Strider: Runtime Support for Optimizing Strided Data Accesses on Multi-cores with Explicitly Managed Memories


Published in:

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen’s institutional repository that provides access to Queen’s research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person’s rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Open Access
This research has been made openly available by Queen’s academics and its Open Research team. We would love to hear how access to this research benefits you. – Share your feedback with us: http://go.qub.ac.uk/oa-feedback

Download date: 20. Oct. 2023
Strider: Runtime Support for Optimizing Strided Data Accesses on Multi-Cores with Explicitly Managed Memories

Jae-Seung Yeom
Department of Computer Science
Virginia Tech, USA
Email: jyeom@cs.vt.edu

Dimitrios S. Nikolopoulos*
Institute of Computer Science
Foundation for Research and Technology–Hellas
Email: dsn@ics.forth.gr

Abstract—Multi-core processors with explicitly-managed local memories provide advanced capabilities to optimize data caching and prefetching in software. Unfortunately, these capabilities are neither easily accessible to programmers, nor exploited to their maximum potential by current language, compiler, or runtime frameworks. We present Strider, a runtime framework for optimizing compilers on multi-core processors with software-managed memories. Strider transparently optimizes grouping, decomposition, and scheduling of explicit software-managed accesses to multi-dimensional arrays in nested loops, given a high-level specification of loops and their data access patterns. In particular, Strider contributes new methods to improve temporal locality, optimize the critical path of scheduling data transfers for multi-stride accesses in regular nested parallel loops, and distribute accesses between cores. The prototype of Strider on the IBM Cell processor performs competitively to hand-optimized code and better than contemporary language frameworks, in both non-trivial parallel applications and important application kernels.

I. INTRODUCTION

Many multi-core processors for high-performance and embedded systems use software-managed on-chip memories, also referred to as scratchpad memories or local stores [1], [2], [3], [4], [5]. Software-managed memories enable application-specific optimization of data caching and prefetching by programmers, compilers, runtime systems, or some combination of the three. Automatic management of software-managed memories in a parallel programming framework improves programmability and simplifies code maintenance. However, the added value of automatic management of software-managed memories depends critically on the extent to which the compiler and the runtime system can optimize performance.

Automating caching and prefetching in software-managed local memories has been studied extensively. Prior solutions can be categorized into solutions that delegate data caching to programmers through a simplified API which hides the details of prefetching [6], [7], solutions that semi-automate caching and prefetching by leveraging annotations provided by programmers [6], [8], [9], [10], [11], [12], [13], and solutions that hide all details of caching and prefetching from programmers, presenting them instead with a high-level programming API based on the abstraction of a single shared address space, such as OpenMP [14], [15], [16], [17].

While prior work has contributed to improving programmability of multi-core processors with software-managed memories, current solutions still fall short in terms of performance, even in common cases. Software prefetching and caching of data accessed with unit and non-unit strides raises challenges, including the overhead for generating the transfers—which adds substantial software latency on the critical path—and the difficulty in distributing and scheduling data transfers between cores to maximize parallelism and minimize non-overlapped memory latency. Furthermore, exploiting temporal reuse across partially overlapping blocks of data, while natural for hardware-managed caches, is a difficult task for software-managed local memories.

This paper presents Strider, a runtime framework for transparent optimization of bundles of strided and non-strided memory accesses on multi-core processors with explicitly managed memories. The framework targets loop-dominated programs with regular nested parallel loops, where loop bounds are affine functions of loop indices. Strider automates the generation and scheduling of explicit data transfers in such loops. In particular, Strider contributes to (a) improving temporal locality, (b) optimizing the critical path of scheduling data transfers for array accesses with multiple, unit and non-unit strides in nested parallel loops, and (c) distributing effectively data accesses between cores. The specific contributions of Strider are:

- **Variable-depth buffering**, a new method for grouping data transfers and exploiting temporal locality between partially overlapping data blocks. Variable-depth buffering benefits common computational patterns such as stencils and enables further optimization of data grouping so that overlapping data blocks can be fetched with predominantly contiguous data transfers instead of strided data transfers.

- A new method for characterizing and grouping strided array accesses based on the concepts of *monochromatics* and *wavelengths*, two terms that we borrow from physics. Strider uses monochromatics as the basis for optimizing the critical path of grouping, preparing, and scheduling explicit strided accesses of arrays. The method is based on the idea of grouping accesses with the same stride across arrays.

- A **multi-level decomposition** method of the iteration space of perfectly and non-perfectly nested parallel loops which does not rely on nested parallelism. The method supports
loop collapsing as proposed by OpenMP [18] and enables aggregation of the working sets of multiple tasks scheduled on the same core, to minimize the latency of the associated data transfers.

We deploy Strider as a plug-in to provide runtime support to source-to-source compilers that implement transparent software caching and prefetching for high-level programming models, specifically OpenMP [14], [16]. Strider is also usable as a stand-alone programming framework for application fine-tuning. We present an implementation of Strider on Cell. In addition to the aforementioned broader contributions, Strider implements two architecture-specific optimizations on the Cell: NUMA-aware data placement for even distribution of data accesses between processor nodes with localized DRAM and block shaping for optimizing DMA transfers.

Our experimental analysis shows that Strider performs competitively to painstakingly hand-optimized code and markedly better (up to $3.6 \times$) than contemporary language frameworks for the Cell processor in non-trivial parallel applications and important application kernels.

The rest of the paper is organized as follows. Section II provides an overview of Strider. Section III discusses the design and implementation of Strider. Section IV provides an example of code using Strider. We present our experimental analysis of Strider in Section V. Section VI discusses related work and Section VII discusses future directions. Section VIII concludes the paper.

