OPTIMIZATION AND PERFORMANCE MODELING OF STENCIL COMPUTATIONS ON MODERN MICROPROCESSORS[‡]

KAUSHIK DATTA[†], SHOAIB KAMIL[*†], SAMUEL WILLIAMS[*†],
LEONID OLIKER[*], JOHN SHALF[*], KATHERINE YELICK[*†]

Abstract. Stencil-based kernels constitute the core of many important 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. In this paper, we explore the impact of trends in memory subsystems on a variety of stencil optimization techniques and develop performance models to analytically guide our optimizations. Our work targets cache reuse methodologies across single and multiple stencil sweeps, examining cache-aware algorithms as well as cache-oblivious techniques on the Intel Itanium2, AMD Opteron, and IBM Power5. Additionally, we consider stencil computations on the heterogeneous multi-core design of the Cell processor, a machine with an explicitly-managed memory hierarchy. Overall our work represents one of the most extensive analyses of stencil optimizations and performance modeling to date. Results demonstrate that recent trends in memory system organization have reduced the efficacy of traditional cache-blocking optimizations. We also show that a cache-aware implementation is significantly faster than a cache-oblivious approach, while the explicitly managed memory on Cell enables the highest overall efficiency: Cell attains 88% of algorithmic peak while the best competing cache-based processor only achieves 54% of algorithmic peak performance.

Key words. Stencil computations, cache blocking, time-skewing, cache-oblivious algorithms, performance modeling, performance evaluation, Intel Itanium2, AMD Opteron, IBM Power5, STI Cell.

AMS subject classifications. 65Y10, 65Yxx, 35R99, 68M20

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 [3].

Stencil calculations perform global sweeps through data structures that are typically much larger than the capacity of the available data caches. As a result, these 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. Reorganizing these computations to take full advantage of memory hierarchies has been the subject of much investigation over the years. These have principally focused on tiling optimizations [11, 16, 17] that attempt to exploit locality by performing operations on cache-sized blocks of data before moving on to the next block. Whereas many tiling optimizations use domain decomposition to improve spatial locality, more recent studies have focused attention on exploiting locality in the time dimension [6, 13, 19, 24].

In this work, we re-examine stencil computations on current microprocessors in light of the growing performance gap between processors and memory, as well as

[‡]Preliminary versions of this article appeared in [8, 9].

[*]CRD/NERSC, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA, 94720.

[†]Computer Science Department, University of California, Berkeley, CA, 94720.

the techniques hardware designers employ to mitigate this problem, including automatic prefetch, large on-chip caches, and explicitly controlled local-store memories. Through a combination of techniques, including the use of targeted benchmarks, a parameterized probe, and analytical modeling, we revisit previously successful optimizations and explain their effectiveness (or lack thereof) on the current generation of microprocessors for three dimensional PDE problems.

First, we examine stencil optimizations across a single iteration — where cache blocking can only be performed in the spatial dimension — and demonstrate that this approach is useful under a very limited set of circumstances on modern microprocessors. Our major observation is that improving cache reuse is no longer the dominant factor to consider in optimizing these computations. In particular, streaming memory accesses are increasingly important because they engage software and hardware prefetch mechanisms that are essential to memory performance. Many of the grid blocking strategies designed to improve cache locality ultimately end up interfering with prefetch policies and thereby counter the advantages conferred by those optimizations.

Our work next examines optimization strategies for multiple iterations, where the stencil algorithms can block computation in both space and time to reduce overall main memory traffic. 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. We begin by exploring an *explicit* cache-aware algorithm known as time skewing [13, 19, 24], where the blocking factor is carefully tuned based on the stencil size and cache hierarchy details. Next, we present a detailed performance model which effectively captures the behavior of the time skewing algorithm, allowing us to analytically determine a near-optimal blocking factor.

Our study then explores alternative approaches to stencil optimizations by evaluating the *implicit* cache oblivious [6] tiling methodology, which promises to efficiently utilize cache resources without the need to consider the details of the underlying cache infrastructure. Performance is evaluated on the Intel Itanium2, 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.

Experimental results show that, while the cache oblivious algorithm does indeed reduce the number of cache misses compared to the naïve approach, it can paradoxically degrade absolute performance due primarily to sub-optimal compiler code generation. 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.

Overall, our work represents one of most extensive analyses of stencil optimizations and performance modeling to date, examining a wide variety of algorithmic approaches and architectural platforms for this important class of computations.

2. Experimental Setup.

This section describes the experimental testbed for our analysis. First, we present a high-level overview of stencil computations, which are an important component of many numerical algorithms. We then introduce the *Stencil Probe*, a parameterized benchmark that mimics the performance of stencil-based calculations. Finally, we describe our evaluated architectural platforms and code development environment.

2.1. Stencil Computations.

Stencil computations on regular grids are at the core of a wide range of scientific codes. In these computations each point in a multi-dimensional grid is updated with contributions from a subset of its neighbors. These "sweeps" (updates of all points in the grid according to the computational rule) are then typically used to build solvers for differential equations. In this work, we examine the performance of the 3D heat equation shown in Figure 3.1, which uses a seven-point stencil. It is taken from Chombo [1], 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 simple 3D heat equation solver that does not use Chombo's more advanced capabilities.

2.2. Stencil Probe.

The experiments conducted in this work utilize the Stencil Probe [9], 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 [8,9] has shown that the Stencil Probe is an effective proxy for the behavior of larger applications; thus, it can simulate the memory access patterns and performance of large applications, while testing for potential optimizations, without having to port or modify the entire application.

2.3. Hardware Platforms.

Our study examines three leading microprocessor designs used in high performance computing systems: the Itanium2, the AMD Opteron, and the IBM Power5. Additionally, we examine stencil performance on the recently-released STI Cell processor, which presents a radical departure from conventional multiprocessors. An overview of each platform's architectural characteristics is shown in Table 2.1.

The 64-bit Itanium2 system used in our study operates at 1.4 GHz and is capable of issuing two FMAs (fused multiply-adds) per cycle for a peak performance of 5.6 GFlop/s. The memory hierarchy consists of 128 floating point registers (of which 96 can rotate) and three on-chip data caches (32 KB L1 cache, 256 KB L2 cache, and 3 MB L3 cache). The Itanium2 cannot store floating point 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 EPIC (Explicitly Parallel Instruction set Computing) technology where instructions are organized into 128-bit VLIW (Very Long Instruction Word) bundles.

The primary floating-point horsepower of the 64-bit AMD Opteron comes from its SIMD (Single Instruction Multiple Data) floating-point unit accessed via the SSE2 or 3DNow! instruction set extensions. The Opteron utilizes a 128b SIMD floating point multiplier and a 128b SIMD floating point adder, both of which are half-pumped (i.e. requires two cycles per instruction). Thus our 2.2 GHz test system can execute two floating-point operations per cycle and deliver a peak performance of 4.4 GFlop/s.

	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	-
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/xlf	xlc

Table 2.1
Architectural characteristics of our evaluated platforms.

The L2 cache on our test system is a 1MB victim cache (allocated 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 IBM Power5 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 engines 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.

2.4. Code Development and Profiling Environment. Our original goal was to implement all of the codes using the C programming language. However, achieving

*The Power5 has a total of 15 GB/s DRAM bandwidth (10 GB/s load, 5 GB/s store).

the highest possible performance across each platform required several exceptions. Notably, the cache-aware (and naïve) Power5 experiments were implemented in Fortran using `xlf` to minimize the high penalty of pointer ambiguity of `xlc` on the Power5. Additionally, the Cell C implementation included hand-coded SIMD intrinsics to ensure effective vectorization and explicit pointer disambiguation (see Section 8.2).

On all three conventional systems, we use the Performance API (PAPI) library [15] 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, PAPI cache miss counters do not include prefetched cache lines, thus preventing the counters from accurately reflecting overall memory traffic. Therefore, we generally only show Itanium2 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.

3. Single Iteration Performance.

More recently, there has been considerable work in memory optimizations for stencil computations, motivated by both the importance of these algorithmic kernels and their poor performance when compared to machine peak. Cache blocking is the standard technique for improving cache reuse, because it reduces the memory bandwidth requirements of an algorithm. In this section we explore single-iteration stencil performance and examine the potential performance improvement of traditional cache blocking techniques.

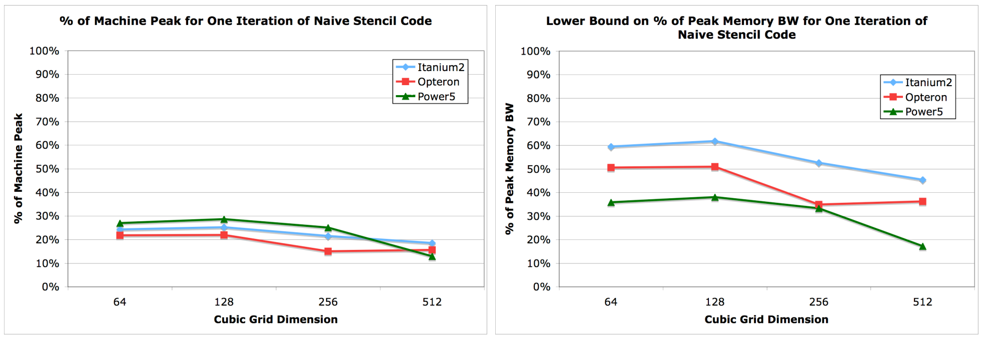
3.1. Naïve Implementation.

