Exploiting Inter-thread Temporal Locality for Chip Multithreading

Jiayuan Meng
Department of Computer Science
University of Virginia
Charlottesville, Virginia
jm6dg@virginia.edu

Jeremy W. Sheaffer
Department of Computer Science
University of Virginia
Charlottesville, Virginia
jws9c@cs.virginia.edu

Kevin Skadron
Department of Computer Science
University of Virginia
Charlottesville, Virginia
skadron@cs.virginia.edu

***Abstract*—Multi-core organizations increasingly support multiple threads per core. Threads on a core usually share a single first-level data cache, so thread schedulers must try to minimize cache contention among threads. While this has been studied for concurrent threads with *disjoint* working sets, the problem has not been addressed for multi-threaded data-parallel workloads in which threads can be scheduled or constructed to improve inter-thread cache sharing. This paper proposes the *symbiotic affinity scheduling (SAS)* algorithm in which work is first partitioned according to the number of cores (i.e., the number of caches), and these partitions are then subdivided and scheduled among each core's available thread contexts so that threads sharing a core operate on neighboring elements to maximize cache locality.**

We demonstrate this concept with a series of data-parallel benchmarks. Simulations on M5 achieve an average speedup of 1.69× and 36% energy savings over conventional scheduling techniques that are oblivious to whether threads share a cache. Even compared to an approach that extends oblivious scheduling to ensure that the sum of the threads' working sets fits in the cache, symbiotic affinity scheduling is able to exploit greater temporal locality and provide 30% performance gains on average. Symbiosis also outperforms adaptive contention reduction techniques by 17%.

***Keywords*-chip multithreading; data locality; fine-grained parallelism; data parallelism; task scheduling;**

I. Introduction

Workloads with significant data parallelism are gaining commercial and social importance and driving processor design in many markets. For applications with sufficient parallelism, it has been shown that a *chip multithreading* (CMT) organization comprising many simple, multithreaded cores maximizes performance within a given area budget [21]. Multi-threading hides latencies and can better utilize precious memory bandwidth. This design philosophy is apparent in a number of contemporary and proposed architectures, including Niagara [30], the Cell Broadband Engine (CBE) [24], and GPUs [17], [2] (which despite their origin in rendering, have proved effective at a variety of non-rendering, general-purpose workloads [13]). The addition of single-instruction, multiple-data (SIMD) hardware further increases thread count. SIMD organizations are appealing because they boost area efficiency for data-parallel workloads and amortize the cost of instruction storage, fetch, decode, and sequencing across multiple processing elements. Wide SIMD loads can also make more efficient use of memory bandwidth [19][1].

Thread counts per core seem likely to increase, even though L1 cache sizes are unlikely to keep pace. Unfortunately, the benefits of increasing threads per core will be limited by cache contention unless threads are carefully co-scheduled. Unfortunately, current schedulers typically treat each thread as if it ran on a separate virtual processor and hence are oblivious to interactions when threads share a cache. Previous techniques attempt to mitigate cache contention by reconfiguring the cache [11], [47], [53] or selecting the best combination of heterogeneous parallel threads that provide the best throughput [40], [48], [50]. Yet, these techniques view threads as competing entities working on disjoint sets of data. Data-parallel tasks share a common data set and typically exhibit predictable access patterns. Work should be partitioned first according to the number of caches, and threads within a core should then be cooperative and access data in a pattern maximizing spatial and temporal locality. We name this concept *Symbiotic Affinity Scheduling* (SAS) and demonstrate its benefits on a wide spectrum of data-parallel applications ranging from media processing and scientific computation, to mining and machine learning.

It may appear that conventional affinity-aware techniques can be straightforwardly adapted to symbiotic threads as a form of SAS. However, as we will discuss in Section II-A, these techniques only address cache affinity for an *individual* thread or among *dependent* threads, rather than among *concurrently executed* symbiotic threads. Even if some cache-aware techniques can be adapted to ensure the joint working set of symbiotic threads fit in the local cache, they may still suffer from conflict misses and a cache block may not be reused in time before it gets replaced.

The solution to this issue is to also exploit *temporal* locality among symbiotic threads to maximize sharing and data reuse. By allowing symbiotic threads to simultaneously process *adjacent* data, the overall sequence of memory addresses they access is similar to that of a single thread that iterates through tasks in order. As a result, they can

[1]Vectors are one form of SIMD. Although they are currently only four lanes wide in most commodity processor ISAs, they will grow to 8- and 16-wide with the introduction of future ISAs [16], [46]. An alternative SIMD organization is the array organization (dubbed "SIMT" by NVIDIA for Single Instruction, Multiple Threads), in which each lane is scalar and executes a separate thread context, and the SIMD lockstep operation is implicit and not directly exposed in the ISA. Array processing has a rich history in high performance computing, and GPUs are the commodity exemplar of this architecture.

achieve parallel performance scaling without sacrificing locality. However, due to run-time dynamics, fine-grained coordination among symbiotic threads is required to ensure inter-thread temporal locality; some data may incur more computation or cache misses than others. In such cases, it is important for symbiotic threads to re-adjust their task[2] distribution so that adjacent data can still be reused in time. Therefore, we propose SAS as a run-time approach with two stages. First, independent tasks are grouped into *blocks* using cache-affinity optimizations based on the layout and capabilities of the cache hierarchy; each block is assigned to an individual core to leverage spatial locality. Secondly, because nearby tasks in the same block are likely to share data for regular access patterns, each core then traverses its block and dispatches *neighboring* tasks to multiple *concurrent* symbiotic threads on the same core to leverage temporal locality.

In SAS, a block of tasks is traversed by a per-core scheduler according to an *affinity graph of independent tasks (AGIT)* as an indicator of locality among tasks. In an AGIT, tasks are represented as vertices and those that share data are connected by undirected edges. For many parallel applications with regular data access patterns, the AGIT may be constructed implicitly without programmer intervention (e.g. the AGIT can be formed into regular meshes, lists or trees according to different data structures and access patterns). Otherwise, AGIT construction would rely on instrumentation.

In this paper, we demonstrate SAS for data-parallel applications whose parallelism is often expressed in nested, parallel `for` loops. Specifically in this scenario, a task refers to the data-parallel computation scoped within the innermost parallel `for` loop, where each task can be identified by a particular loop index. The AGIT is implicitly constructed as a multi-dimensional mesh according to the loop space. Conventional tiling or blocking techniques [10], [27], [45] group a set of neighboring tasks into blocks called *tiles* and execute each tile within an individual thread. These techniques are referred to in this paper as *Individual Tiling* (I-tile). Using SAS, we propose *Symbiotic Tiling* (S-tile) that improves inter-thread temporal locality by allowing a tile to be *collaboratively* processed by symbiotic threads; concurrent threads execute neighboring tasks, which are likely to access adjacent data. We compare S-tile to I-tile on a CMT processor with 16 threads per core and a two-level coherent cache hierarchy. Due to the unavailability of general purpose CMT processors with a high degree of multi-threading, we simulate a set of nine benchmarks selected from Splash2 [54], MineBench [38] and Rodinia [13]. Experiments show that S-tile provides an average speedup of 1.69$\times$ and energy savings of 33% compared to I-tile. Even if I-tile is adapted to assign smaller tiles to each thread so that the aggregate can fit in the L1 cache capacity (I-tile(part)), S-tile still achieves greater locality and 30% performance gains. We also compare S-tile to SOS (Sample, Optimize, Symbios) scheduling, an adaptive contention reduction technique that reduces the number of active threads when contention is detected; results show S-tile outperforms SOS by 17%.

