Understanding Stencil Code Performance On MultiCore Architectures [*]

Shah M. Faizur Rahman
Computer Science Dept.
University of Texas at San Antonio,
srahman@cs.utsa.edu

Qing Yi
Computer Science Dept.
University of Texas at San Antonio
qingyi@cs.utsa.edu

Apan Qasem
Computer Science Dept.
Texas State University
San Marcos, TX
apan@txstate.edu

ABSTRACT

Stencil computations are the foundation of many large applications in scientific computing. Previous research has shown that several optimization mechanisms, including rectangular blocking and time skewing combined with wavefront- and pipeline-based parallelization, can be used to significantly improve the performance of stencil kernels on multi-core architectures. However, the overall performance impact of these optimizations are difficult to predict due to the interplay of load imbalance, synchronization overhead, and cache locality. This paper presents a detailed performance study of these optimizations by applying them with a wide variety of different configurations, using hardware counters to monitor the efficiency of architectural components, and then developing a set of formulas via regression analysis to model their overall performance impact in terms of the affected hardware counter numbers. We have applied our methodology to three stencil computation kernels, a 7-point jacobi, a 27-point jacobi, and a 7-point Gauss-Seidel computation. Our experimental results show that a precise formula can be developed for each kernel to accurately model the overall performance impact of varying optimizations and thereby effectively guide the performance analysis and tuning of these kernels.

1. INTRODUCTION

Stencil computations are used to solve a large number of important scientific computing problems such as Partial Differential Equations and image manipulation, among others. These kernels use an outmost *time* loop to make a large number of sweeps over a multi-dimensional grid so that the value of each grid point is repeatedly modified based on values of neighboring points. While it is safe to restrict parallelism within each sweep of a large grid by updating different portions of the grid using multiple threads, the lack of data reuse and the relatively small amount of computation within each thread makes the performance of this parallelization scheme less than desirable on modern multi-core architectures. To reduce the *computation to synchronization* ratio, each thread needs to operate on multiple sweeps of a data block by coordinating with other threads. Time skewing[13] combined with wavefront or pipelined parallelization can accomplish this goal, but load balancing is a known issue which could seriously degrade performance for these schemes[14].

This paper studies several strategies to effectively explore both the single-sweep and time-skewed parallelism for stencil computations on modern multi-core architectures. We have parameterized each optimization scheme with an array of different configurations and used hardware performance counters to measure the impact of these configurations on various architectural components. Based on a large collection of empirical data, we then apply regression analysis to develop a set of formulas which precisely model the overall performance impact of differently optimized code in terms of their efficiency in utilizing various hardware components. We have applied our methodology to three stencil kernels, a 7-point jacobi (`jacobi7`), a 27-point jacobi (`jacobi27`), and a 7-point Gauss-Seidel (`gauss`) on the intel *Nehalem* architecture. Our experimental results show that a precise formula can be developed for each kernel to effectively guide the performance analysis and tuning of these kernels.

Fig. 1 shows the correlation coefficients between the overall execution time and different hardware counter values measured at runtime for the three stencil kernels when applying cache and parallelization optimizations with different configurations. Most of these optimizations focus on efficiently utilizing individual architectural components such as L1/L2/L3 caches, TLB, and CPU clock cycles. However, the eventual impact of the optimization is often hard to predict. From Fig. 1, we see that the overall performance impact of each hardware counter value varies significantly in spite of the three stencil kernels demonstrating similar computation and data access patterns. Further, some hardware events, e.g., hardware prefetching for the L2 cache (*L2-pref-triggered* and *L2-pref-retired*) could have a positive impact for some kernels but a negative impact on others.

We aim to model the relationship between performance improvements achieved by different optimizations and their efficiency of utilizing various hardware components. The model is *not* intended to predict absolute execution time and thus is supplementary to existing work on performance modeling of scientific applications based on knowledge from static program analysis or profiling [10, 5]. In particular,

[*]This research is funded by the National Science Foundation under Grant No. 0833203 and No. 0747357 and by the Department of Energy under Grant No. DE-SC001770

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'11, May 3–5, 2011, Ischia, Italy.
Copyright 2011 ACM 978-1-4503-0698-0/11/05 ...$10.00.

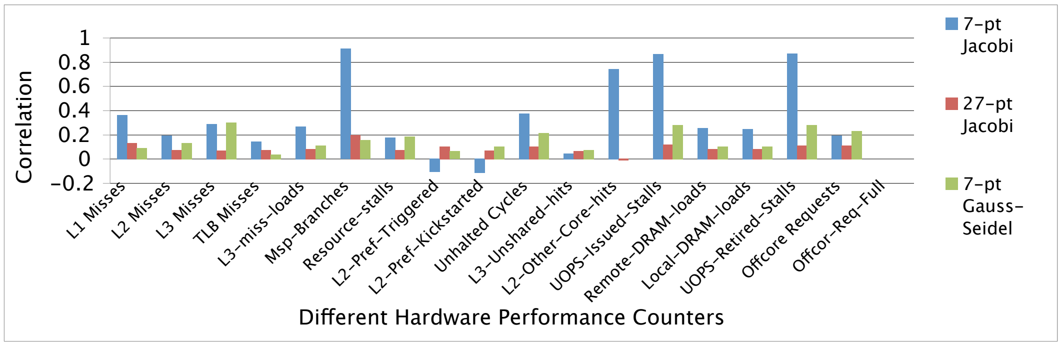


Figure 1: Correlation between overall execution time and different hardware counters

our approach can be used to systematically extract meaningful insights from large collections of experimental data. Such insights can then be used by developers to enable more effective optimization tuning of their applications.

Our approach uses regression analysis to develop a set of formulas for each of the three stencil kernels and show that these formulas can accurately relate the performance impact of optimizing different hardware components and thereby can be used to guide more effective tuning of the benchmarks. For example, our performance formula for optimizing the 7-point jacobi kernel on a single core is shown in Fig. 10 and indicates that the speedup gained by any optimization can be modeled as $1 - NormalizedTime = 0.75 - 0.1 * L1\_miss - 0.18 * L2\_miss - 0.28 * L3\_miss + 0.0008 * TLB\_miss + 0.016 * mis\_branch + 0.014 * hw\_prefetch$. This indicates that in order to achieve better single-threaded performance, developers should foremost try to reduce the L3 cache miss, followed by reducing L2 and L1 misses. Hardware prefetches may help to some extent, but TLB misses have a minimal impact, and mis-predicted branches actually improve performance (which indicate the kernel is memory-bound, where mis-predicted branches may help relieve pressure on memory bandwidth). More details of the performance analysis are discussed in Section 5.

The contributions of this paper include the following.

- We present a detailed performance study of different optimization strategies for stencil computations on the Intel Nehalem multi-core architecture and compare their impact on different architectural components.
- We show that for each stencil kernel, a set of formulas can be used to model the overall performance of differently optimized code based on the impact of optimizations on individual architectural components.
- We show how to apply regression analysis to a large collection of empirical data to derive the formulas and verify the precision of the approach. The methodology can potentially be automated without requiring detailed knowledge of the underlying hardware.