II. OVERVIEW OF STRIDER

Strider consists of a runtime system, an API, and external code modules provided by the compiler or the programmer. We illustrate the components of Strider in Figure 1. White rectangles correspond to runtime components of Strider. Shaded rectangles represent external code modules conforming to Strider specifications and using the Strider API. Arrows connecting modules represent the flow of module dependencies.

External modules define parallel tasks and the computation kernels encapsulated in tasks via the provided API. The API provides capabilities to offload tasks from control-efficient to compute-efficient cores on heterogeneous multi-core architectures. The Strider prototype on the Cell processor, to which we will refer from this point on in the paper, offloads kernel tasks from the PowerPC core (PPE) of the Cell to the Synergistic Processing Elements (SPEs), which provide the bulk of the computational power of the processor [1]. SPEs have software-managed local stores and Strider automates their management.

The Strider runtime system automatically generates explicit data transfers in the form of DMA lists [1]. The block access list generator is the component that integrates Strider’s novel optimizations. It generates DMA lists based on an explicit description of the iteration space of a nested loop and the loop’s data access pattern. SPE offloaders supply this information via the Strider API. The DMA handler provides abstract interfaces to the DMA primitives of the IBM Cell SDK [19].

From the perspective of a compiler or a programmer, calling the buffer manager in SPE offloaders with a description of the iteration space and the layout of data arrays is sufficient to run the code with automated data caching and prefetching performed by Strider. First, a programmer or a compiler identifies parallel loops to offload and data arrays accessed in the loops, and creates data structures to represent loops and arrays using the Strider API and static or dynamic application parameters. The PPE offloader handles signaling and parameter passing to SPEs. In PPE and SPE offloaders, the programmer or the compiler describes whether to deliver each parameter value from PPE to SPEs before a loop starts, let SPEs calculate parameter values using information available at runtime, or use a hard-coded value. The programmer or compiler relies on the Strider’s automatic decomposition API for parallel execution of loops. Computational kernels must be modified to interface with the parameter passing method of Strider’s runtime in advance and then plugged into the framework via the user kernel interface at runtime. The buffer manager allocates data buffers in local stores, schedules DMA transfers, and invokes the externally provided kernels.

Strider implements new optimizations for minimizing the cost of generating DMA lists, grouping data in order to execute more efficient DMA transfers of contiguous data from memory, and reusing data across tasks from shared buffers in local memories to improve temporal locality. The framework includes custom debugging and profiling tools used in the code development and production phases.

III. DESIGN OF STRIDER

We present the design of Strider and the optimizations that Strider uses to improve caching and prefetching for strided data access patterns on processors with explicitly managed memories. Section III-A introduces the architecture-independent design components of Strider, while Section III-B discusses architecture-specific optimizations on the Cell processor.
A. Multi-stride Data Access Optimization

We present Strider’s essential data structures and follow up with a discussion of monochromatics, decomposition of data accesses between cores and variable-depth buffering.

1) Strider Data Structures: The input to Strider is a high-level description of the iteration space of a loop nest and the data accesses to arrays included in this nest. From this description, Strider identifies data access patterns and schedules explicit data accesses using DMA lists and direct buffering of data in local stores.

Strider uses iterators to map loop iterations to offsets in memory for all array elements accessed in the loop. Strider handles both perfectly and non-perfectly nested loops where the loop bounds are affine functions of the loop indices. The runtime system determines a loop iteration space \( I \) from the loop bounds and the loop step using linear inequalities [20]. Formally Strider represents iteration spaces with a data structure \( I = \mathcal{T} \), where \( \mathcal{T} \) is a vector of index variables, i.e. \( \mathcal{T} = [i_1, i_2, \ldots, i_n]^T \), and each index variable \( i_k \) takes values between a lower bound \( l_k \) and an upper bound \( u_k \). The Strider iterator maps each access to an array element in a loop nest to a unique offset in memory. For each access to an array element \( A[i_1,i_2,\ldots,i_n] \) of an array with bounds \( d_1 \times d_2 \times \ldots \times d_n \), Strider calculates the offset as \( ((\ldots((i_n \times d_{n-1} + i_{n-1} \times d_{n-2} + i_{n-2} \times \ldots + i_2) \times d_2 + i_2) \times d_1 + i_1) \) [21]. Strider represents arrays through array descriptors specifying the array rank, dimensions, and base address. The runtime system constructs descriptors for arrays. Thus, Strider handles both statically allocated and dynamically allocated arrays. Compiler frameworks typically estimate the code size and statically defined data structure sizes to check if a local store has enough space for both. Programmers are to statically allocate each data structure with the maximum size required during execution for compilers’ provision to work. With Strider, on the other hand, it is the runtime system’s responsibility to guarantee that the local store is not overflowed.

Calculating offsets of array elements is expensive on processor cores with pipelines that are not optimized for scalar instruction execution. On the Cell SPE, an integer multiplication costs 7 cycles, an integer addition 2 cycles, and a load 6 cycles. Strider reduces the cost of calculating offsets by performing blocking and strip-mining [21]. The runtime system fetches data to local stores in blocks of contiguous or non-contiguous array elements, after labeling each block with the strides between adjacent elements in each of the block’s dimensions. The runtime system then computes the offsets of elements in each block relatively to the block’s base address by iteratively adding strides, thus avoiding expensive multiplication, division, and modulo operations. The runtime system optimizes further offset calculation by vectorizing it across array elements.

2) Monochromatics: On processors with explicitly managed memory hierarchies, data transfers are most efficient if they transfer large blocks of contiguous data. Data transfers of blocks of non-contiguous data suffer from overhead for the preparation of transfer lists, which is dominated by the calculation of offsets of non-contiguous array elements in memory. Strider introduces monochromatics and waves, to perform grouping and aggregation of data with the same access pattern and maximize the efficiency of data transfers.

