# Implicit and Explicit Optimizations for Stencil Computations

Shoaib Kamil†, Kaushik Datta‡, Samuel Williams‡,
Leonid Oliker†, John Shalf†, Katherine Yelick†‡

†Lawrence Berkeley National Laboratory
1 Cyclotron Road
Berkeley, CA, 94720
{sakamil,loliker,jshalf,kayelick}@lbl.gov

‡Computer Science Department
University of California
Berkeley, CA, 94720
{kdatta,samw}@cs.berkeley.edu

## ABSTRACT

Stencil-based kernels constitute the core of many scientific applications on block-structured grids. Unfortunately, these codes achieve a low fraction of peak performance, due primarily to the disparity between processor and main memory speeds. We examine several optimizations on both the conventional cache-based memory systems of the Itanium 2, Opteron, and Power5, as well as the heterogeneous multicore design of the Cell processor. The optimizations target cache reuse across stencil sweeps, including both an implicit cache oblivious approach and a cache-aware algorithm blocked to match the cache structure. Finally, we consider stencil computations on a machine with an explicitly-managed memory hierarchy, the Cell processor. Overall, results show that a cache-aware approach is significantly faster than a cache oblivious approach and that the explicitly managed memory on Cell is more efficient: Relative to the Power5, it has almost 2x more memory bandwidth and is 3.7x faster.

## 1. INTRODUCTION

Partial differential equation (PDE) solvers constitute a large fraction of scientific applications in such diverse areas as heat diffusion, electromagnetics, and fluid dynamics. These applications are often implemented using iterative finite-difference techniques, which sweep over a spatial grid, performing nearest neighbor computations called *stencils*. In a stencil operation, each point in a multidimensional grid is updated with weighted contributions from a subset of its neighbors in both time and space— thereby representing the coefficients of the PDE for that data element. These operations are then used to build solvers that range from simple Jacobi iterations to complex multigrid and adaptive mesh refinement methods [2].

Stencil computations perform global sweeps through data structures that are typically much larger than the capacity of available data caches. As a result, stencil computations generally achieve a low fraction of theoretical peak performance, since data from main memory cannot be transferred fast enough to avoid stalling the computational units on modern microprocessors. Cache blocking in the spatial dimension is useful under a very limited set of circumstances [5]. Thus, more contemporary approaches to stencil optimization are geared towards techniques that leverage tiling in both the spatial and temporal dimensions of computation using loop skewing in order to increase data reuse within the cache hierarchy. Initial work by Wolf [12] showed loop skewing generally did not improve performance, but subsequent studies by McCalpin [6] et al and others [10, 13] have shown a modified form of loop skewing called *time skewing* can improve performance for many stencil kernels.

Cache oblivious optimizations optimize algorithms without using cache sizes as a tuning parameter. Such optimizations have been shown to improve performance for some classes of matrix operations [8] including matrix transpose, fast fourier transform, and sorting [3]. More recently, Frigo et al [4] showed the potential of cache oblivious optimizations for improving stencil kernel performance.

In this paper, we examine the *implicit* cache oblivious tiling methodology which promises to efficiently utilize cache resources without the need to consider the details of the underlying cache infrastructure. Next, we compare performance against an *explicit* the cache-aware algorithm known as time skewing, where the blocking factor is carefully tuned based on the stencil size and cache hierarchy details. For both of these approaches we evaluate performance on the Intel Itanium 2, AMD Opteron, and IBM Power5 microprocessors, where data movement to on-chip caches is automatically (implicitly) managed by hardware (or compiler-managed software) control. Our final stencil implementation is written for the non-conventional microarchitectural paradigm of the recently-released STI (Sony/Toshiba/IBM) Cell processor, whose local store memory is managed *explicitly* by software rather than depending on automatic cache management policies implemented in hardware.

A unique contribution of our work is the comparative evaluation of implicit and explicit stencil optimization algorithms, as well as a study of the tradeoffs between implicitly- and explicitly-managed local store memories. Experimental results show that while the cache oblivious algorithm does indeed reduce the number of cache misses compared to the


*MSPC'06* October 22, 2006, San Jose, CA, USA

naïve approach, it can paradoxically degrade absolute performance due primarily to sub-optimal compiler code generation for certain kernels. In addition, our exploration of the cache oblivious algorithm shows that some refinements can improve performance despite resulting in more cache traffic. We also show that although the time skewed algorithm can significantly improve performance, choosing the best blocking approach is non-intuitive, requiring an exhaustive search of tiling sizes or an effective performance model to attain optimal performance. Finally, we demonstrate that explicitly-managed local store architectures offer the opportunity to fully utilize the available memory system and achieve impressive results regardless of the underlying problem size.

# 2. EXPERIMENTAL SETUP

The experiments conducted in this work utilize the Stencil Probe [5], a compact, self-contained serial microbenchmark developed to explore the behavior of stencil computations on block-structured grids without the complexity of full application codes. As such the Stencil Probe is suitable for experimentation on architectures in varying stages of implementation— from production CPUs to cycle-accurate simulators. By modifying the operations in the inner loop of the benchmark, the Stencil Probe can effectively mimic the kernels of applications that use stencils on regular grids. Previous work [5] has shown that the Stencil Probe is an effective proxy for the behavior of larger applications. Thus the Stencil Probe can be used to easily simulate the memory access patterns and performance of large applications, while testing for potential optimizations, without having to port or modify the entire application.

## 2.1 Stencil Application

In this work, we examine the performance of a 3D seven-point heat equation from the Chombo [1] framework. Chombo is a set of tools for computing solutions of partial differential equations using finite difference methods on adaptively-refined meshes. We use the kernel from `heattut`, a demo application that is a simple 3D heat equation solver that does not use Chombo's more advanced capabilities. In general, performing several sweeps through a grid at once is not always possible because many applications perform other work between stencil sweeps. Our sample application does not, however, suffer from this limitation.

## 2.2 Hardware Platforms

Our study examines three leading microprocessor designs used in high performance computing systems: the Itanium 2, the AMD Opteron, and the IBM Power5. Additionally, we examine stencil performance on the recently-released STI Cell processor, which takes a radical departure from conventional multiprocessors. An overview of each platform's architectural characteristics is shown in Table 1.

