SMT-Aware Instantaneous Footprint Optimization

Probir Roy, Xu Liu
Department of Computer Science
College of William and Mary
{proy, xl10}@cs.wm.edu

Shuaiwen Leon Song
High Performance Computing Group
Pacific Northwest National Laboratory (PNNL)
Shuaiwen.Song@pnnl.gov

ABSTRACT

Modern architectures employ simultaneous multithreading (SMT) to increase thread-level parallelism. SMT threads share many functional units and the entire memory hierarchy of a physical core. Without a careful code design, SMT threads can easily contend with each other for these shared resources, causing severe performance degradation. Minimizing SMT thread contention for HPC applications running on dedicated platforms is very challenging because they typically spawn threads within Single Program Multiple Data (SPMD) models. Since these threads have similar resource requirements, their contention cannot be easily mitigated through simple thread scheduling. To address this important issue, we first vigorously conduct a systematic performance evaluation on a wide-range of representative HPC and CMP applications on three mainstream SMT architectures, and quantify their performance sensitivity to SMT effects. Then we introduce a simple scheme for SMT-aware code optimization which aims to reduce the memory contention across SMT threads. Finally, we develop a lightweight performance tool, named *SMTAnalyzer*, to effectively identify the optimization opportunities in the source code of multithreaded programs. Experiments on three SMT architectures (i.e., Intel Xeon, IBM POWER7, and Intel Xeon Phi) demonstrate that our proposed SMT-aware optimization scheme can significantly improve the performance for general HPC applications.

Categories and Subject Descriptors

C.4 [Performance of systems]: Measurement techniques, Performance attributes; D.2.8 [Metrics]: Performance measures.

Keywords

SMT, Memory Hierarchy, Instantaneous Footprint, SMT-Aware Optimization, Locality, Performance Tools

1. INTRODUCTION

Modern architectures employ simultaneous multithreading (SMT, also known as fine-grained threads) [37] to support many hardware threads. These SMT-based architectures have become increasingly popular, leveraged by seven of the top ten supercomputers in the world today [1]. Unleashing their maximum computing power has become critical for the future missions towards Exascale computing.

In this paper, we refer to the hardware threads on the same physical core as *SMT threads*. SMT threads can avoid CPU stalls and improve the throughput of a physical core. If one SMT thread stalls in execution pipeline, instructions from other SMT threads can be switched in so that the execution can continue without additional overhead. With the trend of increasing thread-level parallelism on chip, more SMT threads have been integrated to a single physical core. For example, Intel Xeon Phi [16] has four SMT threads per core and IBM POWER8 [13] has eight SMT threads per core, both of which double the SMT thread number compared to their previous generation.

Unlike threads on different physical cores, SMT threads share many functional units in the execution pipeline such as instruction issuing slots, load store queues, integer and floating units. Moreover, they share the entire memory subsystem including all levels of cache, prefetchers, and bandwidth. Without a careful code design, SMT threads can easily contend for these shared resources, causing significant performance degradation.

Previous studies [38, 7, 25, 30, 8] analyzed the performance impact of resource sharing between SMT threads. To minimize resource contention, they characterize thread execution and schedule them to the same physical core if they have different resource requirements. This co-scheduling approach works well for mixed workloads running on an SMT architecture, because their threads usually have different resource requests. However, this scheme does not work well for HPC applications which usually run on a dedicated system without sharing resources with co-running applications. Moreover, HPC applications often use Single Program Multiple Data (SPMD) models which create threads with similar characteristics and resource requests. Therefore, such simple thread co-scheduling cannot minimize SMT thread contention for HPC applications that possess similar resource requirements across threads.

In this paper, we propose an SMT-aware optimization scheme to reduce the contention among SMT threads at the source code level for HPC applications. The underlying intuition in our code optimization technique is that threads should collaboratively work on a small data set rather than

©2016 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the United States Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. *HPDC'16*, May 31–June 04, 2016, Kyoto, Japan.
© 2016 ACM 978-1-4503-4314-5/16/05 ...\$15.00.
DOI: <http://dx.doi.org/10.1145/2907294.2907308>.

on a large or multiple data sets. This can be achieved by altering the typical parallel strategy widely used in modern threading models, such as OpenMP [28] and Pthread [4]. Our novel optimization method contrasts and complements the existing approaches based on thread scheduling by developing the first code transformation technique intended to improve the performance for HPC codes running on SMT architectures.

The application of a practical SMT-aware optimization has three major challenges. First, to apply appropriate optimizations, we need to be able to identify the major performance bottlenecks in an SMT architecture. Second, the optimization itself should not be overly complex and should be easy for developers or compilers to adopt and apply. Third, there must be some accompanying technique to explicitly, automatically and accurately highlight the locations in source code where such an optimization should take place in order to achieve the best performance. To address these challenges, this paper makes the following contributions:

- We systematically evaluate the performance impact of SMT on a wide-range of representative HPC and CMP applications on three mainstream SMT architectures. We derive new latency-based metrics to quantify their performance in the memory subsystem. Based on the analysis, we discover that SMT shows inconsistent memory behaviors across applications.
- We introduce a simple but effective code transformation scheme to reduce the contention among SMT threads in the memory hierarchy. To the best of our knowledge, this is the first work on *unconventional* code-level optimization for SMT thread execution. Specifically, we introduce *false sharing* between SMT threads to improve the inter-thread locality.
- We develop a performance tool named *SMTAnalyzer* that can help identify SMT optimization opportunities in existing multithreaded programs. SMTAnalyzer monitors execution of unmodified binaries, collects performance data with low overhead, and provides insights to guide SMT-aware optimizations.

We evaluate our proposed optimization scheme on three modern SMT architectures: Intel Xeon, Intel Xeon Phi, and IBM POWER7. Under the guidance of SMTAnalyzer, we are able to significantly improve the performance for several widely-used and highly-optimized HPC applications.

We organize this paper as follows: Section 2 explores the performance impact of SMT on various applications. Section 3 describes our SMT-aware optimization scheme. Section 4 illustrates the design and implementation of SMTAnalyzer. Case studies under the guidance of SMTAnalyzer are demonstrated in Section 5. Section 6 reviews previous work and distinguishes our approach. We conclude this work in Section 7.

2. PERFORMANCE IMPACT OF SMT: OBSERVATIONS AND ANALYSIS

In this section, we study the performance impact of SMT by analyzing the execution behaviors of parallel applications on modern SMT architectures. To better quantify such impact, we investigate a broad spectrum of 31 widely-used and often highly-optimized benchmarks from chip multiprocessing (CMP) research (i.e., PARSEC [3]), popular HPC

Figure 1: Runtime Ratio (SMT/NONSMT) for applications on Intel SandyBridge, Intel Xeon Phi, and IBM Power7. Two PARSEC benchmarks encounter compilation issues: *facesim* on Xeon Phi and *canneal* on Power7. They are shown as 0 in the figure.

(a) Average memory access latency (L) swing from NON-SMT to SMT on Intel Xeon.

(b) SMT Scaling Factor ($F = L_{SMT}/L_{NONSMT}$)

Figure 2: Memory performance affected by SMT effects on Intel Xeon: (a) Average memory access latency (L) swing from NON-SMT to SMT; (b) SMT Scaling Factor (F).

suites (i.e., Rodinia [5] and NPB [2]), and HPC application benchmarks (i.e., Sequoia Benchmark [18]). They cover a variety of execution patterns including data sharing, locality, inter-thread communication, synchronization, and on-/off-chip traffic. We list all the benchmarks that we investigated from each suite as the following:

- Rodinia Benchmarks: *bp*, *bfs*, *b+tree*, *cfd*, *heartwall*, *hotspot*, *kmeans*, *lavaMD*, *leucocyte*, *lud*, *myocyte*, *needle*, *nn*, *particle_filter*, *pathfinder*, *srad*, *streamcluster1*, *hotspot3D*.
- Sequoia Benchmarks: *Clomp*, *lulesh2.0*, *IRSmk*.
- PARSEC Benchmarks: *blackscholes*, *bodytrack*, *canneal*, *facesim*, *ferret*, *fluidanimate*, *streamcluster2*, *swaptions*.
- NAS Benchmark Suite: *LU*.
- Microbenchmarks: *3D tensor*, *Stencil*.

We evaluate these benchmarks on three SMT architectures that are commonly deployed in modern HPC systems: Intel Xeon (SandyBridge), Intel Xeon Phi, and IBM Power7.

