In-Core Optimization of High-order Stencil Computations

Hikmet Dursun, Ken-ichi Nomura, Weiqiang Wang, Manaschai Kunaseth, Liu Peng,
Richard Seymour, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta
Collaboratory for Advanced Computing and Simulations,
Department of Computer Science,
Department of Physics & Astronomy,
Department of Chemical Engineering & Materials Science,
University of Southern California, Los Angeles, CA 90089-0242, USA
(hdursun, knomura, wangweiq, kunaseth, liupeng, rseymour, rkalia, anakano, priyav)@usc.edu

Abstract

In this paper, we apply in-core optimization techniques to high-order stencil computations, including: (1) cache blocking for efficient L2 cache use; (2) register blocking and data-level parallelism via single-instruction multiple-data (SIMD) techniques to increase L1 cache efficiency; and (3) software prefetching techniques. Our generic approach is tested with a kernel extracted from a 6th-order stencil based seismic wave propagation code on a suite of Intel Xeon architectures. Cache blocking and prefetching techniques are found to achieve modest performance improvement, whereas register blocking and SIMD implementation reduce L1 cache line miss dramatically accompanied by moderate decrease in L2 cache miss rate. Optimal register blocking sizes are determined through analysis of cache performance of the stencil kernel for different sizes of register blocks, thereby achieving over 4.3-fold speedup on Intel Harpertown. We also examine lower precision (3rd, 4th, and 5th orders) stencil computations to analyze the dependency of data-level parallel efficiency on the stencil order.

1. Introduction

The search for ways to improve performance while improving power efficiency has led architecture community to the development of heterogeneous multicore processors with complex cache hierarchies [1]. The shift in architectural design has provided incentives for software developers and application scientists to investigate cache efficient algorithms [2] and on-chip optimizations to utilize underlying hardware for broad computational applications.

Due to ever-increasing cooling challenges, general-purpose commercial microprocessors added single-instruction multiple-data (SIMD) vector extensions to achieve high performance while running at slower clock speed by exploiting data-level parallelism (DLP) with minimal changes to instruction set architecture. The Intel x86 architecture added the MMX instruction extensions, and the PowerPC architecture added Altivec, both of which allowed four single-precision floating-point operations to execute simultaneously using SIMD instructions. The IBM Cell architecture derives most of its performance from DLP thanks to its eight vector processors, called synergistic processing elements (SPE), which execute four single-precision floating-point operations per cycle each.

A common computational kernel used in a variety of scientific and engineering applications is stencil computation (SC). Extensive efforts have been made to optimize SC on multicore platforms with the main focus on low-order SC. For example, Williams et al. [3] have optimized a lattice Boltzmann application on leading multicore platforms, including Intel Itanium2, Sun Niagara2 and STI Cell. Datta et al. have recently performed comprehensive SC optimization and auto-tuning with both cache-aware and cache-oblivious approaches on a variety of state-of-the-art architectures, including NVIDIA GTX280, etc. [4]. Other approaches to SC optimization include tiling [5] and iteration skewing [6-8]. Due to the importance of high-order SC in broad applications and the wide landscape of multicore architectures as mentioned above, it is desirable to optimize SC to get maximal performance.

While it is possible to employ divide-and-conquer strategies for structured grid problems and perfectly scale such algorithms on massively parallel computers [9], utilizing in-core level optimization techniques is essential to exploit flops performance of the underlying computational units. In this paper, we explore in-core optimization techniques addressing high-order stencil computations. Our tests include (1) cache blocking (CB) targeting efficient L2 cache use; (2) register blocking (RB) and data-level parallelism via single-instruction multiple-data (SIMD) techniques to increase L1 cache efficiency; and (3) software prefetching techniques. We test our generic approach with a kernel extracted from a 6th-order stencil based seismic wave propagation code on a suite of Intel Xeon architectures. CB and prefetching techniques achive modest improvement. However, our results show RB and SIMD implementation reduce L1 cache line miss dramatically accompanied by a moderate level decrease in L2 cache miss rate. We analyze cache performance of our stencil kernel for different sizes of register blocks to find optimal RB size and thereby achieve over 4.3-fold speedup on Intel Harpertown. We also explore lower precision stencil computations at 3rd, 4th and 5th orders to analyze dependency of data-level parallel efficiency on the stencil order and to show use of RB technique in applications of other stencil orders.