Pseudocode for a 3D naïve non-periodic stencil is shown in Figure 3.1. The stencil here uses Jacobi iterations, meaning that the calculation is not done in place; thus the algorithm alternates the source and target arrays after each iteration.

```
void stencil3d(double *A, double *B, int niter, int x, int y, int z) {
  for (int t = 0 to niter) {
    for (int i = 1 to x-1) {
      for (int j = 1 to y-1) {
        for (int k = 1 to z-1) {
          B[i,j,k] = C0 * A[i,j,k]
                     + C1 * (A[i+1,j,k] + A[i-1,j,k]
                     + A[i,j+1,k] + A[i,j-1,k]
                     + A[i,j,k+1] + A[i,j,k-1]);
    }}}
    swap A and B
  }
}
```

FIG. 3.1. *Pseudocode for the 3D naïve stencil kernel using non-periodic boundary conditions.*

Figure 3.2 tries to identify the bottleneck in running one iteration of the stencil code by examining both the (a) percentage of machine peak and (b) the percentage



(a) (b)

FIG. 3.2. *Performance of non-periodic naïve stencil code on the three cache-based architectures for varying cubic grid dimensions. Figure (a) shows the percent of machine peak achieved, while (b) shows a lower bound on the percent of peak memory bandwidth achieved.*

of peak memory bandwidth achieved on the three commodity architectures in our study. As seen in Figure 3.2(a), all three architectures achieve below 30% of machine peak. However, Figure 3.2(b) shows that the fraction of memory bandwidth is almost always greater than 30% of peak memory bandwidth. In fact, since the memory bandwidth figure only counts the memory traffic from compulsory cache misses, it actually provides a lower bound on the fraction of peak memory bandwidth. These results show that due to the high fraction of memory traffic, there is limited potential for optimization over a single iteration. Executing across multiple time steps allows for more optimization opportunities due to increased data reuse, as will be explored in Section 6.

3.2. Single-Timestep Cache Blocking. We now consider the challenging problem of improving memory performance within a single sweep. While the potential payoff for a given optimization is lower than for multiple timesteps, the techniques are more broadly applicable. In prior work, Rivera and Tseng [16] concluded that blocking of 2D applications is not likely to be effective in practice. Our analysis in Section 4.4 agrees with and further quantifies this result, showing that enormous grids are necessary for 2D cache blocking to be effective on current machines.

Rivera and Tseng [16] also proposed a blocking scheme for 3D stencil problems that attempts to alleviate the tiny block sizes that result from traditional 2D blocking schemes when applied to three dimensions. Subdividing a 3D grid into cache blocks results in many small blocks because $blocksize^3$ doubles must fit in the cache, as opposed to $blocksize^2$ doubles when blocking in 2D. These small blocking factors cause poor spatial locality because there are frequent discontinuities in the memory stream. Rivera and Tseng attempted to sidestep this limitation by blocking in the two least significant dimensions only (partial 3D blocking). This results in a series of 2D slices that are stacked up in the unblocked dimension, as shown in Figure 3.3(a).

In order to test the effectiveness of partial 3D blocking, we ran problem sizes up to the largest that would fit in the physical memory of our machines. In Figure 3.3(b) we see the best-case cache-blocked results relative to the unblocked version for grid sizes of 128^3, 256^3, and 512^3. The partial 3D blocking speeds up our stencil computation for grid sizes of 512^3 on the Itanium2 and the Opteron, while on the Power5 we obtain no speedups for any of the three grid sizes (due to the the huge L3 cache on the Power5, as quantified in Section 4.4). Observe that in all cases where blocking confers an advantage, the I^{th} blocking dimension is equal to the grid size

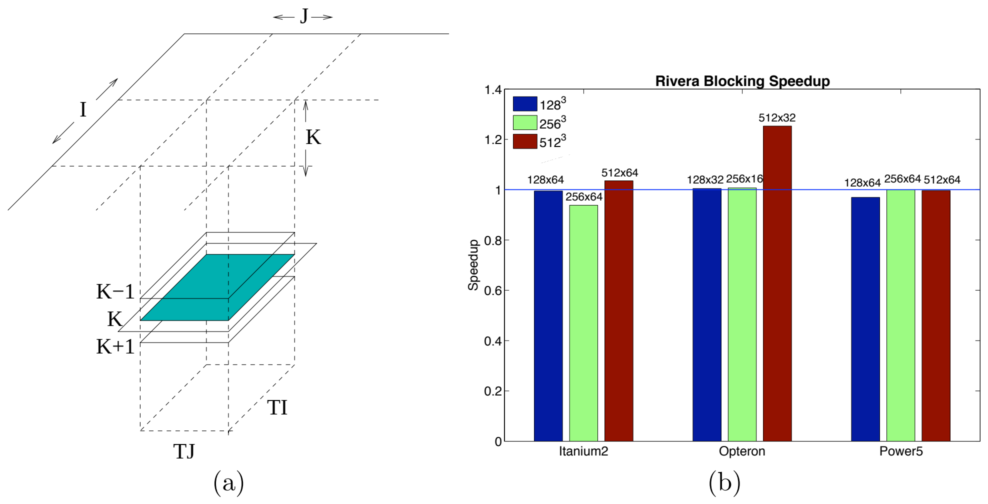


FIG. 3.3. *(a) Partial 3-D blocking using a series of 2D slices stacked up in the unblocked dimension, K, where I is the unit-stride dimension. (b) Speedup results of partial 3D blocking for* 128^3, 256^3, *and* 512^3 *grid sizes using optimal block sizes. Note that the Power5 utilized the* `xlf` *compiler to maximize performance.*

(i.e. maximized).

In order to understand which blocking factors are the most effective for a given architectural configuration, we construct a simple analytical model to predict the cost of memory traffic for a stencil-based computation.

4. Modeling Single Iteration Performance. In order to model the performance of single-iteration cache blocking, we begin by examining the performance of a simpler microbenchmark that has a memory access pattern that nearly matches our cache blocking memory access pattern. We then use the insights gained from the microbenchmark to construct a performance model for cache blocking.

4.1. Stanza Triad. In this section we explore prefetching behavior of modern microprocessors using a simple microbenchmark called *Stanza Triad*. An important trend in microprocessor architectures is the attempt to tolerate the increased memory latency relative to clock frequency. Little's Law [2] asserts that in order to fully utilize the total available bandwidth of the memory subsystem, the number of data elements in-flight concurrently must be equal to the product of the bandwidth and the latency of the memory subsystem. This *bandwidth-delay* product has increased dramatically in recent years. The primary remediation strategy for hiding latency is *prefetch* – both compiler-inserted and automatic hardware prefetch streams. The goal of our work is to demonstrate how the requirements for the efficient use of prefetch can compete with, or even interfere with, traditional strategies employed for cache-blocking, compromising their effectiveness.

To address this issue, we devised a simple microbenchmark called *Stanza Triad*, which is used to evaluate the efficacy of prefetching on various architectures. The *Stanza Triad* (STriad) benchmark is a derivative of the STREAM [12] Triad benchmark. STriad works by performing a DAXPY (Triad) inner loop for a size L stanza before jumping k elements and continuing on to the next L elements, until we reach the end of the array.

Figure 4.1 shows the results of the STriad experiments on the cache-based architectures in our study. The total problem size was set to approximately 48 MB in order to ensure the arrays could not fit in cache. We set k (the jump length) to 2048 double-precision words, which is large enough to ensure no prefetch between stanzas, but small enough to avoid penalties from TLB misses and DDR precharge. Each data size was run multiple times, using a clean cache each time, and we averaged the performance to calculate the memory bandwidth for each stanza length.

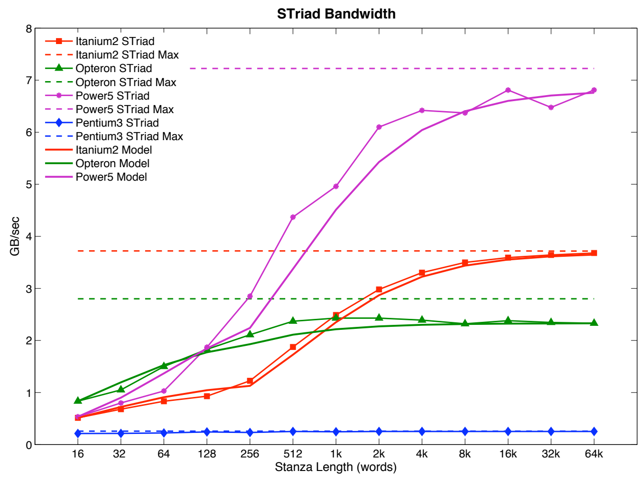


FIG. 4.1. *Performance of STriad on the three evaluated architectures.*

On the Opteron system, we see a relatively smooth increase in bandwidth until STriad reaches peak. In contrast, the Itanium2 and Power5 demonstrate a two-phase increase in performance. Unfortunately, facilities (such as performance counters) to directly capture hardware prefetch behavior are not readily available. Lastly, for historical comparison, we ran STriad on a Pentium 3, a system where prefetch does not offer a significant benefit — notice that the performance behavior here is flat and independent of the stanza length. Finally, it can be seen that (as expected) with increasing stanza lengths, STriad performance asymptotically approaches the "max" bandwidth, which is measured by running STriad and setting the stanza length equal to the array size (similar to STREAM Triad[*]).

4.2. Memory Model for STriad. Based on the measured STriad performance, we now formulate a simple model to predict memory access overhead for a given stanza length. We approximate the cost of accessing the first (non-streamed) cache line from main memory, C_{first}, by the overhead of performing an STriad with a short (single cache line) stanza length. C_{stream}, on the other hand, represents the cost of a unit-stride (streamed) cache miss, as computed by performing an STriad where the stanza length is maximized (set to the total array length). Because the prefetching engines require some number of consecutive cache misses before they ramp up, we also have a third cost, $C_{intermediate}$, which is the cost of accessing a cache line when the prefetch engines have begun ramping up but are not completely at their peak speed.

Therefore, if:

$$\begin{aligned} L &= \text{stanza length (in words)} \\ W &= \text{cache line size (in words)} \end{aligned}$$

[*]The only difference is the loop bound structure.

Then:

Stanza Cache Miss Order	Number of Cache Misses	Cost Per Miss
First	1	C_{first}
Second	k (k is small)	$C_{intermediate}$
Third	$\lceil (L/W) \rceil - k - 1$	C_{stream}

and Total Cost $= C_{first} + k * C_{intermediate} + (\lceil (L/W) \rceil - k - 1) * C_{stream}$.

In other words, we assume that after paying C_{first} to bring in the first cache line from main memory and $C_{intermediate}$ for the next k lines (where k is a small value on the order of several cache lines), the remaining data accesses cost C_{stream} due to enabled stream prefetching. Note that this simplified approach does not distinguish between the cost of loads and stores.