Processor Type	SMT per core	caches shared by SMT per core	L1	L2	L3	Memory
Intel Xeon E5-4650 2.70GHz,	2	\$L2, \$L1	32KB private	256KB private	20MB/8cores shared	64GB, 8 cores 51.2 GB/s
Intel Xeon Phi 1.24GHz	4	\$L2 (Ring Structure), \$L1	32KB private	512KB private	No L3	16GB, 61 cores 352 GB/s
IBM Power7 3.7 GHz	4	\$L2, \$L1	64KB private	4MB private	32MB/8cores shared	64GB, 8 cores 68.2 GB/s

Table 1: Architecture features of the three platforms under investigation.

(L, F)	Benchmarks	Characterization
(high, high)	srad, streamcluster1, Lulesh2.0, IRSmk, LU, 3D tensor, Stencil, streamcluster2, hotspot, Clomp	potentially sensitive to mem-centric SMT optimizations
(high, low)	lud, needle, bfs, nn, bp, canneal, ferret	not clear if they can further benefit from SMT optimizations
(low, high)	leucocyte, heartwall, pathfinder, myocyte	little benefit from mem-centric SMT optimization
(low, low)	b+tree, cfd, kmeans, lavaMD, particle.filter, hotspot3D, blackscholes, bodytrack, facesim, swaptions	good memory performance with SMT enabled

Table 2: Categorization based on the degree of potential performance benefits from reducing the SMT-induced overhead.

Each of them has different architectural features and SMT capabilities, as shown in Table 1. To prevent the non-uniform memory access (NUMA) impact from interfering with our observations, we only experiment on one socket in the IBM Power7 and Intel Xeon platforms (i.e., both have 8 cores per socket). Random noise in the results is reduced through the average measurement of repetitive runs.

2.1 Runtime Ratio Analysis

We first quantify the overall performance impact of applying SMT on benchmarks using a general runtime ratio which is calculated as $\frac{Runtime_{SMT}}{Runtime_{NONSMT}}$. We run the benchmarks on all the three SMT architectures to evaluate their SMT impact, shown in Figure 1. We compute the runtime ratio for each benchmark running with their maximum SMT capability.

From the figure, we observe that a majority of the benchmarks are not close to achieving their expected speedups on all three architectures when SMT threads are enabled. In many cases, they gain no speedup or even perform worse. For instance, on the Intel SandyBridge platform, the average runtime ratio across 31 applications is 1.038 while doubling the threads. Only 18 out of 31 benchmarks achieve some degree of speedup, 13% on average. Among them, **swaptions** is the only benchmark that achieves close to 2x speedup. The other 13 benchmarks either exhibit little to no performance improvement, or have significantly worse performance than the baseline (e.g., **hotspot**, **srad**, **IRSmk**, **Lulesh**, **Stencil** and **streamcluster2**). Although the root causes of this negative performance impact could be a combination of factors at both the computation (e.g., competing for shared resources in execution pipelines) and memory levels, this work primarily focuses on addressing the bottlenecks in the *memory hierarchy* (i.e., private and shared layers) introduced by the SMT effects, which include both suboptimal data locality and scalability issues.

2.2 General Memory Performance Analysis

We further investigate how SMT impacts different benchmarks in the memory hierarchy. We apply the *average memory access latency* as the metric for evaluating the memory-level bottlenecks caused by the SMT effects. We prefer the latency-based metric over hit rate because it reflects the performance change more accurately for SMT architectures which are typically latency-oriented. For example, even if one case has much worse overall performance than the other due to its significant L1 bandwidth contention, they can still have the same L1 hit rate.

We denote the *average memory access latency* for an application as L . L can be calculated as the aggregate latency for accessing the memory subsystem divided by the number of memory accesses, i.e., $L = (\sum_{i=1}^n L_i)/n$, where n represents the number of memory accesses. To obtain an accurate estimation of L , precise and lightweight measurement mechanism is required. Among the three SMT architectures, only Intel SandyBridge provides such capability through Precise Event-Based Sampling (PEBS) [14] (see Section 4.1 for more discussion). We use the aggregate sampled latency divided by the number of memory samples to approximate L .

Figure 2(a) shows the values of L with SMT enabled (L_{SMT}) and with SMT disabled (L_{NONSMT}) for 31 benchmarks. We can observe that L_{NONSMT} for some benchmarks are already high (e.g., **bp**, **canneal**, **needle**, **IRSmk** and **hotspot**), indicating significant memory-level bottlenecks before applying SMT. Additionally, enabling SMT introduces higher L for 17 benchmarks, 10 of which are observed with an over 10% L increment. More interestingly, applying SMT actually reduces L for the other 14 benchmarks, indicating that the data locality between SMT threads on the same core (i.e. inter-thread sharing) is improved and this performance benefit even surpasses resource contention in the memory hierarchy. This unique finding inspires our SMT-aware optimization on memory in Section 3.

2.3 Benchmark Categorization

To understand which benchmark can potentially benefit from reducing SMT-induced overhead, we adopt the differential analysis theory from [23] to construct a metric describing SMT effects with different focus and evaluation standards. We name it *SMT Scaling Factor* (F), where $F = \frac{L_{SMT}}{L_{NONSMT}}$. Figure 2(b) shows the F value for each benchmark.

We combine L_{NONSMT} , short as L , and F to create an evaluation tuple (L, F) for categorizing the benchmarks in terms of SMT-optimization sensitivity, shown in Table 2. In this table, benchmarks are classified into four categories based on their (L, F) values: (low, low), (low, high), (high, low) and (high, high). For the Intel Xeon architecture, we treat L as high if it exceeds 10 cycles (Figure 2(a)) since the average memory access latency with no contention for

Figure 3: Three parallel execution schemes: (a) Default parallel execution: all threads work on different data to enjoy “embarrassingly parallelism”; (b) Reducing instantaneous footprints through intra-thread optimization: data decomposition; (c) Reducing instantaneous footprints through inter-thread optimization: thread collaboration.

the architecture is around 7 cycles. We also treat F as low if $F < 1.05$. The category (low,low) indicates the scenario that SMT does not introduce memory bottlenecks. The category (low,high) indicates that there is probably little benefit from further SMT-oriented optimization, since L is too low. Benchmarks in (high, high) category not only perform worse with SMT threads but also have the high base latency L , which makes them good candidates for SMT-aware optimizations. The category (high, low) includes several interesting cases, which either are sensitive to SMT and their L s are improved with SMT (e.g., **bp**, **canneal**, **ferret**); or react little to the SMT effects (e.g., **lud**, **needle**, **bfs**, and **nn**) but still with high L . It is unclear if these applications can further benefit from SMT-aware optimizations, requiring a deeper investigation.

Based on this categorization, we narrow our focus down to only the benchmarks from (high, high) and (high, low) categories, since their performance can be potentially improved by reducing SMT-induced contention in the memory hierarchy. Note that although the accurate (L, F) values cannot be directly obtained for benchmarks running on IBM Power7 and Intel Xeon Phi due to the profiling limitations in hardware (refer to Section 4.1), coarse-grained approximation using performance counters suggests a similar categorization for (high, high) and (high, low) with **ferret**, **IRSmk**, and **streamcluster2** moving in between the two categories.

2.4 Fine-Grained Analysis in Memory Layers

Table 3 shows the latency inflation in different memory layers on the three architectures for the benchmarks falling into the (high, high) category. The table confirms that SMT effects result in significant memory-level contention for these benchmarks, including cache-level contention caused by limited bandwidth and hardware resources (e.g., limited cache space, limited entries for LFB and TLB), and large memory footprint incurred by a large number of threads. Based on these findings, we propose our SMT-aware optimization techniques to reduce memory contention in the next section.

3. SMT-AWARE CODE OPTIMIZATION

From our analysis in Section 2, optimizing the memory subsystem by reducing the contention in cache space and bandwidth is critically important for SMT architectures. We propose a term *instantaneous footprint* to characterize the contention among SMT threads (defined as the hardware threads on the same physical core), as follows.