This paper is organized as follows: An overview of the stencil problem is given in Section 2 together with the description of the kernel. Section 3 includes our in-core optimization techniques and performance analysis results. Finally, we summarize our study in Section 4.

2. High-order stencil application

This section introduces the general concept of high-order stencil based computation as well as details of our experimental kernel.

2.1. Stencil computations

Stencil computation (SC) is at the heart of a wide range of scientific and engineering applications. A number of benchmark suites, such as PARKBENCH [10] and NAS Parallel Benchmarks [11], include stencil computations. Implementation of special purpose stencil compilers highlights the common use of stencil computation based methods [12].

SC involves a field that assigns values $v_t(\mathbf{r})$ to a set of discrete grid points $\mathbf{\Omega} = \{\mathbf{r}\}$, for the set of simulation time steps $T = \{t\}$. SC routine sweeps over $\mathbf{\Omega}$ iteratively to update $v_t(\mathbf{r})$ using a numerical approximation technique as a function of the values of the neighboring nodal set including the node of interest, $\mathbf{\Omega}' = \{\mathbf{r}' \mid \mathbf{r}' \in neighbor(\mathbf{r})\}$ which is determined by the stencil geometry. The pseudocode below shows a naïve stencil computation:

```
for $\forall t \in T$
  for $\forall \mathbf{r} \in \mathbf{\Omega}$
    $v_{t+1}(\mathbf{r}) \leftarrow f(\{v_t(\mathbf{r}') \mid \mathbf{r}' \in neighbor(\mathbf{r})\})$
```

where f is the mapping function and v_t is the scalar field at time step t. SC may be classified according to the geometric arrangement of the nodal group $neighbor(\mathbf{r})$ as follows: First, the order of a stencil is defined as the distance between the grid point of interest, $\mathbf{r}$, and the farthest grid point in $neighbor(\mathbf{r})$ along a certain axis. (In a finite-difference application, the order increases with required level of precision.) Second, we define the size of a stencil as the cardinality $|\{\mathbf{r}' \mid \mathbf{r}' \in neighbor(\mathbf{r})\}|$, i.e., the number of grid points involved in each stencil iteration. Third, we define the footprint of a stencil by the cardinality of minimum bounding orthorhombic volume, which includes all involved grid points per stencil. For example, Figure 1 shows a 6th order, 25-point SC whose footprint is $13^2 = 169$ on a 2-dimensional lattice. Such stencil is widely used in high-order finite-difference calculations [13, 14].

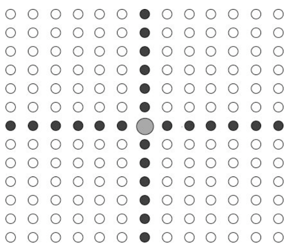

Figure 1. 6-th order, 25-point SC whose footprint is 13^2 on a 2-dimensional lattice.

In Figure 1, the grid point of interest, $\mathbf{r}$, is shown as the central grey circle while the set of neighbor points, excluding $\mathbf{r}$, i.e., $\{\mathbf{r}' \mid \mathbf{r}' \in neighbor(\mathbf{r})\} - \{\mathbf{r}\}$, is illustrated as solid circles. White circles show the other lattice sites within the stencil footprint, which are not used for calculation of $v(\mathbf{r})$.

2.2. Experimental application

This subsection introduces our experimental kernel, which is taken from an application that simulates seismic wave propagation by employing a 3D equivalent of the stencil in Figure 1 to compute spatial derivatives on uniform grids using a finite difference method. The 3D stencil geometry is highly off-diagonal (6-th order) and involves 37 points (footprint is $13^3 = 2{,}197$), i.e., each grid point interacts with 12 other grid points in each of the x, y and z Cartesian directions.