A monochromatic is a data structure that encapsulates one or more distinct arrays with identical access patterns, rank, and dimensions. Strider handles these arrays in a bundle. The access pattern of a monochromatic is formulated as a periodic function, a wave, of the innermost loop index. The monochromatic’s period is a constant positive stride between adjacent array elements along the array dimension accessed by the innermost loop index. A positive stride \( s \) and a negative stride \(-s\) are both represented by the same period \(|s|\). Waves with the same period accessing arrays of the same rank and dimension form the monochromatic. The key optimization enabled by a monochromatic is to calculate offsets of array elements once, for all arrays in the monochromatic, and reuse the same DMA list. This optimization substantially reduces the critical path of data transfer preparation and scheduling.

Besides optimization of offset calculation, monochromatics compress the iteration space representation for multiple arrays with identical access patterns, since only the base effective addresses of different arrays needs to be differentiated in the representation. This optimization saves space in local memories. In addition, when buffers are large enough to hold data blocks for multiple kernel invocations, automatic aggregation of data blocks of a wave reduces the overhead of DMA transfer preparation and completion notification. Furthermore, contiguous blocks are merged for more efficient DMA except when data spans the boundaries of an array dimension.

Strider analyzes monochromatics to efficiently generate DMA lists for transferring in (read-only), out (write-only), and inout (read-write) data to and from local memories. Monochromatics are also the basis for implementing dynamic (runtime) dependence analysis between explicit data accesses and dynamic scheduling of dependent accesses in Strider. Dynamic dependence analysis in Strider is beyond the scope of this paper.

3) Variable-Depth Buffering: Variable-depth buffering improves temporal locality by removing redundant data transfers from external memory to local memories and vice versa. Similarly to static multi-buffering (such as the double- or triple-buffering used commonly on Cell to prefetch data to local stores), variable-depth buffering allocates persistent buffers in local stores and fills some of these buffers with data blocks that will be accessed in future kernel invocations to mask memory latency. Buffers are allocated on a per-array basis, with simultaneous translation of array indices to offsets in local stores. The difference to static multi-buffering is that the number of persistent buffers is variable and defined by the references to each array in one iteration.

Figure 2 illustrates variable-depth buffering. We define the reference depth \( r \) of an array in a loop iteration as the number of distinct references to array elements with indices that vary by a constant in a chosen dimension, minus one. For example, the reference depth for \( a[\ ] \) is 2 in the following statements:
In steady state, an externally provided kernel computes using data from the $r + 1$ preloaded buffers of each array and writes back data from write buffers to memory, while issuing DMA transfers to preload the $r + 2$-th buffer of each array, with the goal of overlapping computation with memory latency. Strider merges data blocks of multiple reference depths from the same array in one buffer, to avoid data redundancy in overlapping buffers and minimize buffer management overhead. The epilogue issues the remaining write-backs to memory.

DMA transfers use tags for allowing the runtime system to poll their status and detect their completion. Tag management can be complex with variable-depth buffering because of potential shortage of tags needed to track all outstanding DMAs. Strider implements a new tag distribution and management strategy to increase the number of arrays that variable depth buffering can handle simultaneously. If the buffering depth of an array is $d$, the runtime system requires $d$ tags for DMAs from and to the array. For optimal performance, the runtime system first attempts to assign a unique tag to each buffer of each array. If this is not possible due to the limited number of available tags (32 on Cell), the runtime system splits the available tags in proportion to the number of tags required by the three different types of data accesses, namely in/out/inout. The runtime system also groups arrays with the same access type and the same buffering depth and reuses the set of tags assigned to an array for other arrays in the same group. The scheduling of DMA transfers for arrays in the same group is identical and is also reused by the runtime system.

4) Iteration Space Partitioning and Aggregate Task Blocking: Strider implements automatic decomposition of the iteration space of parallel loops using multi-level tiling, followed by multi-dimensional loop partitioning [22]. The API provides also an optional decomposition for explicit control of loop scheduling. The default loop partitioning method in Strider divides the entire iteration space of a nested loop across all permutable loop dimensions, leading to smaller tasks, better load balancing and better computation-to-memory latency ratio than decompositions of lower dimensionality [22]. This design choice further increases the opportunities of variable-depth buffering to improve temporal reuse of data in partially overlapping blocks that belong to the working sets of different chunks of loop iterations (tasks) scheduled on the same core.

Variable-depth buffering may incur fragmentation of buffer space in local stores, due to the transfer of potentially many disjoint small blocks of array elements. Fragmentation becomes more pronounced if the number of innermost loop iterations scheduled on an SPE with each kernel invocation (i.e. the task size in conventional parallel programming nomenclature) is small. Strider provides an API for performing aggregation of data sets used in multiple tasks in blocks, which further reduces the latency of data transfers to and from local stores. Through this API, Strider reduces the rank of array blocks transferred with DMAs, effectively creating larger blocks of contiguous data and lower dimensionality. These are in turn transferable with fewer and more efficient DMAs than the blocks generated originally by iteration space partitioning. The runtime system performs aggregation by fusing loop levels in the partition of the iteration space assigned to an SPE and re-blocking the fused loops, under the constraint that the aggregated working sets of tasks do not exceed the available space in local memory. This fusion step enables prefetching of working sets of multiple tasks each time the variable-depth

---

**AN EXAMPLE OF THE REFERENCE DEPTH AND THE VARIABLE DEPTHBuffering with $c[i] = f(a[i+2], a[i], a[i]$, $a[i+1], a[i+2], b[i], b[i]$).