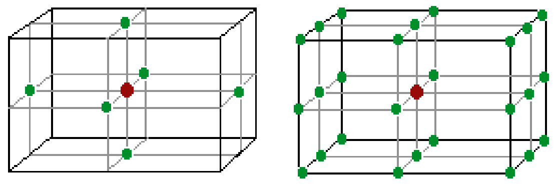
The generality of our methodology and its applicability to other computational kernels are yet to be verified. But our experimental results have demonstrated great potential of this approach and have shown that they can be used to effectively guide the optimization of different stencil kernels.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 provides background on different types of stencil codes and discusses the optimization strategies we have implemented. Sections 4, 5, and 6 present our experimental methodology and results. Finally, section 7 presents conclusions and future work.

2. RELATED WORK

Because of its importance in scientific computing, stencil codes have received considerable attention from the research community. Earlier work on stencils have focused on exploiting data locality [21, 19, 23], while more recent efforts have considered both locality and parallelism issues in concert [13, 14, 6]. Multi-core systems provide ample opportunities for parallelizing stencil applications but the presence of shared caches makes the issues of parallelism and data locality intricately related. For this reason, in recent years, there has been a fair amount of work that targets both data locality and parallelism in stencil computations [12, 3, 6]. These approaches span both manual and automatic code optimizations and also incorporate auto-tuning. Kamil et al. describe a set of optimizations for improving stencil performance on both cache-based systems and architectures with explicitly controlled memory. Their approach yields integer factor speedups over naive implementations mainly because of better utilization of memory bandwidth [13]. Datta et al. [7, 8] and later Kamil et al. [12] extend this work to incorporate more code optimizations (including unroll-and-jam and multi-level blocking for both locality and parallelism) and more architectures (including GPUs). Kamil et al. also propose several models for predicting stencil performance and use auto-tuning to select optimal blocking factors. Krishnamoorthy et al. discuss wavefront parallelism for time-skewed stencil codes. They propose two new techniques, *overlapped tiling* and *split tiling*, both of which can significantly reduce synchronization costs in the pipeline computation without affecting data locality [14]. Bondhugula et al. extend this work to provide an automated framework based on the polyhedral model that performs both effective parallelization and locality optimization of stencil codes [3]. Liu and Li propose an asynchronous algorithm for reducing synchronization costs and improving locality in stencil computations [15]. Christen et al. present a strategy for improving locality and exploiting parallelism in a stencil code appearing in a Bio-heat equation. They specifically target the Cell BE and Nvidia GPUs and thus their strategy exploits some features specific to these architectures [6]. Treibig et al. described a framework for parallelizing iterative stencil computation on multicore architectures. They apply wavefront parallelization combined with temporal blocking to Jacobi and Gauss-Seidel computations and achieve significant speedup on five different multicore architectures [27].

Although many of the approaches mentioned above were successful in achieving high performance for stencil kernels, the focus has been on code optimizations and their interactions. We present a methodology which uses hardware per-



(a) 7-Point Stencil (b) 27-Point Stencil

Figure 2: Stencil Structure

formance counters to relate the efficiency of different architectural events. HW performance counters have been used in performance studies and application tuning since they were first exposed in the Pentium 4 architecture [11]. Tikir and Hollingsworth combine runtime instrumentation and hardware counters on UltraSparc II to improve memory performance for several numerical kernels [26]. Eranian describes how performance counters can be combined on the Core 2 Duo architecture to measure different aspects of memory performance including bandwidth utilization, access latency and remote memory traffic [9]. Marin and Mellor-Crummey present performance studies of several scientific applications where HW performance counters are used to detect opportunities for data locality optimizations [16]. Adhianto et al. provide a framework for analyzing performance of large scale parallel applications using HW counters [1]. This framework has been used to identify and measure bottlenecks in parallel application, including parallel idleness and parallel overhead [25]. Performance counters have also been used for thread scheduling [22], prefetching [24], power estimation [20] and detecting changes in program behavior [17].

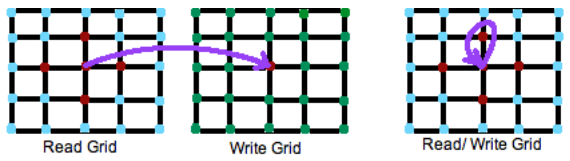
Our strategy is distinct from previous approaches in that it does not focus on a specific performance bottleneck but rather on capturing the interplay of various architectural events. In this regard, our work is similar to the work by Cavazos et al. [4] where they use a wide range of HW performance counters and machine learning algorithms to determine correlations between architectural events and compiler optimizations. Our work is supplementary to this approach as we model correlations between the overall performance and the relative efficiency of different architectural events.

Previous research has applied regression analysis to estimate the power consumption of applications based on runtime hardware counters [20, 18]. We have adopted a similar methodology for a different purpose and use regression analysis to guide performance tuning of the benchmarks.

3. OPTIMIZING STENCIL CODES

A stencil computation typically sweeps over a multi dimensional grid and modifies each point in the grid based on its neighboring values. In most applications they are invoked repeatedly over the data domain and are referred to as *time-step* stencils or *iterative* stencils. Since application context is extremely important for evaluating performance, in this study we only consider iterative stencil computations.

Most stencils exhibit a high degree of temporal locality because each update operation accesses neighboring values on the grid. Typically, in a three-dimensional stencil there is data reuse along all three dimensions. Since each time step sweeps over the same data grid, stencils also exhibit data locality in the time dimension. Exploiting locality in the time dimension is critical for stencil performance because the size of the data grid in real applications exceeds the capacity of L1 and L2 caches on current architectures [7].



(a) Jacobi (b) Gauss-Seidel

Figure 3: Variations in Stencil Update Operation

The amount of data reuse present in the spatial domain depends on the number of neighboring values involved in each update. For example, in *Jacobi Relaxation*, the update of a data point depends on the current position and its neighbors on the left, right, front, back, above and below; thus this constitutes a 7-point stencil. The structure of this stencil is shown in Fig. 2(a). In this 7-point stencil 4 of the 7 data values are reused in each iteration. The structure of a 27-point stencil is shown in Fig. 2(b). In this stencil 8 of the 27 values are reused in every iteration. 7-point stencils are one of the most commonly occurring stencils in scientific code. However, higher order stencils are not uncommon. For instance, 9-point stencils appear in Finite Difference Methods and 27-point stencils appear in multi-grid solvers and advection code. Higher order stencils pose interesting performance challenges since there is more reuse of data but parallelism is harder to exploit because of multiple loop carried dependencies. For this study, in addition to two 7-point stencil codes we also evaluate a 27-point stencil.