Figure 2 shows a pseudocode of our experimental kernel. The pseudocode sweeps the problem domain of size `NX×NY×(NZ-2×STENCILORDER)` in the x—unit stride—direction, to accumulate contributions of grid points neighboring in z direction, which has the largest stride (`NX×NY`). The target grid points (`nextGrid` array) are updated with the contributions of the source grid points (`currentGrid` array) at each sweep of the domain. `LINEARINDEX` is a macro defined to map 3D problem grid onto linear memory allocated for `nextGrid and currentGrid` arrays. The bounds of the uppermost loop are shrunk by `STENCILORDER` (6 in our case) since our kernel does not handle the boundary conditions (in the actual application boundary grids are approximated by lower order derivative approximations). Our experimental kernel has `NX×NY×NZ` $= 384^2 \times 372 = 54.85$ million points. For each grid point, the code allocates 5 floats to hold temporary arrays and intermediate physical quantities and the result, therefore its memory footprint is $5\times4\text{bytes}\times5.5\times10^7 = 1.1$ GB.

```
FOR z = STENCILORDER to NZ-STENCILORDER
   FOR y = 1 to NY
      FOR x = 1 to NX
          FOR s = -STENCILORDER to STENCILORDER
             COMPUTE nextGrid[LINEARINDEX(x,y,z)] as the accumulation of
             currentGrid[LINEARINDEX(x,y,z+s)] contributions
          END FOR
      END FOR
   END FOR
END FOR
```

Figure 2. Pseudocode of the high-order stencil kernel.

3. Optimizations and performance analysis

In this section, we describe our in-core optimizations and present a comprehensive comparison of their effectiveness. We test our implementations in lower and higher models of Intel Xeon Clovertown series quadcore processors (E5320-E5345) and Intel Xeon Harpertown (E5420) architecture. Harpertown, clocked at 2.5 GHz, has a 4×32 KB, 8-way set associative L1 data and instruction cache as well as a large L2 cache (2×6144 KB, 24-way set associative) both featuring 64-byte line size. Clovertown series includes a similar L1 cache, but differs in L2 where each chip shares a 4MB, 16-way set associative cache. Xeon E5320 is clocked at 1.83 GHz and has a 1066 MHz front side bus (FSB) frequency, whereas Xeon E5345 (2.33 GHz clock speed), and Xeon E5420 features a FSB running at 1333 MHz.

3.1. Cache blocking (CB)

The data set in our application is 1.1 GB whereas cache size for the processors in current HPC literature is limited to a few MBs. The fact that higher performance can be achieved for smaller data sets fitting into cache memory suggests a divide-and-conquer strategy for larger problems. We use CB to increase spatial locality, i.e. referencing nearby memory addresses consecutively, and reduce effective memory access time of the application by keeping blocks of future array references at the cache for re-use. Figure 3 shows our implementation of CB.

In Figure 3, loops are subdivided into smaller loops of size `blockParam`. The blocking parameter, `blockParam`, should be selected large enough to amortize the cost of the added loops but the size of the blocks should not exceed physically available cache size. In fact, for a *d* dimensional stencil problem with *m* intermediate physical quantities per grid point, e.g., pressure and/or temperature, and of size *stencilsize* as defined in Section 2.1, given each physical quantity is represented by data of type *datatype*, the blocking parameter should be chosen such that $(\texttt{blockParam})^d \times m \times \mathit{stencilsize} \times \mathit{sizeof}(\mathit{datatype})$ = effective cache size, where we refer to effective cache size instead of physical size to account for memory mapping rules. For example, for an Intel Harpertown (Xeon E5420) with 6 MB L2 cache shared between two cores on each chip, we try a variety of `blockParam` values in the range of 32-48 for our 3D problem which requires 5 float arrays of size equal to the number of global grid points.

```
FOR z = STENCILORDER,NZ-STENCILORDER,blockParam
   FOR y = 1,NY,blockParam
      FOR x = 1,NX,blockParam
         FOR zz = z,MIN(z+blockParam,NZ-STENCILORDER),1
            FOR yy = y,MIN(y+blockParam,NY),1
               FOR xx = x,MIN(x+blockParam,NX),1
                  FOR s = -STENCILORDER to STENCILORDER
                  COMPUTE nextGrid[LINEARINDEX(XX,YY,ZZ)] as the accumulation of
                  currentGrid[LINEARINDEX(XX,YY,ZZ+s)] contributions
                  END FOR
               END FOR
            END FOR
         END FOR
      END FOR
   END FOR
END FOR
```

Figure 3. Pseudocode for cache blocking implementation.