[2]We refer to a set of dynamic instructions that have to be executed sequentially as a *task*, and it often corresponds to the code section in the innermost parallel loop.

II. Background

To avoid cache contention among threads on the same core, we propose to regard threads on the same core as a joint set of work and improve their *joint* cache affinity. We investigate not only *inter-thread spatial locality* where the joint working set of symbiotic threads fits in the cache capacity, but also *inter-thread temporal locality* where threads reuse the same data within a small time interval. In short, for a given partition of the data, both spatial and temporal locality are maximized if threads operate on immediately adjacent data elements, rather than further partitioning the data into disjoint blocks.

There are abundant studies that aim at effectively reordering or distributing computation tasks to maximize memory system throughput. They can be further divided into two complementary categories: techniques that map computation tasks to cores according to cache affinity or data locality, and techniques that reduce contention once tasks have already been mapped to cores. We present a road map of these techniques and study the differences between the proposed technique and the conventional techniques.

A. Affinity-Aware Task Scheduling

Affinity-aware task scheduling can occur during two stages: first, threads are *constructed* as virtually independent cores where each processes a disjoint block of neighboring tasks; and second, threads are *scheduled* on the underlying architecture with a particular order or mapping that optimizes cache-affinity.

1) Affinity-Aware Thread Construction: An *individual* thread can be created in a way that maximizes spatial locality by computing a sequence of neighboring tasks that are likely to access adjacent data. For tasks with regular access patterns, *Tiling* or *Blocking* can be employed to partition the workload into consecutive chunks that each map to a thread [10], [20], [22], [23], [27], [29], [31], [36], [42], [45] When data accesses are irregular, the underlying system can still reorder independent sections of code according to user-instrumented address hints that indicate data locality among code sections [41] or runtime discovery.

These techniques improve data locality within an individual thread. Nevertheless, when multiple threads share the same cache, their working sets do not necessarily overlap, and their joint data-accesses are likely scattered, leading to a higher possibility of conflict and capacity misses. Even if some cache-aware techniques may adapt to multi-threaded cores by subdividing a data partition that fits in the cache capacity among symbiotic threads, they still suffer from a lack of inter-thread temporal locality. For temporal locality within a thread, a cache block may have to persist in the cache for a long time to be reused again. On the other hand, for reuse across threads, if the cache block can be simultaneously reused by multiple threads (our approach), its risk of being replaced before fully reused becomes lower.

This observation is justified in Section VI by comparing the performance of two techniques (i.e. I-tile(part) and S-tile).

2) Affinity-aware Thread Scheduling: Once threads are created, there are several approaches to improve cache affinity:

- An *individual* thread is scheduled on the core where it has run previously to reuse remaining data in the private cache [52].
- An *individual* thread migrates closer to the cache or memory from which it frequently requests data [12], [28], [35].
- *Dependent* threads are scheduled on the same core to save data communication [6], [7], [14], [49].

These scheduling techniques are based on the history about where threads have executed or the knowledge of data dependency among threads, neither of which indicate affinity between concurrent threads sharing data, a challenge raised when data-parallel applications run on multi-threaded cores.

3) Affinity-aware Workload Partitioning: While the above techniques improve data locality either within a thread or among dependent threads, there is limited work in improving affinity among *concurrent* symbiotic threads, an issue raised by multi-threaded cores. Lo *et al.* proposed a workload partitioning technique that subdivides a page of data among concurrent threads on the same core to minimize TLB footprints [33]. However, such static approaches cannot be easily adapted to reduce cache footprints, which requires a more fine-grained orchestration between threads. A more detailed discussion can be found in Section III-A. One adaptation of this technique to reduce cache footprints is to further divide a data partition that fits in the cache capacity among concurrent threads. We refer to it as *I-tile(part)* and Section VI shows that it is inferior to S-tile.

B. Contention Reduction among Threads

Once threads are created and mapped to cores, it may occur that some storage resources may be shared among multiple threads. These resources include the shared last-level cache, private caches and TLBs. There are two main approaches to reduce such contention:

- Only a few threads are selected from a pool of threads whose joint working set minimizes cache contention [40], [48], [50], [57].
- Shared storage can be dynamically partitioned among threads according to their distinct demands [11], [47], [53].

These techniques are mostly intended for heterogeneous threads with different demands in cache capacity or associativity. While they are able to prevent cache thrashing and optimize cache throughput, they do not address data reuse among threads that would further improve the computational throughput. Because only the overall performance is dynamically profiled, these techniques are not aware of data access patterns among concurrent threads and are not able to improve inter-thread data sharing.

Nevertheless, contention reduction techniques can serve as complementary optimizations to cache sharing; there can be severe contention even if data is intensively reused among threads. We study the impact of these techniques in Section VI with I-tile(SOS) and S-tile(SOS).

C. Fine-grained Parallelism and Vector Processing

The trends for deeply multi-threaded cores to support fine-grained parallelism can be observed from Niagara [30], the Cell Broadband Engine (CBE) [24], and graphics processors such as NVIDIA's Tesla [17] that are sometimes used for general purposed computation. While hardware support has been proposed to assist fast dispatching of fine-trained tasks [32], we are not aware of any prior technique that aims to automatically improve affinity among concurrent, fine-grained tasks, even in the newest OpenMP specification [8].

Vector processing may appear similar to our proposed technique; "threads" are grouped into vectors and the need to access data in vector-sized chunks forces the programmer to build affinity among threads in the same vector. Vector processors may have multiple, vector-width threads so that the core can switch among them to hide memory latency, however, there is nothing to force the programmer to assign neighboring tasks to threads in different vectors, which requires an understanding of the data access patterns for each vector. Our proposed technique accomplishes this at runtime without burdening the programmer or compiler. Compared to CUDA [39], whose abstraction of the explicitly-managed, per-block shared memory requires the programmer to manually manage data affinity at a greater effort, our technique works with implicitly managed caches, and it automatically schedules fine-grained tasks with inter-thread data affinity at runtime.

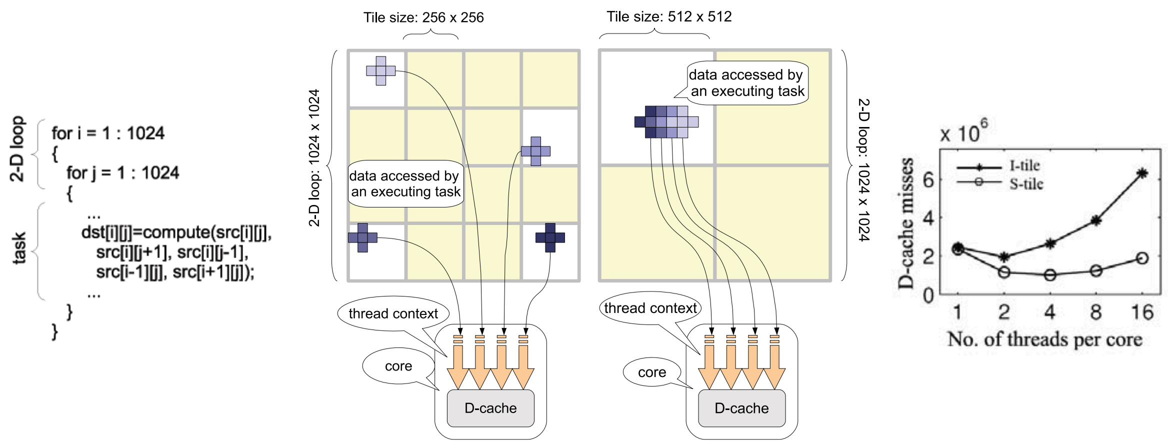
III. Symbiotic Affinity Scheduling for Data-Parallelism

We first introduce some common concepts about tiling. An example is then used to illustrate the conceptual benefits of our technique.