Bandwidth and storage requirements of stencils can be influenced by their separation of the *read* and *write* data. In some stencils such as *Jacobi Relaxation*, the read and write domains are separate, where data are read from one grid and written to a different one. After each iteration the grids are swapped and the process is repeated, as illustrated by Fig. 3(a). On the other hand, in stencils such as the *Gauss-Seidel Computation*, data are read and written to the same grid in each iteration, as illustrated by Fig. 3(b). Although stencils like *Gauss-Seidel* require less bandwidth, they create additional concerns for parallelization because writes to the shared data structure need to be synchronized. Our study includes both *Jacobi* and *Gauss-Seidel* type stencils.

3.1 Exploring Locality and Parallelization

Existing research on optimizing stencil kernels have focussed on exploiting both data locality and parallelism. Data reuse in stencils can be exploited at the register-level by applying unroll-and-jam [2]. Sweeps across the data domain can also be tiled to reduce the working set size and improve cache locality. Because of the carried dependencies, however, locality in the time dimension can only be exploited through a combination of loop skewing and blocking (referred to as time skewing [28]). In terms of extracting parallelism, simple data parallelization can be applied to the spatial loops (e.g., Jacobi Relaxation) in some cases. In other situations pipeline parallelization can be achieved through a combination of skewing and using additional temporary storage [7].

To understand the performance of stencil kernels we explore several optimization strategies to improve both data locality and parallelism. We apply multi-level loop blocking and time skewing to each kernel to exploit data reuse. These blocked and time-skewed variants are then parallelized using

```
for (t = 0; t < timesteps;t++){
 for (k = 1; k < nz - 1; k++) {
 for (j = 1; j < ny - 1; j++) {
 for (i = 1; i < nx - 1; i++) {
  Anext[i,j,k] =
   A0[i,j,k+1] + A0[i,j,k-1] +
   A0[i,j+1,k] + A0[i,j-1,k] +
   A0[i+1,j,k] + A0[i-1,j,k] -
   alpha * A0[i,j,k];
   }
  }
 }
 tmp = A0
 A0 = Anext
 Anext = tmp
}
```

(a) Original code

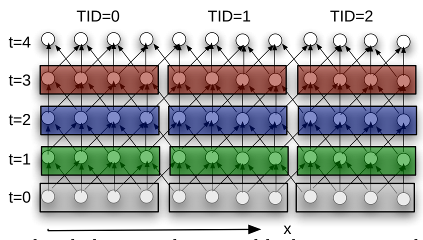
```
for (t = 0; t < timesteps; t++) {
 for (kk = 1; kk < nz-1; kk+=tz) {
 for (jj = 1; jj < ny-1; jj+=ty) {
 for (ii = 1; ii < nx-1; ii+=tx) {
 for (k=1; k<Min(nz-1,kk+tz);k++){
 for (j=1; j<Min(ny-1,jj+ty);j++){
 for (i=1; i<Min(nx-1,ii+tx);i++){
  Anext[i,j,k] =
    A0[i,j,k+1] + A0[i,j,k-1] +
    A0[i,j+1,k] + A0[i,j-1,k] +
    A0[i+1,j,k] + A0[i-1,j,k] -
    alpha * A0[i,j,k];
 } } }
 } } }
 ... swap A0 and Anext ...
}
```

(b) After loop blocking

```
for (kk=1; kk < nz-1; kk+=tz) {
for (jj = 1; jj < ny - 1; jj+=ty) {
for (ii = 1; ii < nx - 1; ii+=tx) {
for (t = 0; t < timesteps; t++) {
 ...set up min_z,max_z,min_y,max_y,min_x,max_x...
 for (k=min_z; k<max_z; k++) {
 for (j=min_y; j<max_y; j++) {
 for (i=min_x; i<max_x; i++) {
  Anext[i,j,k] =
    A0[i,j,k+1] + A0[i,j,k-1] +
    A0[i,j+1,k] + A0[i,j-1,k] +
    A0[i+1,j,k] + A0[i-1,j,k] -
    alpha * A0[i,j,k];
 } } }
  ... swap A0 and Anext ...
 } } } }
```

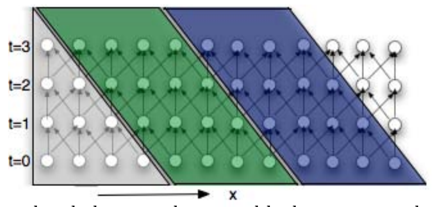
(c) After time skewing

Figure 4: Result of applying locality optimizations to the 7-point Jacobi code



*Grid points that belong to the same block are grouped together. Computation blocks that have the same color are evaluated simultaneously by different threads.

Figure 5: Single-sweep parallelization



*Grid points that belong to the same block are grouped together. Computation blocks that have the same color are evaluated simultaneously by different threads.

Figure 6: Pipeline parallelization

three different schemes, *single-sweep blocking*, *pipeline*, and *wavefront* parallelization. We discuss each variant below.

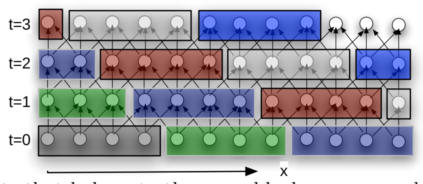
Single-sweep blocking: Fig. 4(b) illustrates the result of applying conventional loop blocking to improve cache reuse of the 7-point Jacobi stencil code show in Fig. 4(a). In this scheme, a single sweep of the grid is partitioned into rectangular blocks. In particular, within a single iteration of the time-step loop t, the k, j, and i loops are blocked so that grid points that are close together in space are grouped to be modified together. This allows each grid point to remain in cache when used to compute new values for their neighboring points in the same iteration of the *time* loop.

Time skewing: Fig. 4(c) illustrates the result of applying time skewing to improve cache reuse of the 7-point Jacobi stencil code. In this scheme, multiple sweeps of the grid are collectively partitioned into *skewed* triangular and parallelepiped blocks. The key difference between this strategy and the single-sweep blocked variant is that different iterations of the t loop are included as part of each computation block. Because of the reordering constraints between updates of neighboring grid points across different iterations of the t loop, each computation block must shift its collection of grid points backward by a constant number of positions (i.e., the skew factor) at each iteration of the t loop. This is accomplished by dynamically setting the lower and upper bounds of the innermost k, j, i loops. In theory, the time-skewed variant is likely to deliver better performance because of improved data reuse in the time dimension. However, the improved performance will depend on the selection of appropriate skewing factors. For this reason, skewing factors are included as a tunable parameter in our study.

Single-sweep parallelization: Fig. 5 illustrates the parallelization scheme we adopt for the blocked code in Fig. 4(b), where the partitioned blocks within a single sweep of the grid are assigned to different threads to be evaluated simultaneously. In this variant, we parallelize the kk loop in Fig. 4(b), so that multiple threads are used to evaluate its iterations, and all threads synchronize before entering the next iteration of the surrounding t loop. Note that this parallelization scheme is applicable to the 7- and 27-point Jacobi iterations but not applicable to the gauss-seidel kernel, which uses a single grid instead of using the two grids $A0$ and $Anext$ shown in Fig. 4(a). As a result, the blocked loops cannot be safely parallelized due to dependence constraints.