The performance results are given in Figure 4, where the wall-clock time per iteration step (for one complete sweep of 3D problem domain) of the original code is compared with that of the CB-optimized code on Xeon Clovertown (E5320-E5345) and Harpertown series (E5420) in Figure 4(a), and the corresponding speedup over the original program is shown in Figure 4(b). (Figure 4(b) normalizes the wall-clock times on each processor with respect to its own timings as it visualizes speedup per architecture.) The figure shows modest speedup due to CB.

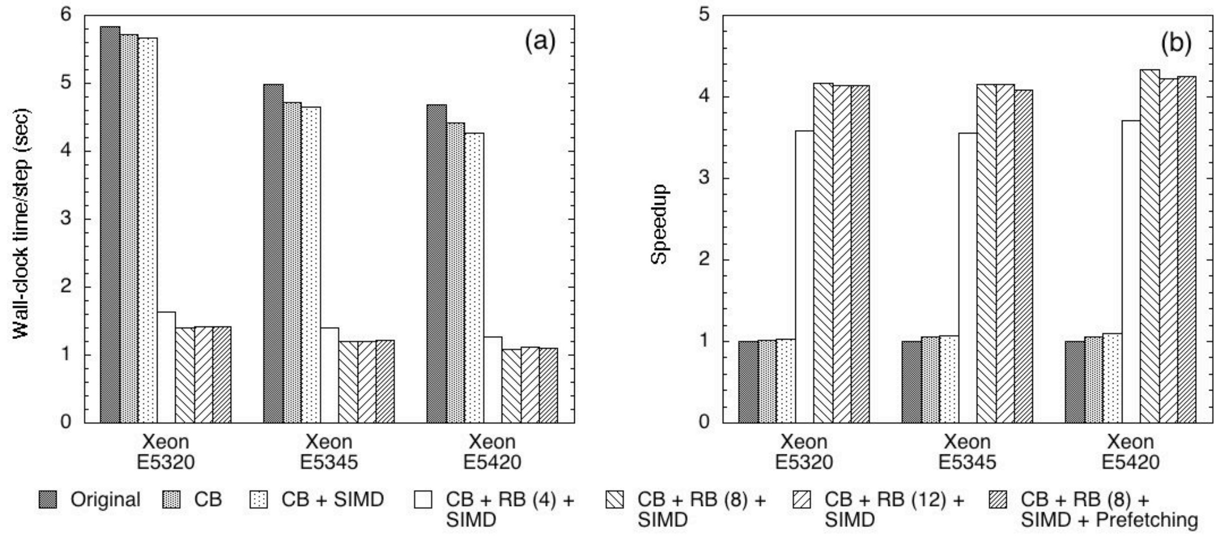

Figure 4. A comprehensive comparison of our optimization techniques.

We next SIMDize the `COMPUTE` statement inside the innermost loop in Figure 3 for implementation of CB and analyze the cache behavior. For SIMDization, we use Streaming SIMD Extensions (SSE3) intrinsics on Intel based architectures, whose prototypes are given in Intel's xmmintrin.h and pmmintrin.h. Figure 4 exhibits slight performance increase of the combined CB + SIMD optimization over the CB-optimized code. For further analysis, we profile the code on Xeon E5320 architecture using Intel's Vtune Performance Analyzer. Cache analysis shows that CB (+SIMD) approach decreases the number of load operations that miss the L1 data cache and send a request to the L2 cache to fetch the missing cache line by 20%. The number of load operations that miss the L2 cache and result in bus request to fetch the missing cache line is also decreased by 12%, confirming the slight speed up in Figure 4. This is consistent with Yelick *et al.*'s findings in their study on sparse matrix operations [15], as stencil computations can be viewed as an extension of sparse linear algebra problems with known patterns. Since high stride accesses are still present in the code even after CB, cache miss rate does not decrease considerably, and even when a cache hit occurs, only one float per cache line is utilized in actual computation. We address this issue in the next subsection with RB technique.