<table>
<thead>
<tr>
<th>array</th>
<th>reference depth</th>
<th>access type</th>
<th>buffering depth</th>
<th>buffering start</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a[]$</td>
<td>4</td>
<td>in</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>$b[]$</td>
<td>2</td>
<td>in</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>$c[]$</td>
<td>0</td>
<td>out</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

---

**TABLE I**

An example of the reference depth and the variable depth buffering with $c[i] = f(a[i+2], a[i], a[i], a[i+1], a[i+2], b[i], b[i+1])$. 

Fig. 2. Variable depth buffering for the example of Table I. P, S, E, BD, and $r_{max}$ stand for prologue, steady state, epilogue, buffering depth, and maximum reference depth respectively. A rectangular block represents a buffering step for an array. Figure 4 shows this step in more detail.

Fig. 3. Buffering rotation types for an array: double-buffering (left), triple-buffering (center), and $d$-depth buffering (right).

Fig. 4. A buffering step in detail.

$b[i] = f(a[i], a[i]$, $a[i], a[i], a[i]$, $a[i]$).

and $b[i] = f(a[i], a[i], a[i], a[i]$).

Variable-depth buffering uses $r + 2$-depth buffering for prefetching both read-only and write-only arrays and $r + 3$-depth buffering for prefetching and writing back read-write arrays with reference depth $r$.

More specifically, variable-depth buffering uses a set of rotating buffers for each array accessed in a loop (Figure 3). The runtime system rotates buffers between data prefetching, computation and data write-back (Figure 4). Innermost loops are strip-mined and scheduled in three stages, a prologue, a steady state, and an epilogue. The prologue spans over the first $r_{max} + 1$ iterations of the innermost loop, where $r_{max}$ is the maximum reference depth of the arrays accessed in the loop. During the prologue, the runtime system preloads $r + 1$ buffers for each array with reference depth $r$ starting at every $r_{max} - r$-th iteration of the innermost loop, each time the runtime executes the innermost loop.

In steady state, an externally provided kernel computes using data from the $r + 1$ preloaded buffers of each array and writes back data from write buffers to memory, while issuing DMA transfers to preload the $r + 2$-th buffer of each array, with the goal of overlapping computation with memory latency. Strider merges data blocks of multiple reference depths from the same array in one buffer, to avoid data redundancy in overlapping buffers and minimize buffer management overhead. The epilogue issues the remaining write-backs to memory.

DMA transfers use tags for allowing the runtime system to poll their status and detect their completion. Tag management can be complex with variable-depth buffering because of potential shortage of tags needed to track all outstanding DMAs. Strider implements a new tag distribution and management strategy to increase the number of arrays that variable depth buffering can handle simultaneously. If the buffering depth of an array is $d$, the runtime system requires $d$ tags for DMAs from and to the array. For optimal performance, the runtime system first attempts to assign a unique tag to each buffer of each array. If this is not possible due to the limited number of available tags (32 on Cell), the runtime system splits the available tags in proportion to the number of tags required by the three different types of data accesses, namely in/out/inout. The runtime system also groups arrays with the same access type and the same buffering depth and reuses the set of tags assigned to an array for other arrays in the same group. The scheduling of DMA transfers for arrays in the same group is identical and is also reused by the runtime system.
for (i = 0; i < nx; i += 3)
for (j = ny-2; j >= 1; j-=2)
for (k = 1; k < nz-1; k++)
for (m = 0; m < 5; m++)
C[i][j][k][m] = dt*A[i][j][k][m]+B[i][j][k][m];

Fig. 5. An example of a nested loop.

void nested_loop(int fid)
{
  uint32_t i_start = 0;
  uint32_t i_stop;
  uint32_t unitsz = A.dim[0];
  IterSpace is;
  Iterator iti, ito;
  BlkDesc blk;
  MonoChromatic mci, mco;
  set_blockdesc(blk,BLK_TYPE1,1,unitsz,NULL,0);
  alloc_iterbases(&is, 4);
  set_ibasis(&is,4,0,nx,A.dim[4],3,4,FWRD);
  set_ibasis(&is,4,1,ny-1,A.dim[4],2,3,FWRD);
  set_ibasis(&is,1,0,5,A.dim[1],2,1,FWRD);
  init_iterpace(&is,unitsz);
  i_stop = get_num_iterations(&is);
  compute_bounds(i_start,i_stop,1,Idspe,Nspes);
  alloc_iterator(&iti, &is);
  alloc_iterator(&ito, &is);
  alloc_iterator(&iti, i_start, i_stop);
  alloc_iterator(&ito, i_start, i_stop);
  alloc_monochromatic(&mci, 2, 1, &iti);
  set_wave(&mci,0,A.ea,0,&blk,NULL,NoApnd,1);
  set_wave(&mci,1,B.ea,0,&blk,NULL,NoApnd,1);
  alloc_monochromatic(&mco, 1, 1, &ito);
  set_wave(&mco,0,C.ea,0,&blk,NULL,NoApnd,1);
  set_kernel(fid, kernel_nested_loop);
  set_double_param(pass.dt, 0);
  strider_nLmD_f_pn(fid,&mci,&mco,NULldlen);
  free_monochromatic(&mci);
  free_monochromatic(&mco);
  free_iterator(&iti);
  free_iterbases(&is);
}

Fig. 6. An SPE offloader.

Fig. 8. The descriptor of array double A[nx][ny][nz][5].

number of
dimensions
dim 4 size
dim 3 size
dim 2 size
dim 1 size
element
(dim 0) size
4 nx ny nz 6 (ns+pad) 8

Fig. 7. Loop parameters for the loop shown in Figure 5.

B. Cell-Specific Optimizations

Architecture-specific optimizations of Strider on Cell include NUMA support and block shaping.