Pipeline parallelization: Fig. 6 illustrates the pipeline parallelization strategy for stencil kernels. Here time-skewed blocks are assigned to designated threads and evaluated in a pipelined fashion. We parallelize the outermost kk loop in Fig. 4(c) by explicitly spawning new threads to evaluate designated regions of its iterations in parallel. To ensure correctness of evaluation, before evaluating each computation block, each thread explicitly synchronizes with others and waits until all pre-requisite computation blocks have been finished. Similarly, after evaluating each block, each thread synchronizes with its neighboring threads to enable evaluation of the dependent blocks. As a consequence of this synchronization scheme, each computation block is evaluated as soon as it is ready and neighboring computation blocks are evaluated by different threads in a pipelined fashion.

Wavefront parallelization: In this strategy, instead of pipelining the evaluation of time-skewed blocks via explicit synchronization, the time-skewed blocks are scheduled collectively in a wavefront fashion. In particular, instead of parallelizing the kk loop in Fig. 4(c) and then inserting synchronizations outside of the t loop, the loops inside t are adjusted so that at each iteration of the t loop, the computation block being evaluated is independent of the other blocks evaluated by different iterations of t. The new t loop can then be safely parallelized. Fig. 7 illustrates the independent



*Grid points that belong to the same block are grouped together. Computation blocks that have the same color are evaluated simultaneously by different threads.

Figure 7: Wavefront parallelization

computation blocks enumerated by different iterations of the t loop. Note that the scheduling of these parallel blocks are identical to that of the pipeline parallelization scheme. The difference is that the parallel blocks are no longer bound to any designated threads. Therefore data reuse across different time steps are no longer guaranteed within each thread.

3.2 Implementation Of Optimizations

We have implemented the optimization strategies described above using POET, a general-purpose program transformation language [29] which supports flexible parameterization of compiler optimizations so that their configurations can be empirically tuned. We have manually written POET scripts for three stencil kernels, 7-point Jacobi, 27-point Jacobi, and 7-point Gauss-Seidel iterations. Each script applies the necessary optimizing transformations with parameterized configurations such as varying tile sizes for the spatial and time dimensions. All optimizations we apply can be integrated within a full-blown optimizing compiler and fully automated for scientific codes that demonstrate a similar behavior as the stencil computations [14].

4. EXPERIMENTAL DESIGN

The goal of our paper is to present a detailed performance study of the optimizations discussed in Section 3.1. We first collect a large set of empirical data by applying these optimizations with different configurations, e.g., using different parallelization and memory blocking factors, shown in Table 1. Based on these empirical data, we then develop a set of formulas for each benchmark to model the overall performance improvement by different optimizations in terms of their impacting factors on various architectural components (e.g., L1/L2/L3 misses). We then verify the accuracy of these formulas using additional samples beyond those used in building the formulas. Note that these formulas are intended to be used not standalone but combined with application-specific knowledge of optimizations, e.g., by developers or iterative compilers, to reduce tuning time of existing optimizations or to guide additional optimizations.

Table 1 lists the main optimization strategies we applied to the three stencil kernels and their configuration parameters. Section 5 presents experimental results of applying the *blocked* and *timeskewed* optimizations, and results of the parallelization schemes are presented in Section 6.

4.1 Performance Measurements

We measured the performance of differently optimized code on an Intel Nehalem 8-core machine. The machine has 2 sockets, each with 4 Intel Xeon 5507 2.27 GHz cores which share a 4MB last level cache (L3 cache), a local 3 channel Integrated Memory Controller, and two Intel QuickPath Interconnect. Each core has a 32 KB private Data cache (L1 cache), a 32KB Instruction cache, and a 256 KB Unified Mid-level L2 Cache. The L3 Cache is inclusive (i.e.. all the data on the L1 and L2 cache are also present in L3). Cache line requests from the remote socket are handled by the Uncore's Global Queue (GQ). Each socket has 4MB memory shared by all the local cores.

We compiled all the benchmarks using the Intel *icc* compiler version 11.1, using -O2 optimization flags. We used PAPI to monitor a large number of hardware performance counters at the thread level granularity. We kill all unnecessary processes and make sure no resource intensive background processes are running on the system before executing our experiments. We ran each experiment five times and took the minimum of the recorded execution time .

4.2 Regression Analysis

We use regression analysis on the empirically collected data to establish relations between execution time and hardware performance counters. While raw execution time can be related directly to hardware counter numbers, their runtime contributions overlap significantly due to internal hardware concurrency, and their relations cannot be accurately modeled using linear equations. To reason about optimization effectiveness, instead of using raw runtime statistics, we normalize both the overall execution time and all the hardware counter values over those of a baseline execution. The goal is to correlate the relative performance improvement of differently optimized code with their impact on individual architectural components such as L1/L2/L3/TLB misses, load imbalance, and synchronization overhead. The derived formulas give indications of possible performance bottlenecks. Therefore they can be combined with other knowledge to enable more effective performance tuning, e.g., by pruning optimizations that are not likely to make a difference. To properly construct these formulas, we follow the following three steps.

Selection of hardware events to monitor.

Fig. 1 shows a subset of the hardware counters that we have monitored on the 8-core Intel *Nehalem* machine during multiple runs of each optimized code. We then correlate the hardware counter values with the corresponding overall execution time (minimum across multiple runs) using Spearman's rank correlation algorithm. Finally, hardware counters that have a relatively high correlation and are directly impacted by our optimizations are selected. The hardware counters (e.g., L1/L2/L3/TLB misses) which can be used to meaningfully guide compiler optimizations are given higher priority than other opaque hardware counters (e.g., UOPS issued stalls). Cumulative hardware counters (e.g., total cycle count and resource stall cycles) that overlap with other higher priority counters are not selected in spite of their high correlation with the overall performance, so that the higher-priority counters are not overshadowed by the cumulative counters. Multiple hardware counter values may be combined to model special architectural events that are not directly monitored by existing hardware counters. For example, *total cycles* and *thread unhalted cycles* can be combined to return the *idle cycle count* of each thread when modeling parallelization speedup, shown in Figure 13.