Definition 1 (Instantaneous Footprint) We define *instantaneous footprint* of a program context (loops, functions,

APPs	Latency Increase (↑) and Decrease (↓)		
	Intel Xeon	Xeon Phi* (miss impact)	Power7*
srad	L1 hit ↑ 62% TLB miss ↑ 104x	↑ 102%	L2 hit ↑ 5% L3 hit ↑ 8% Local_mem hit ↑ 5%
streamcluster1	LFB hit ↑ 69%	↑ 33%	L2 hit ↑ 3x L3 hit ↑ 23%
Lulesh 2.0	L1 hit ↑ 11% L2 hit ↑ 170% LFB hit ↑ 60%	↑ 113%	L3 hit ↑ 24% Local_mem hit ↑ 17%
NAS LU	LFB hit ↑ 87% L2 hit ↑ 115%	↑ 65%	L2 and L3 hit ↓
3D Tensor	L3 miss ↑ 24%	↑ 34%	L2 hit ↑ 15% L3 hit ↑ 17%
Stencil1	LFB hit ↑ 48% L1 hit ↑ 13%	↑ 160%	L2 hit ↑ 23% L3 hit ↑ 9% Local_mem hit ↑ 26%
Clomp	L1 hit ↑ 7% LFB hit ↑ 10%	↑ 13%	L2 hit ↑ 50% L3 hit ↑ 15%
hotspot	L2 hit ↑ 8x L3 hit ↑ 45% mem_bus ↑ 110%	↓ 0.5%	L2 hit ↑ 1.5x

* Unlike Xeon, Xeon Phi and Power7 require workarounds from performance counters to approximate latency inflation for a subset (not all) of memory layers. For instance, L1 miss latency can be estimated through the limited hardware events available on Xeon Phi: Estimated Latency Impact = (CPU_CLK_UNHALTED - EXEC_STAGE_CYCLES - DATA_READ_OR_WRITE) / DATA_READ_OR_WRITE_MISS.

Table 3: Latency inflation of the major memory bottlenecks (based on F and latency contribution of individual memory component) for (high, high) applications on the three architectures.

or the whole program) as the memory bytes required by all the SMT threads at a specific time point.

A large instantaneous footprint potentially leads to high memory contention, as loading these data concurrently competes for cache space and bandwidth. As an example shown in Figure 3(a), threads are working on n different data sets, which may cause increased thread contention due to the large instantaneous footprint (i.e., as high as n data sets). This parallelization is widely used in multithreaded applications with modern programming models, such as OpenMP [28], Pthread [4], and Cilk [11]. While this can benefit from “embarrassing parallelism”, it could also introduce severe contention in the memory subsystem. To reduce the contention, we propose two techniques to shrink the instantaneous footprint with code transformation in Section 3.1, and then use empirical analysis to show their applicability for the real optimization in Section 3.2. Finally, we discuss some limitations of our proposed methods in Section 3.3.

3.1 Instantaneous Footprint Reduction

As shown in Figure 3(b) and (c), we propose two methods to reduce instantaneous footprint, including intra-thread optimization through *data decomposition* and inter-thread optimization through *thread collaboration*.

Intra-thread Optimization. Figure 3(b) shows the basic idea of enhancing intra-thread memory utilization and data locality through data decomposition. Instead of loading the entire data set of each thread to compete for the memory subsystem, we can decompose the data set of each thread into several (e.g., 4 in the figure) small pieces and load them one by one. Thus, the instantaneous footprint across all the threads is reduced to a quarter of the original footprint. For example, as a concrete software-level intra-thread optimization, we regroup n arrays in a loop from structure of arrays (SoA) to array of structure (AoS). In the SoA loop, the instantaneous footprint is n cache lines per iteration, because a thread needs to load a cache line from each array for the computation. However, in the AoS loop, the instantaneous footprint is reduced to $\frac{n}{8}$ cache lines with the assumption that all the arrays contain eight-byte double elements and the cache line size is 64 bytes. Thus, we consider array regrouping as a viable option for reducing the large instantaneous footprint on SMT architectures.

It is worth noting that reducing instantaneous footprint through intra-thread optimization is often complex and sometimes impossible for every loop, since data layouts with different access patterns may not be easily matched between hot loops to obtain good intra-thread locality. Details about array regrouping in highly threaded codes can be found in our previous work [22]. For the rest of the section, we will mainly focus on discussing the inter-thread optimization in SMT architectures.

Inter-thread Optimization. When intra-thread optimization is not feasible or effective, inter-thread optimization can be applied. Figure 3(c) demonstrates the basic idea of inter-thread optimization. As shown in the figure, all the threads collaboratively work on the same data set and then move on to the next upon completion, reducing the instantaneous footprint to $\frac{1}{n}$ of the default parallelization mode in Figure 3(a). Note that such parallelization mode may incur thread synchronization more often. However, the synchronization between the SMT threads occurs in L1 cache as L1 hits, whose cost is much smaller than the gain from the inter-thread optimization that reduces cache miss significantly (see the evaluation in Section 3.2). As a concrete inter-thread optimization, increasing sharing, even *false sharing* between SMT threads can effectively reduce the instantaneous footprint across these threads. In the next section, we elaborate on this idea with empirical studies.

3.2 False Sharing among SMT Threads

3.2.1 General Finding

False sharing is a well-known performance contention in multithreaded programs [19]. Assume that each thread runs on a distinct core with its own private cache. False sharing occurs when threads simultaneously access logically-independent data in the same cache line, and where at least one of the accesses is a write. Usually it causes significant performance degradation because threads keep invalidating cache lines and incur many coherence cache misses. False

Figure 4: Speedups for the SMT-aware optimization on (a) Listing 1 and (b) Listing 2.

```
1 #pragma omp parallel for <schedule(static,1)>
2 for (int i=0; i<N; i++)
3     for (int j=0; j<N; j++)
4         B[j][i] = A[i][j];
```

Listing 1: Threads have no intra-thread locality on array B. We can enhance inter-thread locality to improve performance.

sharing can be avoided by placing the data required by different threads into different cache lines.

Counter-intuitively, false sharing does *not* hurt the performance of SMT threads, which share the private cache to that core. False sharing degrades performance only if threads running on different physical cores, because cache line invalidations and coherence misses only occur between private caches of different cores. Unlike threads in the traditional multicore architectures, SMT threads share all levels of cache within the same physical core. Therefore, false sharing between SMT threads does not hurt performance. This matches the observation we had in Figure 2(a) (although true and false sharing could both occur there). Based on this insight, we discover the following finding, which serves as the foundational knowledge for reducing instantaneous footprint via inter-thread optimization in SMT architectures:

Key Finding: *Introducing false sharing but not breaking the original intra-thread locality can significantly improve performance for SMT threads.*

3.2.2 Empirical Evaluation of the Finding

We empirically illustrate how to use this finding in the optimization and assess its performance gain. Several simple OpenMP [28] microbenchmarks are developed here for evaluation purposes. OpenMP is one of the most popular multithreading models and provides intuitive parallel structures suitable for our investigation. However, the same finding and conclusion can also be applied to other thread models, such as Linux Pthread [4], TBB [34], and Cilk [11]. The experiments for this evaluation are conducted on the POWER7 machine listed in Table 1 with four SMT threads bound to the same physical core. All the code examples used in this section are compiled with gcc-4.9.

Listing 1 shows an OpenMP parallel loop nest for matrix transposition. Referencing array B has long strides, leading to poor spatial locality in each thread. The original code (with no blue keywords) also has no inter-thread reuse because the OpenMP library assigns a chunk of i loop iterations to each thread by default: all threads access different segments of B. It is worth noting that improving the intra-thread locality of the loop via loop transformation is not easy because interchanging the loop nest improves B but hurts A.

We apply a simple SMT-aware optimization to introduce false sharing across threads. We add the blue keywords in Listing 1 to let OpenMP runtime library dispatch ad-

```

1 #pragma omp parallel for shared(array) <schedule(static,1)>
2   for (int i=0; i<N*N; i++)
3     array[i]++;

```

Listing 2: `array` has intra-thread locality in the loop.

```

1 #pragma omp parallel for shared(matrix)
2   for (int i=0; i<N; i++)
3     ...
4     for (int k=0; k<N; k++)
5       ...
6       for (int j=0; j<N; j++)
7         matrix[i][...][j][...][k] = ...;

```

Listing 3: A multi-level loop nest with poor intra-thread locality in `matrix`.

jacent loop iterations to each thread, in an interleaved manner. Therefore, threads (falsely) share the elements of `B` in the same cache line to reduce the instantaneous footprint of `B` from four cache lines (because we have four hardware threads per core) to one cache line. Figure 4(a) shows the speedup of the code after this optimization. The x axis illustrates the values of the loop trip count `N` and the y axis shows the speedup of the optimized code over the original case. From the figure, we find that enhancing inter-thread locality with false sharing can achieve significant speedups as `N` increases. We also examine the hardware performance counters to understand the causes of the speedups. For the case of $N = 30,000$, the optimized code has 8% more instructions than the original case, mainly due to the extra iteration scheduling in the OpenMP library. However, the optimized code reduces L1, L2, and L3 cache misses by 15%, 70%, and 67%, respectively.