1) NUMA Support: Strider implements NUMA aware memory allocation to bind pages to specific nodes and avoid memory accesses crossing the interconnection network between Cell processors on blades. A Cell blade has two nodes with one processor each. When a program starts, the runtime system binds each thread to an SPE and assigns the thread a rank (ID) equal to the order of physical ID of the SPE to which the thread is bound. This ordering performs best with common data access and exchange patterns on SPEs [23].

The runtime system maintains a thread split ratio, which is the ratio of the number of threads with affinity to the SPEs of each node and a page split ratio, which is the ratio of the number of pages to be allocated from each node, out of the total number of pages in the system. The default behavior is to distribute pages between nodes according to the thread split ratio, however the compiler/programmer has the option to modify the split ratio on-demand. The NUMA-aware page binding technique in Strider splits the address space available for program data in N segments, where N is the number of physical nodes. The runtime system logically binds segments to nodes, according to the order in which threads request allocations, the ranks of threads, and their mappings to nodes. This strategy is similar to the first-touch NUMA page allocation scheme [24]. Strider supports also an optional page interleaving pattern which assigns pages round-robin between nodes. Page interleaving often benefits applications that exhibit phase changes in their DRAM access patterns, by spreading out memory traffic more evenly. As an additional optimization, the runtime system may align the memory space allocated to each node so that the boundary between data allocated by different nodes is aligned to the page size.

2) Block Shaping: Software using Strider selects block sizes and shapes through a block descriptor. The block descriptor can represent n-dimensional hypercubes as well as amorphous block shapes. The runtime system uses a DMA list template to describe amorphous blocks, where it adjusts the offsets during execution, based on the loop iteration indices of the currently executed chunk of loop iterations on each core. In our experimental evaluation of Strider (Section V) we use exhaustive search of optimal block shapes and sizes to isolate the impact of these two parameters on performance.

IV. A WORKING EXAMPLE

We use the parallel nested loop in Figure 5 as a working example to demonstrate the use of Strider. Strider composes the offloaded loop from two pieces of code, an SPE offloader shown in Figure 6 and an externally provided compute kernel, which is not shown. The SPE offloader defines the shape of blocks to be transferred, describes the iteration space of the offloaded loop, sets the user-provided kernel, and calls the buffer manager.

Lines 7–10 declare data structures for the iterator, blocks, and monochromatics. Line 12 defines the shape of a block to be transferred. Lines 13–17 describe the parallel loop using the parameters for the loop and the arrays, shown in Figure 7 and Figure 8 respectively. Line 18 pre-computes the strides using the element size of the reference array as input. The block access list generator uses the pre-computed strides in the Iterator to generate memory offsets.
Lines 19–26 define the loop iterator of each SPE and perform iteration space decomposition. Note that SPEs perform the decomposition in a distributed manner, that is, each SPE receives the iterator of the entire loop and computes the local iteration space that the SPE will execute. Statically distributed iteration partitioning is efficient, whereas the fine-grain task partitioning scheme that Strider uses also preserves load balancing. More sophisticated dynamic iteration space partitioning schemes are possible but we do not examine them in this work.

Lines 28–31 specify the monochromatics for the input arrays (A and B) and the output array (C). In this example, all three arrays have identical ranks, dimensions, and access patterns, therefore Strider reuses a single iteration space descriptor (i.e., all three arrays and allocates one monochromatic for input and one for output. The input monochromatic merges arrays A and B, whereas the output monochromatic includes array C. Line 34 links the external kernel. Line 35 passes a parameter to the kernel. The PPE performs parameter passing by directly setting the values via the effective addresses of variables in SPE local stores. Line 36 invokes the Strider buffer manager for the monochromatics.

V. EXPERIMENTAL ANALYSIS

We first present our experimental setup, including our platform and applications. We proceed with the presentation of our experimental results and their analysis.

A. Experimental Platform

We present results from experiments on an IBM QS20 Cell blade with Linux (kernel 2.6.22-5). The blade has two Cell processors with a NUMA organization. The processors reside on two nodes with 512 MB of off-chip XDRAM each. We implement Strider with the IBM Cell SDK 3.0 and compile it with ppu-gcc 4.1.1, and spu-gcc 4.1.1 [19]. We use the time base registers of the PPU and SPU cores to collect breakdowns of execution times.

Our set of benchmarks includes two kernels, copy and transpose, and three applications, Jacobi, Fixedgrid and PBPI. The copy and transpose kernels evaluate the performance of contiguous and non-contiguous data transfers between memory regions in two implementation alternatives, one using Strider as an OpenMP back-end and one using Sequoia, a state-of-the-art programming language with explicit management of the memory hierarchy by the programmer to control locality (see Section VI) [12]. Both kernels are common in scientific applications. The kernels have the minimum operations-per-byte ratio and stress bandwidth utilization and DMA performance. They are sensitive to the overhead of preparation and scheduling of DMA transfers, to DMA block sizes and shapes, and to data placement in DRAM. Furthermore, the shape of the input and output data regions stresses the load balancing capabilities of the runtime system.

Jacobi implements a five-point stencil code [25] and stresses the capability of the runtime system to overlap computation with memory latency and exploit temporal locality. PBPI uses Markov-chain Monte Carlo methods to compute alternative phylogenetic trees, using the maximum likelihood criterion [26]. PBPI stresses the performance of contiguous data transfers with phase changes in the DRAM locations accessed between NUMA nodes. Fixedgrid is the core of a chemical kinetics kernel that operates over a fixed domain grid. The kernel models the evolution of chemical species and is an important component in atmospheric modeling systems [27]. Fixedgrid stresses strided data accesses by performing data discretization along multiple array dimensions.