A. Tiling for Symbiotic Threads

Tiling targets data-parallel applications with nested, parallel `for` loops. An N level nested parallel `for` loop intuitively defines an N-dimensional grid of loop indices. An innermost iteration corresponds to a parallel *task*, which is associated with a particular loop index. A *tile* is a block of tasks that associates with a block of contiguous loop indices. Given regular data access patterns, a tile demonstrates good internal data locality. Conventionally, a tile is assigned to an *individual* thread, and because each thread is assumed to run on a separate processor with its own cache, the only locality is intra-thread data locality. These techniques are referred to in this paper as individual tiling (I-tile) and are widely used in parallel programming models such as OpenMP [18] and TBB [15].

Sadly, I-tile does not address the temporal locality of data accesses among symbiotic threads; even if two threads on the same core are assigned neighboring tiles that have bordering tasks, the amount of data-sharing is negligible compared to threads' overall working set. In addition, because threads in I-tile traverse tiles independently, neighboring tasks belonging to different tiles are seldom executed close in time to



```
for i = 1 : 1024
{
  for j = 1 : 1024
  {
    ...
    dst[i][j]=compute(src[i][j],
      src[i][j+1], src[i][j-1],
      src[i-1][j], src[i+1][j]);
    ...
  }
}
```

(a) code sample (b) I-tile (c) S-tile (d) D-cache misses

Figure 1: Comparing I-tile and S-tile in the context of HotSpot. (a) Pseudocode of HotSpot (b) I-tile: the nested parallel loop is partitioned into smaller tiles, each maps to an available *thread context*. Symbiotic threads work on distinct, scattered data, risking uneven cache-set usage and conflict misses. (c) S-tile: the nested parallel loop is partitioned into larger tiles, each maps to a *D-cache*. Symbiotic threads work on neighboring tasks with data locality. (d) D-cache misses as a function of number of threads per core on a quad core system.

reuse data. It may also appear that I-tile can be adapted to symbiotic threads by creating smaller partitions so that the sum of all the working sets of all threads on the same core will fit in the cache. However, it is very difficult for such an approach to cope with conflict misses, as we will show in Section VI with I-tile(part).

We propose *symbiotic* tiling (S-tile) that performs tiling according to the number of D-caches rather than the number of threads and dispatches data sharing tasks to concurrent threads. Because tasks with neighboring loop indices are more likely to share data given regular access patterns, the conceptual AGIT corresponding to the nested, parallel `for` loop can be simply regarded as a multi-dimensional mesh defined according to the lower bound, upper bound, and stride in each dimension of the tile. The AGIT is then traversed by symbiotic threads in a round robin fashion to ensure threads execute neighboring tasks.

It may appear that S-tile can be achieved by static analysis: tasks can be distributed to symbiotic threads in a cyclic way, similar to what Lo *et al.* proposed in cyclic scheduling where a page of data is distributed cyclicly among simultaneous threads in order to reduce TLB footprints [33]. However, reuse of cache blocks, rather than pages, is more sensitive to run-time dynamics; threads with unbalanced workloads, due to control flow or memory latency, may fall behind, and static approaches fail to re-align their task assignment. These fall-behind threads may not able to reuse cache blocks in time. Threads may fall behind even in SIMD cores — although threads in the same SIMD group are synchronized with their tasks are aligned, there are usually multiple SIMD groups on the same core, and threads in different SIMD groups may execute asynchronously for latency hiding purposes. In addition, static scheduling in a cyclic way requires knowledge of cache capacity and tile size, which either varies in different processors or may not be known until runtime. This leads to high complexity and poor portability. Given all of these factors, we implement our solution as a user-level, runtime library.

B. Case Study: HotSpot

HotSpot [25] is a thermal simulator that estimates processor temperature based on block layout and performance measurement in architectural simulation. It is a member of the structured grid dwarf [4], in which computation can be regionally divided into sub-blocks with high spatial locality. Structured grid applications are at the core of many scientific computations. Other notable examples include Lattice Boltzmann hydrodynamics [43] and Cactus [3].

Figure 1a shows a simplified 2-D version of HotSpot's code. In each iteration, a task gathers five neighboring elements on a 2-D grid to produce a new value. Assuming data array is row-major, the 5 neighboring elements are scattered in three D-cache blocks.

When HotSpot is parallelized using I-tile, it queries for the number of existing thread contexts and tiles its parallel loop accordingly. As Figure 1b illustrates, on an example parallel system with four 4-way multithreaded cores, sixteen 256×256 tiles are created to operate over a 1024×1024 grid to match the number of *hardware thread contexts*. Each thread then executes a distinct tile. Since tiles have minimally overlapping working sets, there is hardly any cache sharing among threads on the same core. Concurrent tasks executing over the same D-cache request 20 elements scattered among 12 D-cache blocks.

S-tile, on the other hand, partitions the same parallel nested loop into four 512×512 tiles to match the number of

D-caches. In our example of a four-core CMT, each tile is then assigned to a core with four threads sharing the same cache. Consecutive, *neighboring* tasks in the same tile are then dispatched to concurrent threads. As Figure 1c shows, concurrent neighboring tasks reuse adjacent data in a short span of time; in fact, at the same time in the case of SIMD access. As a whole, our example requests only 14 elements scattered in three D-cache blocks, assuming each D-cache block is 32 B and can host up to eight elements in a row. In this way, S-tile's L1 cache footprint is similar to that of a single threaded core! Although it may seem that I-tile can achieve the same data locality by only activating one thread per core, this would lose all the benefits of multithreading. This comparison is further evaluated in Section VI-A.

As a result, we show in Figure 1d that as we increase the number of threads per core together with D-cache associativity from one to 16, HotSpot experiences dramatic increases in D-cache misses with I-tile because of D-cache contention. On the contrary, the number of D-cache misses remains relatively constant with S-tile. I-tile(part) helps, but because each thread works on disjoint tiles—even though these tiles were sized to try to fit in the cache—run time dynamics can lead to contention that S-tile does not suffer. The reduced D-cache contention leads to better speedup and energy efficiency, as will be shown in Section VI.

IV. Implementation

Due to the lack of commercialized systems with many general purpose cores that have a high degree of multithreading, we simulated our benchmarks on MV5, an event-driven, cycle-accurate multicore simulator based on M5 [37], [5]. Since TBB [15] and OpenMP [8] programs use the Pthread library, which does not execute properly in system emulation mode, we implemented a user-level runtime threading library supporting basic operations required for a split-join threading model. A nested, parallel `for` loop is abstracted as a generic C++ class whose member function can be derived to encapsulate code sections within the innermost loop. This member function computes an individual task given a particular loop index. When instantiated with loop boundaries and strides, the C++ object invokes the threading library which partitions the loop, schedules and executes all the tasks. Figure 2 gives an example of how our threading API transforms a parallel `for` loop. Such an API is not new, and there are existing techniques that can automatically extract boundaries and strides of a parallel `for` loop such as employed by OpenMP [8]. Given appropriate compiler support or code transformation, our runtime technique can work with with existing APIs without modifying applications' source code; our threading API is only designed to mimic the programming interface in existing parallel APIs in our simulation. Benchmarks are all cross-compiled to the Alpha ISA using gcc 4.1.0.