The 64-bit Itanium 2 system used in our study operates at 1.4 GHz and is capable of issuing two FMAs per cycle for a peak performance of 5.6 GFlop/s. The memory hierarchy consists of 128 FP registers (of which 96 can rotate) and three on-chip data caches (32KB L1, 256KB L2, and 3MB L3). The Itanium 2 cannot store FP data in L1, making register loads and spills potential sources for bottlenecks; however, a relatively large register set helps mitigate this issue. The superscalar processor implements the Explicitly Parallel Instruction set Computing (EPIC) technology where

```
void stencil3d(double current[], double next[],
  int x_n, int y_n, int z_n, int t_n)
{
   for (int t = 0 to t_n)
    for (int x = 1 to x_n - 1)
     for (int y = 1 to y_n - 1)
      for (int z = 1 to z_n - 1)
       X^t_{x,y,z} = a*X^{t-1}_{x,y,z} + b*(X^{t-1}_{x+1,y,z} + X^{t-1}_{x-1,y,z} X^{t-1}_{x,y+1,z}
        + X^{t-1}_{x,y-1,z} + X^{t-1}_{x,y,z+1} + X^{t-1}_{x,y,z-1});
}

void stencil3d_periodic(double current[],
  double next[], int x_n, int y_n, int z_n, int t_n)
{
   for (int t = 0 to t_n)
    for (int x = 0 to x_n)
     for (int y = 0 to y_n)
      for (int z = 0 to z_n)
       X^t_{x%nx,y%ny,z%nz} = a*X^{t-1}_{x%nx,y%ny,z%nz}
        + b*(X^{t-1}_{x+1%nx,y%ny,z%nz} + X^{t-1}_{x-1%nx,y%ny,z%nz}
        + X^{t-1}_{x%nx,y+1%ny,z%nz} + X^{t-1}_{x%nx,y-1%ny,z%nz}
        + X^{t-1}_{x%nx,y%ny,z+1%nz} + X^{t-1}_{x%nx,y%ny,z-1%nz});
}
```

**Figure 1: Pseudocode for the 3D naïve stencil kernel, with non-periodic (top) and periodic (bottom) boundary conditions.**

instructions are organized into 128-bit VLIW bundles.

The primary floating-point horsepower of the 64-bit AMD Opteron comes from its SIMD floating-point unit accessed via the SSE2 or 3DNow instruction set extensions. The Opteron utilizes a 128b SIMD FP multiplier and a 128b SIMD FP adder, both of which are half-pumped. Thus our 2.2 GHz test system can execute two floating-point operations per cycle and deliver peak performance of 4.4 GFlop/s. The L2 cache on our test system is a 1MB victim cache (allocates on evictions from L1). The peak aggregate memory bandwidth is 5.2 Gigabytes/sec (either read or write), supplied by two DDR-266 DRAM channels per CPU.

The latest processor in the IBM Power line, the Power5 processor is a superscalar RISC architecture capable of issuing 2 FMAs per cycle. The 1.9 GHz test system has a 1.9MB on-chip L2 cache as well as a massive 36MB L3 victim cache on the DCM (dual chip module). The peak floating-point performance of our test system is 7.6 GFlop/s. The memory bandwidth is supplied by IBM's proprietary SMI interfaces that aggregate 8 DDR-266 DRAM channels to supply 10 Gigabytes/sec read and 5 Gigabytes/sec write performance (15 GB/s peak aggregate bandwidth) per CPU.

STI's Cell processor is a heterogeneous nine-core architecture that combines considerable floating point resources with a power-efficient software-controlled memory hierarchy. Instead of using identical cooperating commodity processors, Cell uses a conventional high performance PowerPC core that controls eight simple SIMD cores, called synergistic processing elements (SPEs). A key feature of each SPE is the three-level software-controlled memory hierarchy. Instead of transferring data between the 128 registers and DRAM via a cache hierarchy, loads and stores may only access a small (256KB) private local store. The Cell processor utilizes explicit DMA operations to move data from main memory to the local store of the SPE. Dedicated DMA en-

gines allow multiple concurrent DMA loads to run simultaneously with the SIMD execution unit, thereby mitigating memory latency overhead via double-buffered DMA loads and stores. The Cell processor is designed with an extremely high single-precision performance of 25.6 GFlop/s per SPE (204.8 GFlop/s collectively); however, double precision performance lags significantly behind with only 1.8 GFlop/s per SPE (14.6 GFlop/s collectively), for the 3.2 GHz part. The XDR memory interface on Cell supplies 25 GB/s peak aggregate memory bandwidth. Thus for Cell, double-precision performance— not DRAM bandwidth— is generally the limiting factor.

| | Itanium2 | Opteron | Power5 | Cell SPE |
|---|---|---|---|---|
| Architecture | VLIW | super scalar | super scalar | dual SIMD |
| Frequency (GHz) | 1.4 | 2.2 | 1.9 | 3.2 |
| Peak (GFlop/s) | 5.6 | 4.4 | 7.6 | 1.83 |
| DRAM (GB/s) | 6.4 | 5.2 | 15* | 25.6 |
| FP Registers | 128 | 16 | 32 | 128 |
| (renamed/rotating) | 96 | 88 | 120 | 0 |
| Local Mem (KB) | N/A | N/A | N/A | 256 |
| L1 D$ (KB) | 32 | 64 | 64 | N/A |
| L2 D$ (KB) | 256 | 1024 | 1920 | N/A |
| L3 D$ (MB) | 3 | N/A | 36 | N/A |
| Introduction | 2003 | 2004 | 2004 | 2006 |
| Cores Used | 1 | 1 | 1 | 8 |
| Compiler Used | Intel 9.0 | Pathscale | XLC | XLC |

**Table 1: Overview of architectural characteristics.**

## 2.3 Performance Calculation Methodology

On all three conventional systems, we used the Performance API (PAPI) library [7] to measure cache misses at the various levels of the cache hierarchy. PAPI enables us to use a standard cross-platform library to access performance counters on each CPU. Unfortunately, on the Power5 and Opteron platforms, cache miss counters do not include prefetched cache lines, thus preventing cache miss counters from accurately reflecting overall memory traffic. Therefore, we generally only show Itanium 2 cache miss numbers. Memory traffic is calculated as the product of cache misses and cache line size. However, on Cell, as all memory traffic is explicit in the code, it can be computed directly. On the Cell platform, both the SPE decrementers and PowerPC timebase are used to calculate elapsed time, while on the conventional machines, PAPI is used to access cycle timers. Performance, as measured in GFlop/s, is calculated directly based on eight flops per stencil, and one stencil per time step for every point excluding the boundary (if present).

# 3. NAÏVE IMPLEMENTATION

Code for the 3D naïve periodic and non-periodic versions are given in Figure 1. The non-periodic code is straightforward, but the periodic version is more complicated. In the actual periodic implementation, several layers of ghost cells were created. Instead of updating ghost cells after every iteration, this allowed us to update all the ghost cells before doing any computation. The extra ghost cells did introduce slightly more computation and memory traffic, but this was still faster than the alternative. In addition, without this optimization, our cache-aware algorithm could not be implemented.

*Total bandwidth of 15 GB/s (10 GB/s load, 5 GB/s store).

| | Non-Periodic | | Periodic | |
|---|---|---|---|---|
| | Comp. Rate (GFlops/s) | % of Algor. Peak | Comp. Rate (GFlops/s) | % of Algor. Peak |
| Itanium 2 | 1.30 | 41 | 0.83 | 26 |
| Opteron | 0.49 | 17 | 0.51 | 17 |
| Power5 | 1.97 | 45 | 0.99 | 23 |

**Table 2: Performance of non-aliased naïve stencil code on the three cache-based architectures for a $256^3$ problem.**

The stencil from this uses Jacobi iterations, so it is not in-place. Thus each of the algorithms presented alternates the source and target arrays after each iteration.