In the experiments with applications, we compare the performance of Strider against hand-optimized implementations of the applications using the IBM Cell SDK and implementations in Sequoia. We obtain the SDK implementations directly from application domain experts who are not involved in the development of Strider and have already invested significant effort in parallelization and optimization of their applications on Cell. These implementations are documented elsewhere [27], [28], [29]. We introduce additional optimizations to the SDK implementations and present them in detail wherever applicable. We also optimize the Sequoia implementations to the extent of our understanding of the language, compiler, and runtime environment.

To achieve symmetry between different implementations, we use the same SIMD kernel in all implementations. Nevertheless, kernel performance may still differ due to differences in iteration space partitioning, parameter passing between the PPE and SPE kernels, and DMA data marshaling. In addition, we eliminate an additional copy that Sequoia uses to create a replica of the original application arrays on the PPE. Sequoia performs this copy because the data layout used by Sequoia runtime may differ from that in base applications. This capability is non-essential for our experiments but adds overhead as we unify the layout.

B. Experimental Results

1) Kernels: We evaluate the performance of Strider in the copy and transpose kernels using the optimized implementations in Sequoia as our baseline. The kernels copy and transpose 2-D arrays of three sizes, 600×600, 1200×1200 and 1800×1800. We also present additional targeted experiments with regular and irregular shapes of the input arrays to evaluate the impact of block shaping, load balancing, and NUMA-aware page allocation in the runtime systems. In Sequoia, the parallelization of one or more loop levels dictates the block shape and the data decomposition method. The programmer defines this decomposition statically at compile time. By contrast in Strider, the runtime system performs iteration space partitioning and parallelization and allows recalibration on the fly for dynamic optimization.

We select a block size of 30×40 in both experiments following an exhaustive search of the optimized block sizes and shapes for the transpose kernel. The inflicted buffer space is 1,200 double elements. With Strider, we use the first-touch NUMA-aware page placement scheme which optimizes performance in both cases. Unlike Strider, the code generated
blocks of size each framework using DMA-friendly block shapes. We use the Sequoia system, whereas Strider reuses tags (see Section III-A3).

Buffer—28 for 14 double buffers—and 4 tags to the runtime. 14 pairs due to tag shortage. Sequoia assigns a tag to each addition. The array dimensions are 2048 perior bandwidth scaling of Strider in a kernel where we add 3-D matrices and transposes the first and the second dimension of the output matrix. We use two 10×150×200 input arrays, a 10×200×150 output array, and 2×10×20 blocks. Sequoia splits each dimension in 2-, 4- and 2-ways respectively to distribute total 750 blocks to 16 cores as evenly as possible. As a result, 60, 45, 40 and 30 blocks are distributed to 6, 2, 6 and 2 cores respectively. Strider decomposes the iteration space across all dimensions and distributes 47 blocks to 14 cores and 46 blocks to 2 cores. Figure 11(b) shows the difference between the maximum time and the minimum time spent in the computational kernel of the error-transpose. The load imbalance in Sequoia is 36× higher than that in Strider.

2) Applications: Figure 12 shows the speedup of applications. We calculate the speedup as \( \frac{T_1}{T_p} \), where \( T_1 \) is execution time with 1 SPE and the PPE as a control processor and \( T_p \) is execution time with \( p \) SPEs and the PPE as a control processor. Figure 13 shows breakdowns of execution times of applications on 16 SPEs, to provide further insight into the performance of alternative implementations. In all cases, we use exhaustive search to find optimal block shapes and sizes for DMA transfers.

We present results from executions of Jacobi with a 4000×4000 array of doubles as input. The benchmark computes the residual for 1,000 iterations. Strider performs close to the SDK implementation and markedly better (over a factor of 2×) than Sequoia (Figure 12(a)). On 16 SPEs, execution time with Strider is 16% higher than the execution time of SDK code. Compared to Sequoia, Strider reduces memory latency by 97% (shown by the DMA\_wait component in Figure 13(a)). This difference arises because of the data buffering strategy used in each implementation. The implementations with SDK and Strider avoid redundant DMA transfers for the innermost loop of the 5-point stencil and reuse previously buffered data in local stores. Strider implements this optimization via variable-depth buffering (Section III-A3). Sequoia does not provide a mechanism to reuse previously buffered data across tasks. Instead, the Sequoia implementation fetches larger data blocks into local stores compared to those of the Strider and SDK implementations. These blocks need to include additional boundary data from neighboring blocks computed on other SPEs. Furthermore, the Sequoia implementation performs redundant transfers of such data.

In the Sequoia implementation, we experimentally choose a block size and shape (10×256) that minimizes the cost of redundant data transfers and the overhead of handling non-contiguous data at the same time. In the SDK implementation and the Strider implementation, we use linear blocks of size 1×512, i.e. segments of rows that do not span columns. These segments are reused from buffers in local stores whenever needed. Strider provides this optimization automatically whereas the SDK implementation hard-wires it in the code.
Temporal locality optimization not only avoids redundant data transfers, but also minimizes the DMA overhead of transferring non-contiguous data. As a result, the non-overlapped cost of data transfers is higher with Sequoia (Figure 13(a), $DMA_{\text{Await}}$ component). Furthermore, Sequoia implements the reduction of the residual with barrier operations, which are expensive, especially when threads running on different Cell processors exchange data. The other two implementations handle reductions more efficiently by allowing the PPE to directly access the reduction variables on the local stores of SPEs via remote loads and stores.

In Figure 13(a) we show an additional data point corresponding to the original SDK implementation of Jacobi [29], in which the performance of the kernel is significantly lower than the optimized version that we use for comparisons. The original version uses multiple DMA transfer sizes and dynamic buffer resizing in local stores, in order to meet the alignment constraints for DMA transfers. This results in the use of expensive modulo operations for calculating the addresses of scalar array elements while packing them in vectors. Strider avoids this complexity by using fixed-size, variable-depth aligned buffers in local stores and a matching alignment of the application’s arrays in memory. We introduce this optimization in the SDK implementation, which ends up outperforming Strider with lower overhead in preparing DMA (Figure 13(a)).

