Hierarchical Domain Partitioning For Hierarchical Architectures

Univ. of Virginia Dept. of Comp. Sci. Tech Report CS-2008-08

Jiayuan Meng, Shuai Che, Jeremy W. Sheaffer, Jie Li, Jiawei Huang, Kevin Skadron

Department of Computer Science

University of Virginia

June 2008

Abstract

The history of parallel computing shows that good performance is heavily dependent on data locality. Prior knowledge of data access patterns allows for optimizations that reduce data movement, achieving lower data access latencies. Compilers and runtime systems, however, have difficulties in speculating on locality issues among threads. Future multicore architectures are likely to present a hierarchical model of parallelism, with multiple threads on a core and multiple cores on a chip. With such a system, data affinity and localization becomes even more important to efficiently use per-core resources. We show how an application programming interface (API) with the right abstractions can conveniently indicate data locality and that a system can use this information to place threads in a way that minimizes cache miss rates and interconnect traffic. This information is often well understood and easily expressed by the programmer but is typically lost to the system, forcing runtime environments to rediscover it on the fly; a far more costly approach. Our system is particularly well suited for the trend in manycore architectures towards large numbers of simple cores connected by a decentralized interconnect fabric. We study a set of data-parallel benchmarks and show that our technique yields up to a 25% performance gain with 17% reduction in energy.

1 Introduction

As manycore chips scale to tens and hundreds of cores, code performance and energy consumption are increasingly impacted by data movement. Improved data locality reduces memory transactions, placing less demand on the communication infrastructure. Such techniques are especially important for memory-bound, data-parallel applications. Even for regular data parallel applications, sophisticated gather and scatter patterns, together with complex producer/consumer interaction among

threads, pose difficult software engineering challenges in order to best utilize the parallelism of multicore architectures. Even worse, current trends indicate that future architectures will present a *hierarchy* of parallelism: multiple simultaneous threads on a core as well as multiple cores on a chip (e.g. SMT [25] and Hyper-Threading [19]). This suggests that carefully grouping threads based on data affinity will be beneficial to efficiently utilize per-core resources.

The hierarchical organization of parallel architectures motivates efficient support for fine grained parallelism, which can enable easy portability without sacrificing performance [11, 22]. We propose fine grained threads as the natural atoms upon which data partitioning and computation grouping can be carried out. Locality can be improved by considering data affinity when selecting and placing fine grained threads on processing cores.

To better place threads with respect to data locality, the runtime system needs knowledge about data access patterns prior to creating a thread. We present a simple API abstraction that is able to implicitly indicate data affinity. Our API includes the concept of a *domain*, a high level abstraction akin to nested, parallel `for` loops. The hardware interpretation of the domain releases programmers from adapting applications to systems of different scales.

Our API implicitly gives the runtime the necessary metadata to achieve better performance with a consciousness of data affinity. We abstract away the number of physical thread contexts and cores by encapsulating these resources within the domain, and hide data placement with cached, shared memory. This representation allows the programmer to work with the domain without concern for artificial mappings to a limited numbers of hardware contexts.

This work contributes to the state of the art in the following ways:

1. We introduce the *domain* as a conceptual object that encapsulates thread level data affinity for data parallel applications. Domains present a straightforward environment in which programmers may work naturally, and which can be easily interpreted by hardware.

2. We present a hierarchical organization of domain iterators to delegate blocks of data-parallel threads that exhibit data affinity to *processing elements* (PEs). The size of each block is adjusted automatically according to the number of PEs. Our results show that this data-conscious thread placement further increases the speedup by 25% over an otherwise equivalent system that does not use domain partitioning when our system has eight cores or more, together with an extra 17% energy savings.

3. We demonstrate that domain partitioning benefits hierarchical architectures with cores supporting simultaneous multithreading. Specifically, we studied systems with many in-order cores that perform *fine-grained switching* with the goal of hiding memory latency: each core switches among multiple hardware thread contexts on every data access to hide memory latency. We have varied the number of cores and the number of thread contexts per core and found that domain partitioning produces a performance gain of 22% and a energy saving of 16% over otherwise identical systems when the the system has eight cores and each core has four thread contexts.

2 A Domain-based API

We implement our streaming, domain-based programming model over a general purpose, cache-coherent system. Leverich et al. observe that stream programming models are able to block an application's working set and expose locality even if implemented on a cache-coherent system. They find that streaming memory with local stores and cache-based memory perform similarly [24]. While we can model other systems that support other stream programming models [35, 10], including Graphics Processors (GPUs) [5], Cell [13], and Stanford's Imagine [17] and Merrimac [9], cache-coherent systems deliver generality.

2.1 Stream Programming

Our streaming API presents an application as a dataflow graph of heterogeneous computational steps or *kernels*. Kernels interact by way of producer/consumer relationships through *indexable streams* [16], which we implement as n-space ordered arrays of memory data. Each kernel operates over a *domain*; a bounded n-dimensional space with strided integer coordinates. The configuration of a domain does not necessarily depend on the size of its input or output streams. For each domain coordinate, the kernel initiates a thread for which its *domain coordinate* is an implicit parameter, in a data-parallel fashion. Each thread works on a piece of the input stream, and collectively the set of threads processes the entire data set. In this way, one kernel represents a set of homogeneous operations that can execute simultaneously.

Listing 1 shows a C++ implementation of an image convolution operator with a 3×3 filter. Listing 2 and 3 demonstrate our implementation for the same operation with our API. The kernel is configured in Listing 2 and its thread function is defined in Listing 3.

2.2 Domain Implications

A domain constructs a split-join parallel section. Nested `for` loops with constant increments can describe a domain at a low level. Since a domain is a generalization of a common form of nested loop whose iterations can execute concurrently without dependency, several parallel programming models, including X10 [7] and Chapel [6], adopt functionally similar concepts but differ significantly in semantics, as explained below.

In addition to granting implicit parallel execution, domains present a strided, multi-dimensional integer space. We observe that threads with neighboring coordinates are more likely to consume overlapping or neighboring stream data, something which is especially true when threads exhibit a regular data access pattern. Accordingly, threads with nearby coordinates also access proximate memory locations. Figure 1 shows three typical data access patterns for fine-grained, data-level parallelism. Each of these demonstrates a correlation between thread-level data affinity and domain coordinates.