In addition to this three-point model, we also attempted to use just two points (i.e. by setting $k = 0$ and assuming all cache misses are either the first miss in a stanza or are fully-prefetched misses). Results in Figure 4.1 show that our simple performance model reasonably approximates the memory access behavior on all three of our architectures, for both the two point and three point models. However, the three point model more accurately predicts Itanium2 performance at intermediate stanza lengths. Having modeled the timing of this simple proxy code, we now explore a more general model for stencil performance using Rivera blocking.

4.3. Cost Model for Cache Blocking. Several studies have analyzed stencil codes and created metrics to predict performance. Leopold [10] introduced analytic bounds on capacity misses in stencil codes, but did not study actual application or benchmark codes. Through the use of an analytic model, Leopold suggested that rectangular tiles would outperform square tiles, a conclusion supported by the model we build in this section.

We now build on the prefetch-based memory model developed in Section 4.2 to capture the behavior of stencil computations using various cache blocking arrangements, as seen in Section 3.2.

Given an N^3 grid we first approximate the lower and upper bounds on traffic between cache and main memory for a given $I \times J \times N$ blocking. Recall that a 3D stencil calculation accesses 6 columns in three adjacent planes. The lower bound for traffic assumes perfect reuse, where all three $I \times J$-sized planes fit in cache — thus the grid is only transfered twice from/to main memory (one read and one write). The upper bound (pessimistically) assumes no cache reuse of the $I \times J$ planes due to conflict and capacity misses; therefore, for each sweep of a given plane, the 'front' and 'back' planes required for the 7-point stencil calculation must be reloaded from main memory. The upper bound thus requires the grid to be transferred four times from/to main memory (three reads and one write): twice the cost of the lower bound. Note that this is not a strict upper bound, since we assume an optimal number of loads and only consider the costs of memory operations (ignoring registers, ALU, etc). Additionally, our simplified performance model does not differentiate between the cost of load and store operations.

Having established lower ($2\times$) and upper ($4\times$) bounds for the grid memory traffic, we now compute the cost of transferring the appropriate stencil data for a given $I \times J \times N$ blocking. Given a system with W words per cache line, a sweep through an

N^3 grid requires a total of $T_{total} = \frac{\lceil (I/W) \rceil N^3}{I}$ cache lines. Because the grid is accessed in a blocked fashion, we compute the number of non-streamed (non-prefetched) cache line accesses:

$$T_{first} = \frac{N^3}{I} \text{ if } I \neq N, \text{ or } \frac{N^3}{IJ} \text{ if } I = N \neq J, \text{ or } \frac{N^2}{IJ} \text{ if } I = J = N.$$

The number of intermediate (partially-prefetched) cache line accesses is the next k accesses in each stanza. Lastly, the total number of streamed (prefetched) cache lines is then the remaining number of accesses: $T_{stream} = T_{total} - T_{intermediate} - T_{first}$.

We now apply the cost model derived in Section 4.2, where we established that non-streamed access to a cache line from main memory requires a higher overhead (C_{first}) than subsequently streamed cache lines (C_{stream}), due to the benefits of prefetching. Thus the total cost of sweeping through a 3D stencil in a blocked fashion is approximated as

$$C_{stencil} = C_{first}T_{first} + C_{intermediate}T_{intermediate} + C_{stream}T_{stream}.$$

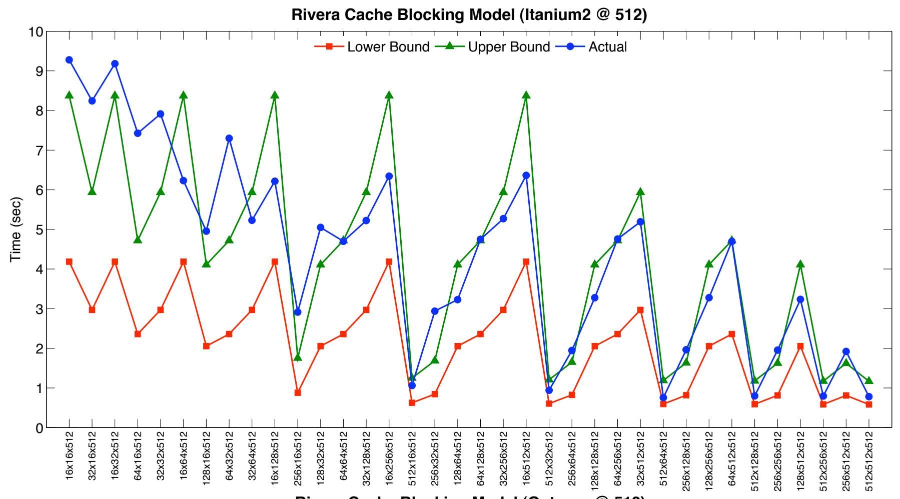
The lower bound of the memory cost for the stencil computation is thus $2C_{stencil}$, while the upper bound is $4C_{stencil}$. Therefore, setting the block size too small will incur a penalty on memory system performance because prefetch is not engaged or is only partially engaged.

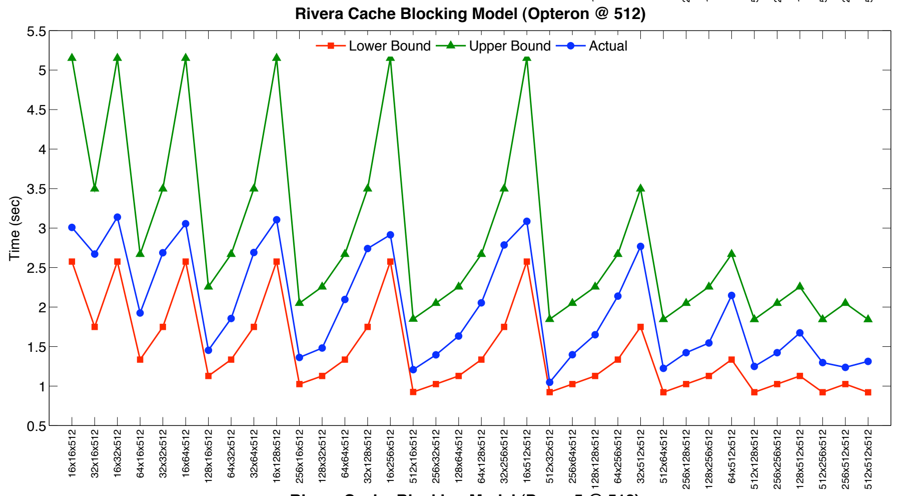
Figure 4.2 shows the lower and upper bounds of our cost model compared with the measured results of the Stencil Probe using Rivera blocking across a complete set (powers of two) of $I \times J \times N$ blocking factors. Results show that our analytical model performs extremely well in capturing the behavior of the stencil computation for all three evaluated architectures. The actual data does occasionally fall outside of the computed lower/upper bounds, but it is clear from the overall performance trends that our methodology is effective in quantifying the tradeoffs between cache blocking and prefetching efficacy.

In the next section, we discuss trends in modern architectures in light of our analysis of partial blocking and the impact of automatic prefetching engines.

4.4. Impact of Architectural Trends. It is important to understand our cache-blocking findings in the context of evolving architectural trends. As silicon lithography techniques improve, processor architects are able to migrate more levels of the cache hierarchy onto the same chip as the microprocessor core. In addition to reducing the latencies for cache misses at each level of the hierarchy, this has also enabled the designers to operate on-chip caches at the same clock frequency as the core. In these cases, an on-chip L2 (and in the case of the Itanium, the on-chip L3) can deliver operands to the core at the same rate as the L1 caches.

Consider that the 360MHz Sun UltraSparc2i platform, studied in the cache tiling work of Rivera and Tseng [16] (described in Section 3.2), used a 16KB on-chip L1, but the off-chip L2 operated at half the processor's cycle time. Likewise, the Pentium II that was used to demonstrate another effective blocking technique [17] operated the off-chip L2 at half the clock rate of the on-chip L1. In contrast, all three of the cache-based processors reviewed in this paper employ a cache hierarchy that is entirely on-chip and operates at the same clock frequency as the core — allowing each level of the hierarchy to operate at the nearly the same effective bandwidth. Since the on-chip cache sizes (operating at the same clock rate as the core) have increased dramatically in recent processor generations, block-sizes that improve bandwidth locality have increased correspondingly.





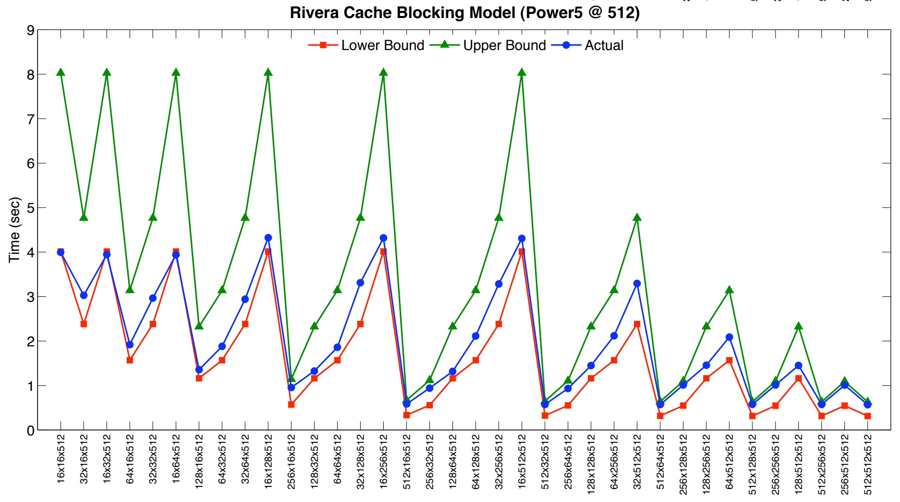


FIG. 4.2. *Comparison between partial 3D-blocking runs and the lower/upper bounds of our memory model. Results show that our analytical approach is extremely effective in predicting blocked stencil performance. On all graphs, the x-axis shows the cache block size with the contiguous dimension listed first.*

The benchmark data in the previous sections suggests that code optimizations should focus on creating the longest possible stanzas of contiguous memory accesses in order to maintain peak performance. These requirements are driven by the behavior of prefetch engines, which are fully engaged via long stanzas of unit-stride stream accesses. Thus, in practical terms, stencil computations must be blocked for the largest level of cache hierarchy that operates at core bandwidth, as was empirically and analytically demonstrated in Sections 3.2 and 4.3.