Table 2 shows the performance of the naïve stencil algorithm on the three commodity architectures. The Opteron platforms achieves a low percentage ($< 20\%$) of algorithmic peak, while the other two cannot achieve even half of their algorithmic peak. This serves as our motivation.

# 4. IMPLICITLY BLOCKED: CACHE OBLIVIOUS ALGORITHM

There are limited opportunities for cache reuse in stencil computations when relying exclusively on spatial tiling because each point is used a very small number of times. Tiling in both the spatial and temporal dimensions opens up additional opportunities for cache reuse for stencil-based applications that allow multiple timesteps (or sweeps) to be computed simultaneously. The cache oblivious stencil algorithm [4] further leverages the idea of combining temporal and spatial blocking by organizing the computation in a manner that doesn't require any explicit information about the cache hierarchy. The algorithm considers an $(n+1)$-dimensional *spacetime trapezoid* consisting of the $n$-dimensional spatial grid together with an additional dimension in the time (or sweep) direction. We briefly outline the recursive algorithm below; details can be found in [4].

Consider the simplest case, where a two-dimensional spacetime region is composed of a one-dimensional space component (from $x0$ to $x1$) and a dimension of time (from $t0$ to $t1$) as shown in Figure 2(a). This trapezoid shows the traversal of spacetime in an order that respects the data dependencies imposed by the stencil, (i.e. which points can be validly calculated without violating the data dependencies in spatial and temporal dimensions).

In order to recursively operate on smaller spacetime trapezoids, we cut an existing trapezoid either in time or in space and then recursively call the cache oblivious stencil function to operate on the two smaller trapezoids. Figure 2(b) demonstrates an example of a space cut. Note that since the stencil spacetime trapezoid itself has a slope ($dx0$ and $dx1$), we must preserve these dependencies when performing a space cut, as demonstrated in Figure 2(b). The two newly-created trapezoids, $T1$ and $T2$, can now be further cut in a recursive fashion. In addition, note that no point in the stencil computation of $T1$ depends on a point in $T2$, allowing $T1$ to be completely calculated before processing $T2$.

Similarly, a recursive cut can also be taken in the time

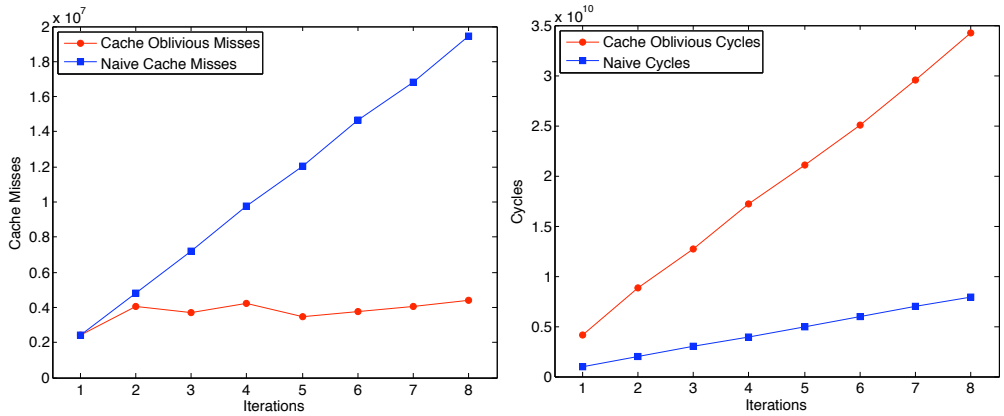



**Figure 3: Performance of the initial cache oblivious implementation for a $256^3$ periodic problem on our Itanium 2 test system. The algorithm reduces cache misses but performs worse.**

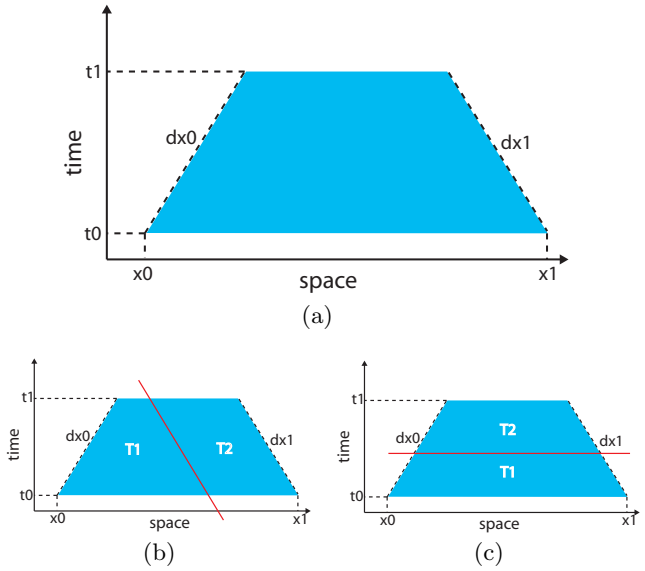



**Figure 2: (a) 2D trapezoid space-time region consisting of a 1D space component and 1D time component, and an example of cache oblivious recursive (b) space cut and (c) time cut.**

dimension, as show in Figure 2(c). Because the time dependencies are simpler, the cut divides the time region $(t0, t1)$ into $(t0, tn)$ and $(tn, t1)$ regions which are then operated on recursively. Again, recall that no point in the $T1$ computational domain depends on a point in $T2$. Note, however, that cutting in time does not in itself improve cache behavior; instead, it allows the algorithm to continue cutting in the space dimension by creating two trapezoids that are shaped amenably for space cutting. The recursion calls the function on smaller and smaller trapezoids until there is only one timestep in the calculation, which is done in the usual fashion (using a loop from $x0$ to $x1$). The multidimensional algorithm is similar, but attempts to cut in each space dimension before cutting in time.

## 4.1 Periodic Performance

First, we compare performance between the implementation of the cache oblivious code given in [4] and the naïve non-recursive version (consisting of four simple loops), using the 3D heat equation within the Stencil Probe as described in Section 2, both using periodic boundaries, because the cache oblivious algorithm as originally designed uses periodic boundaries. Figure 3 shows the raw performance (in cycles) and measured cache misses for a $256^3$ problem on our Itanium 2 test system.

Observe that the runtime of the cache oblivious approach is substantially poorer than that of the naïve algorithm (this is actually the case on all three cache-based platforms). However, the cache oblivious approach is indeed effective in dramatically improving cache efficiency compared with the naïve implementation, as can be seen in the measured number of cache misses in Figure 3. In fact, the cache miss model developed in [4] accurately predicts the volume of misses measured for the cache oblivious algorithm on the Itanium 2, while the volume of misses for the naïve version is exactly what is to be expected from the simple algorithm. It is therefore critical to gain insight into the seemingly contradictory trend of improving caching efficiency and worsening performance, as much algorithmic effort has been invested over several decades to improve program performance by reducing cache misses.