When executing threads with neighboring coordinates on the same PE, it is likely that cache blocks accessed by a current

Listing 1. An image convolution in C++

```
/* define variables */
float inImage[height][width];
float outImage[height][width];
float coef[3][3];
/* ... initialization */
for(long i = 1; i < height-1; i++) {
  for(long j = 1; j < width-1; j++) {
    /* convolve with a neighborhood */
    float val = 0.f;
    for(long y = -1; y <= 1; y++) {
      for(long x = -1; x <= 1; x++) {
        val += inImage[i + y][j + x] *
               coef[y + 1][x + 1];
      }
    }
    outImage[i][j] = val;
  }
}
```

Listing 2. Kernel invocation in our streaming API for an image convolution operator. Each coodinate on the 2-D domain triggers an invocation of kernel "foo" with a uniform parameter "coef". The kernel consumes the stream "inImage" and produces the stream "outImage".

```
frContext cxt; /* Define a context */
/* Then define the streams */
idxflow<float> inImage(2, height, width);
idxflow<float> outImage(2, height, width);
domain2D(1, height - 1, 1, 1, width - 1, 1)
{
  kernel(foo, coef) << inImage >> outImage ;
}
activate(cxt); /* Activivate the context */
```

thread will also be touched sooner or later by co-located threads, a boon for a system that is conscious of temporal locality. Similarly, by co-locating nearby threads and designating them to the same PE, we expect better spatial locality since each PE will then have a confined working set. As a whole, thread level data affinity can be exploited.

Functionally a strided array, a domain is easily interpreted by hardware. Given that threads are distinguished only through their domain coordinates, co-locating nearby threads takes no more effort than partitioning the domain and instantiating threads according to the chunked coordinates.

3 Implementation

We first introduce a one level hardware domain iterator that allows fast thread creation. We then show how a hierarchical layout of the hardware domain iterators can help localize computation. For load balancing purposes, we explore work stealing

Listing 3. The definition of a kernel in our streaming API

```
foo(coord /* domain coordinate */):
 upar(float **, coef),/* shared parameter */
 ixflow(float, src),  /* input stream     */
 oxflow(float, dst)   /* output stream    */
{
  /* Function body */
  long i    = coord[0];
  long j    = coord[1];
  float val = 0.f;
  /* convolve a 3-by-3 neighborhood */
  for(long y = -1; y <= 1; y++) {
    for(long x = -1; x <= 1; x++) {
      val += src[i + y][j + x] *
             coef[y + 1][x + 1];
    }
  }
  dst[i][j] = val;
}
```

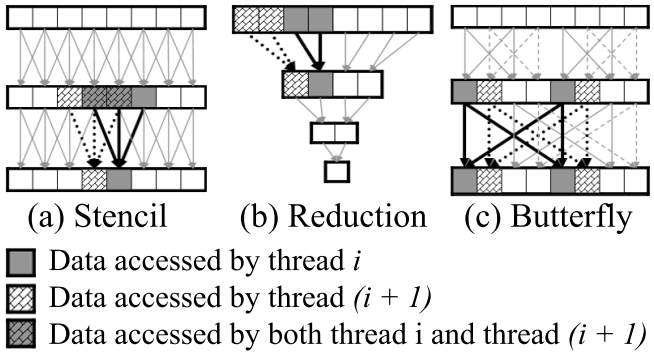


Figure 1. Threads with nearby coordinates access nearby data. Neighboring elements are likely to reside in the same cache block. Rows represent data that was produced by the previous stage and to be consumed afterwards. Edges denote data dependency. We show three common access patterns for data parallel operations. (a) Each thread gathers a neighborhood. Data is shared among threads. (b) Each thread gathers a neighborhood. Data is not shared. (c) Each thread gathers two elements and scatters two elements.

as well.

3.1 Hardware Domain Iterators

Observing that threads are homogeneous except for their coordinates, we need only distribute the shared PC and uniform parameters once to each core. All that remains is to generate a set of coordinates and instantiate threads with them. A centralized hardware domain iterator can quickly generate coordinates and distribute them to the per-core queues that buffer the coordinates. Cores are able to instantiate threads simultaneously. This reduces average thread dispatch time, which is

Listing 4. Pseudocode demonstrating the algorithm of the domain iterator. Assuming a domain of "nLayers" dimensionality

```
for n := from nLayers-1 to 0:
  i = prevCoordinate[n]+strides[n]
  if i < upperBound[n]:
    prevCoordinate[n] = i
    for m := from n+1 to nLayers-1:
      prevCoordinate[m] = lowerBound[m]
    return prevCoordinate
return NoMoreCoordinates
```

important for fine grained parallelism [22].

The hardware domain iterator interprets the domain with three register arrays and is denoted with $Iter_n(\vec{L}, \vec{U}, \vec{S})$, where n is the dimensionality of the domain, and $\vec{L}, \vec{U}, \vec{S} \in \mathbb{N}^n$ are the lower bounds, the upper bounds, and the strides. This representation implicitly imposes hardware limitations on domain dimensionality; however, data of very high dimension is rarely used in real code, and architects can very easily and cheaply plan for even unreasonable worst cases.

An *output register array* stores the current coordinate for the next generated thread invocation. The pseudocode for computing the next coordinate is shown in Listing 4. We have modeled our hardware domain iterator based on Virtex-II Pro XC2VP30 FPGA. With an equivalent gate count as small as 3117, it is able to generate a domain coordinate every cycle. We extend the Alpha ISA to include instructions to copy coordinate values from the *output register array* to user memory space.

3.2 Domain Partitioning

We assume that threads exhibit spacial locality. This means that threads with neighboring domain coordinates are more likely to consume and share nearby data, and therefore we batch threads with neighboring coordinates into a *thread block* and associate each thread block with a PE.

This batching is easily achieved by partitioning the domain into smaller *blocks*, or *subdomains*. A block can be represented in the same way as the domain: lower bound, upper bound, and stride; The lower bound and upper bound have to fall within the domain boundary, however. With a two level hierarchy of hardware domain iterators, partitioning can be easily performed through the following steps:

1. The user application invokes the runtime, which registers a domain by copying its lower bound, $\vec{L}_d$, upper bound, $\vec{U}_d$, and stride, $\vec{S}_d$. Based on specific hardware and software parameters, the runtime library then determines the size of each block, $\overrightarrow{Size_b}$, stores it in an array the block's size in each dimension.

2. A *centralized domain iterator* is initialized with $Iter_n(\vec{L}_d, \vec{U}_d, \overrightarrow{Size_b})$. It adopts the lower bound and upper bound of the user specified domain, but uses the block size as its stride. Effectively, the granularity of the domain is coarsened

to blocks, instead of individual coordinates. The output of the centralized domain iterator becomes the lower bound, $\vec{L}_b$, of a new block. A *centralized block queue* buffers these outputs in FIFO fashion.