Note that it may possible to increase stanza length by fusing two or more spatial loops together, as was performed on the CDC CYBER 205 vector architecture in the late 1970's and early 1980's [7] to maximize vector length. For modern processors, loop fusion has the advantage of eliminating potentially costly nested loops at the expense of additional computation. However, the choice of periodic or constant boundary conditions adds significant complexity to the stencil. Fused loops with a periodic boundary results in extremely complex address calculation, and a constant boundary results in a conditional store for the stencil. Neither of these are attractive solutions on traditional superscalar architectures as they would likely add a branch to the inner loop. This is unlike the vector CYBER 205 machine, which contained a bit mask to dictate which elements of the destination array could be written to. For future work we will consider the more tractable solution of unrolling and loop interchange in the intermediate spatial dimension.

Figure 4.3 describes the conditions where tiling may offer a benefit for 2D and 3D stencil computations, based on our analysis. Six microprocessor architectures are plotted on this graph, based on the the largest tile size that would derive a performance gain for stencil computations. This is equivalent to the deepest level of cache capable of communicating with the processor at full bandwidth. Two different generations of microprocessors are plotted, where the vertical position is based on the on-chip cache size, while the horizontal position is based on the memory footprint for a given sized 3D stencil (128^3, 256^3, and 512^3). Any processors that are below the top red line (bottom blue line) may see a performance benefit from tiling for the 3D (2D) problems. (Note that there is more opportunity for effective cache-blocking for 3D computations than for 2D.) Processors above these lines will likely see a performance degradation from attempts to use a tile-based optimization strategy, since all the appropriate data-sets already fit in the on-chip caches without blocking. It can be seen clearly from this graph that the growth of on-chip L2 and L3 caches have dramatically raised the size of problems that would see any benefit from cache-blocking for stencil computations.

The predictions of Figure 4.3 can be compared against the results presented Section 3.2. As Figure 3.3(b) shows, the Itanium2 and Opteron clearly benefit from blocking for 512^3 grid sizes, while no benefit is seen on Power5 due to its large 36 MB L3 cache. These observations were also validated on the PowerPC G5 in a previous study [9].

It is also important to understand the range of grid sizes currently being utilized in large-scale scientific applications. For example, the grid sizes for Cactus [4], a computational framework for astrophysics, are typically 80^3 per processor for parallel General Relativity simulations, and will occasionally be stretched to 128^3 per processor, if the memory is available. Chombo, on the other hand, is an AMR code that uses adaptive hierarchical meshes to refine the computation where needed. Cells are selected for refinement on a per-cell basis, which are then aggregated in order to create the largest possible grid that can achieve a specified filling ratio of selected cells. While larger grids can be formed, the typical grid size formed by this strategy is 32^3

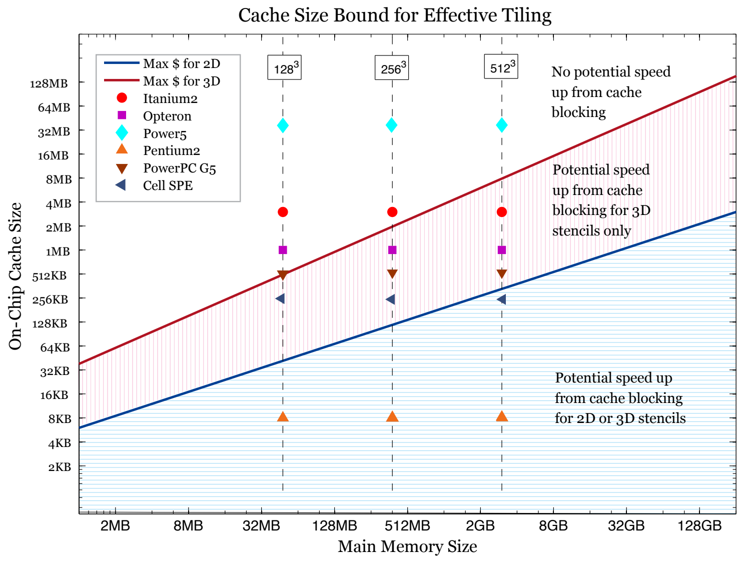


FIG. 4.3. *Conditions where tiling offers a potential benefit for 2D and 3D stencil computations. The upper red line (lower blue line) shows the cache limit for a given problem size that could benefit 3D (2D) problems. Six microprocessor on-chip cache sizes are plotted. Processors below the line may benefit from cache blocking for the specified problem sizes* (128^3, 256^3, 512^3) *whereas those above a given line will generally not.*

to 64^3 elements in size. Thus, it is our observation that high-end stencil computations are currently not run at a grid scale that would benefit from tiling a single sweep, due to the large on-chip caches of the underlying microprocessors.

5. Multiple Iteration Time Skewing. As seen in Section 4.4, 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. Thus, more contemporary approaches to stencil optimization are geared towards tiling techniques that leverage blocking in both the spatial and temporal dimensions of computation, in order to increase data reuse within the cache hierarchy [6,13,19,24]. In the remainder of this paper we examine optimization strategies for stencil algorithms that block computation both in space and time to reduce overall main memory traffic.

Note that performing several sweeps through a grid at once is not always possible, as applications may require other types of computation between stencil sweeps. However, there are important cases where consecutive sweeps are required, such as: relaxes during the multigrid algorithm [17], sweeps in a conjugate gradient preconditioner [20], or convergence steps within a dual-time stepping scheme in computational fluid dynamics [14, 21]). Similarly, our sample application performs multiple stencil sweeps without intermediate work.

A logical extension to single-iteration cache blocking, the time skewing algorithm [13, 19, 24] blocks in both space and time while respecting stencil dependencies. The algorithm uses explicitly defined cache block sizes; however 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, one must perform an exhaustive 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

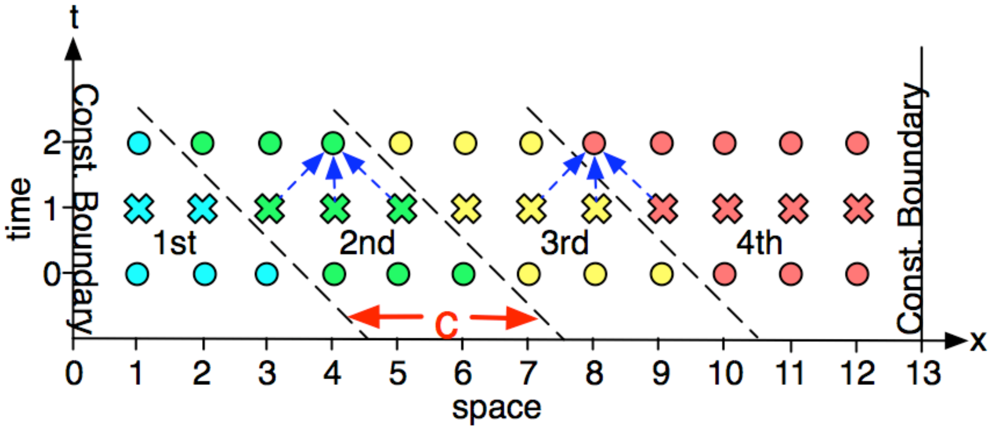


FIG. 5.1. *A simplified two-dimensional spacetime diagram of time skewing with a 3-point stencil. The dotted blue arrows show dependencies for two different points. In order to preserve these dependencies, the cache blocks need to be executed in the order shown. In addition, the X's and O's indicate which of two arrays is being written to.*

(the unit stride dimension) is left uncut to allow for longer unit-stride memory streams as demonstrated in Sections 3.2 and 4.3.

5.1. Time Skewing Algorithmic Description. Figure 5.1 shows a simplified diagram of time skewing for a 3-point stencil where the grid is divided into cache blocks by several skewed cuts. These cuts are skewed in order to preserve the data dependencies of the stencil. To clarify this concept, two points in the figure are shown with blue arrows indicating dependencies. For the green point, all three of its dependencies lie within the second cache block, and therefore it too can be computed within the same block. On the other hand, the red point's dependencies span the third and fourth cache blocks. In this case, since the last dependency is in the fourth cache block, the red point must also be computed in that block. In general, as long as the blocks are executed in the proper order, the algorithm respects the stencil dependencies.

However, the blocks generated from time skewing do not all perform the same amount of work, despite equal partitioning in the first time step. As time progresses, the shifting causes the cache blocks at the boundaries to perform unequal work. As shown in Figure 5.1, the number of points per iteration slowly decreases for the first cache block, while it slowly increases for the final cache block. The interior cache blocks perform the same number of stencil operations, since shifting does not change the number of points per iteration.

In general, the time skewing algorithm requires two sets of loops. The outer loops iterate over every cache block in the usual manner (i.e. the more contiguous dimensions are iterated through more quickly than the less contiguous dimensions). These outer loops have fixed loop bounds. Within these loops is a set of inner loops that iterates over the points within each cache block. These inner loops have varying loop bounds depending on the location of the specific cache block. For instance, all interior cache blocks have the same shape, and they always skew toward the completed portion of the grid. On the other hand, exterior cache blocks do not skew on the sides where they touch a grid boundary.

A closer representation to our actual 3D time skewing code is illustrated in Figure 5.2. 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.

There are few potential performance limitations caused by the skewing algorithm. The first is that extra cache misses may be incurred by shifting, 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 mitigate, if not eliminate, the extra memory traffic.

A second concern is that skewing 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 5.1, 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 7.1(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.

Additionally, there are concerns about the practicality of applying the time-skewing approach to real applications. Although it can be difficult to utilize this optimization on complex codes that contain more than one algorithm aside from stencils, there are significant benefits that could be realized by applying time-skewing approach. For example, in dual-stepping methods [14, 21], the outer-loop evolves the system forward in time, while the inner-loop requires hundreds of iterations to drive residual errors to zero. Thus, the inner loop is sufficiently isolated from the rest of the code that it could greatly benefit from time-skewing optimizations.