While modern architectures typically provide a large number of different hardware counters for performance monitoring, selecting a reasonable subset of relevant events, e.g., L1/L2/L3 cache misses, is not difficult given a computational kernel and the set of possible optimizations being in-

Notation	Optimizations applied	Parameters	Parameter range
naive	Original code. No optimization	none	none
blocked	Blocking within a single time step	tile size	[4-512, 4-512]
timeskewed	Blocking via time skewing	skew factor	[4-512, 4-512, 4-512]
blocked_par	Blocking+single-sweep parallel.	tile size	[4-512, 4-512]
pipelined_par	Timeskewing+pipeline parallel.	skew factor	[4-512, 4-512, 4-512]
wavefront_par	Timeskewing+wavefront parallel.	skew factor	[4-512, 4-512, 4-512]

Table 1: Different optimization strategies

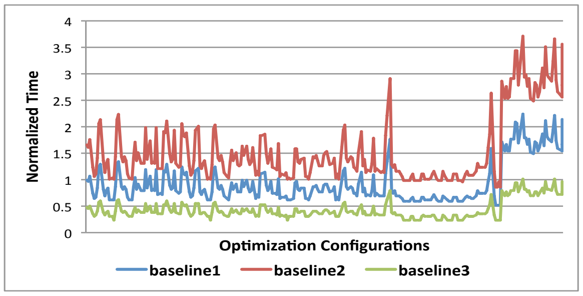


Figure 8: Performance of 7-point Jacobi normalized using different baseline executions

vestigated. The events do not have to chosen exactly right to derive meaningful formulas from empirical data.

Estimation of correlation coefficients.

To reason about optimization effectiveness, we normalize both the overall execution time and the hardware counter values over those of a baseline execution. For each kernel, when modeling sequential performance, we use runtime statistics of the *naive* version (see Tab. 1) as the baseline; for parallel performance models, we use that of a randomly chosen parallelized code as the baseline.

To demonstrate that different selections of the baseline does not affect the accuracy of the resulting performance models, Fig. 8 shows the normalized execution time of differently optimized 7-point jacobi kernel against different baselines. Note while the degree of performance variation may change when using different baselines, the relative variation patterns remain the same. Similarly, while the absolute correlations between the overall performance and the individual hardware counters may vary when using different baselines, their relative variation pattern remain constant.

We use multivariate Ordinary Least Square (OLS) regression to determine the linear coefficient for each normalized hardware counter value. Two performance formulas are derived for each kernel: one for modeling the sequential performance of the *blocked* and *timeskewed* optimizations, and the other for modeling the parallel performance of the *block_par*, *pipeline_par*, and *wavefront_par* optimizations. The resulting performance formulas are shown in Figures 10 and 13.

Putting things together.

To verify the accuracy of our performance models, we use 2/3 of the empirical data to derive the performance formulas and then use the formulas to predict the actual execution time of the other 1/3 of execution samples. Figures 11 and 14 present the verification results. From these figures, we see that our performance formulas are highly accurate and can be used to precisely model the overall performance impact of different optimizations.

5. SEQUENTIAL PERFORMANCE

We experimented with two main optimizations for sequential stencil kernels: single-sweep blocking and time-skewing. The choice of blocking and skewing factors is critical to the effectiveness of these optimizations. Therefore, for each kernel, we generate a large collection of alternate variants with different tile sizes and skew factors.

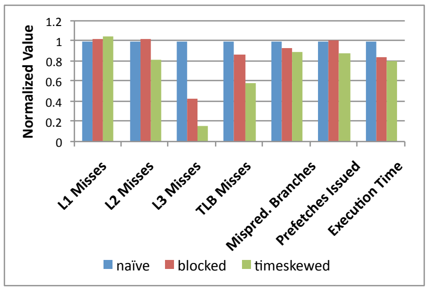
5.1 Comparing Optimizations

Fig. 9 presents best tuned performance results for both the blocked and timeskewed optimizations through an exhaustive search of the parameter space. The numbers reported in these figures are normalized to those of the unoptimized *naive* implementation of each kernel.

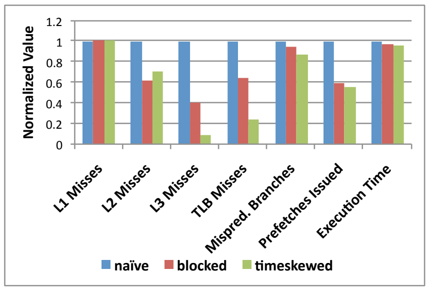
In terms of overall speedup, time-skewing has a slight advantage over single-sweep blocked code. For all three kernels the best performing variant is the one that applies time-skewing. This is not surprising, since the timeskewed versions exploit locality across the time dimension as well as the spatial dimension. This is indicated by the significant reduction in L3 misses for timeskewed codes for both `jacobi7` and `jacobi27`. For `gauss` the reduction happens for L2 misses, as seen in Fig. 9(c). In addition to the reduced cache misses, the timeskewed variants also exhibit better TLB locality. Interestingly, however, the reductions in cache and TLB misses do not always translate into a proportional gain in performance. For example, for `jacobi27`, there is almost an 80% reduction in L3 cache miss rates for the timeskewed code over the blocked code. There is also a huge reduction in the number of TLB misses. However, the performance gain is minimal. Part of this anomaly is explained by the increased L2 misses for the timeskewed code and part of it is explained by the computational intensity of `jacobi27`. These issues are hard to predict through pure analytical models or empirical tuning based solely on raw execution time. This makes a case for using HW counter based models to discover the actual impact of optimizing different hardware components. Such information can then be combined with application-specific knowledge of optimizations to better understand their performance.

5.2 Performance Models

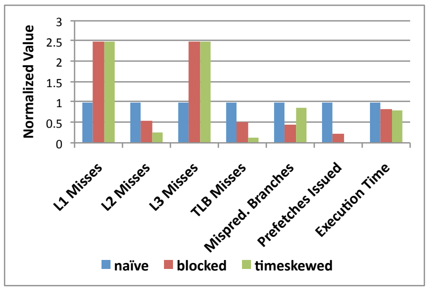
Fig. 10 shows the sequential performance models we derived using the methodology described in Section 4. The performance counters we include in these models are L1 data cache misses , L2 cache misses , L3 cache misses, TLB Misses and number of L2 hardware prefetches issued. Also, since both time skewing and blocking directly impact the branch predictor, we consider mis-predicted branches as one of the key architectural events. The R^2 values for the regression models for `jacobi7` and `jacobi27` are 93% and 94%, respectively. For `gauss` it is slightly lower, at 88%. Thus, each model provides a good fit to the data. To further verify the accuracy of the constructed performance models, we generated 200 *additional* blocked and timeskewed code variants (about 33% of the original sample size used in the model). Fig. 11 shows the results of applying our models to these variants, where each model provides a good fit for the per-



(a) 7-point Jacboi



(b) 27-point Jacobi



(c) 7-point Gauss Seidel

Figure 9: Performance of different sequential optimizations with best tuned configurations

$$\begin{array}{rcl} Time & = & 0.25 + 0.10 * L1\ Misses \\ & + & 0.18 * L2\ Misses \\ & + & 0.28 * L3\ Misses \\ & - & 0.0008 * TLB\ Misses \\ & - & 0.016 * Misp.\ Branches \\ & - & 0.014 * HW\ Prefetches \end{array}$$

(a) 7-point Jacboi

$$\begin{array}{rcl} Time & = & 0.22 + 0.57 * L1\ Misses \\ & - & 0.01 * L2\ Misses \\ & + & 0.03 * L3\ Misses \\ & - & 0.002 * TLB\ Misses \\ & - & 0.003 * Misp.\ Branches \\ & - & 0.154 * HW\ Prefetches \end{array}$$

(b) 27-point Jacobi

$$\begin{array}{rcl} Time & = & 0.91 - 0.09 * L1\ Misses \\ & + & 0.0005 * L2\ Misses \\ & + & 0.05 * L3\ Misses \\ & + & 0.007 * TLB\ Misses \\ & - & 0.001 * Misp.\ Branches \\ & + & 0.13 * HW\ Prefetches \end{array}$$

(c) 7-point Gauss Seidel

*Both the execution time and hardware counters are normalized against a base line execution of the *naive* code

Figure 10: Performance Models for Sequential Stencil Kernels

formance curve for all three kernels. Except for a few points where the performance is really poor, the model is able to determine performance with almost perfect accuracy.