3. The runtime library also notifies each PE with $\overrightarrow{Size}_b$, $\vec{U}_d$ and $\vec{S}_d$. This information is recorded in a per-PE buffer. The starting PC and uniform parameters of the homogeneous threads are recorded as well. After an idle PE fetches $\vec{L}_b$ from the centralized block queue, its *private domain iterator* is initialized with $Iter_n(\vec{L}_b, \min(\vec{U}_d, \vec{L}_b + \overrightarrow{Size}_b), \vec{S}_d)$. Subsequently, the private domain iterator generates an ordered sequence of coordinates that fall within the block (the subdomain) and new threads from that subdomain are created accordingly.

Figure 2 demonstrates this process. Note that each core can process only one subdomain at any time. When it finishes its current subdomain, the private domain iterator fetches a new subdomain from the centralized domain iterator.

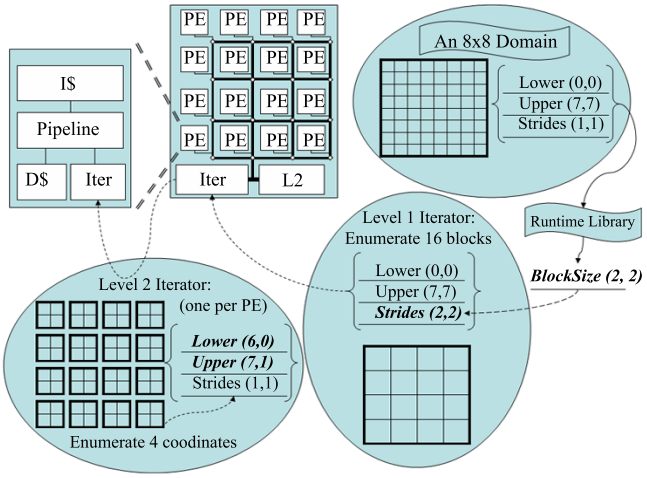


Figure 2. System Description. The high level domain is partitioned by the runtime library and interpreted by a hierarchical tree of hardware domain iterators. The runtime library determines the partitioning granularity (block size), which is used by the first level domain iterator. This centralized domain iterator generates blocks that become subdomains for the second level domain iterator, which finally generates each thread's domain coordinates.

3.3 Partition Schemes

The low level representation of the domain allows the system to explore various partition mechanisms. The block size can be decided by the programmer statically, or it can be computed adaptively according to the runtime environment.

We exploit a simple partition scheme that adapts to the number of PEs utilized by the process. The expected number of blocks is $2^{\lfloor \lg n \rfloor + 1}$, where n is the number of accessible PEs. Our algorithm initializes the block size to the size of the domain. It then visits each dimension in round robin and shrinks the block size by half until the expected number of blocks

Technology Node	70 nm
Processing Elements	Alpha ISA, 2.0 GHz, 0.9 Vdd in order 1-wide dispatch/retirement
L1 Caches	16 KB I-cache and 16 KB D-cache 4-way associative, 32 B line size 12 MSHRs, 3 cycle hit latency
L2 Cache	Unified, 4096 KB, 16 banks 32-way associative, 128 B line size 64 MSHRs, 32–56 cycle hit latency
Interconnect	2-D Mesh, wormhole routing, 300 MHz 57 GB/s, 1 cycle routing latency
Memory Bus	266 MHz, 16 GB/s
Main Memory	1 GB, 50 ns access latency

Table 1. System parameters

is reached. The algorithm is performed once for each domain, with a time complexity of $O(\lg(n))$, where n is the number of accessible PEs.

We compare static and adaptive partition schemes in Section 4.7. Our remaining experiments are all based on the adaptive partitioning scheme.

4 Evaluation and Analysis

The trend for future manycore architectures appears to be toward systems with large numbers of simple, in-order cores that have little area cost compared to out-of-order cores. Moreover, hierarchical systems present cores with multiple thread contexts. Every data access triggers a switch to another thread, overlapping computation with communication. With multiple thread contexts contending for per-core resources, data locality becomes an important concern. The behavior of these manycore systems is studied in this section.

Fine-grained context switching requires one extra register file for each additional hardware thread context. Upon switching, intermediate register values are saved. The pipeline is reloaded with intermediate values from another thread context and the core continues execution. We assume there is one stall cycle for each fine grained switch.

We have built a manycore simulation infrastructure based on M5 [3], an event-driven, cycle-accurate simulator. We choose a cache coherent system for its generality and programming ease. Our system has two levels of cache, with an MSI directory based coherence protocol. Every core has its own separated L1 instruction and data caches. These caches communicate with a shared L2 cache through a 2-D mesh interconnect. The L2 cache is the last level of on-chip memory, connected to physical memory through a PCI-E bus. Table 1 lists the major parameters of our system. We model cache latency with Cacti [34] together with data about non-uniform cache accesses as presented by Kim [18]. Pullini et al. provide the basis for out interconnect latency modeling [31]. A variety of systems are compared, outlined in Table 2.

System	Description
Convent	Conventional multicore system. Threads are executed one by one
HWIter	With a centralized domain iterator, threads are spawned with little overhead
Partition	Two level domain iterators distribute thread in blocks for data locality
PartSteal	*Partition* system that enables work stealing in cases of imbalanced computation
Convent-n	*Convent* system with fine grained switching n threads on each core
HWIter-n	*HWIter* system with fine grained switching n threads on each core
Partition-n	*Partition* system with fine grained switching n threads on each core
PartSteal-n	*PartSteal* system with fine grained switching n threads on each core

Table 2. Description of all the system configurations in our comparison. Note that with a one-level hardware iterator, setups prefixed with "HWIter" merely accelerate thread creation processes. Hierarchical domain partitions are performed on setups prefixed with "Partition" and "PartSteal", which have two-level hardware domain iterators.

Benchmark	Configuration
FFT	1-D array of 4096 floating point numbers
Edge	a grayscale image of size 100×100
HotSpot	60×60 2-D grid, 100 iterations
LUD	100×100 matrix
Merge	1-D array of 20000 integers
N-W	2-D array of size 250×250
ShortPath	6 steps each has 10000 paths
Variance	200×200 2-D array

Table 3. Input size for each benchmark.

4.1 Benchmarks

We chose a series of eight data-parallel benchmarks for evaluation of our system. These benchmarks are selected to cover distinct *dwarves*, as defined in a recent technical report out of Berkeley [1]. We adjust the input such that our benchmarks exhibit sufficient parallelism for evaluation while maintaining manageable simulation times. See Table 3 for details on input parameters.

1. **Fast Fourier Transform** (FFT) falls in the category of spectral methods. Its butterfly computation involves strided access patterns during gather and scatter data operations (shown in Figure 1 (c)).

2. **Edge Detection** (Edge) is representative of many common image processing applications where the entire image is convolved with a filter. A thread gathers a 3×3 neighborhood of elements and calculates a single output value (similar to Figure 1 (a)).