Finally, we note that in this work time skewing is described exclusively as an out-of-place algorithm, simply because it is easier to visualize. However, it is a trivial change to modify the algorithm to be in-place. As a result, this work applies to Gauss-Seidel [17] and the Successive Overrelaxation (SOR) in addition to Jacobi iterations [18]. In all cases, the final result will be numerically identical to doing consecutive sweeps over the entire grid. For example it is necessary to reformulate the computation as in-place algorithm to support the dual-stepping of implicit solvers required for the unsteady incompressible of flow problems [14, 21].

5.2. Time Skewing Performance. We first verify that the per-iteration memory traffic does in fact decrease with more iterations. These results are only shown for the Itanium2 since it is the only platform in our study with accurate cache miss counters (see Section 2.4). Figure 5.3(a) confirms that for small block sizes, overall memory traffic decreases drastically from the first iteration (left) to the fourth (right). 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.

Figure 5.3(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 shows the greatest improvements in performance.

Table 5.1 shows time skewing performance for 128^3, 256^3, and 512^3 problem sizes on the Itanium2. Notice that the overall gains in computational speed are not as dramatic as the savings in memory read traffic. This is because the problem shifts from being memory bound to being computation bound, at which point further

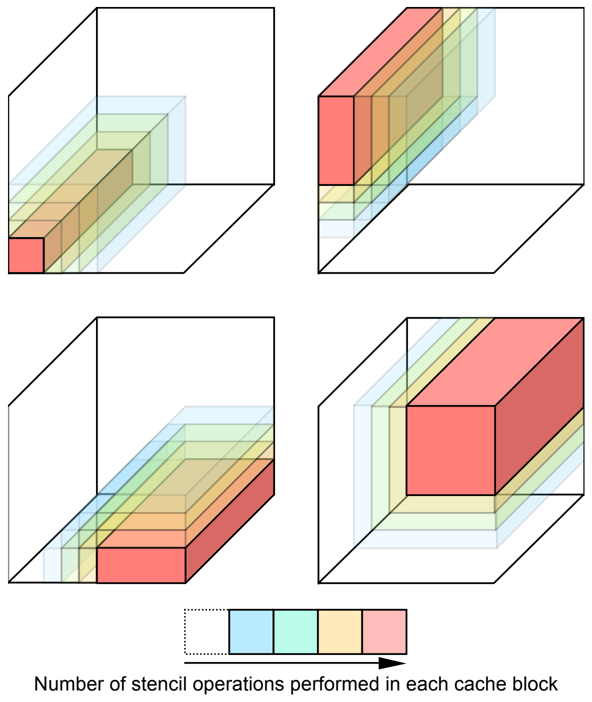


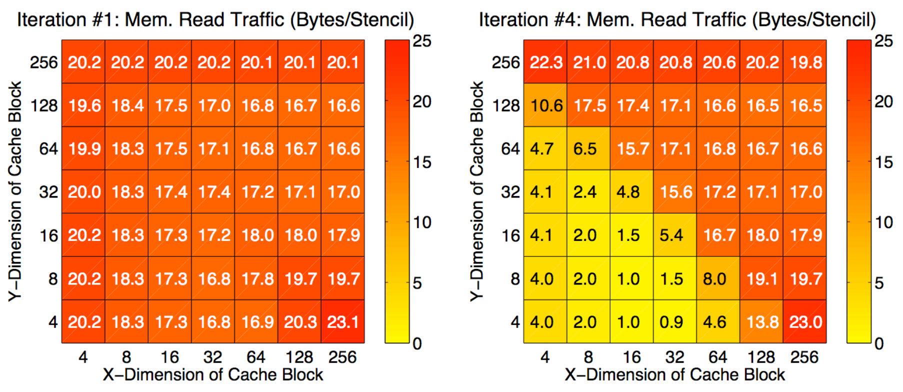
FIG. 5.2. *Color coded plots of the number of stencils operations performed on a 10^3 grid using four iteration time skewing with 5x5x10 cache blocks. There is one plot for each cache block. Blue halos represent only a single stencil operation for that region, where red blocks show the cores where the 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.*

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 code 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 [9] for details). Thus, the same point in the source array needs to be brought into cache several times during a single iteration, resulting in significant main memory traffic.

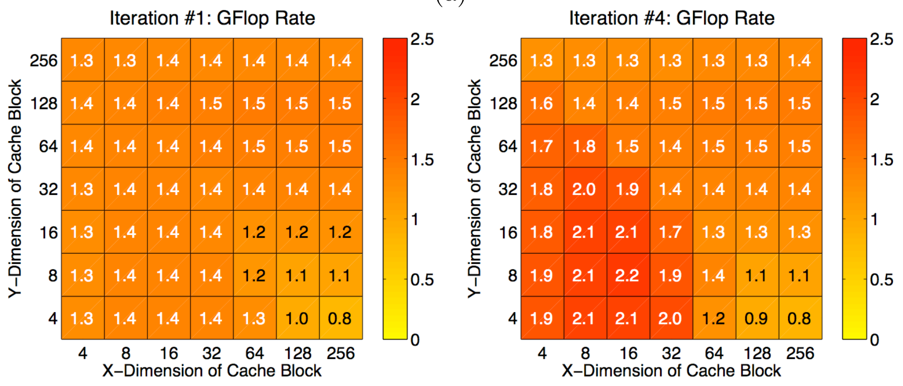
Time skewing addresses this problem by processing individual cache blocks one at a time. This effectively shrinks the size of each plane, allowing all the iterations for a point to be completed after bringing it into cache only once. The result is a drastic drop in memory traffic (84%) and consequently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

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

TABLE 5.1
Time skewing for four iterations of varying problem sizes on the Itanium2.



(a)



(b)

FIG. 5.3. *Finding the optimal cache block size using time skewing on the Itanium2. 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 Itanium2 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 5.4 shows the GFlop rates for the Opteron and Power5 in addition to the Itanium2. The data shown is for the best runtime on each of the platforms for four iterations, determined by an exhaustive search across all block sizes. As expected, the graph indicates that for all three platforms, time skewing produces a significant speedup over the naïve code during later iterations.

6. Time Skewing Performance Model. We now develop a performance model to identify an optimal blocking size for a time skewed stencil calculation. Having an effective model obviates the need to conduct an exhaustive search across all block sizes, as was performed in the previous section. Additionally, an analytical model allows us to gain greater insight into potential bottlenecks and architectural behaviors. For instance, identifying whether performance is computation or memory bound allows the appropriate optimization strategies to be applied.

6.1. Modeling Cache Misses. First, we examine the case where the stencil code is memory bound, by modeling the number of cache misses resulting from different cache block sizes. For each cache block, there are two sources of cache misses: the misses from initial cache loading, and misses from shifting the block (when performing multiple iterations). Cache misses are mostly compulsory during initial data loading. Shifting the blocks, however, may cause capacity misses since blocks are

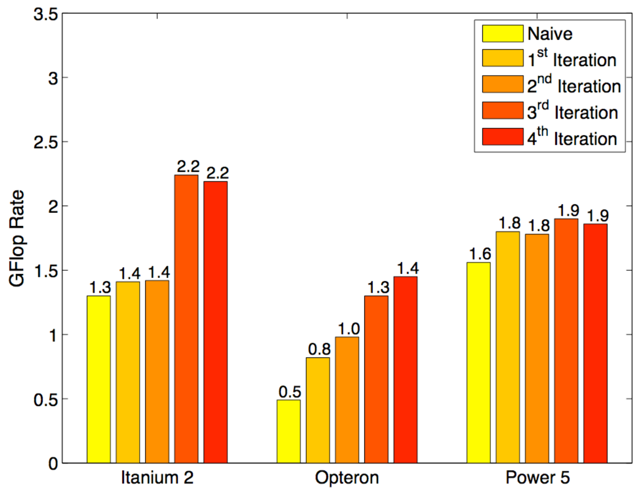


FIG. 5.4. *GFlop rates for time skewing, where each platform's best block size was determined by the fastest running time for four iterations. This is a* 256^3 *problem with constant boundaries. Note that this chart shows performance per iteration, not average over all iterations, and that Power5 experiments use* `xlf`.

always shifted towards the completed portion of the grid; thus, if the cache is large enough, these misses can be avoided.

Our performance model categorized cache misses as being streaming or non-streaming (as described for the STriad microbenchmark in Section 4.1). When streaming through memory, prefetch engines (if present) will retrieve the next cache line in advance. As a result, loading a successive cache line will typically be a fast operation. Non-streaming cache misses, on the other hand, occur when loading non-adjacent cache lines. In this case, prefetch engines are usually not useful, and these misses become more expensive. Our model differentiates between streaming and non-streaming cache misses, assigning them different costs based on the two-point STriad microbenchmark. However, both streaming and non-streaming cache misses are equivalent in terms of the volume of memory read traffic.

As explained in Section 3.2, optimized cache blocking strategies do not cut in the contiguous memory dimension. Our model also assumes that, like in the time skewing algorithm, all the iterations are completed for one cache block before proceeding to the next block. In addition, we assume a 7-point 3D stencil on the source grid is written to a single point in the target grid. Note that the this 7-point stencil simultaneously utilizes three planes of the source grid.

Based on these assumptions, we can divide the first iteration misses into five cases, as shown in Figure 6.1. Note that the first three cases are preferable, since they can reuse data across iterations, while the final two cases cannot benefit from data reuse. Thus we can optimize performance by choosing block sizes that avoid these undesirable cases.