The code example in Listing 1 has no reuse in array `B` inside each thread due to the long accessing stride, so increasing false sharing between SMT threads can significantly improve performance. However, if the code transformation breaks the temporal or spatial reuse that originally exists within the thread, it may suffer from performance degradation. Listing 2 shows such an example. Since the threads stream `array` with a unit stride, they have good intra-thread spatial locality in their own chunk of loop iterations. If we interleave loop iterations to different SMT threads (shown in the blue keywords), the performance degrades. Figure 4(b) illustrates the degradation. Regardless of the array size, the interleaved version has about a 40% longer execution time than the original code. The main reason is that interleaving loop iterations add more parallel overhead from the OpenMP runtime library to the execution. Moreover, it prevents each thread to benefit from the good intra-thread locality and causes memory-bandwidth underutilization. The performance counters show that the interleaved loops result in 83%, 49%, and 9% more L1, L2, and L3 cache misses respectively than the original case.

3.2.3 Generalizing the Optimization Scheme

To generalize the idea, we extend our optimization method to deep loop nests and multiple physical cores.

Deep loop nests: We consider the multi-level loop nest in Listing 3, which has poor intra-thread locality. The loop does not iterate the right-most `k` dimension of `matrix` in the inner-most loop, so all the accesses to `matrix` suffer from non-unit stride. To enhance the inter-thread locality, we apply the SMT-aware optimization, as shown in Listing 4. Instead of parallelizing the outer-most loop, we parallelize

```

1 <#pragma omp parallel>
2 {
3   for (int i=0; i<N; i++)
4     ...
5     <#pragma omp for shared(matrix) schedule(static,1)>
6     for (int k=0; k<N; k++)
7       ...
8       for (int j=0; j<N; j++)
9         matrix[i][...][j][...][k] = ...;
10 }

```

Listing 4: Inter-thread optimization for the loop nest. The blue code highlights the transformation.

```

1 #pragma omp parallel shared (matrix)
2 {
3   // n is the number of physical cores in the system
4   int n = omp_get_num_threads()/SMT_NUM;
5   int DN = N/n; // suppose N is dividable by n
6   int t = omp_get_thread_num(); // thread ID
7   int dt = t/SMT_NUM;
8   int rt = t%SMT_NUM;
9   for(int i= dt*DN+rt; i<(dt+1)*DN; i+=SMT_NUM)
10     for (int j=0; j<N; j++)
11       B[j][i] = A[i][j];
12 }

```

Listing 5: Code transformation to ensure `B` is only falsely shared by the SMT threads on the same physical core, avoiding any false sharing between threads running on different physical cores. `SMT_NUM` is the number of SMT threads on a core.

the `k` loop by interleaving its iterations to threads, which effectively introduces false sharing between SMT threads.

Multiple physical cores: Enhancing inter-thread locality with false sharing should only happen between SMT threads. For threads running on different physical cores, false sharing should be avoided because it can cause significant execution slow down [19]. We explicitly schedule thread and parallel loop iterations, as shown in Listing 5, which is an example code transformation for Listing 1. We explicitly divide the entire `B` into `n` disjoint chunks, where `n` is the number of physical cores. Each chunk is assigned to one core to avoid false sharing between cores. To process each chunk, SMT threads interleave the accesses to the elements to introduce the false sharing between threads within a core. This OpenMP parallel region uses software thread ID (obtained from `omp_get_thread_num()`) and thread affinity set by an environmental variable to guarantee that each hardware thread executes the desired iterations of the outer `i` loop. With this optimization, running on 32 hardware threads in our POWER7 machine achieves a $1.5\times$ speedup. Moreover, the L1, L2, and L3 cache misses are reduced by 39%, 53%, and 46%, respectively. *It is worth noting that such explicit thread scheduling beyond simply adding `schedule(static,1)` is a more general use case of our inter-thread optimization.*

3.2.4 Summary

We summarize our findings related to the inter-thread optimization as follows:

- False sharing between SMT threads does not hurt performance. Thus, it is not harmful to introduce false sharing in SMT-aware optimizations.
- Exploiting inter-thread locality between SMT threads can significantly improve performance, even via false sharing. However, the code transformations cannot be applied to the memory accesses that already have intra-thread locality.
- The performance gains from enhancing inter-thread locality come from a significant instantaneous footprint

reduction, resulting in miss rate reduction at different cache levels.

Through the findings, we have illustrated a simple but effective SMT-aware optimization: explicit iteration scheduling of parallel loops to different SMT threads. This inter-thread optimization complements the intra-thread optimization, i.e., array regrouping, for reducing the instantaneous footprint across threads.

3.3 Limitations and Challenges

There are a few limitations for our technique. First, transforming from SoA to AoS is not always possible and beneficial, as arrays may have different access patterns. Second, explicitly introducing false sharing may not work for irregular memory accesses, as it may be difficult to understand their random and complex patterns. Thus, although our SMT-aware optimization (including both intra- and inter-thread) for reducing instantaneous memory footprint is powerful, it is not applicable to every data layout and parallel loop. For example, if we interleave every parallel loop in NAS LU, a case study program in this paper, we get more than 20% performance degradation. The reason is that some loops in LU have good intra-thread locality, which does not benefit from our SMT-aware optimization. It motivates the needs of a performance tool to identify the potential data structures and parallel loops for optimization, especially for a program with thousands of lines of code. We describe SMTAnalyzer in Section 4 to address these challenges.

4. SMTANALYZER

SMTAnalyzer has three unique capabilities: (1) it is a compiler-independent tool and works with unmodified binaries, (2) it provides rich information to guide SMT-aware optimizations, and (3) it incurs very low overhead. These capabilities make it applicable to real massively parallel applications. SMTAnalyzer employs the methods from our previous work [22] to identify poor-intra thread locality (e.g., long accessing strides and multiple memory streams) and arrays that can be regrouped for intra-thread optimization. In this section, we mainly focus on the design of SMTAnalyzer for pinpointing inter-thread optimization opportunities.

According to the findings described in Section 3, a code section that potentially benefits from the inter-thread optimization should meet the following three requirements. First, it is a parallel loop that has regular access patterns across iterations, so one can control memory access patterns by dispatching loop iterations to threads. Such dispatching can enhance inter-thread sharing between SMT threads and avoid false sharing between threads on different physical cores. Second, the loop is hot and memory bound. Since our SMT-aware optimization reduces memory contention, applying it to such loops may lead to significant whole-program speedups. Third, access patterns in the loops have poor intra-thread locality to benefit from our inter-thread optimization.

In this section, we describe the design and implementation of SMTAnalyzer. Section 4.1 describes the background knowledge of hardware sampling mechanisms utilized by SMTAnalyzer. Section 4.2 describes the method SMTAnalyzer uses to identify parallel loops and quantify their ‘hotness’ and memory intensiveness. Section 4.3 shows the method used by SMTAnalyzer to analyze loop access

patterns. It filters out irregular access patterns that cannot be optimized by our scheme.

4.1 Hardware Sampling Support

Modern microprocessors employ performance monitoring units (PMU) to provide insights into an application’s behavior. The PMU periodically generates interrupts as samples during an application’s execution. It can effectively associate work (e.g., instructions executed), resource consumption (e.g., cycles), and inefficiency (e.g., latency) with code regions. Such asynchronous sampling serves as the foundation of SMTAnalyzer’s analysis of program execution. SMTAnalyzer uses these performance events to identify hot loops with significant memory accesses.

In addition, PMUs on recent processors support address sampling, which collects instruction and data address pairs to associate memory references with the data that they touch. In modern Intel Xeon architectures, SMTAnalyzer can leverage Precise Event-Based Sampling (PEBS) [14] supported by PMUs. PEBS provides the following insights:

- It can record the memory address read or written by a sampled memory access instruction.
- It can glean memory related metrics. Such information is useful for understanding the performance of memory accesses (e.g., collection of L in Figure 2(a)).
- It can capture the precise instruction pointer for each sample to avoid “skid” [6] on processors with out-of-order cores. This information guarantees precise measurement.

SMTAnalyzer uses the information collected by address sampling to attribute samples to both code regions and data objects [21]. SMTAnalyzer’s analyses are based on these two attribution techniques, which are described as follows:

Code-centric attribution: SMTAnalyzer analyzes a loop’s access pattern by aggregating all the samples that are taken in the loop. Because PMU triggers memory access samples that include instruction pointers (IP), SMTAnalyzer can associate samples with instructions that can be further aggregated to the loop level.