3. **HotSpot** is a widely used thermal simulation tool to estimate processor temperature based on block layout and simulated performance measurement [15]. It iteratively solves a partial differential equation over a regular structured grid.

4. **LU Decomposition** (LUD) decomposes a matrix into the product of an upper and a lower triangular matrix. Subsequent kernels switch among row-major and column-major computation. LUD exhibits a complex interaction between streams and kernels in that streams have multiple producers and consumers.

5. **Merge Sort** (Merge) iteratively merges two sorted sequences of neighboring elements into a single sequence of sorted elements. The number of data elements accessed by one thread doubles with every iteration.

6. **Needleman-Wunsch** (N-W) is a global optimization method for DNA sequence alignment. Representing the dynamic programming dwarf, it fills a 2-D matrix with scores that represent the value of the maximum weighted path ending at that matrix element; each stage of the computation is performed diagonally across the matrix, consuming a 2×2 neighborhood.

7. **Shortest Path** (ShortPath) calculates the shortest path through a field. In each step, the position can move diagonally right forward, diagonally left forward, or straight forward, given a cost function. It is an optimization problem that is solved by dynamic programming.

8. **Variance** represents the map-reduce dwarf. It uses one reduce pass to calculate a sum, followed by another pass to compute the square of each element. A final reduction sums the squares. The reduction process is illustrated in Figure 1(b).

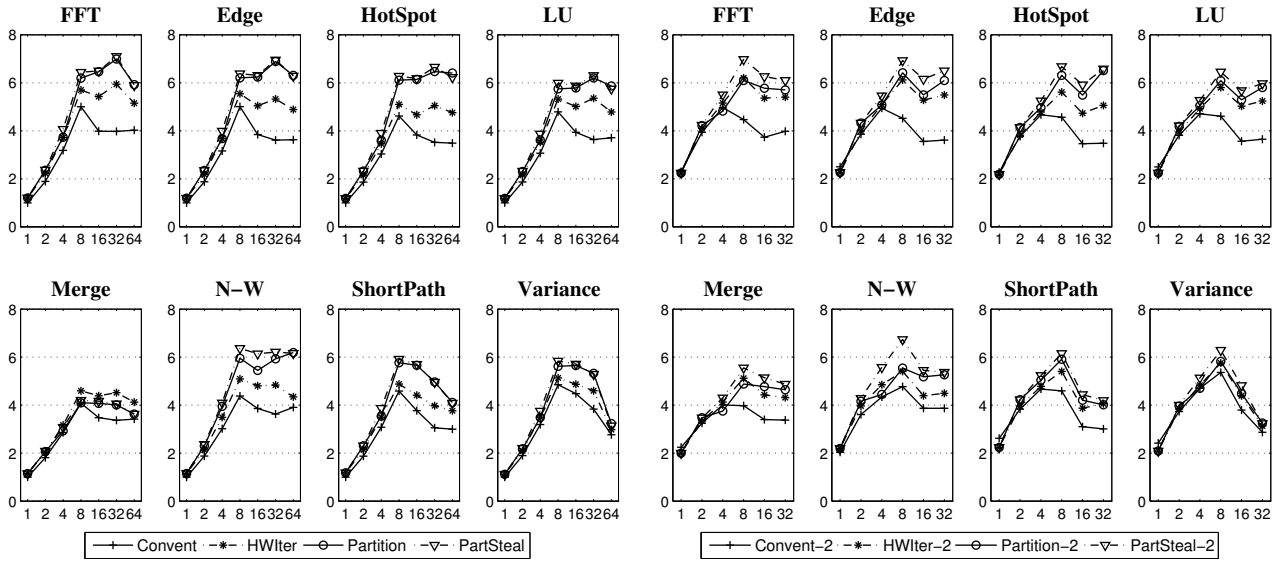
The benchmarks are simulated in *system-emulation mode*; besides threads created by the application, our API implementation invokes a runtime library that downloads the domain information, manages the memory allocation of streams, determines the granularity of the domain partitions, and signal the hardware to spawn data-parallel threads.

4.2 Performance Scaling

We show in Figure 3 (a) that while conventional systems scale with a small number of cores, their peak performance is reached at around eight cores and then performance degrades. Equipped with a single level domain iterator, more parallelism is exploited and the system scales better. Domain partitioning leads to further performance gains by localizing computation and exploiting thread-level data affinity.

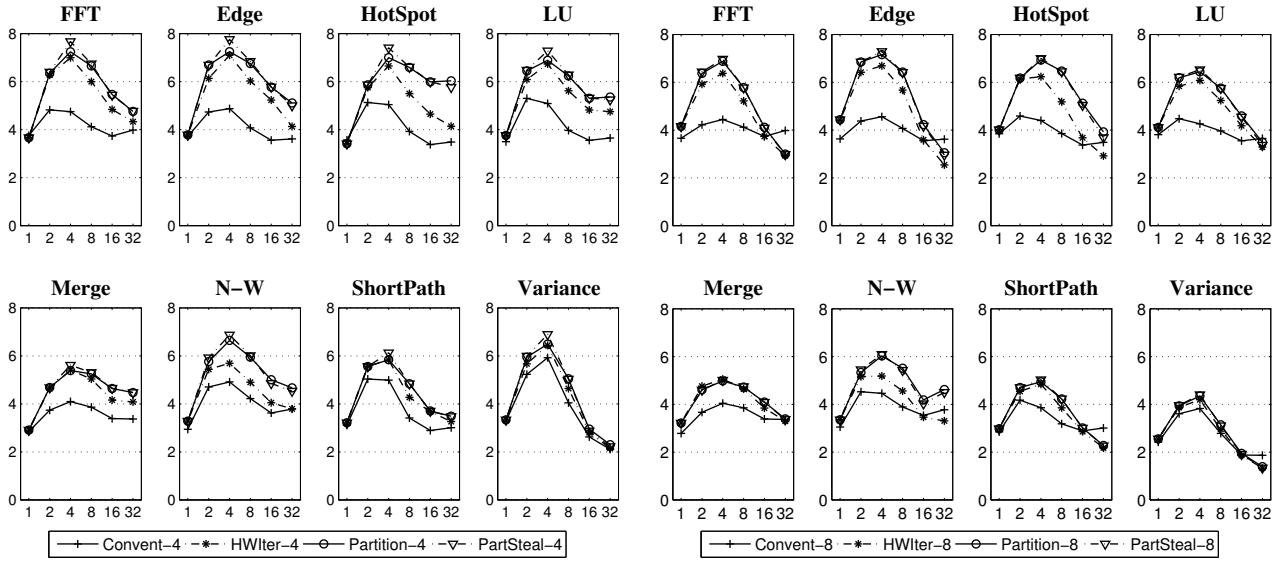
Most of our benchmarks demonstrate fine-grained parallelism. Each fine-grained thread accesses limited data. Therefore it is very likely that cache blocks brought in by the current thread may remain in the cache and be reused by subsequent threads. This hypothesis holds true for fine grained tasks with overlapping or interleaving data access patterns. As a result, *PartSteal* systems uniformly improve performance by 12% to 25% over *HWIter* systems.