In all of the first three cases, there will be compulsory cache misses from initially loading the current source and target cache blocks. However, the cases vary in how many misses result from the shifting. In the first case, one plane of cache blocks from each array fits into cache. This scenario results in the fewest cache misses, since two sides of the source block are still in cache as the current blocks are updated and there no cache misses from shifting. In the second case, the current and previous blocks

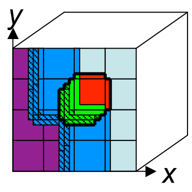
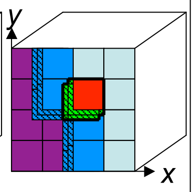
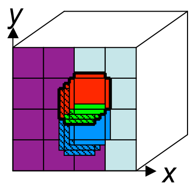
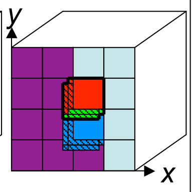
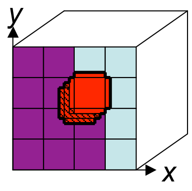
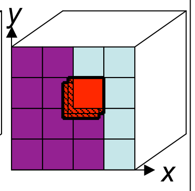
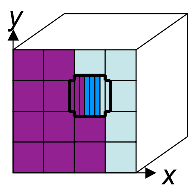
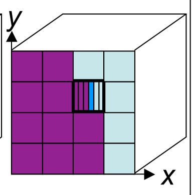
Case #	First Array	Second Array	What Fits Into Cache	Cache Misses for Current Block
1			One plane of cache blocks in each array	Compulsory misses from loading current block, but no misses from skewing
2			The previous and current cache block in each array	Compulsory misses from loading current block and misses in one direction from skewing
3			The current cache block in each array	Compulsory misses from loading current block and misses in two directions from skewing
4			Three planes of the source cache block and one plane of the target cache block	The full cache block is reloaded into cache during each iteration
5	None of the Above		Case #4 does not fit into cache	Each point in the source block is loaded multiple times during each iteration

FIG. 6.1. *The different cases for the time skewing performance model, where the x-axis represents the least contiguous dimension and the y-axis is the intermediate dimension. Each case should be considered only after the previous (lower-numbered) cases are ruled out. In the diagram, the current block is indicated by a thick black line surrounding it, and areas that are loaded due to shifting are indicated by diagonal lines. In addition, the colors represent the following: purple blocks may or may not be in cache, dark blue blocks must be in cache, red indicates misses in the current block, green indicates hits in the current block, and the light blue areas have not yet been traversed.*

from each array fit into cache. Under these circumstances, only one side of the source block is still in cache, so shifting only causes cache misses in one direction. In the third case only the current source and target blocks are kept resident in cache, thus shifting causes cache misses in two directions.

The final two cases do not reuse data across multiple iterations, thereby defeating the main thrust of cache blocking. Consequently, these cases should be avoided. The fourth case occurs when both blocks do not fit into cache, but at least three planes of the source block and one plane of the target block do. In this scenario, every point in each grid will be brought into cache once per iteration. Finally, the fifth case occurs when three planes of the source block and one plane of the target block do not fit into cache. This exhibits the worst cache behavior, since each interior point in the source block needs to be loaded into cache three times per iteration (once for each plane of the 7-point stencil). Here there is no data reuse within a single iteration, let alone across multiple iterations.

Having outlined the possible model scenarios, we now validate our model against the actual memory read traffic. Figure 6.2(a) shows this comparison for the 3D stencil computation on the Itanium2 using a 256^3 problem size for one (top) and four (bottom) iterations. Note that our memory read traffic model does not incorporate prefetching, thus to make a fair comparison we deactivated the software prefetch for the Itanium2 experiments. Additionally, we incorporated conflict misses into our performance model as a cumulative Gaussian distribution that matched the data from a simple microbenchmark. Observe that, while not perfect, the performance model accurately predicts the actual memory read traffic for both one and four iterations of the stencil computation. We now explore how to extend this memory traffic model to predict the overall stencil running time.

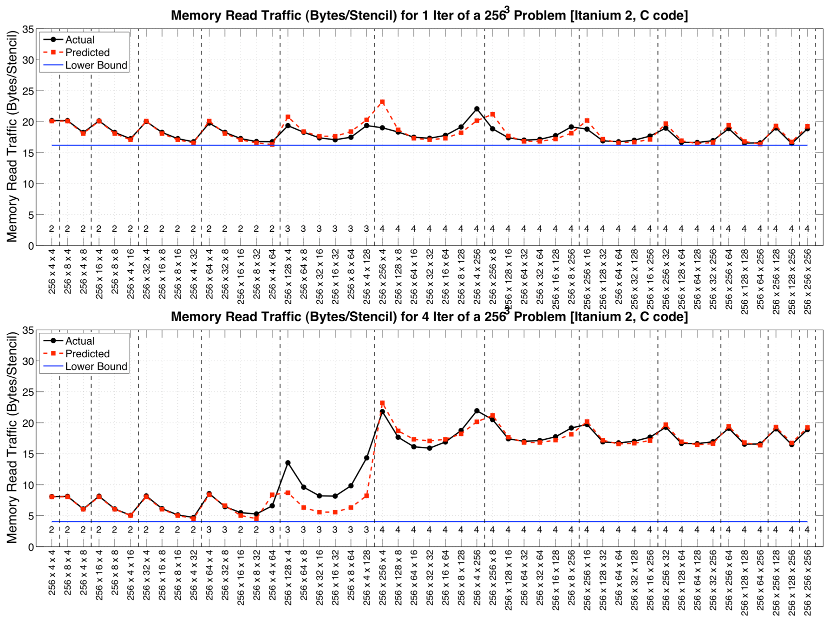
6.2. Modeling Performance. Having developed a memory traffic model, we now develop a model for compute-bound stencil computations, as (depending on the block size) the overall runtime will be a combination of these two factors.

We first normalize by converting the memory read traffic into a running time. This is done by using the STriad microbenchmark (described in Section 4.1) to determine the time for both a streaming and non-streaming cache miss. Combining this with our cache miss model of Section 6.1 allows us to predict running times for stencil codes that are memory-bound. Next, we develop a model for compute-bound stencils by running multiple iterations over a small problem that fits into the processor's L1 cache. Once the problem is loaded into cache during the first iteration, all subsequent iterations are then processor bound. The computation rate for these processor-bound iterations represents the maximum compute rate that can be achieved for this code. The final step is to reconcile the memory-bound and processor-bound models so that we reasonably predict the running time. For the predicted running time of the first iteration, the memory-bound model is always used, since the problem needs to be loaded into cache. However, for subsequent iterations, the maximum of the two memory- and compute-model overheads is chosen, since that will be the limiting factor.

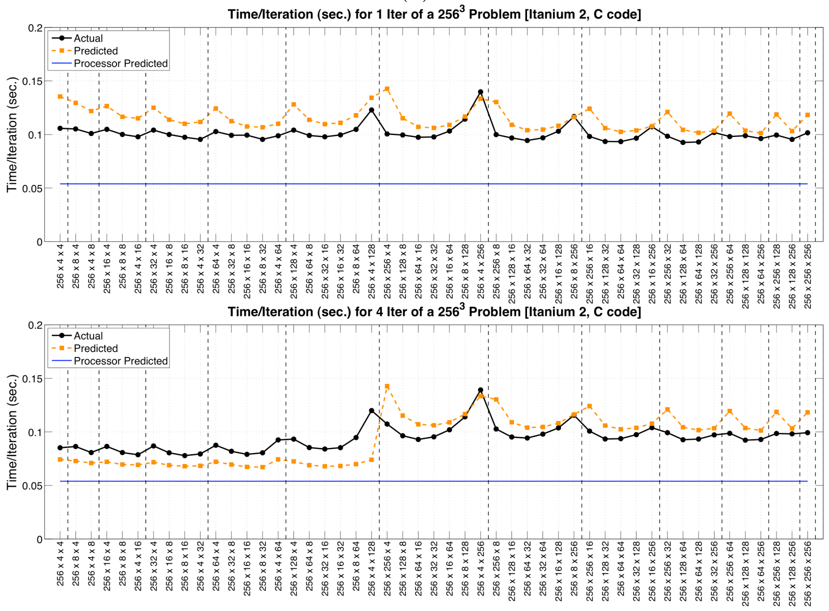
Itanium2 results comparing actual and predicted runtimes for one (top) and four (bottom) iterations are shown in Figure 6.2(b). Overall our model is reasonably accurate in predicting overall running time for varying block sizes. Note that, as expected, the runtime model is not as accurate as the memory traffic model. Since the memory traffic model is used in creating the runtime model, errors present in the memory model are propagated to the running time model, along with any additional errors in the running time model itself.

Figure 6.3(a) and (b) presents a comparison of actual and predicted runtime on the Opteron and Power5 (respectively), for one (top) and four (bottom) iterations. Observe that our model generally predicts the runtime (and trends) of both systems under varying blocking factors; although the model's accuracy is lower on the Opteron than for the Itanium2.

Turning to the Power5, we see that our performance model very accurately captures the actual runtime behavior of the stencil computation. However, contrary to the other platforms in our study, the Power5 is insensitive to cache blocking for the time skewed experiments, because here the stencil code is computationally bound. This is due to the Power5 high memory bandwidth relative to its computational per-

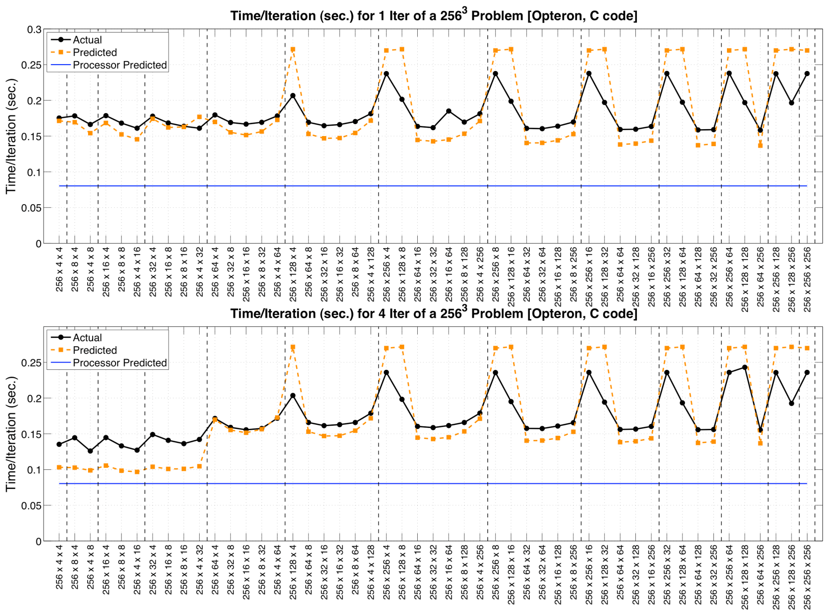


(a)

