Using the loop chain abstraction to schedule across loops in existing code

Ian J. Bertolacci* and Michelle Mills Strout

The University of Arizona,
Tucson, AZ 85721, USA
Email: ianbertolacci@cs.arizona.edu
Email: mstrout@cs.arizona.edu
*Corresponding author

Jordan Riley and Stephen M.J. Guzik

Department of Mechanical Engineering,
Colorado State University,
1374 Campus Delivery,
Fort Collins, CO 80523, USA
Email: jriley2@rams.colostate.edu
Email: stephen.guzik@colostate.edu

Eddie C. Davis and Catherine Olschanowsky

Boise State University,
1910 University Drive,
Boise, ID 83725, USA
Email: eddiedavis@boisestate.edu
Email: catherineolschan@boisestate.edu

Abstract: Exposing opportunities for parallelisation while explicitly managing data locality is the primary challenge to porting and optimising computational science simulation codes to improve performance. OpenMP provides mechanisms for expressing parallelism, but it remains the programmer's responsibility to group computations to improve data locality. The loop chain abstraction, where a summary of data access patterns is included as pragmas associated with parallel loops, provides compilers with sufficient information to automate the parallelism versus data locality trade-off. We present the syntax and semantics of loop chain pragmas for indicating information about loops belonging to the loop chain and specification of a high-level schedule for the loop chain. We show example usage of the pragmas, detail attempts to automate the transformation of a legacy scientific code written with specific language constraints to loop chain codes, describe the compiler implementation for loop chain pragmas, and exhibit performance results for a computational fluid dynamics benchmark.

Keywords: loop optimisations; loop transformations; loop chain abstraction; data locality; source-to-source transformation; performance optimisation; high performance computing; scientific computing; parallel programming; legacy scientific code.

Reference to this paper should be made as follows: Bertolacci, I.J., Strout, M.M., Riley, J., Guzik, S.M.J., Davis, E.C. and Olschanowsky, C. (2019) 'Using the loop chain abstraction to schedule across loops in existing code', *Int. J. High Performance Computing and Networking*, Vol. 13, No. 1, pp.86–104.

Biographical notes: Ian J. Bertolacci is a PhD student at the University of Arizona. His focus is on providing scientists with high performance computing resources through programming language and compiler technologies. He earned his undergraduate degrees in computer science, psychology, and applied computing technology from Colorado State University.

Michelle Mills Strout is a Professor at the University of Arizona and her research areas are high performance computing and compilers. She earned her PhD at the University of California, San Diego in 2003 with co-Advisors Jeanne Ferrante and Larry Carter. In 2008, she received a CAREER Award from the National Science Foundation for research in parallelisation techniques for irregular applications, such as molecular dynamics simulations. In 2010, she received a DOE Early Career award to fund research in separating the specification of scientific computing applications from the specification of implementation details such as how to parallelise such computations.

Jordan Riley is a PhD student at Colorado State University in the CFD and Propulsion Laboratory. His focus is on developing and applying new programming models for computational fluid dynamic applications. He earned his Bachelor of Science in Aerospace Engineering from the University of Florida, and his Master's of Science in Mechanical and Aerospace Engineering from Illinois Institute of Technology.

Stephen M.J. Guzik is an Assistant Professor in the Department of Mechanical Engineering at Colorado State University. He directs research in the CFD and Propulsion Laboratory with a focus on high-performance computing. He earned his PhD in Aerospace Sciences and Engineering from the University of Toronto Institute for Aerospace Studies in 2010. Before joining Colorado State University, he worked as a Post-Doctoral Researcher in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory.

Eddie C. Davis received his undergraduate degree in Computer Science and Applied Mathematics from Montana State University, Master of Computer Science from Boise State University, and is currently pursuing a PhD in Computing with a scientific emphasis at the same. He has performed distributed bioinformatics research in comparative genomics, and has been an engineer in the semiconductor industry for several years, characterising devices for emerging memory technologies.

Catherine Olschanowsky is currently faculty in the Computer Science Department at Boise State University. She was previously a Research Professor in the Computer Science and Mechanical Engineering Departments at CSU. Her research area is high performance computing application performance. She earned her PhD in Computer Science from University of California at San Diego in the Performance Modeling and Characterization Laboratory. She also previously worked as a Research Scientist and Software Engineer at the San Diego Supercomputer Center.

This paper is a revised and expanded version of a paper entitled 'Identifying and scheduling loop chains using directives' presented at Third International Workshop on Accelerator Programming Using Directives, Salt Lake City, 14 November 2016.