3.2. Register blocking (RB)

In stencil applications, it is not possible to have small strides for all directions of computation at the same time, e.g., for computations in z direction stride size is 59 KB in our kernel whereas x stride is unit stride, i.e. 4 bytes. CB methodology suffers from suboptimal use of the referenced cache lines and pays the penalty of accessing to memory units at a later time for a word located adjacent to a previously addressed memory location.

We completely eliminate these multiple sweeps over the neighboring memory addresses by rearranging computation and exploit spatial locality through use of machine registers maximally. Figure 5 schematically represents our approach to RB for high-order stencil computation. In the original kernel, red square indicates target grid points to be updated at certain time. Blue, yellow and green squares respectively show memory locations of the nearest neighbor points in x, y, and z Cartesian coordinates. During each iteration, the red square is updated by accessing all neighbors (not necessarily the nearest neighbors, and the stencil order determines the distance of the furthest accessed neighbor.) Memory stride for each direction is 1, `NX`, and `NX`×`NY`, respectively, as shown in Figure 2. Instead of the original kernel shown in the upper part of Figure 5, RB deals with a chunk of target grid points contiguous in the x direction. The same size of neighboring cells is fetched to update the block of target grids, which maximally utilize register. This is shown in the lower part of Figure 5.

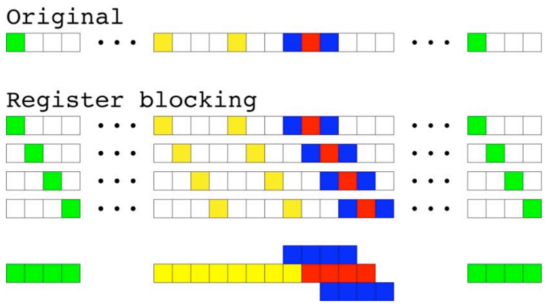

Figure 5. Memory access pattern for register blocking technique compared to original version.

The use of register blocks saves 3 accesses to floats in neighboring grid points per 16-byte block and clearly increases cache performance at both levels as the following performance analysis shows. Note that we also unroll the innermost loop shown in Figure 3 for RB scheme since the block sizes are fixed. This avoids loop overhead and improves opportunity for better instruction scheduling with increased instruction level parallelism.

Figure 4(a) compares RB performance for several block sizes (4, 8, and 12) to find the optimal block size. In Figure 4(b), 8 float-sized register blocks show over 4-fold speedup on all platforms, with the best speedup of 4.3 on Harpertown. Even though E5345 times are less than Xeon E5320 in Figure 4(a), Xeon E5320 and Xeon E5345 normalized speedups almost exactly match in Figure 4(b). It should be noted that two processors have the same cache organization, therefore our cache-utilization targeting optimizations showed similar speedups on both processors albeit run times for E5345 is less due to its higher clock frequency. The Harpertown (E5420) processor however shows slightly better speedup than both Clovertowns, due to its enhanced cache capacity.

Figure 6 shows cache behavior on Xeon E5320 processor for RB + CB + SIMD implementation with a variety of block sizes shown in numerals. Here, Figure 6(a) shows the number of cache lines fetched from the L2 cache by the retired loads as the number of missed L1 data cache lines on the first y-axis. Second y-axis has the number of cache lines fetched from memory by retired loads as the number of L2 cache line misses. Both counts only account for loads from cacheable memory. It should also be noted that Figure 6 counts multiple misses from the same cache line as single line miss. Figure 6(b) scales the numbers in Figure 6(a) with the number of instructions that retire execution of each RB implementation. We observe the highest cache performance for register blocks of 8 float numbers, which we propose as the optimal register block size for our problem depending on speedup and cache line miss reduction: 2.68-fold decrease in missed L1 line rate and 23% decrease in L2 cache line miss rate. As one might expect, overuse of machine registers saturates the speedup, therefore 12 float sized register blocks show suboptimal performance improvement.