Data-centric attribution: SMTAnalyzer allows the analysis of access patterns for each data object referenced in the loop, so it attributes samples to data objects. Because PMU captures the effective address accessed by each memory access sample, SMTAnalyzer leverages this address to associate the sample with the memory range allocated statically or dynamically for a data object. SMTAnalyzer inherits the data-centric attribution from HPCToolkit [21].

4.2 Identifying Hot Parallel Loops

SMTAnalyzer using its code-centric attribution to first identify hot parallel loops with significant memory accesses. SMTAnalyzer can easily identify loops parallelized by OpenMP, because OpenMP compilers convert the parallel loops into outline functions. However, some other threading models, like Pthread, do not create outline functions associated with parallel loops. In this case, SMTAnalyzer parses the machine code for a program, builds a control flow graph (CFG) for each procedure, and then uses interval analysis [12] to identify loops in each procedure’s CFG. If multiple threads execute the same loop, SMTAnalyzer marks the loop as a parallel loop.

Figure 5: Relax iteration-dependent patterns to thread-dependent patterns: instead of associating memory elements with iterations, we associate them with threads.

SMTAnalyzer measures the memory access latency consumed in each parallel loop to identify the hot ones, which narrows down the analysis to only the parallel loops with intensive memory accesses. In addition, SMTAnalyzer associates the average memory access latency obtained from PEBS with each loop that is investigated. Loops with high average memory access latencies are memory bound and suffer from poor locality. The threshold to determine which loops have high latencies in a program varies with the underlying architectures.

4.3 Analyzing Memory Access Patterns

Our SMT-aware optimization enhances inter-thread locality via explicitly allocating loop iterations to threads. It is effective only if access patterns can be controlled via assigning threads to different loop iterations for execution. We call such access pattern *iteration-dependent*. The iteration-dependent patterns are always regular. In contrast, irregular access patterns, such as random or indirect accesses lead to iteration-independent patterns. Changing the loop iteration assignments for applications with such patterns may not improve their inter-thread locality. SMTAnalyzer uses its code- and data-centric attributions to identify parallel loops with iteration-dependent access patterns.

Usually, analyzing access patterns involves heavyweight code instrumentation for memory accesses and loop iterations, incurring unaffordable overhead in execution. Instead, SMTAnalyzer employs a lightweight method to identify parallel loops with iteration-dependent access patterns. To avoid the instrumentation to each loop iteration, SMTAnalyzer relaxes the iteration-dependent patterns to *thread-dependent patterns*. To avoid instrumentation to all the memory accesses, SMTAnalyzer only monitors part of them using PEBS described in Section 4.1.

4.3.1 Problem Relaxation

For a simplified explanation, Figure 5 shows a loop iteration-dependent access pattern for `array` in Listing 2. The elements of `array` are referenced in each iteration in an ascending order. We find that iteration-dependent patterns can be relaxed to thread-dependent patterns to avoid monitoring each loop iteration. For parallel loops, threads are usually assigned a block of iterations for execution, as the dashed rectangles shown in Figure 5. Therefore, instead of correlating memory references with each iteration, we can associate ranges of memory references with each thread. This relaxation provides the possibility of using sampling to identify parallel loops with iteration-dependent patterns.

4.3.2 Sampling-Based Pattern Analysis

SMTAnalyzer samples memory accesses in each thread to identify whether access patterns are thread-dependent. For the analysis, SMTAnalyzer needs two pieces of information for each sample: (1) which parallel loop the sample belongs to, and (2) which data object the sample accesses. SMTAnalyzer uses the code- and data-centric attribution described in Section 4.1 to obtain (1) and (2), respectively.

For each sampled memory access m to a data object x in the parallel loop l , SMTAnalyzer computes its relative access offset o in x with Equation 1. In the equation, m_a is the memory address accessed by m ; x_{begin} and x_{end} are the boundaries of the memory range allocated for x .

$$o = \frac{m_a - x_{begin}}{x_{end} - x_{begin}} \quad (1)$$

Because m_a is always smaller than x_{end} , o is always between $[0,1]$. At the loop l level, SMTAnalyzer clusters o of all the sampled accesses by computing a $[\min, \max]$ interval of o per thread and reveals how x is accessed in each thread. Figure 6(a) shows two access pattern examples. The left pattern illustrates that each thread works on the segments in a cyclic distribution, while the right pattern shows that each thread accesses a segment of the data object. They match the pre-transformation access patterns for `B` in Listing 1 and `array` in Listing 2, respectively. Figure 6(b) shows an example output of SMTAnalyzer by plotting the $[\min, \max]$ intervals across threads to represent the right pattern of Figure 6(a). These regular access patterns imply that they are both thread-dependent and furthermore, iteration-dependent. It is worth noting that SMTAnalyzer may not accurately report the memory chunk boundary accessed by each thread, because the sampling technique does not monitor every memory address accessed by each thread. However, from our experiments, such inaccuracy does not affect the results of the pattern analysis.

In contrast, if a thread references the offset of a data object randomly, the algorithm will not recognize its access pattern as thread-dependent. SMTAnalyzer will then reason that memory accesses in the loop cannot be optimized by the proposed transformation. *SMTAnalyzer highlights hot, regular parallel loops with poor intra-thread locality as the final candidates for the inter-thread SMT optimization.*

5. CASE STUDIES

Benchmarks: To evaluate the effectiveness of our SMT-aware optimization and SMTAnalyzer for instantaneous footprint reduction, we investigated all the benchmarks categorized as (high, high) and (high, low) from Table 2 with large inputs. SMTAnalyzer does not pinpoint significant optimization opportunities in all the benchmarks studied in these two categories. For instance, further code analysis through SMTAnalyzer suggests that neither `Clomp` nor `hotspot` in (high, high) is an ideal candidate for our SMT-aware optimization. Although SMTAnalyzer identifies that `Clomp` has bad intra-thread locality (i.e., parallel loop:328-338), it performs pointer chasing which hinders itself from performing SMT-aware optimization. Additionally, `hotspot` is observed with unit stride memory accesses, but it has no temporal reuse which causes high memory access latency even without SMT effects. For `lud` in (high, low), stride analysis shows that its loop accesses to key data objects with

Benchmarks	Suite	Code Analysis from SMTAnalyzer			SMTAnalyzer Overhead	
		bottleneck region	% of total latency	OPT method	total time (s)	overhead
lulesh2.0	Sequoia [18]	lulesh.cc: 604-609	3.6%	inter-thread	389.9	+3.1%
IRSmk	Sequoia [18]	rmatmult3.c: 86-103	78.6%	intra-thread	116.7	+3.2%
needle	Rodinia [5]	needle.cpp:185-187	20%	inter-thread	37.4	+2.99%
srad	Rodinia [5]	srad.cpp:136-167	80.1%	intra-thread	599.2	+2.47%
LU	NAS [2]	rhs.f:318-328	8.4%	inter-thread	148.1	+10.6%
Stencil	Micro	stencil.c:16-21	95.7%	inter-thread	481	+1.55%
3D tensor	Micro	mt.c: 22-22	69.4%	inter-thread	96	+2.4%
streamcluster2	PARSEC [3]	streamcluster.cpp:653	14.1%	inter-thread	120.7	+15.2%

Table 4: The analysis results reported by SMTAnalyzer. SMTAnalyzer pinpoints problematic loops with only 5.2% overhead on average.

Figure 6: Figure 6(a) shows two possible thread-dependent patterns identified by SMTAnalyzer. The left and right patterns match the pre-transformation access patterns for **B** in Listing 1 and **array** in Listing 2, respectively. Figure 6(b) shows the output of SMTAnalyzer on the pattern shown in the right of Figure 6(a), across the memory address touched by 16 threads.

Benchmarks	Optimization techniques	Speedups
lulesh2.0	Enhance False sharing	1.43×
IRSmk	Array regrouping	4.86×
needle	Enhance False sharing	2.37×
srad	Array regrouping	1.74×
LU	Enhance False sharing	1.36×
Stencil	Enhance False sharing	10.9×
3D tensor	Enhance False sharing	1.44×
streamcluster2	Enhance False sharing	6.72×

Table 5: Speedups of the benchmarks running on SMT architectures after applying our optimization. We report the best speedup across different SMT architectures. The detailed analysis is described in the following sections.

a hybrid pattern using both unit stride and non-unit stride. Thus, it is not suitable for our proposed optimization. Finally, the eight selected benchmarks for further analysis are shown in Table 4.