In order to understand the performance potential of the cache oblivious methodology, we explore a series of optimizations, building on those that successfully result in reduced time-to-solution:

| Optimization | Speedup over Naïve | | |
|---|---|---|---|
| | IA64 | AMD64 | Pwr5 |
| Original | 0.26 | 0.13 | 0.30 |
| Inline Kernel | 0.50 | 0.14 | 0.61 |
| Inline Kernel + Explicit Stack | 0.46 | 0.14 | 0.57 |
| Inline Kernel + Early Cutoff | 1.23 | 0.19 | 0.96 |
| Inline Kernel + No Modulo | 1.23 | 0.68 | 2.00 |
| Inline Kernel + Early Cutoff + No Modulo | 1.52 | 1.25 | 3.85 |
| Inline Kernel + Early Cutoff + No Modulo + Preserve Stride-1 | 1.67 | 1.56 | 4.17 |
| All Opts + Exhaustive Cutoff | 1.69 | 1.59 | 4.17 |

**Table 3: Summary of all attempted optimizations.**

- Explicit inlining of the kernel. The original cache oblivious algorithm in [4] performed a function call per point. Instead, we inline the function.
- Using an explicit stack instead of recursion. Because

the algorithm is not tail-recursive, we cannot completely eliminate recursion. Instead, we attempted to explicitly push and pop parameters on a user-controlled stack in place of recursion. However, this did not yield a speedup on any of our test platforms.

- Cut off recursion early. Instead of recurring down to a single timestep, we stop the recursion when the volume of the 3D trapezoid reaches an arbitrary value. This optimization results in somewhat greater memory traffic when compared to the original cache oblivious algorithm yet decreases overall runtime.
- Use indirection instead of modulo. We replaced the modulo in the original algorithm with a lookup into a preallocated table to obtain indices into the grid.
- Never cut in unit-stride dimension. Previous work [5] showed that long unit-stride accesses were important in achieving good performance. We preserve the long unit-stride accesses by not cutting in space in the unit-stride dimension. Although this raised total memory traffic, it substantially improved overall performance.

Table 3 shows a comparative summary between naïve aliased and cache oblivious performance for the periodic boundary condition test case. Using the best set of optimizations, we observe a speedup relative to the naïve version of 1.69 on the Itanium 2 system, and speedups of 1.58 and 4.2 for the Opteron and Power5 machines, respectively. This demonstrates that the cache oblivious approach can indeed outperform the naïve computation for this kernel on a problem with periodic boundaries through careful code design.

## 4.2 Non-Periodic Performance

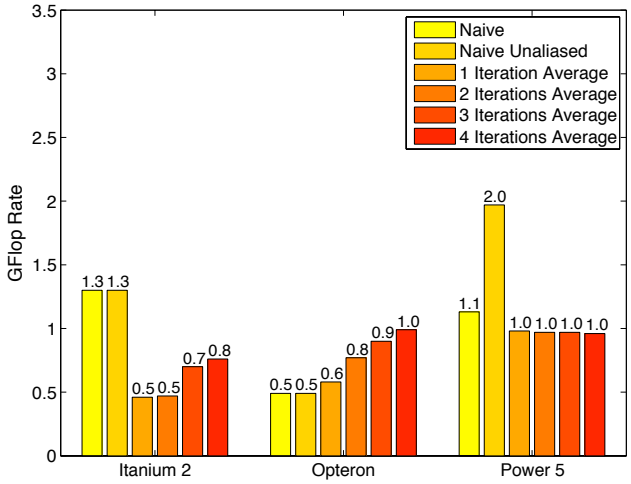



**Figure 4: Performance of non-periodic cache oblivious implementation. Note that this chart shows *average* performance over four iterations.**

Thus far, we have focused on stencil computations with periodicity in the boundary; however, most computational science codes utilize non-periodic (or constant) boundary conditions. Although the cache oblivious stencil algorithm is better suited for periodic conditions, the algorithm can be converted to use constant boundaries by setting the initial slopes of the grid edges grid to zero (thus ensuring the timespace trapezoid is really a rectangle) and utilizing ghost cells to ensure that the stencil does not access areas outside of the grid.

| Problem Size | Speedup over Naïve | |
|---|---|---|
| | Memory Read Traffic | Computation Rate |
| $128^3$ | 0.25 | 0.70 |
| $256^3$ | 0.41 | 0.55 |
| $512^3$ | 0.12 | 0.59 |

**Table 4: Cache oblivious performance for four iterations of varying problem sizes on the Itanium 2, with non-periodic boundary conditions. Despite large reductions in cache misses, the cache oblivious algorithm performs up to 45% slower.**

A summary of performance for one to four iterations using constant boundaries is shown in Figure 4. Note that the non-periodic naïve algorithm is quite a bit faster than the periodic version of the naïve algorithm on some architectures. We attempted all the same optimizations used in the non-periodic version and used the best-case performance.

In addition, the graph shows two different implementations of the naïve stencil code. This is because the Power5 platform shows drastically different computation rates depending on whether the source and target arrays are aliased or not. If they are unaliased, then the code achieves 2.0 GFlops/s. Otherwise, the code runs at 1.1 GFlops/s, likely due to the xlc compiler's inability to infer aliasing. We attempted to improve performance with a no-alias directive, but it had no effect. IBM engineers are currently investigating this problem.

Observe that (as expected) on the Opteron and Power5 platforms the cache oblivious and aliased naïve implementations show similar performance for a single iteration since the cache oblivious approach essentially executes the same code as the naïve case when there is a single iteration. However, on the Itanium 2, the compiler-generated code for the cache oblivious case performs poorly compared with the naïve version (about a third of the speed). This is apparent in Figure 4, which shows that one iteration of cache oblivious and one iteration of the naïve stencil have vastly different performance on the Itanium 2, although they essentially execute the same source code[†].

As a result, the overall performance of the non-periodic cache oblivious implementation is much worse than the naïve case on the Itanium 2, at best achieving only 55% of naïve performance at four iterations, despite reducing the overall main memory read traffic substantially. On the Opteron, however, we see that the cache oblivious implementation outperforms the naïve implementation, achieving double the performance. Lastly, on our Power5 test system, the cache oblivious version of the code performs slightly slower than the naïve version probably due to unavoidable aliasing issues.

# 5. EXPLICITLY BLOCKED: TIME SKEWING ALGORITHM

Unlike the cache oblivious algorithm, the time skewing algorithm [6, 10, 13] uses *explicit* space cuts, requiring the

[†]The two versions calculate loop bounds slightly differently.

user to specify a cache block size. In the absence of a performance model, we typically do not know which block size will execute fastest. Therefore, for each platform where time skewing is run, we perform a search to determine the optimal block size. While the cache block's x- and y-dimensions (both non-contiguous in memory) are allowed to vary, the z-dimension (the unit stride dimension) is left uncut to allow for longer unit-stride memory streams.