The last few stages of merge sort involve only a few threads. Each gathers and scatters a large amount of data over distinct data subsets. This inevitably leads to many D-cache capacity misses. It is thus beneficial to explore producer consumer data



(a) Each core executes only one thread context at any time.

(b) Each core switches among two thread contexts to hide memory latency



(c) Each core switches among four thread contexts to hide memory latency

(d) Each core switches among eight thread contexts to hide memory latency

Figure 3. Speedup versus number of cores. Speedup is normalized to that of a uni-core "Convert" system. Generally, systems with domain partitioning outperforms others. Work stealing raises the speedup further in some cases.

locality by placing consumer threads on the same core as their producers. Unfortunately, the domain construction does not indicate producer consumer data locality.

On the other hand, we find that work stealing yields limited performance gains. Because all our benchmarks involve fine-grained threads that share similar amounts of work, workload is rarely imbalanced.

4.3 On-Chip Data Movement

When threads are distributed without concern for locality, scaling the number of cores potentially increases D-cache miss rates because the same data can be demanded at multiple locations. Moreover, it is common that fine grained threads share the same characteristics and that their memory bound phases coincide with each other. As a result, the on-chip network is more inclined to suffer from congestion caused by bursts of data requests.

Lack of data locality also imposes more pressure on the shared L2 cache. It is more likely that requests to the same L2 cache line come from multiple upper level caches. When a *miss status holding register* (MSHR), holding information about outgoing L2 cache misses [20], saturates with incoming requests and can no longer serve more, further requests will have to be rejected and re-sent from the upper level, adding more on-chip communication and amplifying D-cache miss latency. This saturation seldom occurs in our experiments, which are configured with MSHRs which each can serve as many as eight requests from the upper level.

Without consideration of data affinity, *HWIter* systems suffer from increased D-cache misses when the system scales up (Figure 4). Despite the increased number of cores, *Partition* and *PartSteal* systems keep the number of D-cache misses from increasing, thereby demonstrating data locality as a benefit of domain partitioning. This holds true even for *Merge*. With *N-W*, the number of D-cache misses is even reduced, due to an increased total size of on-chip cache. Note that with work stealing, there is a very slight increase in D-cache misses. Threads stolen from remote cores do not share data affinity with previous local threads. Since only a small portion of threads are stolen remotely, it does not hurt the effectiveness of domain partitioning. Overall, *Partition* and *PartSteal* systems reduce D-cache miss rates by 16%–30% from *HWIter* systems.

4.4 Off-Chip Communication

As the system scales, there will be more contention over the L2 cache due to its limited capacity, associativity, and number of MSHRs. We find that the number of L2 cache misses increases dramatically as we scale up a system with only a few (≤ 16) MSHRs. We use 64 MSHRs in all of our experiments for which data is reported in this paper.

Domain partitioning does not have significant impact on off-chip communication though. The number of L2 misses remains almost identical across all four system categories. When we scale up to 32 cores, all the systems in our benchmark suite show L2 misses due to contention.

4.5 Fine Grained Switching

Figure 3 (b), (c) and (d) illustrate performance scaling when each core has 2, 4, or 8 thread contexts. Over a small number (≤ 4) of cores and thread contexts, performance scales almost linearly with the total number of thread contexts. We have noticed that for all the systems and benchmarks, peak performance is achieved around a total of 16 thread contexts.

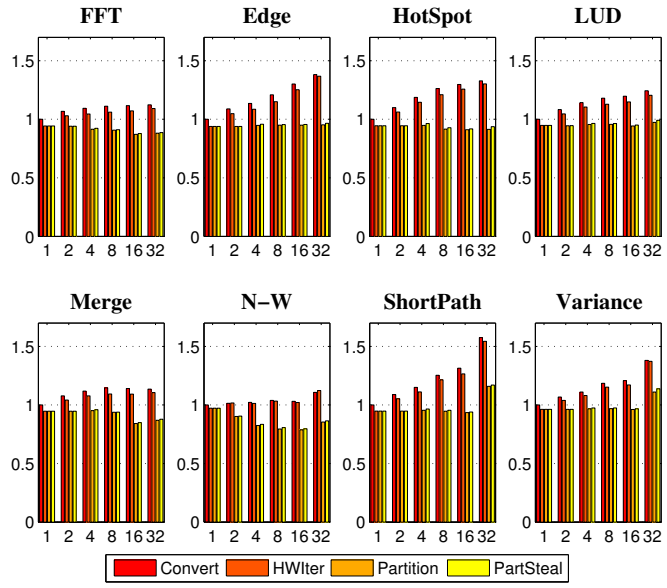


Figure 4. D-cache misses versus number of cores. D-cache misses are normalized to the total number of D-cache misses on a uni-core "Convert" system.

It is particularly beneficial to exploit thread level data affinity with this hierarchical organization of a parallel architecture. Threads executing on the same core may contend for per-core resources such as D-cache blocks and MSHRs. With knowledge of data locality, *PartSteal-*n systems almost uniformly improve performance by 10% to 22% over *HWIter-*n systems.