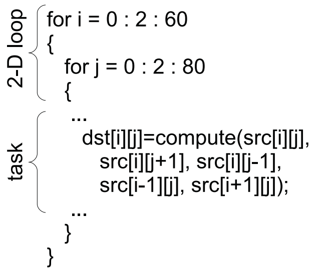
Internally, the run-time library interprets the upper bounds, lower bounds, and strides of nested parallel `for` loops. The run-time library then creates a monitor thread that executes on a randomly chosen core. The monitor thread in turn spawns threads on all available hardware thread contexts and then acts as the *centralized tile scheduler*, which is responsible for constructing the AGIT and partitioning the loop according to either I-tile or S-tile.

The AGIT for regularly strided, parallel for loops is a two dimensional array with three rows, each representing the upper bounds, and lower bounds, and strides in all levels of the nested loop. It is allocated in the heap of the monitor thread, and the cost in the memory space is negligible (N words, where N is the number of levels in the nested loop. Note N does not increase with a larger input size). The monitor thread is only activated during the sequential phase right before the parallel phase. After it partitions the loop space and distributes the tiles, it is suspended with its context switched out. The amount of work it performs does not scale with the input size and is small compared to the actual tasks performed by the parallel threads. For heterogeneous architectures, the monitor thread can run on a latency-oriented out-of-order core, while other parallel threads run on a set of throughput-oriented cores.

In I-tile, the centralized tile scheduler evenly partitions the loop according to *the number of hardware thread contexts*. Tiles are represented concisely by their boundaries and strides, and are stored in a shared memory structure. On each core, an individual thread repeatedly checks a flag to see whether a new tile is available, and acquires the tile if so. It then computes the tile by repeatedly calling the function representing the innermost loop within the tile boundary.

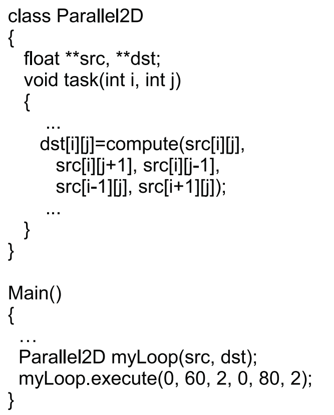
In S-tile, the centralized tile scheduler evenly partitions the loop according to *the number of cores*. Each multithreaded core has one of its threads accept the tile in a manner similar to I-tile. This thread then acts as the *per-core task scheduler*. It first allocates queues for storing loop indices for *each* thread on that core. It then iterates through the tile; instead of computing tasks, it only generates loop indices which are then pushed into the pre-allocated queues in a round robin fashion. Afterwards, threads on the same core, including the per-core task scheduler, fetch loop indices from their associated queues and execute tasks in parallel. As a result, multiple threads do not compete for the same task queue. The per-core task scheduler keeps filling a queue that is nearly empty. A thread will be busy waiting once it finds its queue empty.

The runtime library in S-tile effectively dispatches fine-grained tasks to symbiotic threads; hardware support for fine-grained parallelism, such as that proposed by Kumar *et al.* [32], is helpful but not necessary for three reasons. First, overhead in the operating system is negligible because everything takes place at user-level except for creating the initial threads. Secondly, dispatching a task is as simple as copying the next loop index from a queue and calling the function representing the innermost loop. Finally, it usually takes the per-core task scheduler fewer than 10 instructions to advance to the next loop index and buffer it. Therefore it does not create noticeable overhead since each task usually takes hundreds or thousands of instructions to execute. By manually experimenting with the run-time overhead, we found that a latency of 10 cycles in scheduling each task yields an average performance overhead of 0.7%. Note that some of the computational latency resulting from task scheduling can be hidden due to the overlap of memory



```
for i = 0 : 2 : 60
{
  for j = 0 : 2 : 80
  {
    ...
    dst[i][j]=compute(src[i][j],
      src[i][j+1], src[i][j-1],
      src[i-1][j], src[i+1][j]);
    ...
  }
}
```

(a) Conventional code with a parallel `for` loop.



```
class Parallel2D
{
  float **src, **dst;
  void task(int i, int j)
  {
    ...
    dst[i][j]=compute(src[i][j],
      src[i][j+1], src[i][j-1],
      src[i-1][j], src[i+1][j]);
    ...
  }
}

Main()
{
  ...
  Parallel2D myLoop(src, dst);
  myLoop.execute(0, 60, 2, 0, 80, 2);
}
```

(b) The expression of a parallel `for` loop in our threading API

Figure 2: Representation of a parallel `for` loop.

accesses. However, if tasks are only a few cycles long, the overhead in the run-time may still be significant. In such cases, a hardware task scheduler can be used instead. We have modeled it using a Virtex-II Pro XC2VP30 FPGA [55]. The hardware implementation requires an equivalent gate count of 3117, while even a simplest, single-threaded in-order core would still take at least tens of thousands of gates (35K gates for a 32-bit LEON2 processor compliant with the SPARC V8 architecture [1]). The hardware implementation is able to generate a loop index every cycle at 1.0 GHz.

A. Over-Decomposition and Load Balancing

Workloads are *always* balanced among symbiotic threads because the per-core task scheduler keeps filling the queues of loop indices, and any idle thread can fetch a new task from its queue. Under most circumstances, tiles are similar in size with similar amounts of computation, and workloads among cores are balanced as well. Therefore, the number of tiles by default equals the number of cores in S-tile. However, in the case of those applications that may have unbalanced workloads among tiles, programmers can over-decompose the parallel loop into smaller tiles with a larger quantity than the number of cores [31]. Load-balancing is then achieved by having idle cores fetch remaining tiles. Over-decomposition is not used when generating the results presented in Section VI.

V. Methodology

Our simulations model multi-threaded cores that operate over coherent cache hierarchies, resembling those found in Niagara [30], Larrabee [46], and Fermi []. Our core model can host hundreds of thread contexts. With such a degree of multi-threading, a deeply pipelined, out-of-order core may be neither area nor energy efficient. Instead, SIMT (Single Instruction, Multiple Threads) is more common for highly multi-threaded cores, and therefore we base our experiments upon SIMT modeling. SIMT cores group multiple scalar threads into SIMD batches that operate under a common instruction sequencer. Different from traditional SIMD structures based on vectorization, SIMT is formed by multiple scalar threads that maintain their own registers and they may follow different control flows. These properties entitle the processor to accommodate SPMD (Single Program, Multiple Data). SIMT is now used in commodity graphics processors such as NVIDIA's Tesla [17]. It is used not only for graphics but also for a wide range of general purpose applications [13]. While our evaluation focuses on SIMT, the principles of SAS and symbiotic tiling apply to multithreaded cores of any width. To explore the applicability of symbiotic tiling on cores with various SIMD widths and multithreading depths, e.g. Niagara or Larrabee, we scale the SIMT width from one to eight in Section VI-A.

A. Modeling and Configuration

We model cache latency using Cacti [51]. Pullini *et al.* [44] provide the basis for our interconnect latency modeling. The per-thread IPC (instructions-per-cycle) is assumed to be one except for memory references, which are modeled faithfully through the memory hierarchy (although we do not model memory controller reordering effects). Cores switch SIMT groups in zero cycles upon a cache access by pointing to another set of register files, as commodity GPUs do [17].