## 5.1 Algorithm Description

Time skewing is a type of cache tiling, that attempts reduce main memory traffic by reusing values in cache as often as possible. Figure 5 shows a simplified diagram of time skewing for a 3-point stencil. The grid is divided into cache blocks by several skewed cuts, similar to the space cuts from the cache oblivious algorithm (see Figure 2(b)). These cuts are skewed in order to preserve the data dependencies of the stencil. For example, the cut between the first and second cache blocks allows the first cache block to be fully calculated before starting on the second cache block. In general, this holds true between the $n^{th}$ and $(n+1)^{th}$ cache blocks. As long as the blocks are executed in the proper order, the algorithm respects the stencil dependencies.

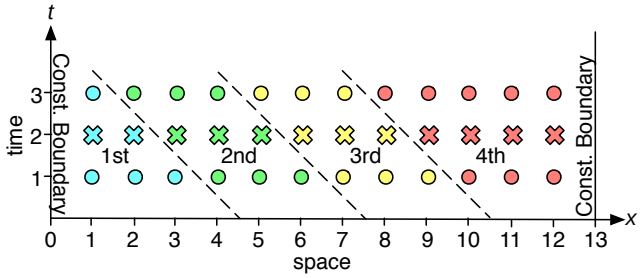



**Figure 5: A simplified two-dimensional spacetime diagram of time skewing with a 3-point stencil. The cache blocks need to be executed in the order shown to preserve dependencies. The X's and O's indicate which of two arrays is being written to.**

However, the blocks generated from time skewing do not all perform the same amount of work, despite being equally partitioned initially. For instance, the points in Figure 5 are equally divided for the first time step. However, as time progresses, the shifting causes the cache blocks at the boundaries to perform unequal work. The number of points per iteration slowly decreases for the first cache block, while it slowly increases for the final cache block. For interior cache blocks, the shifting does not change the number of points per iteration, and so they all perform the same number of stencil operations.

There are two major points of concern caused by this shifting. The first is that extra cache misses may be incurred, thereby hindering our efforts to minimize memory traffic. Fortunately, this shift is always towards the completed portion of the grid, so the needed points are often already resident in cache. This helps in mitigating, if not eliminating, the extra memory traffic.

The second concern is that the shifting limits the number of iterations that can be performed. Specifically, some of the cache blocks along the boundary can be shifted off the grid as time progresses. Once a cache block is off the grid, any further iterations will cause dependency violations. This is seen in Figure 5, where the first cache block shifts completely over the boundary after the third iteration. In these cases, we can perform a time cut (as explained in Figure 2(c)) to "restart" the algorithm. After the time cut, we can either execute the remaining number of iterations or, if needed, perform another time cut. Of course, this problem can also be addressed by simply using a larger cache block.

A closer representation to our actual 3D time skewing code is illustrated in Figure 6. By showing how the number of stencil operations performed varies within each cache block, the diagram sheds light on how time skewing works in higher dimensions.

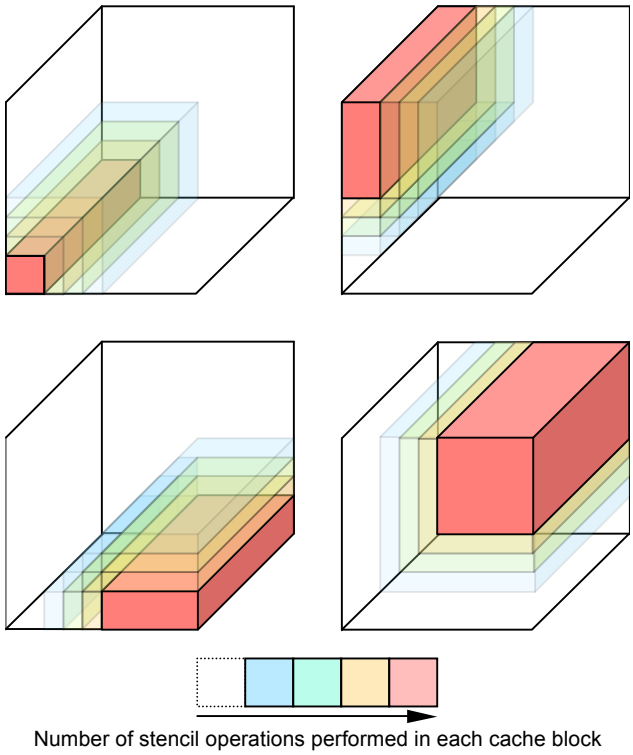
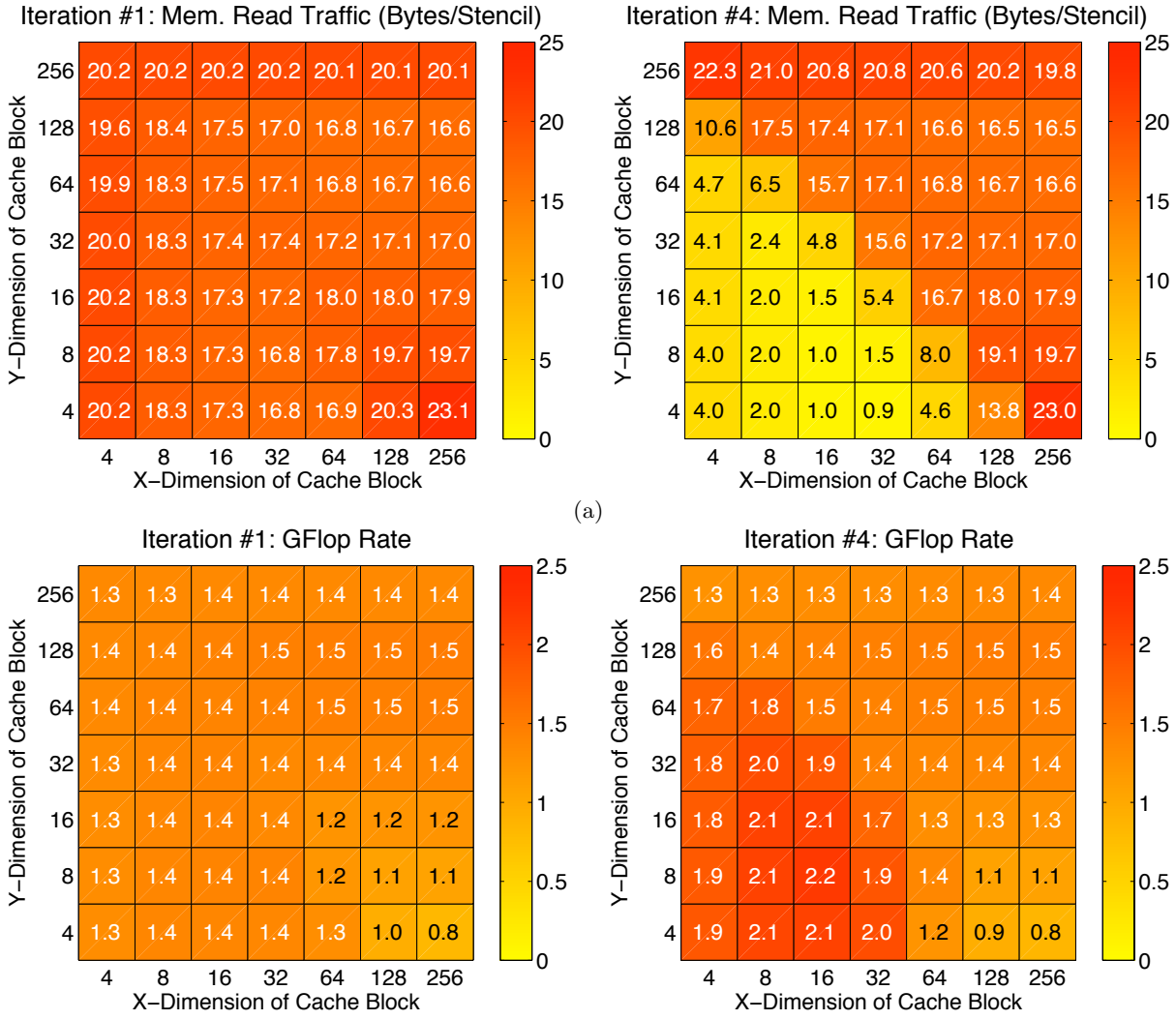