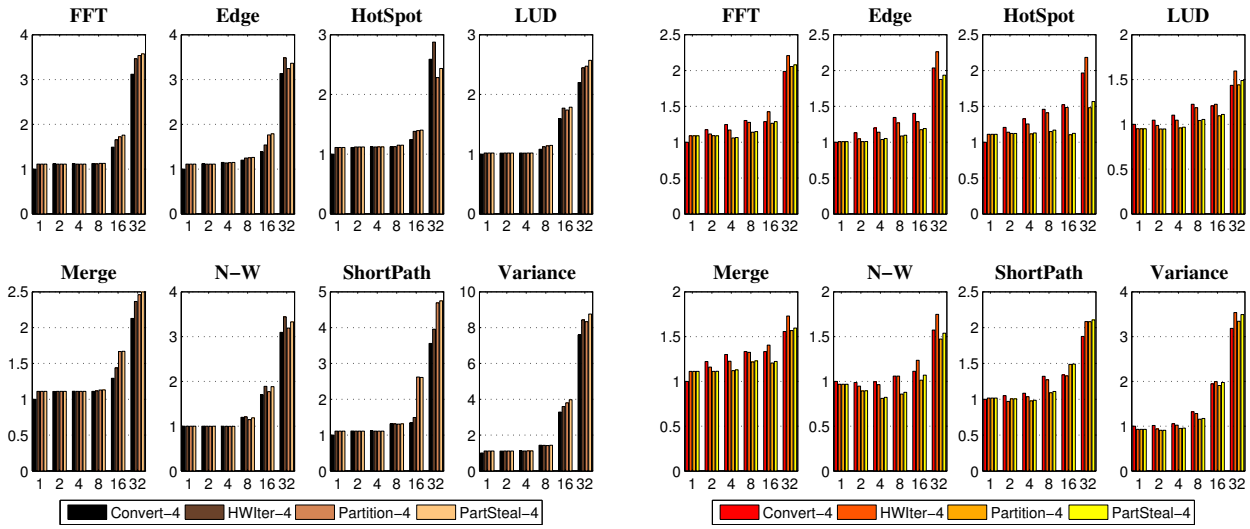
However, the benefit of overlapping communication with more computation may suffer a penalty of leveraged resource contention. Since threads are merely temporally multiplexed on each core upon data accesses, the effectiveness of fine grained switching depends on several factors:

- **Computation to memory-access ratio**: This determines the amount of computation that may be overlapped with communication.
- **D-cache miss latency**: This imposes an upper bound on how much computation can be overlapped while the communication occurs.
- **D-cache capacity and associativity**: These provide shared resources to the threads. Limited capacity and associativity lead to contention.

The advantages of fine grained switching saturate with eight thread contexts per core in our configurations. We observe a performance degradation after scaling the total number of thread contexts beyond 16. We further analyze the cause for this degradation.

With eight cores or more, there is an obvious increase in the number of L2 misses (Figure 5 (a)). L2 misses not only increase the L1 miss latency, but also add to L1 misses. This is because the coherence protocol invalidates corresponding

upper level cache blocks when the lower level cache block is replaced. When the same block of data is reused by another thread, data has to be brought back in to D-cache all the way from main memory. The increased number of D-cache misses are shown in Figure 5 (b). We have conducted sensitivity studies over a range of L2 associativities and L2 cache sizes. Results show that the bottleneck here is not the L2 cache size, but rather its associativity.



(a) Normalized L2 cache misses versus number of cores

(b) Normalized D-cache misses versus number of cores

Figure 5. On-chip and off-chip communication on systems with four threads per core, attempting to hide memory latency of each other. Data is normalized to the execution time of a uni-core "Convert-4" system.

Overall, resource contention at the shared L2 cache is the major cause for the performance degradation. The increased communication overhead due to more hops to and from the L2 cache also contributes to this degradation. In Figure 6, we raise the L2 associativity from 32 to an unrealistic value of 256 on a *PartSteal-4* system. We also increase the interconnect's clock frequency from 300 MHz to 2 GHz. Performance degradation is alleviated and some benchmarks continue to benefit from more cores. We have also conducted several other experiments with an ideal scenario where each L1 cache is connected to the L2 cache with a dedicated line. The same trend holds: systems with domain partitioning (*Partition* and *PartSteal*) perform consistently better that other systems (*Convent* and *HWIter*) across a wide range of scaling parameters and bottlenecks.

4.6 Energy Comparison

Our power parameters for caches are computed by Cacti [34]. We model our router's dynamic power and leakage after the work of Pullini [31]. The pipeline power numbers are based on Wattch [4]. Since we are modeling only in-order cores, our pipeline power model measures only register files, integer and floating point ALUs, result buses, and the clock.

Energy consumption correlates with performance in our experiments and it is dominated by leakage with a technology

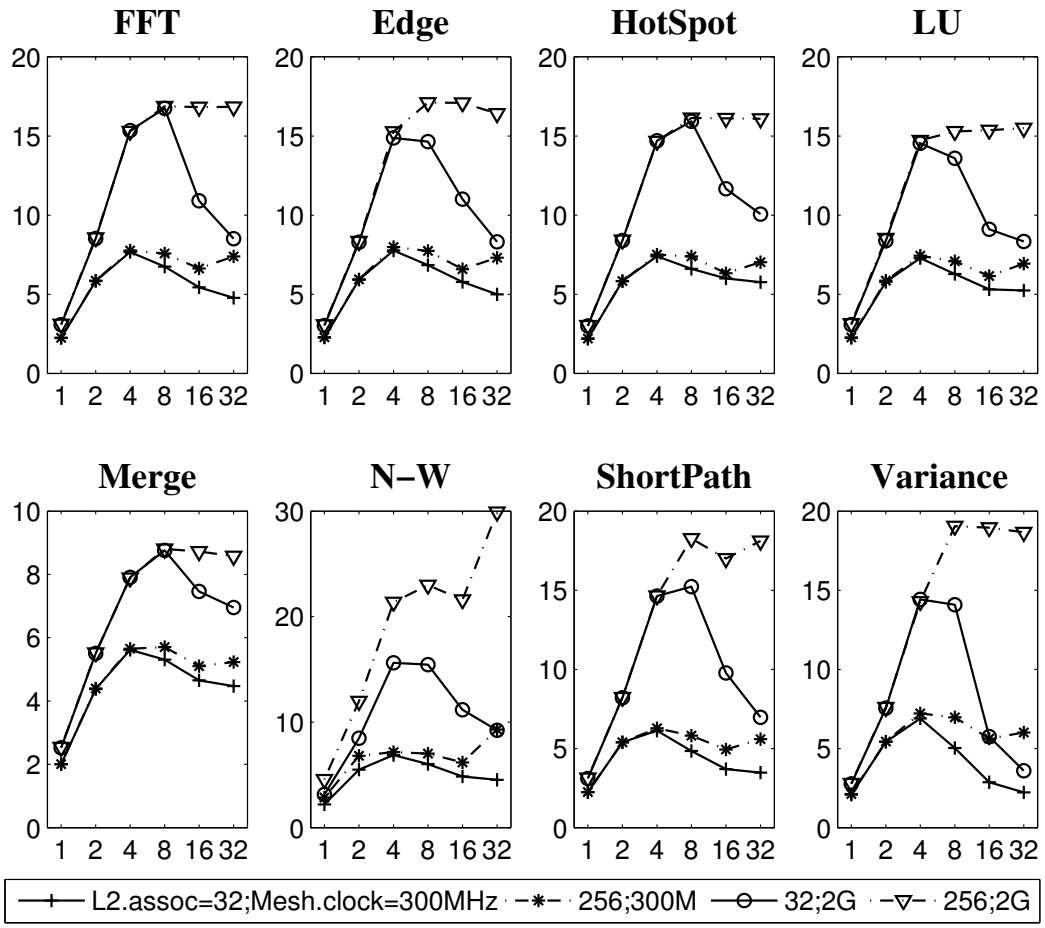


Figure 6. Optimistically scaling L2 associativity and on-chip network frequency with "PartSteal-4" configuration.

node of 70 nm. Figure 7 compares on-chip energy consumption over eight cores. *Partition* systems are able to save 7%–15% energy over *HWIter*, while *PartSteal* systems saves further 3%–5%. The energy advantages bought by way of domain partitioning are evident as well with hierarchical systems with four thread contexts per core.