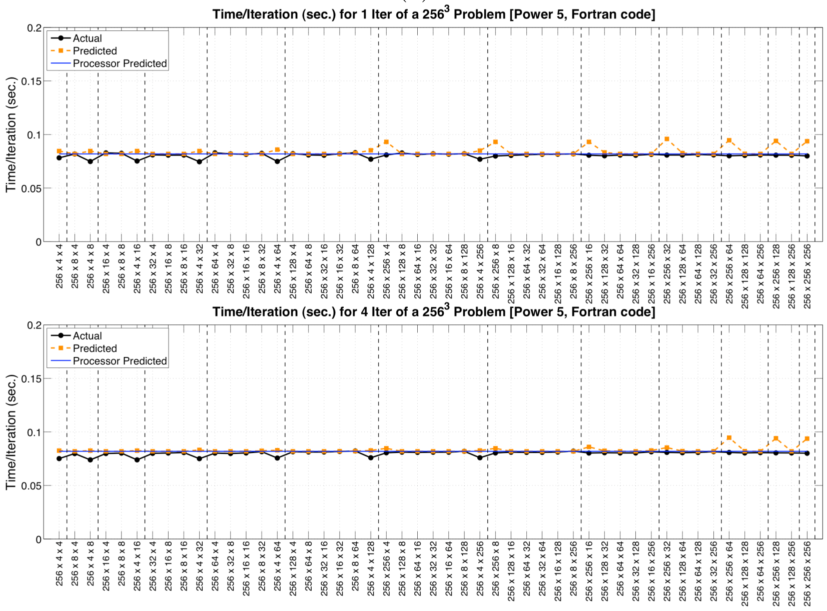


(b)

Fig. 6.2. *A comparison of the time skewing performance model against reality on the Itanium2. The graphs show (a) main memory read traffic and (b) running times for a* 256^3 *problem with constant boundaries. In both pairs of graphs, the top graph shows data for one iteration, while the bottom graph shows average data over four iterations. On all graphs, the x-axis shows the cache block size with the contiguous dimension listed first (in this experiment, the contiguous dimension is never cut). The x-axis is ordered such that the block sizes are monotonically increasing, and the vertical dotted lines divide areas of equal-sized cache blocks.*



(a)



(b)

FIG. 6.3. *A comparison of the time skewing performance model against reality on non-Itanium architectures. The graphs show running times on the (a) Opteron and (b) Power 5 for a* 256^3 *problem with constant boundaries. In both pairs of graphs, the top graph shows data for one iteration, while the bottom graph shows average data over four iterations. On all graphs, the x-axis shows the cache block size with the contiguous dimension listed first (in this experiment, the contiguous dimension is never cut). The x-axis is ordered such that the block sizes are monotonically increasing, and the vertical dotted lines divide areas of equal-sized cache blocks.*

formance (see Table 2.1). Note that, unlike the experiments of Section 3.2 where the Power5 was sensitive to cache blocking, here we maximize stanza length by not cutting (blocking) in the unit-stride dimension. This approach amortizes loop overheads and enables more effective prefetching, thus more efficiently utilizing the available memory bandwidth and increasing code performance.

7. Cache Oblivious Stencil Computations.

Having examined and analytically modeled the well-established time skewing approach, we now explore alternative methodologies for improving stencil computation performance. However, unlike the cache-aware time skewing approach, the cache oblivious stencil algorithm [6] 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 [6]. Note that like time skewing, the cache oblivious approach can be easily modified from an out-of-place to an in-place algorithm.

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 7.1(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).

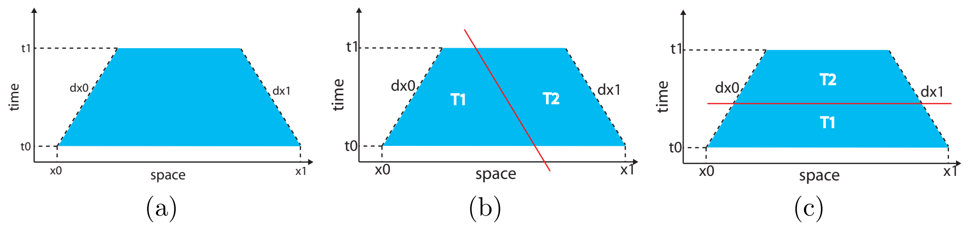


Fig. 7.1. *(a) 2D trapezoid space-time region consisting of a 1D space component and 1D time component, and an example of cache oblivious recursive (b) space cut and (c) time cut.*

In order to recursively operate on smaller spacetime trapezoids, a cut is performed in either the space or time dimension. That is, 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 7.1(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 7.1(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 dimension, as shown in Figure 7.1(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

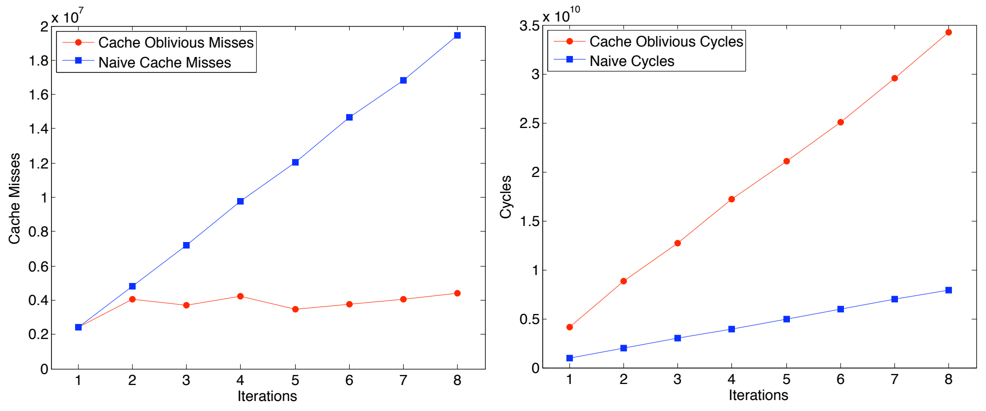


FIG. 7.2. *Performance of the initial cache oblivious implementation for a* 256^3 *periodic problem on our Itanium2 test system showing (a) cache misses and (b) runtime cycles. Although the algorithm reduces cache misses the performance worsens.*

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.

7.1. Cache Oblivious Performance. Performance results, in terms of cache misses and cycles, for our initial implementation of the cache oblivious stencil algorithm (based on the pseudocode in [6]) are shown in Figure 7.2(a) and (b). Although the implementation successfully reduces the number of cache misses, the overall time-to-solution is much slower for the cache oblivious code than for the naïve stencil implementation.

In an attempt to mitigate this problem, we performed several optimizations on the original version of the stencil code, including:

- Explicit inlining of the kernel. The original cache oblivious algorithm in [6] performed a function call per point. Instead, we inlined the function.
- Using an explicit stack instead of recursion. Because the algorithm is not tail-recursive, we could not completely eliminate recursion. Instead, we explicitly pushed and popped 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. Sections 3.2 and 4.3 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.

A summary of the optimized cache-oblivious performance for one to four itera-

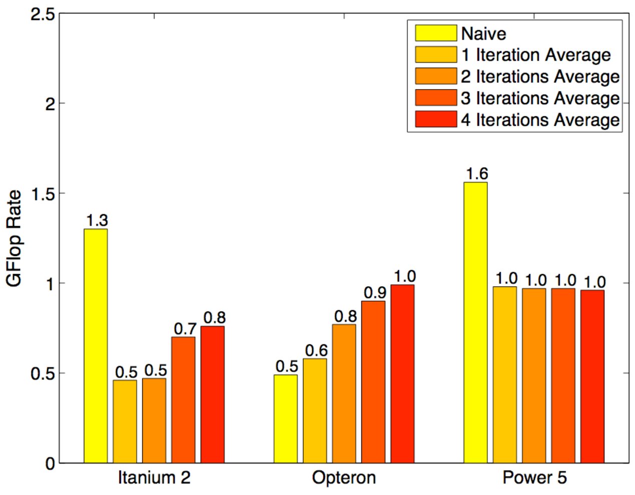


FIG. 7.3. *Performance of optimized (non-periodic) cache oblivious implementation for a* 256^3 *problem. Note that this chart shows average performance over the specified number of iterations.*

tions using constant (non-periodic) boundaries is shown in Figure 7.3. Observe that on the Opteron, the cache oblivious and 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). At four iterations, the cache oblivious methodology on the Opteron attains a performance improvement of 2x compared with the naïve approach.

However, on the Itanium2 and Power5, the compiler-generated code for the cache oblivious case performs poorly compared with the naïve version. This is apparent in Figure 7.3, which shows that one iteration of cache oblivious and naïve stencil have vastly different performance on these two platforms, although they essentially execute the same source code[†]. As a result, the overall performance of the cache oblivious implementation is much worse than the naïve case on the Itanium2 and Power5, achieving approximately only 60% of the naïve runtime at four iterations, despite reducing the overall main memory read traffic substantially. These results highlight the potential limitations of the cache oblivious approach — despite several layers of optimizations — due to (in part) the compiler's inability to generate optimized code for the complex loop structures required by the cache oblivious implementation.

8. Stencil Computations on Cell.

Our final stencil implementation is written for the Cell processor's unconventional microarchitecture whose local store memory is managed explicitly by software rather than depending on automatic cache management policies implemented in hardware. This approach is in sharp contrast to the cache oblivious algorithm as both cache blocking and local-store data movement are explicitly managed by the programmer.

Before implementing the stencil algorithm on a Cell SPE, we examine 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

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

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 when performing only a single iteration. As a result, there is no benefit for time skewing in double precision on a single Cell chip even at 3.2 GHz. It should be noted that in single precision, the stencil algorithm on the Cell changes from computationally-bound to memory-bound. This is because the 14x increase in computational performance overwhelms the benefit of a 2x decrease in memory traffic.

8.1. 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 [22]. In this case, six blocked planes must be stored simultaneously within a single SPE's local store. Figure 8.1 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.