**Figure 6: Color coded plots of the number of stencils operations performed on a $10^3$ grid using four iteration time skewing with 5x5x10 cache blocks. There is one plot for each cache block. Blue halos represent only a single stencil operation for that region, where red blocks show the cores where the full four stencils operations were performed. When processed in order, the full $10^3$ has completed four iterations—i.e. a blue cell in four different cache blocks implies one stencil performed in each cache block or four total.**

## 5.2 Performance

We first verified, on our Itanium 2 test machine, that per-iteration memory traffic does in fact decrease with more iterations. Figure 7(a) confirms that for small block sizes, overall memory traffic decreases drastically from the first iteration to the fourth. More importantly, during the fourth iteration the memory traffic for the smaller cache blocks is much lower than for the naïve case (the upper right corner of the graph). Assuming the code is memory-bound, this suggests that some of these block sizes will have lower running times than the naïve case.




(b)

**Figure 7: A search to find the optimal cache block size using time skewing. Each cache block's z-dimension (contiguous in memory) is uncut. The graphs show (a) main memory read traffic and (b) GFlop rates on the Itanium 2 for a $256^3$ problem with constant boundaries. The graphs on the left show first iteration data, while the right graphs show data for the fourth iteration.**

Figure 7(b) shows that this is indeed the case. The fourth iteration exhibits speedups of up to 60% over the naïve code. Not surprisingly, the block sizes with the largest reductions in memory traffic also showed the greatest improvements in performance.

Table 5 shows how well time skewing performs for other problem sizes on the Itanium 2. The general trend is that the computational speedups are not as large as the decreases in memory read traffic. This is because the problem has now shifted from being memory bound to being computation bound. At this point, further reductions in memory traffic are no longer useful. However, the overall speedups are still substantial. The computational speedup is particularly dramatic in the $512^3$ case, since the naïve case is especially slow at this problem size. The problem is large enough so that three planes of the source array and one plane of the target array cannot fit into L3 cache (see [5]). Thus, the same point in the source array needs to be brought into cache several times during a single iteration. This is very expensive.

Time skewing addresses this problem by working with a single cache block at a time. This effectively shrinks the size of each plane, allowing all the iterations for a point to be completed after bringing it into cache only once. The result is a drastic drop in memory traffic (84%) and consequently a large speedup in performance (1.67).

| Problem Size | Best Block Size | Speedup over Naïve | |
|---|---|---|---|
| | | Memory Read Traffic | Computation Rate |
| $128^3$ | 4x4x128 | 0.29 | 1.33 |
| $256^3$ | 16x8x256 | 0.26 | 1.27 |
| $512^3$ | 16x4x512 | 0.16 | 1.67 |

**Table 5: Time skewing for four iterations of varying problem sizes on the Itanium 2.**

Figure 8 shows the GFlop rates for the Opteron and Power5 in addition to the Itanium 2. The data shown is for the best block size on each platform, which was determined by the fastest running time for four iterations. As expected, the graph indicates that for the Itanium 2 and Opteron, time

skewing produces a significant speedup over the naïve code during later iterations.

However, compared to the aliased naïve code, time skewing does not have any impact on the Power5. This is because the time skewing code also does array aliasing. As explained earlier, the xlc compiler generates conservative code when the arrays are aliased, so both codes are limited by the compiler. By removing all array aliasing, the naïve stencil code speeds up from 1.1 to 2.0 GFlops/s. We also attempted to remove all aliasing from the time skewing code by inlining the method calls, but in this case it did not help; the code still ran at about 1.0 GFlops/s. We cannot fully explain this– perhaps the xlc compiler is unable to optimize the extra loops involved in time skewing.

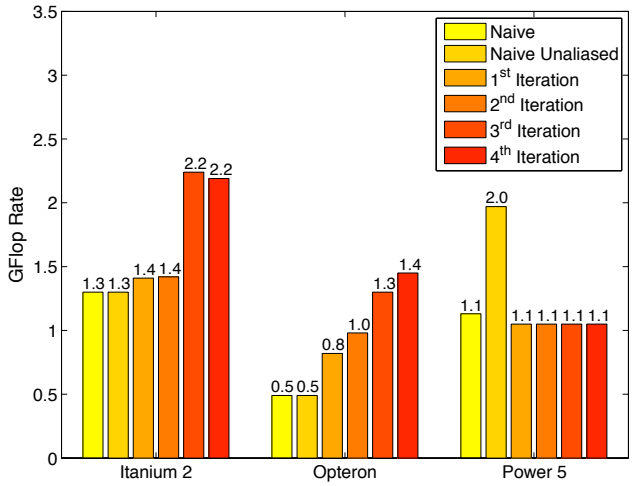



**Figure 8: GFlop rates for time skewing, where each platform's best block size was determined by the fastest running time for four iterations. This is a $256^3$ problem with constant boundaries. Note that this chart shows performance per iteration, not average over all iterations.**

# 6. EXPLICITLY BLOCKED: SOFTWARE MANAGED MEMORY

Before implementing this stencil on a Cell SPE, we began by examining some of the algorithmic limitations. First, aggregate memory bandwidth for the Cell processor is an astounding 25.6 GB/s. As each stencil operation requires at least 8 bytes to be loaded and 8 bytes stored from DRAM, we can expect that performance will be limited to at most 12.8 GFlop/s regardless of frequency. Second, we note that double precision performance is fairly weak. Each adjacent pair of stencil operations (16 flops) will require 7 SIMD floating point instructions, each of which stalls the SPE for 7 cycles. Thus peak performance per SPE will never surpass 1.04 GFlop/s @ 3.2 GHz. With only 8 SPEs (8.36 GFlop/s), it will not be possible to fully utilize memory bandwidth, and thus Cell, in double precision, will be heavily computationally bound performing only a single iteration. Thus, there is no benefit in time skewing in double precision on a single Cell chip at even 3.2 GHz. It should be noted that in single precision, the opposite is true. The 14x increase in computational performance overwhelms the benefit of a 2x decrease in memory traffic.

