optimization Orchestration," *ACM Trans. Program. Lang. Syst.*, vol. 30, pp. 17:1–17:43, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1353445.1353451
[23] Y. Cui, K. Olsen, T. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. Panda, A. Chourasia, J. Levesque, S. Day, and P. Maechling, "Scalable Earthquake Simulation on Petascale Supercomputers," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.45