4.7 Partitioning Algorithms

We compare static partitioning and our adaptive partitioning schemes in Figure 8. We vary the block size in static partitioning from 4 to 64. We find that static partitioning as adopted by prior work [29], is a poor tool for capturing the optimal partition granularity. Our adaptive partitioning scheme is preferable in the general case, with or without work stealing.

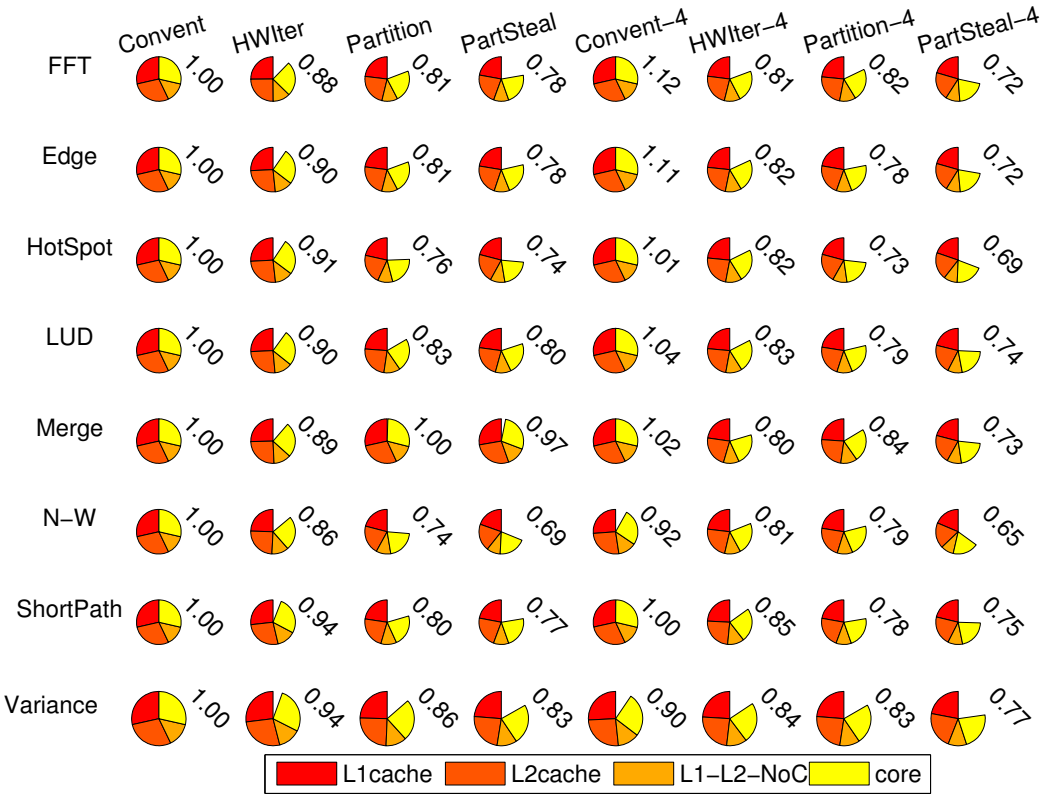


Figure 7. Comparison of energy consumption over eight cores. The total energy consumption is normalized to that of the "Convert" system over eight cores. Energy is broken down into four components: L1 caches, L2 cache, the on-chip network, and the pipeline.

5 Related Work

Several techniques have been proposed to hide data access latency, including *hardware prefetching* [23, 33, 37, 38], *software controlled* or *source level prefetching* [27, 28, 30], *direct memory access* (DMA), and *thread-level speculation* (TLS) [2, 32]. Despite the effectiveness of latency hiding techniques, they do not reliably reduce, and may even increase, the number of data transactions. Prefetching, as an example, can lead to misspeculation. As Vanderwiel et al. have pointed out, prefetching requires a memory system with higher bandwidth. This is to avoid saturation, which is a particular concern for multiprocessors [36]. If care is not taken, misspeculation penalties potentially negate all speed benefit of prefetching and lead to increased energy requirements to boot.

Rather than attempting to ease the pain of long memory latency, data can be partitioned and processed with consideration of locality. Recently, Kulkarni et al. have investigated data partitioning on the Galois system for worklist based applications [21]. For data-parallel applications that exhibit fine-grained parallelism, Debattista et al. identified that short dispatch time is critical for fine grained parallelism on shared memory multiprocessors [11]. They batched threads using the per-processor

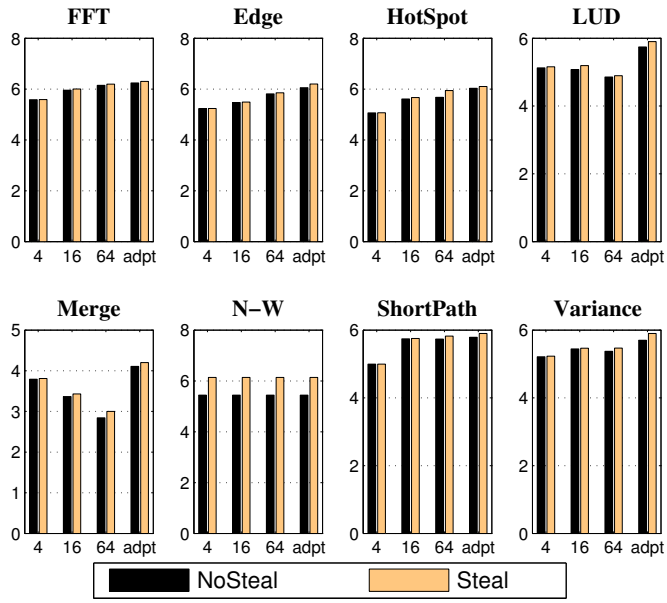


Figure 8. Speedup versus thread block size over 16 cores, comparing different thread blocking implementations including static and adaptive block sizes. In general, adaptive partitioning performs best. The same trend holds with or without work stealing.

run queues. In addition, they promoted cache affinity by scheduling threads on the same processor where they had executed before. Kumar et al. proposed hardware task queues and task prefetchers as hardware support for fine grained parallelism on chip multiprocessors (CMPs) [22]. Both the approaches fail to capture data affinity among homogeneous threads with different input data.