In PBPI, we use a dataset of 107 taxa with 9,994 nucleotides. PBPI executes three dominant computational kernels in parallel loops. We use three 1-D block shapes of size 32, 32, and 16 for the three loops in the SDK implementation, those of size 112, 96, and 32 in the Strider implementation and those of size 512, 512, and 256 in the Sequoia implementation. In principle, our exhaustive search of block sizes yields larger blocks for the runtime systems that incur higher overheads. Strider runs no more than 13% longer compared to the SDK implementation and up to a factor of $3.6 \times$ better than Sequoia, which does not scale beyond 8 SPEs. Strider reduces DMA wait time by 53% compared to Sequoia (Figure 13(b), $DMA_{\text{Await}}$ component). Most of the difference in performance between the three implementations of PBPI (Figure 12(b)) arises due to synchronization and page allocation.

The overhead of signaling and barrier synchronization during reduction is high in Sequoia (Figure 13(b)). Sequoia uses the mailbox mechanism for inter-core synchronization. A PPE thread writes in the inbound mailbox of each SPE to initiate tasks and reads from the outbound mailbox of each SPE to detect completion of tasks. Before a PPE thread reads from the outbound mailbox of an SPE, it needs to poll the status register in the memory flow controller (MFC) of the SPE to make sure that there is a new unread message in the mailbox. This generates traffic on the ring interconnect of the Cell and degrades performance significantly, especially when it has to reach SPEs across nodes of the blade and if the number of tasks offloaded on SPEs is large. There are 324,941 times task offloading in a PBPI run, while there are 2,592 and 1,000 times in Fixedgrid and Jacobi respectively. In the other two implementations of PBPI, a PPE thread directly writes to and reads from the effective address of a flag in the local store of each SPE to initiate tasks and detect task completion.

Both vectorization and NUMA-aware page allocation improve the performance of PBPI by large margins. The application accesses a fixed-length segment of a one-dimensional array starting from a nondeterministic position in each iteration. This access pattern creates memory hot spots on individual nodes on the Cell blade with evenly distributed pages between nodes using first-touch (Figure 14). PBPI performs better with round-robin page interleaving. This provides a more significant performance benefit in conjunction with vectorization (shown by the difference between $pg-intvl$ and $noNUMA$ versions), because the non-overlapped DMA latency is higher in the vectorized kernel without page interleaving than in the scalar kernel without page interleaving.

The SDK implementation relies on statically allocated buffers, while Strider and Sequoia rely on dynamically allocated buffers. When we plug the original kernel into Strider and Sequoia, we observe significant kernel performance degradation. We also notice the performance difference of the original hand-optimized kernel when we replace static buffers with dynamically allocated buffers (shown as $SDK_{org}$ in Figure 13(b)). The difference stems from additional local store accesses and an inefficient vectorization scheme. The original kernel prepares vector arrays by aliasing scalar arrays to point to buffers. The kernel uses multiple aliases to the buffers to improve code readability. We fix this problem by removing pointer aliasing. This modification improves the
DMA wait is the DMA data transfer time that is not overlapped with computation. DMAprep is the time to prepare DMA lists. In Jacobi, Others accounts for signaling overhead between PPE and SPE and the cost of reduction operations. In PBPI, Others accounts for the cost of executing part of the workload between offloaded regions on the PPE and the cost of reduction operations. In Fixedgrid, Others accounts for the cost of executing part of the workload between offloaded regions on the PPE.

Fixedgrid calculates the concentrations for species of interest, such as ozone, in a M×N domain of size 600×600 from a two-component wind vector, a horizontal diffusion tensor, and the current concentration. The computation in each time step executes five phases of row discretization, parameter updates, column discretization, parameter updates, and row discretization. The implementation on Cell offloads row discretization and column discretization on SPEs using fixed block sizes of 2×600 and 600×2 respectively.

Fixedgrid has high compute density and its vectorized kernels dominate performance. DMA latency is almost entirely overlapped with computation (Figure 13(c), where the DMAwait component is almost invisible). Therefore, all three implementations perform similarly (Figure 12(c)). Execution time with Strider is 2.4% higher than execution time with the SDK implementation, whereas execution time with Sequoia is 14% higher than with the SDK implementation. This result arises from subtle differences in signaling mechanism, parameter passing, and decomposition in the runtime systems. With SDK and Strider, SPEs performs decomposition of iteration space and calculation of starting addresses for DMA transfers in parallel, where each SPE calculates the bounds of its own partition of the iteration space. With Sequoia, the PPE performs these, thereby creating a centralized bottleneck during loop scheduling. We illustrate this difference in Figure 13(c) by comparing the Others component among the three implementations. Sequoia also uses a different SPE-PPE signaling method and incurs higher DMA preparation costs on the PPE. Strider reduces the DMA preparation cost by 21% compared to Sequoia (Figure 13(c), DMAprep component).

The data point denoted as SDKorg in Figure 13(c) shows the performance of the original SDK implementation of Fixedgrid [27]. This implementation vectorizes only the column discretization kernel, which specifically discretizes two columns simultaneously on each SPE. The code uses strided DMA transfers to pack column elements for vectorization in local stores. Vectorizing row discretization requires shuffling of data in local stores. The original implementation interleaves scalar elements in vector arrays, leaving half of the vector execution units unutilized. We modify the row discretization kernel to fully utilize the vector execution units and use the modified kernel for comparison with Strider and Sequoia. This modification brings about a 52% kernel performance improvement and a 55% overall performance improvement.