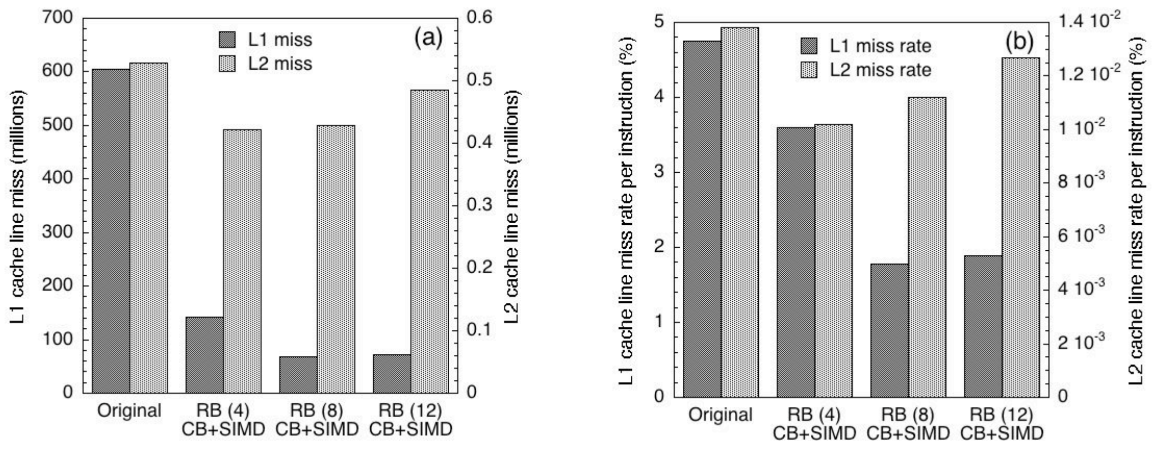

Figure 6. L1 and L2 cache line misses.

In the RB approach, contrary to the common SIMDization frameworks, we do not use zero padded registers to have dense blocks, i.e. to fill them. Therefore machine registers are maximally used. In order to show this property, we modify our kernel to simulate lower-order stencil problems requiring less precision, and implement by previously mentioned CB + SIMD technique employing zero padded SIMD vectors. Figure 7 compares the wall-clock time of this approach to that of the optimally sized RB scheme. A dramatic increase in wall-clock time of CB + SIMD approach is observed between 4th and 5th order stencil computations, since 5th order stencil requires another 4 float sized SIMD vector, which is padded by 3 zeros—only effectively uses 4 bytes. Since RB approach packs consecutive addresses instead of adjacent nodes in stencil geometry, which are away in memory for non-unit stride dimension, it avoids the jump between 4th and 5th order stencils.

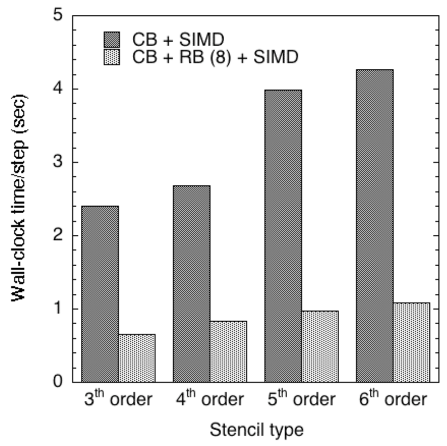

Figure 7. Effects of stencil order.

3.3. Prefetching

The RB method significantly reduces the number of cache misses, but still missed L2 data cache lines pays the penalty of accessing to memory which is application specific and usually around ~150 clock cycles, an order of magnitude more than L2 access penalty. Therefore, we use software prefetching to hide memory latency as the highest level of our memory optimization framework.

We use Intel's `_mm_prefetch` intrinsic to implement fetching data from memory to second level cache. Since prefetch scheduling distance (PSD) is not a well-defined metric, and to achieve better performance, we spread the prefetch instructions inside the instruction sequence of the innermost loop rather than clustering prefetches together and experimentally try variety of PSDs up to 3 full iterations of the innermost loop. Considering high stride access due to the topology of stencil computation, we also combine this by translation lookaside buffer (TLB) priming to preload the page table entry for the z-neighboring grid points, which are 59KB away. This is similar to prefetch, but instead of a data cache line, the page table entry is being loaded in advance of its use to avoid TLB miss.