The on-chip memory system has a two level hierarchy. Each core has a private I-cache and a private D-cache. D-caches are banked to cater to the bandwidth demands of multiple thread contexts. A thread context can access any D-cache bank. If bank conflicts occur, memory requests are serialized and a one cycle queuing overhead is charged; the queuing overhead is much smaller than the hit latency because we assume requests can be pipelined. I-caches are not banked because only one instruction is fetched every cycle for an entire SIMT group. All L1 caches share the L2 cache through a crossbar. The L2 cache is inclusive and can hold more than twice as much data as all of the L1 caches combined. Caches are physically indexed and physically tagged. We employ the MESI directory-based coherence protocol. Table I summarizes the main system parameters.

Energy is modeled in four parts. We use Cacti 4.2 [51] to calculate dynamic energy for reads and writes as well as the leakage power of the caches. We estimate the energy consumption of the cores using Wattch [9]. The pipeline energy is divided into seven parts including fetch and decode, integer ALUs, floating point ALUs, register files, result

Tech. Node	65 nm
Cores	Alpha ISA, 1.0 GHz, 0.9V Vdd. in-order. 16-way multithreaded: two SIMT groups of width eight
L1 Caches	physically indexed, physically tagged 16 KB I-cache and 16 KB D-cache, 32 B line size 16-way associative, 16 MSHRs, write-back 3 cycle hit latency, 4 banks, LRU
L2 Cache	physically indexed, physically tagged 16-way associative, 128 B line size, 16 banks 1024 KB, LRU, 32 cycle hit latency, write-back 64 MSHRs, $\leq$ 8 pending requests each
Interconnect	crossbar, 300 MHz, 57 GB/s
Memory Bus	266 MHz, 16 GB/s
Memory	50 ns access latency

Table I: Default system configuration

bus, clock and leakage. Dynamic energy is accumulated each time a unit is accessed. Energy in crossbar's switches and routers are also modeled after the work of Pullini *et al.* [44], and we assume the physical memory consumes 220 nJ per access [26]. We neglect refresh power.

B. Benchmarks

We select a set of parallel benchmarks from several benchmark suites. Our primary objective is to obtain representative data-parallel applications with distinct data access and communication patterns. To maintain manageable simulation times, the input size is carefully chosen so that it assigns sufficient work to each core and the overall working set is larger than the capacity of the L2 cache size. This is large enough, since our technique mainly addresses L1 cache misses. We have tried some experiments with larger input sizes and the speedups stayed almost the same. Table II summarizes our selected benchmarks, including the dominant application behavior.

	Benchmark Description
FFT	Fast Fourier Transform (Splash2 [54]) Spectral methods. Butterfly computation Input: a 1-D array of 32,768 (2^{15}) numbers
Filter	Edge Detection of an Input Image Convolution. Gathering a 3×3 neighborhood Input: a gray scale image of size 500×500
HotSpot	Thermal Simulation (Rodinia [13]) Iterative partial differential equation solver Input: a 300×300 2-D grid, 100 iterations
LU	LU Decomposition (Splash2 [54]). Dense linear algebra Alternating row-major and column-major computation Input: a 300×300 matrix
Merge	Merge Sort. Element aggregation and reordering Input: a 1-D array of 300,000 integers
N-W	Needleman-Wunsch DNA alignment (Rodinia [13]). Dynamic programming: Updating matrix with a diagonal wavefront Input: a 2-D array of size 512×512
Short	Winning Path Search for Chess. Dynamic programming. Neighborhood calculation based on the previous row Input: 6 steps each with 150,000 choices
KMeans	Unsupervised Classification (MineBench [38]). Map-Reduce. Distance aggregation. Input: 10,000 points in a 20-D space
SVM	Supervised Learning (MineBench [38]) Support vector machine's kernel computation. Input: 1,024 vectors with a 20-D space

Table II: Simulated benchmarks with descriptions and input sizes.

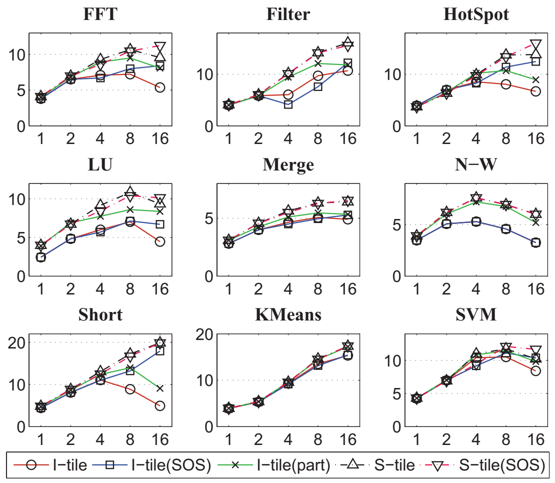


Figure 3: Speedup vs. Number of threads per core, measured on four cores each with two SIMT groups except for the case with one thread per core. D-cache associativity equals the number of threads. Speedup is normalized to single-threaded execution.

Configuration	SAS	Adaptive No. of Threads	Cache-aware Data Partition
I-tile	N	N	N
I-tile(SOS)	N	Y	N
I-tile(part)	N	N	Y
S-tile	Y	N	N
S-tile(SOS)	Y	Y	N

Table III: Different partitioning and scheduling combinations

VI. Evaluation

We use I-tile as a baseline and compare its performance to S-tile. Two other implementations, adapted from conventional techniques, are also compared: I-tile(part) subdivides a data partition that fits the cache capacity among symbiotic threads; I-tile(SOS) is named after *Sample, Optimize, Symbios* (SOS) scheduling that selects an appropriate set of threads to reduce cache contention [48]. In our adaptation, SOS first experiments with different SIMT widths and then chooses the one that yields the best throughput—wider SIMT execution increases parallelism but also increases cache contention. The optimal SIMT width can be equal to or less than the available *pipeline width* (i.e. the number of SIMD lanes) provided by the hardware. SOS can be applied to S-tile as well as a complementary technique, and we name it S-tile(SOS). Table III summarizes the combinations of partitioning and scheduling techniques that we study.

To take into account the run-time dynamics, the performance of each configuration is presented as the mean of the outputs from five simulations. Within each simulation we randomly assign tiles to threads (in I-tile systems) or cores (in S-tile systems). For simulation results belonging to the same configuration, the coefficient of variation for executed cycles usually falls below 1%.

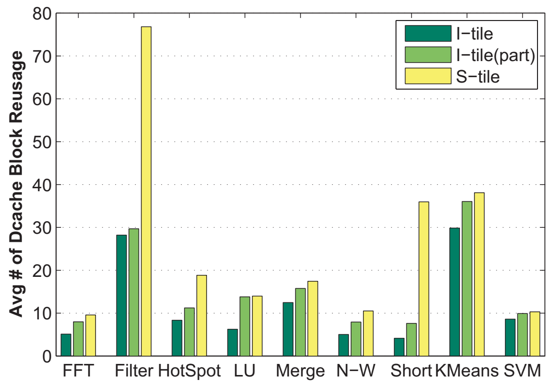


Figure 4: The average number of reuses for D-cache blocks. Data are measured with four cores each with two SIMT groups of width eight. The D-cache associativity is 16.