Our target workload overlaps with that of OpenMP, an extensively used programming model for shared memory programming [8]. Our work differs with OpenMP in several key ways:

- **Thread-level data affinity.** OpenMP threads are treated as equivalent virtual processors. There is no guarantee for thread-level data affinity when it comes to hierarchical architectures with multithreading. With our approach, threads with data affinity are placed on each PE, improving locality.

- **Task-level data affinity.** OpenMP encapsulates a large number of loop-independent tasks in one thread. When it comes to parallel nested loops, OpenMP *collapses* the nested loop into a single level loop. Consecutive tasks in the single level loop are then grouped together and distributed to threads. Loop collapsing loses information about the multi-dimensional layout of tasks, which often corresponds to a multi-dimensional layout of data, which in turn may lead to loss of locality.

  Consider the scenario where a parallel, two level, nested loop operates over a large 2-dimensional floating point data set. Each iteration performs a task that consumes a 5×5 neighborhood of elements. A group of 100 consecutive

tasks accesses a 2-D data block with a size of 5 rows and 104 columns, yielding a total of 520 elements. Assuming a four byte representation of each floating point value and a 32 byte cache line, this corresponds to 65 cache lines. Our approach groups the same 100 tasks in a 10×10 domain block. As a whole, tasks within a domain block consume a 14×14 data block, yielding a total of 196 elements and only 28 cache lines. As a consequence, task-level data affinity is improved. Although multi-dimensional loop partitioning is not currently employed in OpenMP, it could be added and our results suggest that this would be beneficial.

- **Robustness over different platforms and environments.** OpenMP relies on the programmer to determine the number of parallel threads. Care has to be taken when the same program is ported to another platform with a different number of PEs. By exploiting fine-grained parallelism and regrouping threads at runtime, our approach adjusts to different platforms automatically. Moreover, stream architectures [5, 9, 13, 17] have emerged and demonstrated performance advantages over traditional architectures in several application domains [14]. Our domain abstraction is not limited to our API nor shared memory programming.

Similar domain concepts have been proposed in other general purpose programming models [6, 7]. Existing graphics programming APIs and languages [26] render polygons while binding pixel shaders that treat screen-space coordinates analogously with domain coordinates and textures like stream data in order to perform general purpose, parallel computation. None of these programming models interpret domains as indicators of data affinity.

Other languages, like Sequoia [12], have been proposed to develop memory hierarchy aware parallel programs. Such techniques help reduce communication and localize computation; however, programmers often find the explicit memory hierarchies daunting.

NVIDIA's *CUDA* (Compute Unified Device Architecture) accommodates completely general purpose programming on graphics processors [29]. CUDA is able to issue a group of threads on several *multiprocessors*. It requires programmers to define threads over a *grid* of *blocks*. While the concepts of grids and blocks bare some similarity with our domain construct, their flexibility and programmability are limited. The lack of a logical shared memory and data communication across multiprocessors raises the need for further programmer optimizations.

6 Conclusions and Future Work

We have demonstrated how an easily expressed domain abstraction can indicate data affinity among threads at a high level. We use low cost hardware domain iterators to interpret the domain at a low level to assist adaptive, hierarchical partitioning for parallel systems. Over hierarchical architectures that execute multiple threads simultaneously on each PE, locality becomes even more important, and domain partitioning is able to co-locate threads with data affinity.

Our experiments show that domain partitioning reduces D-cache miss rates significantly. It also improves performance

and saves energy consistently cross a wide range of scaling parameters and bottlenecks.

On a multiprogramming system, multiple parallel processes may execute at the same time and contend for computational resources. Further study will approach how to share the use of domain iterators among processes, considering their priority and degree of parallelism.

7 Acknowledgements

This work was supported in part by SRC grant No. 1607, NSF grant. No. IIS-0612049, and a professor partnership award from NVIDIA Research.

References

[1] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, UC Berkeley, 2006.

[2] S. Balakrishnan et al. Program Demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *ISCA'06*, 2006.

[3] 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.

[4] D. Brooks et al. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *ISCA 27*, June 2000.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04*, 2004.

[6] B. Chamberlain. An introduction to Chapel: Cray Cascade's high productivity language. *AHPCRC DARPA Parallel Global Address Space (PGAS) Programming Models Conference*, 2005.

[7] P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, 2005.

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

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

[10] A. Das, W. J. Dally, and P. R. Mattson. Compiling for stream processing. In *PACT'06*, 2006.

[11] K. Debattista, K. Vella, and J. Cordina. Wait-free cache-affinity thread scheduling. *IEEE Proceedings - Software*, 150(2), 2003.

[12] K. Fatahalian et al. Sequoia: Programming the memory hierarchy. In *SC'06*, 2006.

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

[14] J. Gummaraju et al. Stream programming on general-purpose processors. In *MICRO 38*, 2005.

[15] 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.

[16] N. Jayasena, M. Erez, J. H. Ahn, and W. J. Dally. Stream register files with indexed access. In *HPCA 10*, February 2004.

[17] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, 2003.

[18] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGARCH Comput. Archit. News*, 30(5):211–222, 2002.

[19] K. Krewell. Intel embraces multithreading. *MP Report*, 15(9), Sep. 2001.

[20] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA'81*, 1981.

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

[22] 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.

[23] R. L. Lee. *The effectiveness of caches and data prefetch buffers in large-scale shared memory multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 1987.

[24] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *ISCA '07*, 2007.

[25] J. L. Lo et al. Converting thread-level parallelism to instruction-level parallelism via Simultaneous Multithreading. *ACM Trans. on Computer Systems*, 15(3), 1997.

[26] W. R. Mark et al. Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. on Graphics*, 2003.

[27] S. A. McKee et al. Dynamic access ordering for streamed computations. *IEEE Trans. Comput.*, 49(11), 2000.

[28] T. Mowry et al. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 12(2), 1991.

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

[30] 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*, 2002.

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

[32] H. Rui et al. Accelerating sequential programs on Chip Multiprocessors via Dynamic Prefetching Thread. *Microprocess. Microsyst.*, 31(3), 2007.

[33] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. *SIGARCH Comput. Archit. News*, 34(2), 2006.

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

[35] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *ICCC*, 2002.

[36] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2), 2000.

[37] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. *SIGARCH Comput. Archit. News*, 33(2), 2005.

[38] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, C. Gniady, A. Ailamaki, and B. Falsafi. Store-ordered streaming of shared memory. In *PACT'05*, 2005.