## 6.1 Local Store Blocking

Any well-performing implementation on a cacheless architecture must be blocked for the local store size. This paper implements a more generalized version of the blocking presented in [11]. In this case, six blocked planes must be stored simultaneously within a single SPE's local store. Figure 9 presents a visualization of cache blocking and plane streaming. As with the previous implementations discussed in this paper, we chose not to cut in the unit-stride direction, and thus preserved long contiguous streams. A simple algebraic relationship allows us to determine the maximum dimensions of a local store block:

$$8bytes*6planes*(ZDimension+2)*(BlockSize+2) < 224KB$$

For example, if the unit-stride dimension were 254, then the maximum block size would be 16, and each plane including ghost zones would be 256x18. We found that on Cell, performance is most consistent and predictable if the unit stride dimension plus ghost zones are a multiple of 16.

## 6.2 Register Blocking

For each phase, the stencil operation must be performed on every point in the current local store block. Instead of processing the plane in "pencils", we process it in "ribbons" where the ribbon width can easily hide any functional unit latency. As Cell is heavily computationally bound, it is imperative that the inner kernel be as fast as possible. As such we implemented it using SIMD instrinsics. This constituted about 150 lines for a software pipelined four wide ribbon that is extruded in the unit stride dimension two elements (for SIMDization) at a time. The resultant code requires about 56 cycles per pair of points. Although this may sound excessive, we must remember the 49 stall cycles consumed by double precision instructions. Thus each pair of points only sees 7 cycles of overhead. It should be noted that for optimal performance, register blocking necessitates that the y-dimension of the grid be divisible by four and the unit stride dimension be even— neither of which is unreasonable.

## 6.3 Parallelization

Using the threaded approach to parallelization, we observe that each local store block is completely independent and presents no hazards aside from those between time steps. Thus assigning batches of local store blocks to SPEs allows for very simple and efficient parallelization on this architecture. If, however, the selected maximum block dimension leaves one or more SPEs heavily or lightly loaded, the code will attempt to select the smallest block size (a single ribbon) in the hope that this will better load balance the machine. Thus for best performance, the y-dimension of the grid should be divisible by four times the number of SPEs the code is run on.

## 6.4 Performance

As unit stride dimension grows, the maximum local store block width shrinks. However an inter-block ghost zone must be maintained. As such the ratio of bytes transferred to stencils performed can increase significantly. Conversely, it should also be noted that an explicitly managed memory allows for the elimination of cache misses associated with writing to the target grid— i.e. one less double must be loaded for each stencil operation. Cell performance is de-

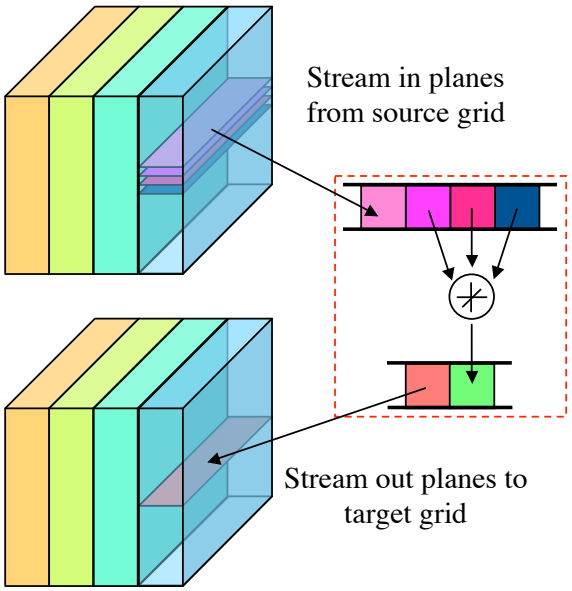



**Figure 9: Cell's blocking strategy is designed to facilitate parallelization, as such a single domain is blocked to fit in the local store and have no intra-iteration dependencies. Planes are then streamed into a queue containing the current time step, processed, written to a queue for the next time step, and streamed back to DRAM.**

tailed in Table 6. It was clear that Cell is heavily computationally bound performing just one iteration at a time, and the potential impact of inefficient blocking was completely hidden by the significantly improved memory efficiency and vastly improved memory bandwidth. We were able to run on both a 2.4 GHz machine and a 3.2 GHz machine and show nearly linear scaling that reinforces our assertion of being computationally bound. It should be noted that at 3.2 GHz, each tiny, low power SPE delivers 0.92 GFlop/s, which compares very favorably to the far larger, and power hungry, Power5.

| Problem size | GFlop/s @2.4GHz | GFlop/s @3.2GHz | Read memory traffic per stencil (in bytes) |
|---|---|---|---|
| 126x128x128 | 5.36 | 6.94 | 9.29 |
| 254x256x256 | 5.47 | 7.35 | 9.14 |
| 510x512x64* | 5.43 | N/A | 12.42 |

**Table 6: Performance characteristics using 8 SPEs. *There was insufficient memory on the prototype blade to run the full problem, however performance remains consistent on the simulator.**

## 6.5 Time Skewing

Although Cell is currently computationally bound in double precision, it clearly is not in single precision. A 4 step time skewed version similar to the blocking algorithm developed by Sellappa and Chatterjee [9] was demonstrated in [11]. Unlike the time skewing implementation described earlier in this paper, the version on Cell was simplified to allow for parallelization. In the 1D conceptualization, the Cell version overlaps trapezoids, where the optimized version utilizes non-overlapping parallelograms. This is less efficient as work is duplicated. Nevertheless, Cell delivers an impressive 49.1 GFlop/s @ 2.4 GHz and a truly astounding 65.8 GFlop/s @ 3.2 GHz for single precision stencils.

Cell blades are a two chip (16 SPE) NUMA. Each chip may access DRAM directly attached to it at 25.6 GB/s (51.2 combined), but are connected to each other via a substantially slower I/O bus. Thus if memory affinity cannot be guaranteed (i.e. a single thread per blade), effective memory bandwidth will plummet to the point where it is the bottleneck. This however presents the opportunity to perform perhaps two steps of time skewing and fully utilize the blade. This is an area for future research.

# 7. CONCLUSION

We explored a combination of software optimizations and hardware features to improve the performance of stencil computations that form the core of many scientific applications. The optimizations include cache oblivious algorithms and (cache-aware) time skewed optimizations, both of which improve cache reuse by merging together multiple sweeps over a grid, thereby enabling multiple iterations of the stencil to be performed on each cache-resident portion of the grid. These optimizations may be used on blocked iterative algorithms and other settings where there is no other computation between stencil sweeps.