Figure 4 shows experimental results. Prefetching does not provide significant speedup, which we have also confirmed by micro-architecture level analysis of second-level cache utilization. This is because the innermost loop in stencil kernel is predominately memory bandwidth-bound and already features competing techniques such as CB and loop unrolling. It should also be noted that RB technique improves cache performance and this also contributes in less effectiveness of prefetching the data to L2 cache. Similar results were reported by Kaushik *et al.* for similar memory bound problems [16].

4. Conclusions

In summary, in-core optimization techniques, including CB, RB and prefetching, have been explored on a suite of Intel Xeon architectures using a kernel extracted from a 6th-order stencil based seismic wave propagation code. Modest improvements are seen for implementation of CB and prefetching, due to the slightly decreased number of missed L1 and L2 cache lines. On the contrary, the implementation of RB shows as high as 4.3-fold speed up on Intel Harpertown as a result of the 2.68-fold decrease in L1 line miss rate and 23% decrease in L2 cache line miss rate. We have also quantified the effect of above mentioned optimization techniques for lower order stencil applications on the multicore processors with same complex cache hierarchies. Results reveal that, compared to CB and prefetching, RB can potentially be a more generic optimization technique because its speed up on cache performance is less affected by the order of the application's stencil. Future work will include further micro-architecture level optimization approaches. This work was partially supported by Chevron—CiSoft, NSF, DOE, ARO, and DTRA. Performance tests were carried out at Collaboratory of Advanced Computing and Simulations and High Performance Computing and Communications cluster of the University of Southern California.

References

[1] J. Dongarra *et al.*, "The impact of multicore on computational science software," *CTWatch Quarterly,* vol. 3, pp. 11-17, 2007.

[2] S. Sandeep *et al.*, "Towards a theory of cache-efficient algorithms," *J. ACM,* vol. 49, pp. 828-858, 2002.

[3] S. Williams *et al.*, "Lattice Boltzmann simulation optimization on leading multicore platforms," in *Proceedings of the 22nd International Parallel and Distributed Processing Symposium* Miami, Florida, 2008.

[4] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* Austin, Texas: IEEE Press, 2008.

[5] G. Rivera *et al.*, "Tiling optimizations for 3D scientific computations," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* Dallas, Texas: IEEE Computer Society, 2000.

[6] M. Frigo *et al.*, "Cache oblivious stencil computations," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing* Cambridge, Massachusetts: ACM, 2005.

[7] D. Wonnacott, "Using time skewing to eliminate idle time due to memory bandwidth and network limitations," in *Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium* Cancun, Mexico, 2000.

[8] L. Renganarayanan *et al.*, "Towards optimal multi-level tiling for stencil computations," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium* Long Beach, California, 2007.

[9] L. Peng *et al.*, "High-Order Stencil Computations on Multicore Clusters," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium* Rome, Italy: (to appear), 2009.

[10] "PARKBENCH: PARallel Kernels and BENCHmarks," Available from http://www.netlib.org/parkbench.

[11] S. Kamil *et al.*, "Implicit and explicit optimizations for stencil computations," in *Proceedings of the 2006 Workshop on Memory System Performance and Correctness* San Jose, California: ACM, 2006.

[12] M. Bromley *et al.*, "Fortran at ten gigaflops: the connection machine convolution compiler," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation* Toronto, Canada: ACM, 1991.

[13] F. Shimojo *et al.*, "Divide-and-conquer density functional theory on hierarchical real-space grids: Parallel implementation and applications," *Physical Review B,* vol. 77, 085103, 2008.

[14] A. Stathopoulos *et al.*, "Parallel methods and tools for predicting material properties," *Computing in Science and Engineering,* vol. 2, pp. 19-32, 2000.

[15] S. Williams *et al.*, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing* Reno, Nevada: ACM, 2007.

[16] D. Kaushik *et al.*, "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors," *SIAM Review,* vol. 51, pp. 129-159, 2009.