Analyzing the coefficients of the performance counters, we observe that for `jacobi7`, L3 cache misses have the most significant impact, which is supported by the performance results shown in Fig. 9(a). The tuned timeskewed version of `jacobi7` provides a 23% reduction in execution time over the baseline version, where close to an 83% reduction in L3 miss count is observed. The blocked version of `jacobi7` performs slightly worse than the timeskewed version. But again, the most significant gains come from reduced L3 misses. According to the model, L1 and L2 misses also play an important role but their impact is not as significant as L3 misses.

The performance model for `jacobi27` is in Fig. 10(b) and indicates that the impact of L1 cache misses is most significant. Here `jacobi27` accesses 27 different points and performs 30 floating operations during each update operation. Thus, this kernel is more computationally intensive than `jacobi7`. For this reason L2 and L3 cache misses have less performance impact than L1 misses, and as shown in Fig. 9(b), reduction in L2 and L3 misses does not lead to a corresponding decrease in execution time. This situation is captured by the small coefficients for the L2 and L3 cache miss parameters in our model. None of the block sizes or skew factors that we tried for `jacobi27` was able to reduce the L1 cache misses (although some variants increased them). As a result even the best tuned version of `jacobi27` shows very little performance improvement. For `gauss`, the most significant parameter is the number of issued HW prefetch instructions. Surprisingly, this event has a negative impact on performance (i.e., the more HW prefetching instruction issued, the worse the performance). We observe in Fig. 9(c) the best performing versions of both the blocked and timeskewed codes have the least number of HW prefetches issued. For this code, we also see a large increase in the L3 cache misses. However, the performance loss due to the increase in L3 misses is offset by the reduction of number of HW prefetches issued. The much larger coefficient of the HW prefetching parameter in our model explains this phenomenon.

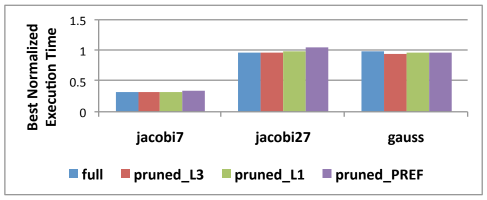
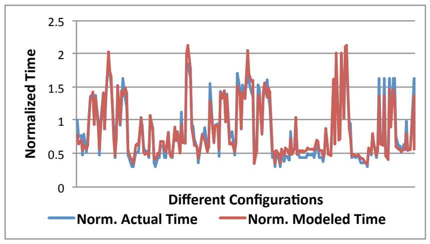


Figure 12: Performance of Pruned Search Spaces

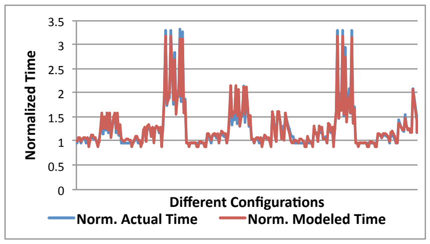
5.3 Implications for Optimization Guidance

In addition to characterizing performance of stencil codes, we aim to use the constructed performance models to guide optimization and tuning. To this end, we conduct a set of search-based experiments where we use our model to reduce the size of the search space. The search space for these experiments consists of 800 points of blocked and timeskewed code variants for each kernel. To prune the search space, we take the most significant performance parameters from each model (L3 for `jacobi7`, L1 for `jacobi27` and HW Prefetch for `gauss`). We then sort the entire search space using each parameter and take the top k values as our search space. We apply random search on this pruned search space for i iterations and compare the performance to the performance of a random search of i iterations on the entire search space.

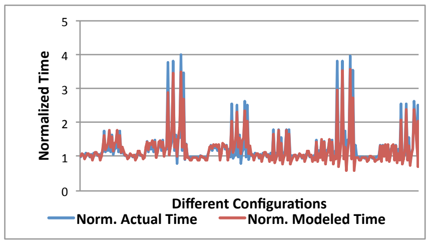
Fig. 12 shows the results of these experiments, where $k = 50$ and $i = 10$. In the figure, `pruned_L3` refers to the pruned search space generated using the best L3 cache miss values, whereas `pruned_L1` and `pruned_PREF` refer to the pruned search spaces for L1 misses and HW prefetch, respectively. In each case the performance obtained through exploring the pruned search space is comparable to that obtained from a random search of the entire space. For `gauss`, searching the pruned space of L3 and L1 misses actually leads to better performance than searching over the entire space. The pruned space for HW prefetching leads to slightly worse performance for `jacobi7` and `jacobi27`. However, this difference is small. Although our choice of k and i values for this experiment is somewhat arbitrary, the results do suggest that many of the best values within the search space are concentrated in areas where the values of



(a) 7-point Jacboi



(b) 27-point Jacobi



(c) 7-point Gauss Seidel

Figure 11: Verification of sequential model by comparing normalized actual and modeled execution time

the most significant performance parameters are best. Thus, the constructed models can be effectively utilized in picking out representative search spaces when tuning stencil kernels.

6. PARALLELIZATION PERFORMANCE

We experimented with three parallelization optimizations, *blocked_par*, *pipelined_par*, and *wavefront_par*, shown in table 1. Since *blocked_par* is illegal for *gauss*, we do not investigate this strategy for this kernel. Since *Nehalem* has 8 cores, we parallelized all kernels using eight threads.

6.1 Performance Models

Fig. 13 shows the formulas we attained via regression analysis of the empirical data collected when applying the three parallelization schemes with different configurations in Table 1. In addition to the set of hardware counters selected to model sequential performance in Figure 10, two extra hardware events are selected: *off-core requests*, which model the amount of communication across different processing cores; and *thread idle cycles*, a derived hardware event which monitors the load imbalance of parallelization, i.e., the number of clock cycles that each thread spent sitting idle while waiting for its task to get ready. *Thread idle cycles* are computed by subtracting the *total clock cycles* with the unhalted cycles for each thread, both monitored via hardware counters. All the runtime statistics are normalized to a baseline of a randomly selected parallelized code for each kernel.