Evaluation Platforms: The evaluation is conducted on three SMT architectures shown in Table 1. All the benchmarks are evaluated with the full SMT capabilities on these platforms. Without specific mention in the individual case studies, experiments are run with 8 cores $\times$ 2 SMT threads on Intel Xeon, 8 cores $\times$ 4 SMT threads on IBM POWER7, and 61 cores $\times$ 4 SMT threads on Intel Xeon Phi. We compile all the benchmarks using `icc 13.0.0` on Intel Xeon Phi, and `gcc 4.9` on Intel Xeon and IBM POWER7, with the highest optimization-level flag.

Code Analysis: As discussed in Section 4.1, SMTAnalyzer uses PEBS to conduct monitoring on the Intel Xeon platform. During the profiling runs on Intel Xeon, the benchmarks are run with 2 hardware threads per core on

```

1 #pragma omp parallel for private(j,k) schedule(static,1)
2 for (i = 0; i < N; i++)
3   for (j = 0; j < N; j++)
4     for (k = 0; k < N; k++)
5       tensor_a[i][j][k] = tensor_b[k][j][i];

```

Listing 6: Optimization for the 3D tensor transposition.

```

1 #pragma omp parallel for shared(array) schedule(static,1)
2 for (int i=T; i<N-T; i++)
3   for (int j=T; j<N-T; j++)
4     for (int k=0; k<T; k++)
5       R[i][j] = matrix[i][j]+matrix[i-k][j]+matrix[i][j-k]
6         +matrix[i+k][j]+matrix[i][j+k];

```

Listing 7: Optimization for the 2D Stencil computation.

8 cores. Because POWER7 and Xeon Phi do not provide necessary PMU sampling capabilities such as PEBS to support SMTAnalyzer, we use the analysis insights from Intel Xeon to optimize the benchmarks and then apply them to the other two platforms.

Table 4 summarizes the analysis results from SMTAnalyzer. SMTAnalyzer reports the locations of problematic loops in source code, percentage of memory access latency caused by these loops over the entire program, and optimization suggestions. SMTAnalyzer incurs an average of 5.2% runtime overhead. Additionally, the memory space consumed by SMTAnalyzer is less than 32 MB. Table 5 reports the optimization techniques we utilized to reduce instantaneous footprint and the best speedups obtained from our Intel Xeon, Xeon Phi, and IBM POWER7 platforms. In the rest of this section, we conduct individual case study for the benchmarks in Table 4, show the code transformations on the problematic loops reported by SMTAnalyzer, and quantify the speedups on all the three SMT architectures.

5.1 3D Tensor Transposition

Listing 6 shows a transposition on three-dimensional tensors. SMTAnalyzer identifies that both **tensor_a** and **tensor_b** in the parallel loop have iteration-dependent accesses. Figure 6(b) shows the access pattern analysis output from SMTAnalyzer. Moreover, **tensor_b** has poor locality as it has a long accessing stride. Therefore, we can apply the SMT-aware optimization to enhance the inter-thread locality on **tensor_b** via introducing (false) sharing. The blue code in Listing 6 shows the transformation sufficient for this purpose to enhance sharing on **tensor_b** between SMT threads on the same core. It is worth noting that interleaving the outer-most loop does not hurt intra-thread locality of **tensor_a** because each thread still works on continuous elements of **tensor_a** in **k** loop. We evaluate this benchmark with the full thread capability on all the three platforms. This simple optimization achieves a 1.05 $\times$ speedup (1.06 $\times$ at loop) on POWER7, a 1.44 $\times$ speedup (2.49 $\times$ at loop) on

```

1 for (i=0; i<iter; i = i + SMT) {
2     change += pgain(feasible[x+(pid%SMT)], points, z, k, pid, barrier);
3 }
4 ...
5 pgain(long x, Points *points, double z, long int *numcenters, int pid,
        pthread_barrier_t * barrier)
6 {
7     ...
8     k1 = ((int)(pid / SMT)) * bsize;
9     k2 = k1 + (bsize * SMT);
10    for ( i = k1; i < K2; i++ ) {
11        float x_cost = dist(points->p[i], points->p[x], points->dim)
12        ...
13    }
14    ...
15 }

```

Listing 8: A parallel loop that applies inter-thread optimization (explicit thread scheduling to make false sharing only between SMT threads) in StreamCluster.

Intel Xeon Phi and a $1.03\times$ speedup ($1.11\times$ at loop) on Xeon throughout the application.

5.2 Stencil computation

Listing 7 shows an example two-dimensional stencil computation. It uses k neighbors on each dimension in the matrix to compute an element. Thus, this 2D stencil code has good spatial locality on the x-axis direction but poor spatial locality along the y-axis direction. SMTAnalyzer identifies that the loop has poor locality due to the long access stride incurred by `matrix[i-k][j]` and `matrix[i+k][j]`.

We use SMT-aware optimization to interleave iterations of the i loop to enhance the inter-thread locality for `matrix[i-k][j]` and `matrix[i+k][j]`. Moreover, the transformation does not hurt the intra-thread locality for `matrix[i][j]`, `matrix[i][j-k]`, and `matrix[i][j+k]`. The SMT-aware optimization shows $1.28\times$ and $1.25\times$ speedups on POWER7 and Xeon Phi, respectively, running with 32 hardware threads. Additionally, it can achieve up to $10.9\times$ speedups on Xeon Phi while running on 244 threads (i.e., 61 cores). The optimization obtains a mild $1.03\times$ speedup running with 16 hardware threads on Intel Xeon.

5.3 StreamCluster

StreamCluster from PARSEC is used to solve online clustering problems. SMTAnalyzer identifies the problematic loop of StreamCluster, as shown in Listing 8. Accessing to different fields of `point` incurs significant contention in the memory hierarchy between SMT threads. To increase inter-thread sharing and reduce the instantaneous footprint, we modify the loop to have SMT threads working on the adjacent memory blocks, as shown in Listing 8. Note that this optimization is similar to the one shown in Listing 5 by explicitly controlling the thread and work allocation according to our inter-thread optimization. This SMT-aware optimization speeds up the loop by $6.72\times$ on POWER 7 and $1.14\times$ on Intel Xeon.

5.4 IRSmk

IRSmk solves a diffusion equation on a three-dimensional block-structured mesh. The loop kernel performs matrix multiplication in a three nested level. SMTAnalyzer identifies that this parallel loop has significant memory access latency. Further code analysis shows that there are 27 streams in the loop. Although threads have regular access patterns with unit access stride, too many streams per thread contend for cache and interfere prefetchers, causing bad intra-thread locality. We apply array regrouping to this loop nest by converting multiple streams into one, as shown in List-

```

1 #pragma omp parallel for private(jj,ii,i)
2 for (kk = kmin; kk < kmax; kk++) {
3     for (jj = jmin; jj < jmax; jj++) {
4         for (ii = imin; ii < imax; ii++) {
5             i = ii + jj * jp + kk * kp;
6             b[i]=ABCD[i].dbl*xdbl[i]+ABCD[i].dbc*xdbc[i]+ABCD[i].dbr*xnbr[i]+
7             ABCD[i].dcl*xdcl[i]+ABCD[i].dcc*xdcc[i]+ABCD[i].dcr*xdcr[i]+
8             ABCD[i].dfl*xdfl[i]+ABCD[i].dfc*xdfc[i]+ABCD[i].dfr*xdfr[i]+
9             ABCD[i].cbl*xcbl[i]+ABCD[i].cbc*xcbc[i]+ABCD[i].cbr*xcbr[i]+
10            ABCD[i].ccl*xccl[i]+ABCD[i].ccc*xccc[i]+ABCD[i].ccr*xccr[i]+
11            ABCD[i].cfl*xcfl[i]+ABCD[i].cfc*xcfc[i]+ABCD[i].cfr*xcfr[i]+
12            ABCD[i].ubl*xubl[i]+ABCD[i].ubc*xubc[i]+ABCD[i].ubr*xubr[i]+
13            ABCD[i].ucl*xucl[i]+ABCD[i].ucc*xucc[i]+ABCD[i].ucr*xucr[i]+
14            ABCD[i].ufl*xufl[i]+ABCD[i].ufc*xufc[i]+ABCD[i].ufr*xufr[i];
15         }
16     }
17 }

```

Listing 9: A parallel loop that applies intra-thread optimization (array regrouping) in IRSmk.

```

1 do j = jst, jend
2 !$omp do schedule(static,1)
3   do i = ist, iend
4     do k = 1, nz
5       utmp(1,k) = u(1,i,j,k)
6       utmp(2,k) = u(2,i,j,k)
7       utmp(3,k) = u(3,i,j,k)
8       utmp(4,k) = u(4,i,j,k)
9       utmp(5,k) = u(5,i,j,k)
10      utmp(6,k) = rho_i(i,j,k)
11    end do
12    ...
13  end do
14 !$omp end do
15 end do

```

Listing 10: An candidate loop nest for SMT-aware optimization identified by SMTAnalyzer in LU.

ing 9. The new array layout successfully reduces instantaneous memory footprint in the loop from 27 to 4 cache lines per iteration and achieves $4.86\times$, $4.83\times$, and $2.92\times$ speedups on POWER7, Xeon Phi, and Xeon, respectively.