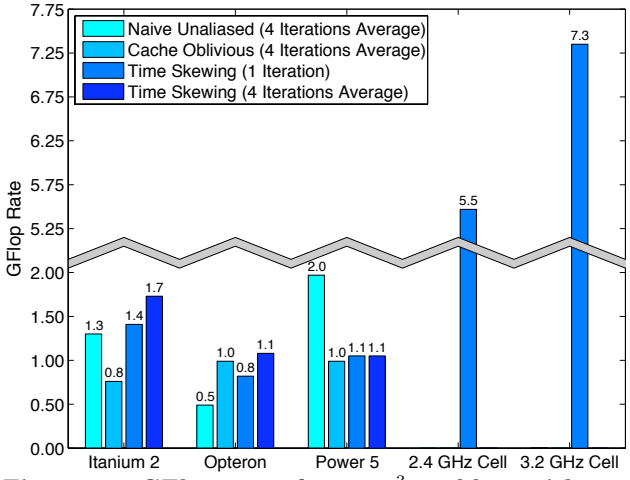



**Figure 10: GFlop rates for a $256^3$ problem with constant boundaries. The "Time Skewing (1 Iteration)" data is essentially space-only cache blocking.**

| Stencil Version | Read Memory Traffic Per Stencil (bytes) |
|---|---|
| Naïve | 20.0 |
| Cache Oblivious (4 Iter) | 8.21 |
| Time Skewed (1 Iter) | 17.28 |
| Time Skewed (4 Iter) | 5.14 |
| Cell | 9.14 |

**Table 7: Total main memory traffic per point for naïve, cache oblivous, and time skewing on the Itanium 2 as well as for Cell for a $256^3$ problem.**

A summary of our results is presented in Figure 10. The results confirm that the cache oblivious approach using non-periodic boundaries is only effective at improving performance on the Opteron. The poor results are partly due to the compiler's inability to generate optimized code for the complex loop structures required by the cache oblivious implementation. The performance problems remain despite several layers of optimization, which include techniques to

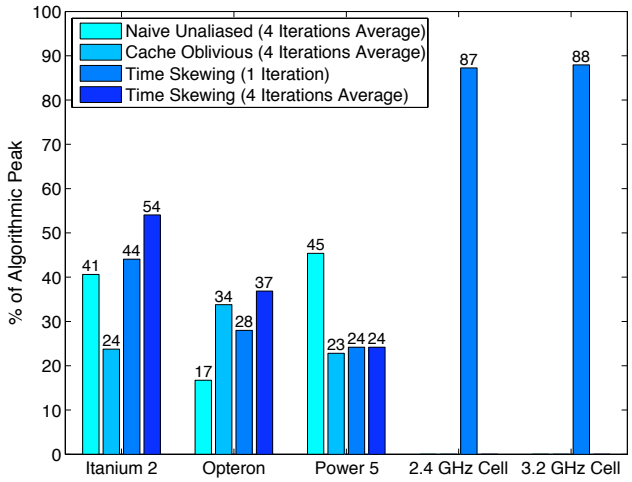



**Figure 11: Percentage of algorithmic peak for a $256^3$ problem with constant boundaries. The "Time Skewing (1 Iteration)" data is essentially space-only cache blocking.**

reduce function call overhead, eliminate modulo operations for periodic boundaries, take advantage of prefetching, and terminate recursion early. Cache-aware algorithms that are explicitly blocked to match the hardware are more effective. Within a single iteration (time skewed with 1 iteration), blocking is effective if three planes of the problem do not fit in the cache. Time skewing with multiple iterations, which can be directly compared to cache oblivious, produces better performance overall, although on the Power5 neither algorithm improves performance over the unaliased naïve algorithm. Much of this is due to the xlc compiler's inability to infer array aliasing. With multiple iterations, time skewing is usually effective whenever the total problem size exceeds that of the cache. However, no speedup is seen on the Power5, again due to conservative code generation by the compiler.

Overall, our results indicate a surprising lack of correlation between main memory traffic and wallclock run time. Although the cache oblivious stencil algorithm reduces misses (as seen in Table 7), it does not generally improve the run time for non-periodic problems. Furthermore, some of the lower-level optimizations we implemented, such as never cutting the unit-stride dimension, increase memory traffic but actually improve the time to solution. These optimization can prove effective because they make better use of automatic hardware and software prefetch, which has proven just as important to optimizing memory performance as cache locality on cache-based systems.

The most striking results in Figure 10 are for the Cell processor. Cell has a higher off-chip bandwidth than the cache-based microprocessors (nearly 2x compared to Power5), although Cell cannot take full advantage of that bandwidth due to the handicapped double precision performance of the chip. Still, the explicit management of memory through DMA operations on Cell proves to be a very efficient mechanism for optimizing memory performance. For example, code that is written to explicitly manage all of its data movement can eliminate redundant memory traffic due to cache misses for stores. The performance of Cell relative to the other systems is up to 7x faster and is limited by floating point speed rather than bandwidth. In terms of percentage of algorithmic peak, Cell approaches an incredible 90% of peak, as shown in Figure 11, while the best set of optimizations on the cache-based architectures are only able to achieve 54% of algorithmic peak. Thus Cell's improved performance is not just a result of higher peak memory bandwidth, but is also due to the explicit control the programmer has over memory access as well as explicit SIMDIzation via intrinsics.

Future work will focus on developing predictive performance models for the optimization strategies examined in our study. These models will help us gain a deeper understanding of the observed performance behavior and allow us to analytically derive optimal blocking strategies for a given problem size and architectural specification.

## 8. ACKNOWLEDGMENTS

We would like to thank Nehal Desai from Los Alamos National Labs for running our Cell code on their prototype 2.4 GHz machine. We would also like to thank Otto Wohlmouth from IBM Germany for running our code on the new 3.2 GHz Cell machines.

## 9. REFERENCES

[1] Applied Numerical Algorithms Group (ANAG), Lawrence Berkeley National Laboratory, Berkeley, CA. Chombo website. `http://seesar.lbl.gov/ANAG/software.html`.

[2] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[3] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms (extended abstract).

[4] M. Frigo and V. Strumpen. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *Proc. of the 19th ACM International Conference on Supercomputing (ICS05)*, Boston, MA, 2005.

[5] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *3rd Annual ACM SIGPLAN Workshop on Memory Systems Performance*, Chicago,IL, 2005.

[6] J. McCalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Department of Computer Science, Rugers University, 1999.

[7] Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/`.

[8] H. Prokop. Cache-oblivious algorithms, June 1999. Master's thesis, MIT Department of Electrical Engineering and Computer Science.

[9] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. *International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

[10] 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.

[11] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing Frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

[12] M. E. Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Stanford University, Stanford, CA, USA, 1992.

[13] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *IPDPS:Interational Conference on Parallel and Distributed Computing Systems*, Cancun, Mexico, 2000.