The accuracy of the formulas is verified in Fig. 14, where we used the derived models to predict the execution time of around 200 additional randomly generated parallelization variants. The predicted execution time has matched almost precisely the actual time for all cases. The R^2 values for the regression models are around 98%, which are quite high.

Based on the performance formulas, for all three kernels, the number of L3 cache misses and *off-core* requests are critical factors determining the overall performance of the parallelized code. The impact of L1 cache misses ranges from significant in *jacobi*27 to minor in *jacobi*7, indicating the computation is becoming less memory-bound. On the other hand, L2 cache misses have surprisingly played a negative role in the overall performance, where the formulas seem to indicate that the more L2 misses incurred, the better the performance. We suspect the possible reason for this is that the extra L2 misses can serve as informed prefetches for the other threads, which share a significant amount of data due to the nature of our stencil parallelization strategies. The hardware prefetches, however, have been consistently degrading performance as data are not fetched continuously in the parallelized codes. Note that although the impacting factor of *thread idle cycles* is low (ranging from 0.013 to 0.12), it is a critical reason explaining the poor performance of some parallelized code where serious load imbalance has essentially sequentialized all the threads. Some of these cases can be observed in Figure 15.

6.2 Comparing Parallelization Strategies

Figure 15 compares the best performance achieved by different parallelization strategies after an exhaustive search of the configuration space. For all kernels, the best performance achieved by the first parallelization scheme is used as a baseline to normalize the performance of other strategies.

For *jacobi*7, the *pipeline_par* strategy achieves the best overall performance. This strategy parallelizes multiple sweeps of timeskewed blocks in a pipelined fashion, treating each CPU core as a pipeline stage by permanently binding a thread to it. It is able to significantly outperform the *blocked_par* version by dramatically reducing both L3 cache misses and off-core requests, the two most significant hardware counters that impact the overall performance. In contrast, although *wavefront_par* follows a very similar parallelization strategy as *pipeline_par*, it restarts a new wave of different threads to evaluate each new set of time-skewed blocks simultaneously. As different cores may process a time-skewed block at different wave iterations, poor data locality is resulted. This is reflected in the dramatic increase of L3/TLB misses and off-core requests, and as a result *wavefront_par* performed significantly worse than both *blocked_par* and *pipeline_par*.

For *Jacobi*27, the relative performance of different parallelization schemes is similar to that of *jacobi*7, except that *pipeline_par* achieves similar performance as that of *blocked_par* in spite of significantly reducing both L3 misses and off-core requests. Here because *pipeline_par* has increased *thread idle cycles* by a factor of 50 (in contrast to a factor of 13 for *jacobi*7), the serious load imbalance has reduced the parallelization efficiency. For the same reason the performance of *wavefront_par* is much worse. For 7-point Gauss-Seidel, *pipeline_par* has dramatically outperformed the *wavefront_par* strategy due to similar reasons.

6.3 Guiding Optimization Tuning

From studying the performance models in Fig. 13, the most important factors in achieving high parallelization efficiency is improving L3 cache locality, reducing off-core requests, and keeping load imbalances under control. Fig. 16 shows how we might use such knowledge to guide the optimization tuning of these stencil kernels. In particular, for each kernel, we compare four hypothetical search strategies to explore the optimization configuration space, where each strategy always tries to improve the efficiency of a single architectural component. For example, the L3-misses based strategy takes random samples from an underlying exhaustive search and selects only those points where the number of L3 cache misses are reduced; that is, the search strategy will

$$\begin{aligned} Time &= 0.077 + 0.001 * L1\ Misses \\ &- 0.14 * L2\ Misses \\ &+ 0.44 * L3\ Misses \\ &- 0.002 * TLB\ Misses \\ &+ 0.021 * Misp.\ Branches \\ &+ 0.05 * HW\ Prefetches \\ &+ 0.5 * Off\text{-}core\ Requests \\ &+ 0.013 * Idle\ Cycles \end{aligned}$$

(a) 7-point Jacboi

$$\begin{aligned} Time &= 0.054 + 0.39 * L1\ Misses \\ &- 0.15 * L2\ Misses \\ &+ 0.25 * L3\ Misses \\ &+ 0.008 * TLB\ Misses \\ &+ 0.013 * Misp.\ Branches \\ &+ 0.15 * HW\ Prefetches \\ &+ 0.25 * Off\text{-}core\ Requests \\ &+ 0.016 * Idle\ Cycles \end{aligned}$$

(b) 27-point Jacobi

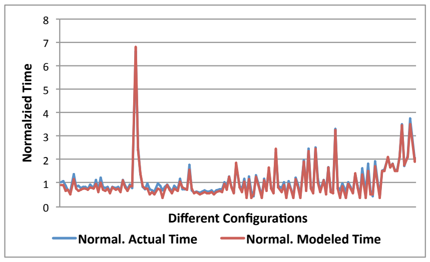
$$\begin{aligned} Time &= 0.157 + 0.13 * L1\ Misses \\ &- 0.57 * L2\ Misses \\ &+ 0.19 * L3\ Misses \\ &- 0.001 * TLB\ Misses \\ &- 0.03 * Misp.\ Branches \\ &+ 0.09 * HW\ Prefetches \\ &+ 0.89 * Off\text{-}core\ Requests \\ &+ 0.12 * Idle\ Cycles \end{aligned}$$

(c) 7-point Gauss Seidel

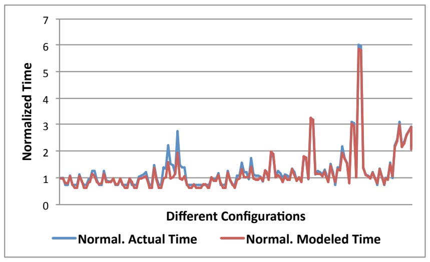
*Idle Cycles = Total Cycles - Unhalted Cycles

All runtime statistics are normalized to a baseline execution of a randomly selected parallelized code for each kernel

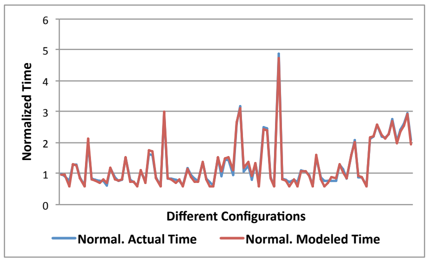
Figure 13: Parallel Performance Models



(a) 7-point Jacboi



(b) 27-point Jacobi



(c) 7-point Gauss Seidel

Figure 14: Verification of parallel model by comparing normalized actual and modeled execution time

not attempt any optimization configuration that incur more L3 cache misses than those incurred by the best configurations found so far. Other hypothetical search strategies are derived similarly and focus on optimizing L1 misses, off-core requests, and thread idle cycles respectively.