5.5 NAS LU

LU, a computational fluid dynamic application, is a lower-upper Gauss-Seidel solver. SMTAnalyzer reports a problematic parallel loop nest that calculates *zeta-direction flux differences*. This loop has iteration-dependent access patterns and poor spatial locality due to the long stride of accessing array u ¹. This piece of code from `rhs.f` computes right hand sides from the z dimension. This parallel loop accounts for around 10% of total LU execution time. According to our SMT-aware optimization, we move the OpenMP parallel directives to the loop that iterates the inner-most dimension i of array u , as shown in the Listing 10. The optimization obtains $1.36\times$ and $1.22\times$ speedups for this loop nest on IBM POWER7 when running with 4 and 8 hardware threads, respectively.

5.6 SRAD

SRAD is an image processing benchmark based on partial differential equation. SMTAnalyzer identifies a hot parallel loop in SRAD with significant memory access latency and suggests to regroup the arrays to reduce instantaneous footprint. Listing 11 shows the code after the array regrouping, which leads to $1.43\times$ and $1.74\times$ speedups on the full thread capability of POWER7 and Xeon, respectively. It is worth noting that our SMT-aware optimization hurts performance of SRAD running on Xeon Phi. The reason is that the AoS data layout suggested by our intra-thread optimization breaks the code vectorization that is significantly important on Xeon Phi as it supports 512-bit SIMD instructions. Unlike IRSmk, the benchmark we obtain speedup on

¹LU is a Fortran program, whose memory layout is column major.

```

1 #pragma omp parallel for shared(J, cnswe, rows, cols, iN, iS, jW, jE)
   private(i, j, k, Jc, G2, L, num, den, qsqr)
2 for (int i = 0; i < rows; i++) {
3   for (int j = 0; j < cols; j++) {
4     ...
5     cnswe[k].dN = J[iN[i] * cols + j] - Jc;
6     cnswe[k].dS = J[iS[i] * cols + j] - Jc;
7     cnswe[k].dW = J[i * cols + jW[j]] - Jc;
8     cnswe[k].dE = J[i * cols + jE[j]] - Jc;
9     G2 = (cnswe[k].dN*cnswe[k].dN + cnswe[k].dS*cnswe[k].dS
10          + cnswe[k].dW*cnswe[k].dW + cnswe[k].dE*cnswe[k].dE) / (Jc*Jc);
11
12     L = (cnswe[k].dN + cnswe[k].dS + cnswe[k].dW + cnswe[k].dE) / Jc;
13
14     num = (0.5*G2) - ((1.0/16.0)*(L*L));
15     ...
16   }
17 }

```

Listing 11: A parallel loop that has many memory access streams in SRAD. SMTAnalyzer suggests to use intra-thread optimization for SRAD.

```

1 #pragma omp parallel for firstprivate(numNode) schedule(static,1)
2 for (Index_t gnode=0; gnode<numNode; ++gnode) {
3   ...
4   for (Index_t i=0; i < count; ++i) {
5     Index_t ElemId = nodeElemCornerList[start+i];
6     Index_t CornerId = Find_Pos(modelist, gnode, ElemId);
7     nodefx += f_elem[ElemId][xDir][CornerId];
8     nodefy += f_elem[ElemId][yDir][CornerId];
9     nodefz += f_elem[ElemId][zDir][CornerId];
10   }
11   ...
12 }

```

Listing 12: A candidate loop nest for SMT-aware optimization identified by SMTAnalyzer in LULESH.

Xeon Phi with array regrouping, the problematic loop in SRAD has only four streams, causing mild cache contention and prefetcher interference. Thus, the gain of contention reduction is difficult to trade off the vectorization loss. The trade-offs between the footprint reduction and vectorization is out of the scope and will be studied in our future work.

5.7 LULESH

LULESH, an application benchmark developed by Lawrence Livermore National Laboratory (LLNL), is an Arbitrary Lagrangian Eulerian code that solves the Sedov blast wave problem in 3D. SMTAnalyzer pinpoints a parallel loop in LULESH as a candidate for the SMT-aware optimization. This loop, shown in Listing 12 accounts for 3.6% of total memory access latency. SMTAnalyzer reports that the accesses to `f_elem` are iteration-dependent, although from the static code analysis, `f_elem` is accessed indirectly via `ElemId` and `CornerId`. The access pattern of `f_elem` across threads is similar to the one in Figure 6(b), but chunks have overlaps accessed by adjacent threads. Moreover, SMTAnalyzer reports a non-unit stride of accessing `f_elem` in the loop. The reason for the long stride is that `ElemId`, the left-most index is also updated in the inner loop body besides `CornerId`, the right-most index. Therefore, this parallel loop in LULESH qualifies all the conditions for the SMT-aware optimization. We can interleave iterations of the `gnode` loop to let threads work on adjacent `f_elem` elements for better inter-thread locality. This SMT-aware optimization leads to $1.21\times$ and $1.43\times$ speedups for this loop on POWER7 and Xeon Phi, respectively.

5.8 Needle

Needle is a bioinformatics application that performs dynamic programming. SMTAnalyzer pinpoints a problematic parallel loop shown in Listing 13. Memory analysis from Table 3 shows that Needle incurs high memory access overhead due to the increasing TLB misses for L2 hit when running

```

1 #pragma omp parallel for shared(input_itemsets) firstprivate(i,max_cols,
   penalty) private(idx, index) schedule(static,1)
2
3 for (idx = 0; idx <= i; idx++){
4   index = (max_cols - idx - 2) * max_cols + idx + max_cols - i - 2;
5   input_itemsets[index] = maximum( input_itemsets[index-1-max_cols]+
   reference[index],
6   input_itemsets[index-1] - penalty,
7   input_itemsets[index-max_cols] - penalty);
8 }

```

Listing 13: Optimizing the parallel loop that has long strided accesses in Needle.

with SMT. The reason can be inferred from the loop, which accesses array `input_itemsets` using a long stride, indicating inferior intra-thread locality. If the SMT threads are assigned consecutive `idx`, the inter-thread sharing increases. Thus, we perform the inter-thread optimization to reduce its instantaneous footprint, resulting in a $2.37\times$ speedup at loop level on Intel Xeon.

5.9 Discussion

To understand the performance behaviors in the memory hierarchy, we capture the cache miss events with hardware counters. Figure 7 quantifies the miss reduction due to our SMT-aware optimization at different cache layers on the three architectures. The data is collected on the whole-program level. We can observe that the benchmarks that obtain significant speedups enjoy cache miss reduction, such as 3D tensor transposition, Stencil, IRSmk, Needle, and StreamCluster. This shows that our SMT-aware optimization for instantaneous footprint reduction can efficiently improve the memory-level performance. However, for LU and LULESH2.0 on Intel Xeon and Xeon Phi, cache misses do not improve much and even get worse. The reason is that the optimized loops in these benchmarks are not hot enough and optimizing them does not show a high impact in the caches miss reduction of the entire program. Moreover, introducing the extra code by controlling thread and work allocation adds extra instructions and pollutes caches, resulting in higher cache misses.

It is also worth noting that the observed speedups are not consistent across different SMT architectures. Some benchmarks, e.g., Needle, LULESH, LU, and SRAD, obtain mild or even no speedups on one architecture but significant speedups on the others. Moreover, the number of physical cores involved in the experiments also impacts the performance gain. For example, LU can achieve speedups running with one or two physical cores on POWER7, but no speedups with more cores. A possible reason is that inter-core behaviors may influence the gains from the SMT-aware optimization. Such influence on performance may become dominant in some cases when the core number increases. Studying detailed characterization of the inter-core influence across different SMT architectures is a strong interest of our future research.