A. *Performance Speedup*

To compare the overall performance scalability for the five systems, we increase the number of threads per core from 1 to 16. Experiments are conducted on a simulated four-core CMT. Within each core, execution switches among active threads or thread groups to hide memory latency (except of course when only one thread context is available). Switching among threads costs zero cycles.

As Figure 3 shows, for two, four, eight and 16 threads, the average speedup of S-tile over I-tile is 1.08×, 1.25×, 1.43×, and 1.69×, respectively. Not surprisingly, S-tile is more beneficial to *high-dimensional access patterns* — strided accesses resulting from tasks nested in multilevel parallel loops (FFT, Filter, HotSpot, LU) or tasks that gather and scatter high-dimensional data (Filter, HotSpot, LU, N-W, Short). Tasks in these applications involve scatter or gather memory addresses with large strides, leading to less locality. Cache conflicts are more likely to happen and therefore exploiting cache affinity becomes more critical. This phenomenon is more evident in Figure 4 where D-cache misses and reuses are characterized; S-tile improves data reuse significantly in those workloads.

Although I-tile(SOS) is able to improve I-tile's performance to some extent, or to degrade more gracefully, it does not achieve as much performance gain as S-tile does alone; different from I-tile(SOS), S-tile reduces cache contention without decreasing SIMT pipeline utilization. I-tile(SOS) may also fail to adapt to runtime phase changes, and its performance may even degrade. In general, S-tile outperforms I-tile(SOS) by 31% on an 8-way CMT and 17% on a 16-way CMT.

On the other hand, Figure 3 shows that S-tile(SOS) performs similarly to S-tile, however, SOS may benefit S-tile where computational resources are extremely limited, as we will show in Figure 5. In such cases, S-tile and SOS can serve as complementary techniques; while S-tile improves cache sharing extensively, SOS reduces cache contention when necessary.

B. *Data Reuse and Contention Reduction*

Figure 4 shows S-tile's drastic improvement in data reuse. While I-tile(part) is able to improve cache reuse as well, its improvement falls far behind that of S-tile in many benchmarks due to the lack of temporal locality. As a result, cache blocks are more likely to be replaced before they are reused again. Compared to I-tile(part), S-tile brings additional performance gains of 13% on an 8-way CMT and 30% on a 16-way CMT, as illustrated in Figure 3.

The effect of improved cache sharing is reflected on conflict and capacity misses. The D-cache associativity is varied from four-way associative to fully associative and the resulting performance is compared in Figure 5a. As soon as the D-cache associativity increases to the number of symbiotic threads, S-tile's performance dramatically improves and reaches optimal, leaving I-tile behind. The performance gain with fully associative caches shows that S-tile saves capacity misses as well. On the other hand, while I-tile(part) reduces capacity misses in the case of fully associative caches, it suffers from conflict misses more than S-tile.

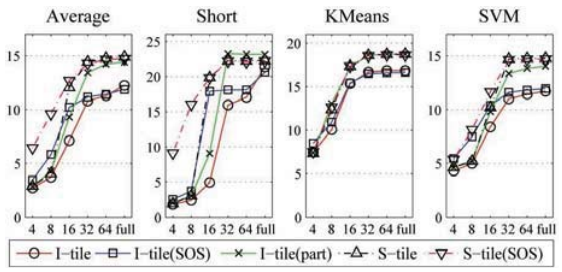
Cache sharing also leads to better storage utilization, and D-cache size is much less of a scaling bottleneck in S-tile than in I-tile. We compare the performance of I-tile and S-tile by varying the D-cache size from 4 KB to 128 KB. Results are shown in Figure 5b. With the exception of KMeans and SVM, all benchmarks show that I-tile's performance drops more drastically than S-tile's when we shrink the D-cache size. S-tile's reduced demand on D-cache size also indicates that the same D-cache can host more thread contexts in S-tile than in I-tile.

S-tile's improvement in storage utilization also leads to area efficiency and power savings. Several workloads show that S-tile's performance achieved at a D-cache size of 8 KB or 16 KB is similar to, or sometimes even better than, I-tile's with 128 KB D-caches. Using Cacti [51] to model the area cost, we show that a decrease in D-cache size from 128 KB to 8 KB saves 1.89 mm^2 for a system with 4 cores. On a system with 8 cores, this saving can account for another in-order core! In addition, several techniques provide opportunities to power down some cache segments [34], [56]. In fact, seven out of eight cache segments can be powered down with a 128 KB cache in S-tile to achieve the same performance as in I-tile.

C. *Scalability through Reduced L1 D-cache Footprints*

S-tile achieves more scalable performance because it is able to accommodate a larger number of symbiotic threads with little increase in D-cache footprints. We breakdown D-cache misses into first misses (i.e. misses that allocate MSHRs and send data requests to the lower level caches) and secondary misses (i.e. misses captured by MSHRs whose requests have already been sent to the lower level storage). We focus on the comparison of first misses since secondary misses are not part of the critical loop of memory accesses.

Figure 6 shows the number of D-cache first misses when the number of symbiotic threads increases for both I-tile and S-tile. With more threads per core, I-tile often experiences an explosion of D-cache first misses caused by contention,


(a) Speedup vs. D-cache associativity. The D-cache size is 16 KB.

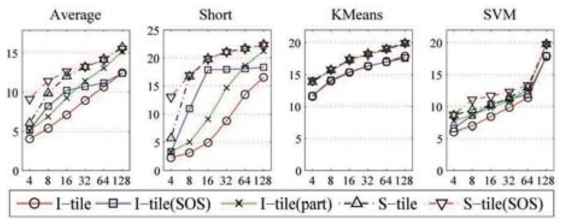

(b) Normalized speedup vs. D-cache size (KB). The D-cache associativity is 16.

Figure 5: D-cache sensitivity study conducted over a 16-way CMT. Each core has a SIMT width of eight. In each configuration, the average speedup across all benchmarks is shown as `Average`.

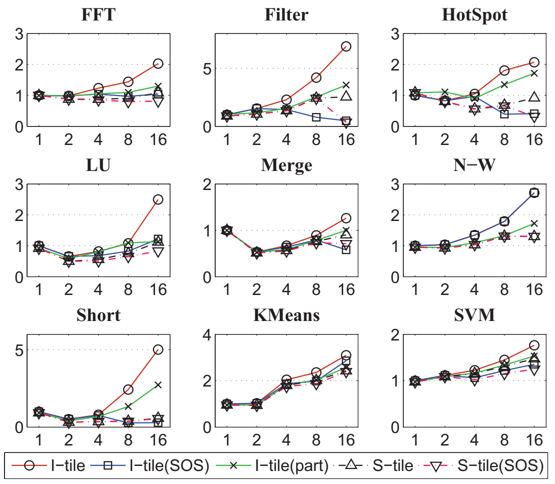
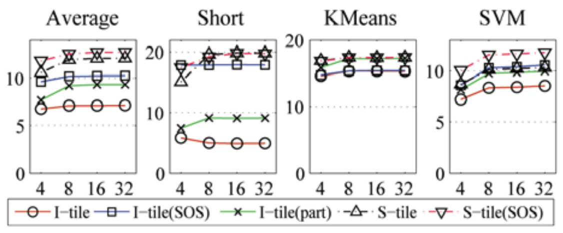


Figure 6: Number of D-cache first misses vs. Number of threads per core, measured with the same system configuration as Figure 3.

even though the D-cache associativity scales accordingly to the number of threads per core. This inevitably leads to cache thrashing. The number of D-cache misses are reduced by I-tile(part), however, it still keeps increasing with more threads.