8.2. 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, our implementation utilizes SIMD intrinsics. 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 high, it is important to keep in mind that 49 stall cycles are consumed by double precision instructions. Thus, each pair of points only requires 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).

8.3. 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. Therefore assigning batches of local store blocks to SPEs allows for simple and efficient parallelization on the Cell 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 order to minimize load imbalance. 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.

8.4. Cell Performance. Cell results are detailed in Table 8.1, attaining impressive performance of approximately 7 GFlop/s for the 3.2 GHz test system. Note

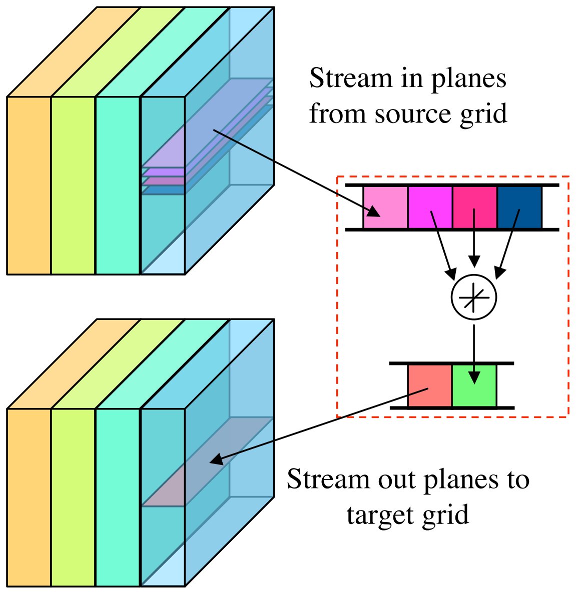

Stream in planes from source grid
Stream out planes to target grid

Fig. 8.1. *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.*

that 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, the 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). Results show that Cell is heavily computationally bound even when performing just one iteration at a time, and the potential impact of inefficient blocking is completely hidden by the significantly improved memory efficiency and vastly improved memory bandwidth. Comparing performance between the 2.4 GHz and 3.2 GHz machines shows nearly linear scaling (relative to clock speed), confirming our the assertion that the stencil code on the Cell is indeed 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 processor.

	GFlop/s	GFlop/s	Read memory traffic
Problem size	@2.4GHz	@3.2GHz	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 8.1

*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.*

8.5. Time Skewing. As described in Section 8 the stencil calculation is computationally bound in double precision but memory bound in single precision. A four step time skewed version similar to the blocking algorithm developed by Sellappa and Chatterjee [17] was demonstrated in [22]. 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. Although this is somewhat less efficient since work is duplicated, the Cell delivers an impressive 49.1 GFlop/s at 2.4 GHz and a truly astounding 65.8 GFlop/s at 3.2 GHz for single precision stencils.

Our experiments have not yet explored the Cell blade, which consists of two NUMA chip (16 SPE). In this configuration, each chip may access the DRAM directly attached to it at 25.6 GB/s (51.2 GB/s combined), but communication between chips is substantially slower via the I/O bus. Thus, if memory affinity cannot be guaranteed (i.e. a single thread per blade), effective memory bandwidth will suffer greatly, becoming the bottleneck. This may therefore presents an opportunity to fully utilize the blade by performing two time skewing steps, and will be the subject of future investigation.

Looking ahead, the forthcoming Cell eDP (extended double precision) promises over 100 GFlop/s of double precision performance [5]. Using the performance model suggested in [23], we can estimate both naïve and time skewed performance. Our analysis indicates that naïve performance will come very close to the arithmetic intensity-bandwidth product (12.2 GFlop/s); however, we predict that the cell time skewing approach will rapidly loses efficiency — achieving approximately 21, 26, and 21 GFlop/s on average for 2, 3, and 4 time steps respectively. This is an artifact of the relatively small local store and the inefficiency introduced to parallelize the code.

9. Conclusion.

Stencil-based computations are an important class of numerical methods that are widely used in high-end scientific applications. Although these codes are characterized by regular and predictable memory access patterns, the low computational intensity of the underlying algorithms results in surprisingly poor performance on modern microprocessors. It is therefore imperative to effectively maximize cache resources by tiling in both the spatial and temporal dimensions when possible.

In this paper we explored the impact of trends in memory subsystems on a variety of stencil optimization techniques and developed performance models to analytically guide our optimizations. Our work first examined stencil computations where, due to computation between stencil sweeps, blocking is restricted to a single iteration (i.e. only in the spatial direction). Results show that modern processors contain relatively large on-chip caches in comparison to main memory size, meaning that single-iteration cache blocking is now effective only for very large (sometimes unrealistic) problem sizes. We also observe that prefetching, both in hardware and software, improves memory throughput for long stride-1 accesses, but also makes performance more sensitive to discontinuities in the access patterns. Finally, we devised a simple analytical model for partial 3D blocking on stencil grids, which demonstrates the importance of avoiding cache-blocking in the unit stride direction.

Our work then focused on optimizations that improve cache reuse by merging together multiple sweeps over the computational domain, enabling multiple stencil iterations 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. We explored a combination of software optimizations and hardware features to improve the performance of stencil computations, including cache oblivious algorithms, (cache-aware) time skewed optimizations, and the explicitly managed local-store of the Cell processor. Additionally, we developed accurate performance models to analytically identify the optimal blocking schemes on cache-based architectures, without the need for exhaustively searching the tile space.

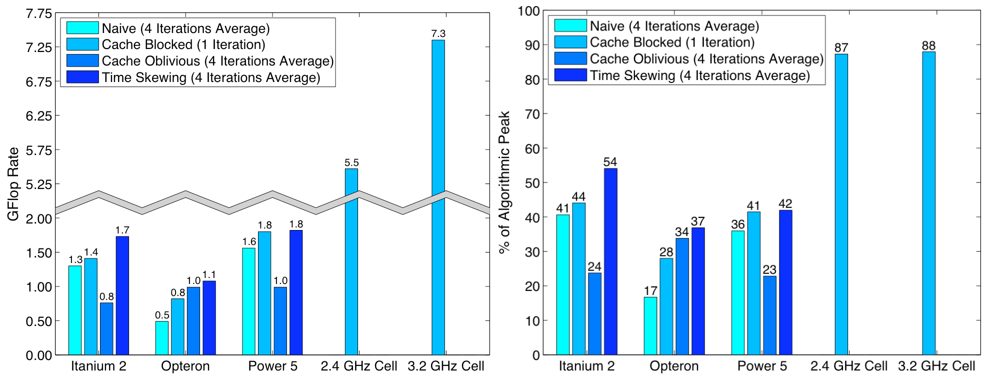


Fig. 9.1. *(left) GFlop rates and (right) percentage of algorithmic peak, for a* 256^3 *problem with constant boundaries.*

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

Table 9.1

Total main memory traffic per point for naïve , cache blocked, cache oblivious, and time skewing on the Itanium2 as well as for Cell for a 256^3 *problem.*

A summary of our multiple-iteration results is presented in Figure 9.1(left). Observe that the cache oblivious approach 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 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, as can be seen in Figure 9.1(left) where time skewing consistently outperforms the cache oblivious approach.

Another surprising result of our study is the lack of correlation between main memory traffic and wallclock run time. Although cache oblivious algorithms reduce misses (as seen in Table 9.1), they do not generally improve the overall run time. Furthermore, some of the lower-level optimizations we implemented, such as never cutting the unit-stride dimension, increase memory traffic while actually reducing the time to solution. These optimizations, while reducing cache reuse, can prove advantageous because they more effectively utilize the automatic hardware and software prefetch facilities.

The most striking results in Figure 9.1(left) are for the Cell processor. Cell has a higher off-chip bandwidth than the cache-based microprocessors (nearly 2× 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 7× 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 9.1(right), 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 continue our investigation of predictive performance models into the realm of multi-core processors. We also plan to extend our scope of stencil optimization techniques, with the ultimate goal of developing automatic performance tuners for a wide variety of stencil-based computations.

10. Acknowledgments. The authors would like to thank Parry Husbands for his many contributions. We also thank Nehal Desai from Los Alamos National Labs for running our Cell code on their prototype 2.4 GHz machine as well as Otto Wohlmouth from IBM Germany for running our code on the new 3.2 GHz Cell machines. All LBNL authors were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231.

REFERENCES

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

[2] D. Bailey. Littleś law and high performance computing. *RNR Technical Report*, 1997.

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

[4] Cactus Homepage. `http://www.cactuscode.org`, 2004.

[5] The Cell project at IBM Research. `http://www.research.ibm.com/cell/`, 2007.

[6] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proc. of the 19th ACM International Conference on Supercomputing (ICS05)*, Boston, MA, 2005.

[7] W. Gentzsch. *Vectorization of Computer Programs with Applications to Computational Fluid Dynamics*, volume 8 of *Notes on Numerical Fluid Mechanics*. Friedr. Vieweg and Sohn, 1984.

[8] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *ACM SIGPLAN Workshop Memory Systems Performance and Correctness (MSPC)*, San Jose, CA, 2006.

[9] 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.

[10] C. Leopold. Tight bounds on capacity misses for 3D stencil codes. In *Proceedings of the International Conference on Computational Science*, volume 2329 of *LNCS*, pages 843–852, Amsterdam, The Netherlands, April 2002. Springer.

[11] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2001.

[12] J. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCAA Newsletter*, December 1995.

[13] 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.

[14] J. C. Oefelein. Large eddy simulation of turbulent combustion processes in propulsion and power systems. *Progress in Aerospace Sciences*, 42:2–37, 2006.

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

[16] G. Rivera and C. Tseng. Tiling optimizations for 3d scientific computations. In *Proceedings of SC'00*, Dallas, TX, November 2000. Supercomputing 2000.

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

[18] G. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 2004.

[19] 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.

[20] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[21] S. Venkateswaran and C. Merkle. Analysis of preconditioning methods for the euler and navier-stokes equations. In *VKI Lecture Series 1999-03 (Computational Fluid Dynamics)*, pages 113–114. Von Karman Institute, March 1999.

[22] 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.

[23] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the cell processor. In *International Journal of Parallel Programming*, pages 263–298, June 2007.

[24] 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.