6. RELATED WORK

Thread optimization on SMT architectures: There is some previous work [38, 7, 25, 30, 10, 9] focusing on scheduling different workloads on SMT architectures. Workloads with different resource requirements are put on the same physical core and executed by its SMT threads due to minimal contention. The common drawback of these approaches is treating programs as black boxes. They change the execution environment rather than the code itself. AI-

Figure 7: The miss reduction due to our SMT-aware optimization at different cache layers on different SMT architectures. The positive numbers represent cache miss reduction, while the negative values are cache miss increase.

though these methods can avoid contention, they omit the optimization opportunities in the source code. Moreover, they do not achieve an appreciable performance improvement for HPC applications coded with Single Program Multiple Data (SPMD) models, because all threads have similar resource requests. A more recent work [32] characterizes the HPC benchmarks on SMT architectures and leverages machine learning methods to predict the performance with SMT enabled. However, this work does not provide any specific optimization method for SMT architectures.

Code optimization for shared caches: Some recent studies introduce code transformations to enhance thread sharing for programs running on multi-core systems to avoid thread contention, through adapting code to make threads share data non-uniformly [39]. Moreover, fine-grained control of data placement [20, 17, 24, 31] in source code can significantly reduce memory contention in multi-socket systems. Finally, some previous studies leveraged cache partitioning [29] and cache bypassing [36] to reduce thread contention in shared cache. However, the code transformations introduced in these approaches target programs running on multi-core and multi-socket architectures, rather than SMT architectures. Our work contrasts and compliments these approaches by developing the first code transformation technique intended to improve the performance of HPC applications running on SMT architectures. The most related work to ours is the code transformation to optimize SMT execution. Nikolopoulos [27] leverages specific tiling techniques to partition shared cache and avoid contention between SMT threads. Unlike our approach, his method does not target instantaneous footprint reduction to avoid memory contention through a general systematic optimization scheme. Moreover, there is no tool associated with his method to guide optimization.

Lightweight performance tools: There are a few tools that guide source code transformations to minimize thread contention in the shared memory hierarchy. Memphis [26], MemProf [17], and HPCToolkit [20] identify thread contention on shared data in multi-socket systems. They can associate the performance losses with data objects due to threads from different sockets congesting the memory bus. MACPO [33], ThreadSpotter [35], PTU [15], and Predator [19] pinpoint false sharing between threads running on multi-core systems. To the best of our knowledge, there is

no existing performance tool like SMTAnalyzer that directly identifies opportunities in source code for SMT-aware optimizations.

7. CONCLUSIONS

In this paper, we first vigorously conduct systematic performance analysis to characterize SMT performance impacts on a wide range of benchmarks, and quantify their detailed memory-level contention on three mainstream SMT architectures (i.e., Intel Xeon, Intel Xeon Phi, and IBM Power7). Based on these architecture-level analysis, we introduce the concept of *instantaneous footprint*, which serves as a good indicator for memory contention in SMT architectures. We then propose a simple yet effective scheme to reduce instantaneous footprint including both intra- and inter-thread SMT-aware optimizations. To reduce the optimization efforts, we develop SMTAnalyzer, a performance tool that identifies SMT-aware optimization opportunities in multi-threaded programs with low overhead. Guided by SMTAnalyzer, we optimize several widely-used SMT-sensitive applications on three platforms, and the results indicate that our proposed techniques can significantly improve the performance of these programs through drastic reduction on memory contention.

Acknowledgements

This research was supported by the National Science Foundation (NSF) under Grant No. 1464157 and DOE CENATE (66150) project. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

8. REFERENCES

- [1] Top 500 lists. <http://www.top500.org/lists/2015/11>, Nov. 2015.
- [2] D. H. Bailey et al. The NAS parallel benchmarks – summary and preliminary results. In *Proc. of SC*, 1991.
- [3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [4] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

- [5] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of IISWC*, 2009.
- [6] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors, Fall 2010.
- [7] S. Eyerman and L. Eeckhout. Probabilistic modeling for job symbiosis scheduling on smt processors. *ACM Trans. Archit. Code Optim.*, 9(2), June 2012.
- [8] S. Eyerman and L. Eeckhout. The benefit of SMT in the multi-core era: Flexibility towards degrees of thread-level parallelism. In *Proc. of ASPLOS*, 2014.
- [9] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. L1-bandwidth aware thread allocation in multicore SMT processors. In *Proc. of PACT*, 2013.
- [10] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Addressing bandwidth contention in SMT multicores through scheduling. In *Proc. of ICS*, 2014.
- [11] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of PLDI*, Montreal, Quebec, Canada, 1998.
- [12] P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.
- [13] IBM Corp. POWER8 Processor. In *Hot Chips: A Symposium on High Performance Chips*, 2013.
- [14] Intel Corporation. Intel 64 and IA-32 architectures software developer’s manual, Volume 3B: System programming guide, Part 2, Number 253669-032, June 2010.
- [15] Intel Corporation. Intel Performance Tuning Utility 4.0 Update 5. <https://software.intel.com/en-us/articles/intel-performance-tuning-utility>, October 2012. Last accessed: Aug. 10, 2014.
- [16] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.
- [17] R. Lachaize, B. Lepers, and V. Quéma. MemProf: A memory profiler for NUMA multicore systems. In *Proc. of USENIX ATC*, 2012.
- [18] Lawrence Livermore National Laboratory. LLNL Sequoia Benchmarks. <https://asc.llnl.gov/sequoia/benchmarks>. Last accessed: Dec. 12, 2013.
- [19] T. Liu et al. PREDATOR: Predictive false sharing detection. In *Proc. of PPOPP*, 2014.
- [20] X. Liu and J. Mellor-Crummey. A tool to analyze the performance of multithreaded programs on NUMA architectures. In *Proc. of PPOPP*, 2014.
- [21] X. Liu and J. M. Mellor-Crummey. A data-centric profiler for parallel programs. In *Proc. of the 2013 ACM/IEEE Conference on Supercomputing*, Denver, CO, USA, 2013.
- [22] X. Liu, K. Sharma, and J. Mellor-Crummey. ArrayTool: a lightweight profiler to guide array regrouping. In *Proc. of PACT*, Edmonton, Alberta, Canada, 2014.
- [23] X. Liu and B. Wu. ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proc. of the 2015 ACM/IEEE Conference on Supercomputing*, Austin, TX, USA, 2015.
- [24] Z. Majo and T. R. Gross. Matching memory access patterns and data placement for NUMA systems. In *Proc. of CGO*, 2012.
- [25] S. Manousopoulos et al. Characterizing thread placement in the IBM POWER7 processor. In *Proc. of IISWC*, 2012.
- [26] C. McCurdy and J. S. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proc. of ISPASS*, 2010.
- [27] D. Nikolopoulos. Code and data transformations for improving shared cache performance on SMT processors. In *High Performance Computing*. 2003.
- [28] OpenMP Architecture Review Board. OpenMP application program interface, version 4.0. <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>, July 2013.
- [29] A. Pan and V. S. Pai. Imbalanced cache partitioning for balanced data-parallel programs. In *Proc. of MICRO-46*, 2013.
- [30] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for SMT processors. Technical report, University of Washington, Department of Computer Science and Engineering, 2000. Available as <http://www.cs.washington.edu/research/smt/papers/threadScheduling.pdf>.
- [31] G. Piccoli et al. Compiler support for selective page migration in numa architectures. In *Proc. of PACT*, 2014.
- [32] L. Porter et al. Making the most of smt in hpc: System- and application-level perspectives. *ACM Trans. Archit. Code Optim.*, 11(4):59:1–59:26, Jan. 2015.
- [33] A. Rane and J. Browne. Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In *Proc. of PACT*, Minneapolis, MN, USA, 2012.
- [34] J. Reinders. *Intel Threading Building Blocks*. O’Reilly, Sebastopol, CA, 2007.
- [35] Rogue Wave Software. ThreadSpotter manual, version 2012.1. http://www.roguewave.com/documents.aspx?Command=Core_Download&EntryId=1492, August 2012. Last accessed: Dec. 12, 2013.
- [36] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proc. of SC*, 2010, 2010.
- [37] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of ISCA*, 1995.
- [38] V. Čakarević et al. Characterizing the resource-sharing levels in the UltraSPARC T2 processor. In *Proceedings of MICRO*, 2009.
- [39] E. Z. Zhang et al. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proc. of PPOPP*, 2010.