On the other hand, the number of S-tile's D-cache misses remains relatively constant under the same scaling. In most benchmarks, the number of D-cache misses remains almost identical to that of single-threaded cores even with eight threads per core or more. It is also observed that S-tile alone is equally as effective as I-tile(SOS) in reducing the number of D-cache first misses, and this is achieved without decreasing the number of active threads. As a result, S-tile is able to scale parallel performance beyond other systems.

D. Sensitivity on Various Shared Cache Designs


(a) Speedup vs. LLC associativity

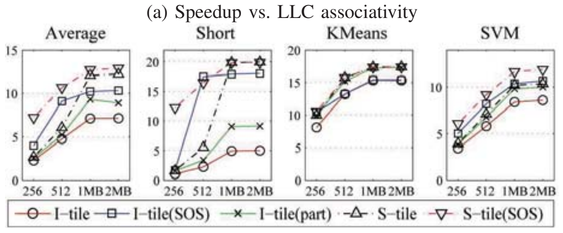

(b) Speedup vs. LLC size (KB)

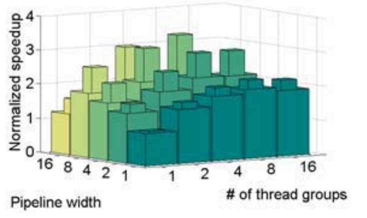
Figure 7: Sensitivity to the LLC cache design on a 16-way CMT. Each core has a SIMT width of eight. S-tile and I-tile respond similarly to different LLC cache designs. S-tile performs consistently better than I-tile. In each configuration, the average speedup across all benchmarks is shown as `Average`.

Performance sensitivity on shared cache designs is also investigated. We scale the last level cache's (LLC's) associativity from 4 to 32 and show the resulting performance in Figure 7a. All systems show similar sensitivity to LLC associativity. S-tile does not improve LLC cache sharing by much because concurrent threads over different cores still operate on distinct tiles. Instead, SOS may be more effective in reducing LLC contention when the LLC's associativity is extremely small. Nevertheless, systems with S-tile always outperform their I-tile equivalent.

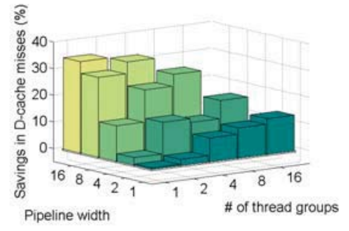
Both S-tile and I-tile suffer when the available LLC capacity is small, as illustrated in Figure 7b. In fact, S-tile's performance may even degrade to that of I-tile. In this scenario, both S-tile(SOS) and I-tile(SOS) perform significantly better than S-tile and I-tile, and S-tile(SOS) achieves the best speedup. With an LLC larger than 1024 KB, its capacity is no longer a scaling bottleneck, and performance starts to benefit more significantly from S-tile than from I-tile(SOS).

E. Applicability with Various Pipeline Configurations

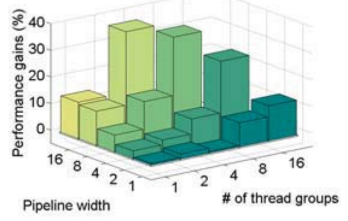
To demonstrate the wide variety of multi-threaded cores S-tile can benefit, we vary the number of scalar pipelines per core, or the width of a thread group, from 1 to 16. The



(a) Performance comparison of S-tile and I-tile over various multi-threaded cores. Speedup is normalized to the execution time on four single-threaded cores. Narrower bars denote S-tile speedup and wider bars denote I-tile.



(b) S-tile's average savings in D-cache misses compared to I-tile over various multithreaded cores..



(c) S-tile's average performance gains compared to I-tile(part).

Figure 8: Applicability for different pipeline configurations. Data is averaged across all benchmarks.

degree of multithreading, or the number of thread groups, is also varied from 1 to 16. Figure 8a illustrates the average speedup relative to the performance of 4 single-threaded cores over the memory system specified in Table I, with D-cache associativities equal to the total number of threads per core. The maximum speedups for I-tile and S-tile are 2.39× and 3.54× respectively, and both are achieved with a moderate degree of multithreading and pipeline width; too few thread groups are not able to sufficiently hide memory latency, while too many thread groups may involve more threads than necessary and increase cache contention. On the other hand, since we only model pipelines that operate in SIMT lockstep, a wider pipeline is more likely to suffer from stalls due to cache misses by individual threads or from under-utilization caused by divergent branches.

The speedup of S-tile over I-tile under the same configuration is maximized with a moderate degree of multithreading and pipeline width as well. This phenomenon has to do with the interaction between D-cache contention and the effectiveness of latency hiding; the benefit of S-tile can be increased with more thread groups due to more D-cache contention, at the same time, it may also be obscured by the improved latency hiding. While I-tile's performance may get closer to S-tile with more effective latency hiding, Figure 8b demonstrates that S-tile's savings in D-cache misses always grows with the number of threads per core. On the other hand, S-tile does not always benefit more with a wider pipeline; if S-tile is not able to eliminate cache misses simultaneously for *all* threads within the same group, a SIMT thread group has to stall anyway.

Compared to I-tile(part), S-tile's performance gains increase with a larger number of symbiotic threads, as shown in Figure 8c. Similar to the comparison of I-tile, S-tile benefits most with a modest degree of multi-threading and pipeline width.

It is also important to note that S-tile does not only benefit cores with SIMT pipelines. Even for a single scalar pipeline that switches among multiple threads, S-tile can lead to 10% performance gains over I-tile as well as I-tile(part). This indicates that S-tile and latency hiding techniques are complementary.

F. Energy Savings

Since SAS promotes constructive cache sharing and significantly reduces the number of D-cache misses, it also significantly reduces the number of L2 cache lookups. As Figure 9 suggests, the energy budget on the L2 cache is reduced significantly using S-tile.

The performance gains also translate into lower leakage energy. Leakage accounts for a significant portion of total energy consumption at a technology node of 65nm or smaller. Therefore, execution time correlates closely with energy consumption. All benchmarks save energy with S-tile, with an average energy savings of 36% compared to I-tile and 17% compared to I-tile(SOS) and I-tile(part). We observe the same trend that programs with high-dimensional access patterns benefit more from symbiotic tiling in energy consumption.

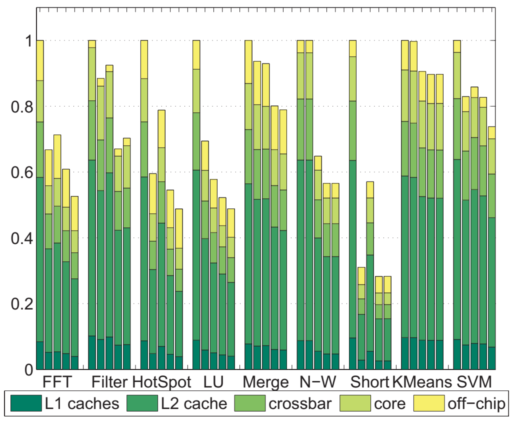


Figure 9: Energy consumption with four cores each with two SIMT groups of width eight. For each benchmark, five systems are shown and from left to right they are: I-tile, I-tile(SOS), I-tile(part), S-tile, and S-tile(SOS). Energy is normalized to each system's I-tile equivalent.