From Fig. 16, for all three kernels, tuning for L3 cache misses and off-core requests yield the best result and require the fewest tuning samples to find a reasonably good optimization configuration. Tuning for off-core requests incurs more performance variations but could find better optimizations in the end. Tuning for L1 cache misses yield unpredictable results. Tuning for thread idle cycles can converge quickly but may miss out better optimizations.

The results in Fig. 16 have confirmed the effectiveness of applying optimizations to target the most significant architectural components. However, different hardware events may become the most significant when applying different optimizations and tuning different benchmarks. Regression analysis is therefore necessary to build offline models to guide optimization tuning of different benchmarks.

7. CONCLUSIONS AND FUTURE WORK

This paper presents a performance study of both memory system and parallelization optimizations for stencil computations on modern multi-core architectures. Based on an extensive collection of empirical data, we develop a set of formulas via regression analysis to model the overall performance of differently optimized code in terms of their impact on various hardware performance counters. We apply our methodology to three stencil computation kernels, 7-point jacobi, 27-point jacobi, and 7-point Gauss-Seidel iterations. Our experimental results show that precise formulas can be developed for each kernel to model the overall performance impact of varying optimizations and thereby effectively guide the optimization and tuning of these kernels.

Both the selection of hardware events and the coefficients of the formulas are determined by the performance characteristics (e.g., memory vs compute bound) of an application. Therefore distinct computational kernels typically have different performance formulas. As optimizations are applied, the performance bottleneck may change and so may the performance formulas. It is our future work to model the transitions of application behavior more comprehensively via more advanced regression analysis methodologies.

8. REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCtoolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience, To Appear*, 2009.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–113, New York, NY, USA, 2008. ACM.

[4] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

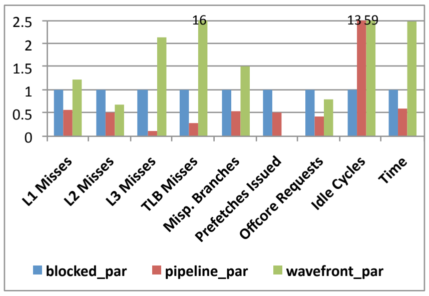
[5] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, March 2005.

[6] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart. Parallel data-locality aware stencil computations on modern micro-architectures. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[7] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

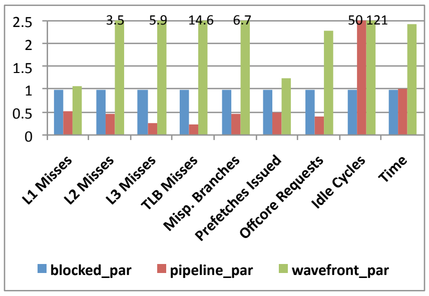
[8] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC08)*, 2008.

[9] S. Eranian. What can performance counters do for memory subsystem analysis? In *MSPC '08: Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness*, pages 26–30, 2008.



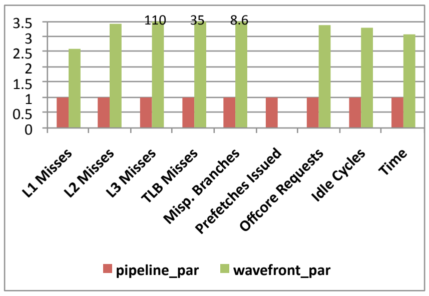
(a) 7-point Jacobi

*Baseline parallelization speedup: 5.2



(b) 27-point Jacboi

*Baseline parallelization speedup: 7.4

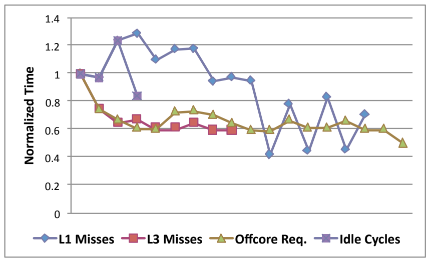


(c) 7-point Gauss-Seidel

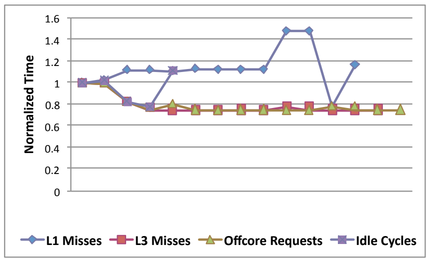
*Baseline parallelization speedup: 7.8

* All parallelized codes are evaluated using 8 threads, each thread occupying a different processing core.
All values are normalized against the best performance achieved by the first parallelization strategy for each kernel.

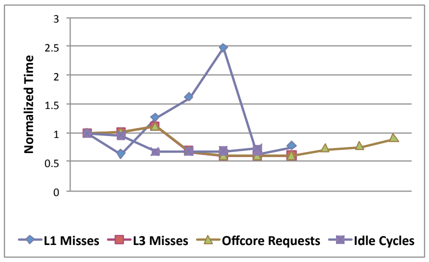
Figure 15: Parallel performance achieved by different optimizations with the best configurations



(a) 7-point Jacboi



(b) 27-point Jacobi



(c) 7-point Gauss Seidel

Figure 16: Using hardware counters to guide the search of best optimizations

[10] B. Fraguela, Y. Voronenko, and M. Puschel. Automatic tuning of discrete fourier transforms driven by analytical modeling. In *PACT'09: Parallel Architectures and Compilation Techniques*, Raleigh,NC, Sept. 2009.

[11] Intel Pentium 4 Processor Optimization Reference Manual. Intel Corporation, 2000.

[12] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 2010.

[13] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60, New York, NY, USA, 2006. ACM.

[14] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235–244, 2007.

[15] L. Liu and Z. Li. Improving parallelism and locality with asynchronous algorithms. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 213–222, New York, NY, USA, 2010. ACM.

[16] G. Marin and J. Mellor-Crummey. Pinpointing and exploiting opportunities for enhancing data reuse. In *In Proceedings of the 2008 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'08)*, 2008.

[17] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT07)*, 2007.

[18] S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *HIPEAC:High-Performance and Embedded Architectures and Compilers (to appear)*, Heraklion, Greece, Jan 2011.

[19] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 32, Washington, DC, USA, 2000. IEEE Computer Society.

[20] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37(2):46–55, 2009.

[21] Song, Yonghong, and Z. Li. New tiling techniques to improve cache temporal locality. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 215–228, New York, NY, USA, 1999. ACM.

[22] F. Song, S. Moore, and J. Dongarra. Feedback-directed thread scheduling with memory considerations. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, 2007.

[23] Y. Song, R. Xu, C. Wang, and Z. Li. Data locality enhancement by memory reduction. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, June 2001.

[24] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *13st International Conference on High-Performance Computer Architecture (HPCA-13)*, 2007.

[25] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *roceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPOPP09)*, 2009.

[26] M. M. Tikir and J. K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC*, 2004.

[27] J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *Journal of Computational Science*, In Press, 2011.

[28] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS00)*, page 171, Washington, DC, USA, 2000. IEEE Computer Society.

[29] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, Mar 2007.