VI. RELATED WORK

Automated caching and prefetching in software-managed local memories has been studied extensively due to its importance for performance and programmer productivity. Prior solutions can be categorized into solutions that delegate data caching to programmers through a simplified API which hides the details of prefetching [6], [7], [30], solutions that semi-automate caching and prefetching by leveraging data
annotations provided by programmers [10], [11], [12], [13], and solutions that hide all details of caching and prefetching from programmers, presenting them instead with a high-level programming API, such as OpenMP [14], [15], [16], [17].

Library frameworks that delegate software caching and prefetching to programmers such as IBM’s ALF [7], RapidMind [6], and CellMT [30] provide APIs to express contiguous (unit-stride) and non-contiguous (non-unit-stride) data transfers to and from parallel tasks executing on heterogeneous cores. The implementation of these data transfers is hidden in the respective runtime libraries. ALF and RapidMind delegate caching to programmers and support static prefetching schemes via programmer annotations that pin-point opportunities for double- or triple-buffering. CellMT on the other hand implements multithreading on SPEs, which simplifies the task of overlapping computation with communication via a thread-level API. The available documentation of these frameworks does not address issues related to optimizing the code for preparing and scheduling strided data transfers. Strider optimizes strided data transfers through monochromatics, multi-level decompositions of data accesses and framework-specific optimizations. Furthermore, Strider provides variable-depth buffering for improving temporal locality, a capability not available in existing library frameworks. Finally, Strider provides a high-level, yet rich API for controlling the blocking and partitioning of data transfers to programmers and compilers alike.

Parallel programming models such as CellSs [10], [11] and Sequoia [12], [13] rely on user-provided data annotations to perform data transfers to and from software-managed local memories. Similarly to library frameworks, these programming models delegate the responsibility of caching to programmers. The models provide static prefetching capabilities through double-buffering or triple-buffering of selected data streams of array elements. Similar optimizations are provided by the compilers of certain stream programming languages [31], [32]. Strider introduces optimizations of data transfers that extend beyond a single stream with a single stride, while being suitable as a replacement of the data transfer framework in the aforementioned programming models. Recent extensions of both CellSs [33] and Sequoia [34] include runtime techniques (based on scheduling tasks that access common data blocks on the same core) and static techniques (based on loop fusion for reducing data reuse distance) for improving temporal locality. Strider’s variable depth buffering is complementary to these techniques, none of which is applicable to stencil codes. Variable depth buffering targets temporal locality of partially overlapping data blocks accessed by multiple tasks scheduled on the same core and optimizes effectively stencil computations.

Recently, implementations of OpenMP [14], [16] on Cell have demonstrated high performance in parallel applications, compared to hand-crafted implementations of the same implementations using the IBM SDK. Earlier implementations based on a software cache that exhibited low efficiency [35] have been replaced with implementations that include direct data buffering, multi-buffering, and loop optimizations such as fusion and tiling [14], [15] to improve performance. Related work on polyhedral models [17], [36] enables the automatic generation of DMA transfers from affine nested loops. IBM’s DBDB framework [15] is perhaps the closest to Strider. Strider departs from DBDB in several aspects, including the methods for grouping data accessed with multiple strides, the methods for optimizing the code that prepares and schedules DMA lists, and the methods that improve temporal locality. A point-to-point comparison between Strider and IBM’s OpenMP compiler technology would be instrumental but is not feasible due to the proprietary nature of IBM’s code. Our qualitative comparison suggests that Strider provides supplemental optimizations that can benefit OpenMP compiler technologies on multi-core processors with software-managed memories.

VII. Future work

Strider extensions that we are currently exploring include automatic data aggregation for dependent tasks based on input from dependence analysis and automatic derivation of optimal block shapes and sizes. We are also introducing new capabilities for dependence-driven parallel execution, in particular, pipelining and wavefront execution schemes [37]. Future plans include further integration of Strider with the OpenMP standard, exploration of Strider as a back-end for advanced static optimization frameworks [17], and introduction of techniques to exploit on-chip communication and direct core-to-core transfers to further reduce memory latency in Strider. Since IBM’s cancellation of next generation Cell processors, we engaged in porting Strider to platforms based on graphics processing units (GPUs) for supporting applications with low computational density, experimental multi-core processor architectures with explicitly managed memories [38], and Intel SCC [2]. We are also investigating specific aspects of Strider that can benefit multi-core processors with coherent caches.

VIII. Conclusions

We presented Strider, a runtime system for the optimization of strided data accesses in parallel loops, targeting multi-core processors with explicitly managed memory hierarchies. Strider leverages (a) variable-depth buffering to improve temporal locality in stencil codes, (b) monochromatics and aggregate task blocking to reduce the critical path for preparing, initiating, and detecting completion of DMA transfers, (c) multi-level loop decomposition to improve load balancing in nested loops. The runtime system abstracts away details of data transfers and generates efficient code from high-level specifications of loop nests and array references. These specifications conform to common representations used in parallelizing and optimizing compilers, for which Strider serves as an optimization module. The prototype of Strider on Cell outperforms modern language frameworks by up to a factor of 3.6× in kernels and applications dominated by strided access patterns, while performing competitively to hand-optimized code.
Acknowledgments: This work has been supported by grants from NSF (CCF-0715051, CNS-0720750, CNS-0720673, OCI-0904844) and the European Commission (FP7-PEOPLE-224759, FP6-IP-27648, FP7-IST-217068, FP7-ICT-248647). We are grateful to our colleagues at the Barcelona Supercomputing Center (BSC) for providing us access to the IBM QS20 Cell blades.

REFERENCES