VII. Conclusions and Future Work

With emerging multithreaded cores, there has been some work on thread scheduling that reduces resource contention, but nothing that considers cache sharing among dozens or hundreds of concurrent threads sharing a common L1. Conventional data partitioning techniques can be adapted to subdivide data partitions among symbiotic threads for spatial locality; however, threads would still operate on disparate data and may suffer from conflict misses. The solution is to also exploit temporal locality among symbiotic threads. We propose symbiotic affinity scheduling (SAS) that partitions work according to the number of L1 caches, and then use all threads on that processing unit to work *simultaneously* on adjacent data in the same partition. This becomes particularly important for high-dimensional data accesses because their strided accesses penalize data-locality. The scheduler demonstrates an average speedup of 1.69× and average energy savings of 33% on the data-parallel benchmarks we studied. Based on conventional cache-blocking techniques, the exploited temporal locality brings additional 30% performance gains. It also outperforms adaptive contention reduction techniques by 17%.

Combining SAS with temporal blocking or hierarchical tiling [10] is a natural extension for future work. In fact, in this case, SAS becomes *essential* for CMTs, otherwise independent threads sharing the same cache are not able to coordinate and work on potentially time-skewed tiles optimized for individual caches.

Future work also includes extending SAS to cover parallel operations over trees and graphs. It would also be useful to integrate SAS into existing parallel programming APIs such as OpenMP [8].

VIII. Acknowledgements

This work was supported in part by SRC grant No. 1607, NSF grant nos. IIS-0612049 and CNS-0615277, and a grant from Intel Research. We would like to thank Shuai Che and Jiawei Huang who helped us on the coding of HotSpot and LU for benchmarking, Jie Li who modeled the hardware schedulers in FPGA, and Michael Boyer and Mario Donato Marino for their helpful comments. We would also like to thank the anonymous reviewers for their insightful comments.

References

[1] LEON2 Processor. http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.gaisler.com/products/leon2/leon.html.

[2] NVIDIAs next generation CUDA compute architecture: Fermi. *NVIDIA Corporation*, 2009.

[3] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H.-C. Hege, G. Lanfermann, A.é Merzky, T. Radke, and E. Seidel. Cactus grid computing: Review of current development. In *Euro-Par '01*, pages 817–824, London, UK, 2001. Springer-Verlag.

[4] K. Asanovic, R. Bodik, B. Christopher C., J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[5] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4), 2006.

[6] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *ACM Proc. of Annu. Symp. on Para. Alg. and Archi.*, pages 1–12, 1995.

[7] R. D. Blumofe. *Executing multithreaded programs efficiently*. PhD thesis, Cambridge, MA, USA, 1995.

[8] OpenMP Architecture Review Board. OpenMP application program interface, May 2008.

[9] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *ISCA 27*, June 2000.

[10] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *IPPS '95*, pages 239–245, Washington, DC, USA, 1995.

[11] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05*, pages 340–351, Washington, DC, USA, 2005.

[12] M. Chaudhuri. PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *HPCA*, pages 227–238, Feb. 2009.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphisc processors using CUDA. *JPDC'08*, 2008.

[14] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA '07*, pages 105–115, New York, NY, USA, 2007.

[15] Intel Corporation. Intel threading building blocks.

[16] Intel Corporation. Pircture the future now: Intel AVX. http://software.intel.com/en-us/avx/.

[17] NVIDIA Corporation. GeForce GTX 280 specifications. 2008.

[18] L. Dagum. OpenMP: A proposed industry standard API for shared memory programming, October 1997.

[19] William J. Dally et al. Merrimac: Supercomputing with streams. In *SC'03*, 2003.

[20] 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 *SC '08*, pages 1–12, 2008.

[21] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing cmp throughput with mediocre cores. In *PACT '05*, pages 51–62, Washington, DC, USA, 2005.

[22] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *SC'06*, 2006.

[23] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS '05*, pages 361–366, New York, NY, USA, 2005.

[24] M. Gschwind. Chip multiprocessing and the Cell Broadband Engine. In *CF'06*, New York, NY, USA, 2006.

[25] W. Huang, M. R. Stan, K. Skadron, S. Ghosh, K. Sankaranarayanan, and S. Velusamy. Compact thermal modeling for temperature-aware design. In *DAC'04*, 2004.

[26] I. Hur and C. Lin. A comprehensive approach to dram power management. *HPCA '08*, pages 305–316, 2008.

[27] W. Jalby and U. Meier. Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system. In *Proc. Int. Conf. Parallel Processing*, pages 429–432, 1986.

[28] S. Jenks and J.-L. Gaudiot. An evaluation of thread migration for exploiting distributed array locality. In *HPCA*, pages 190–195, 2002.

[29] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *PLDI '97*, pages 346–357, New York, NY, USA, 1997.

[30] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[31] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS 13*, 2008.

[32] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2), 2007.

[33] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *MICRO 30*, pages 114–124, Washington, DC, USA, 1997.

[34] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. In *ISCA '00*, pages 161–171, New York, NY, USA, 2000.

[35] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *SC'92*, pages 104–113, 1992.

[36] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Trans. Comput. Syst.*, 17(4):288–336, 1999.

[37] J. Meng and K. Skadron. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. In *ICCD*, Oct 2009.

[38] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. *IISWC '06*, pages 182–188, Oct. 2006.

[39] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide, 2007.

[40] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for smt processors. Technical report, 2000.

[41] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *ASPLOS-VII*, pages 60–71, New York, NY, USA, 1996. ACM.

[42] V. K. Pingali, S. A. McKee, W. C. Hseih, and J. B. Carter. Computation regrouping: restructuring programs for temporal data cache locality. In *ICS '02*, pages 252–261, New York, NY, USA, 2002.

[43] K. N. Premnath and J. Abraham. Three-dimensional multi-relaxation time (MRT) lattice-boltzmann models for multiphase flow. *J. Comput. Phys.*, 224(2):539–559, 2007.

[44] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. D. Micheli, and L. Benini. Bringing NoCs to 65 nm. *IEEE Micro*, 27(5), 2007.

[45] J. Ramanujam. Tiling of iteration spaces for multicomputers. In *Proc. 1990 Int. Conf. Parallel Processing, Vol*, pages 179–186, 1990.

[46] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[47] A. Settle, D. Connors, E. Gibert, and A. González. A dynamically reconfigurable cache for multithreaded processors. *J. Embedded Comput.*, 2(2):221–233, 2006.

[48] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS '00*, pages 234–244, New York, NY, USA, 2000.

[49] S. Subramaniam and D. L. Eager. Affinity scheduling of unbalanced workloads. In *SC '94*, pages 214–226, New York, NY, USA, 1994.

[50] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02*, page 117, 2002.

[51] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. Cacti 4.0. Technical Report HPL-2006-86, HP Laboratories Palo Alto, 2006.

[52] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 24(2):139–151, 1995.

[53] S. Wang and L. Wang. Thread-associative memory for multicore and multithreaded computing. In *ISLPED '06*, pages 139–142, New York, NY, USA, 2006.

[54] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *ISCA '95*, pages 24–36, June 1995.

[55] Inc. XILINX. Virtex-ii pro and virtex-ii pro x fpga user guide.

[56] S.-H. Yang, B. Falsafi, M. D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA '02*, page 151, 2002.

[57] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss. An adaptive OpenMP loop scheduler for hyperthreaded SMPs. In *PDCS '04*, 2004.