1 Introduction

Many large scientific applications expose parallelism at a shared memory level using a data parallel paradigm. The applications are 'modularised' into a series of parallel and reduction loops. Several programming models, languages, and abstractions expose parallelism in this manner including OpenMP, OpenCL, and OpenACC. The problem is that exploiting all possible parallelism without regard to data locality leads to insufficient arithmetic intensity (i.e., the ratio of computation to memory accesses) and excessive memory traffic, resulting in poor performance and lack of parallel scaling. This paper presents a mechanism to expose and exploit both parallelism and data locality in a series of data parallel loops.

The main limitations of previous work for expressing data locality are that:

1 the programmer is responsible for aggregating computations into tasks

2 tasks are limited to groupings of iterations within a single loop or user-defined functions

3 the programmer has to rewrite full computations in another programming model.

The principal advantage of the loop chain abstraction presented here is that it depends on inserting pragmas, a familiar mechanism, and they can be added to legacy applications, meaning that only the high-level loop chain annotations need to be adjusted for efficient execution on various hardware configurations.

The *loop chain* abstraction represents a sequence of parallel and/or reduction loops that explicitly share data (Krieger et al., 2013). Figure 1 illustrates an example loop chain with two loop nests and the pragmas we propose in this paper. Such coding patterns are often found in stencil codes and other kinds of buffered producer/consumer codes. The loop chain abstraction requires that each loop in the chain is parallel or a reduction (typically an array reduction), has regular, non-sparse, domain, and static access functions that indicate how each iteration accesses data spaces. With these requirements, the loop chain abstraction can be used to derive a partially ordered set of iterations that makes scheduling and determining data distributions across loops possible for a compiler and/or run-time system. The flexibility to schedule across loops enables better management of the data locality and parallelism trade-off.

Providing data access information that enables the compiler to determine dependences and high-level schedule information in pragmas enables domain scientists to incrementally parallelise large production codes. The pragmas specify the loop chain abstraction and schedules for loop chains, which were developed to navigate the trade-off between data locality and parallelism while requiring minimal extra information from programmers.

Figure 1 Example of annotated source code (schedule omitted)

```
#pragma omplc loopchain schedule(…)
{
    #pragma omplc for domain (lb:ub) \
        with (i) \
            write A {(i)}, \
            read B {(i–1), (i), (i+1)}
    for ( int i = lb ; i <= ub; i += 1 )
        A[ i ] = (B[i–1] + B[i] + B[i+1])
    #pragma omplc for domain (lb:ub) \
        with (i) \
            write A {(i)}, \
            read A {(i)}
    for ( int i = lb; i <= ub; i += 1 )
        A[i] = A[i] * (1.0/3.0);
}
```

The current implementation focuses on expressing schedules that balance data locality and parallelism for shared-memory multicore architectures. However, the information provided could be used to automate high-level schedule specifications beneficial to accelerators as well. For example, Grosser et al. (2013) demonstrated the performance advantages of split tiling for stencil codes on GPUs. Additionally, they presented a mechanism for automatic code generation of split tiling code. While currently beyond the scope of this work, this is precisely the type of transformation that is a candidate for inclusion in loop chains.

Manual implementation of the transformations has demonstrated their potential impact. In a previous paper, we manually applied the loop chain abstraction and explored the trade-offs between parallelism and data locality by employing different loop chain scheduling strategies (Olschanowsky et al., 2014). In this paper, we use Jacobi2D as an illustrative example and show how many of those same transformations can be specified at a high level with the loop chain schedule pragma. For performance experiments, we used mini-flux-div, a stencil code representative of computational fluid dynamics (CFD) applications.

The specific contributions of this work include:

- a pragma grammar to specify loop chains and their schedules
- an informal description of the semantics of the schedule commands
- examples of how a user would annotate existing code with the pragmas
- a discussion of the challenges associated with preparing an existing application to accommodate the loop chain abstraction
- a prototype source-to-source translator that implements the pragmas
- a discussion of the current limitations in the implementation.

Our preliminary results indicate that this programming abstraction can serve as a useful tool for developers and maintainers seeking to improve the performance of their application without having to overhaul their code.

2 Loop chain syntax and semantics

A *loop chain* is comprised of a sequence of loop nests with no code occurring between them. The annotations around a loop chain describe the iteration space of each loop nest as an unordered integer set and the data usage patterns as a mapping between iterators and representative data spaces. The combination of the iteration spaces and data usage information describes a partial ordering of iterations that is required for correctness.

A *schedule* indicates the transformations that should be applied to the loop chain. The application of the schedule takes advantage of the fact that the ordering of iterations is partial. The goal is to exploit any flexibility in the partial ordering to balance data locality and parallelism and improve performance.

Figure 2 Full LoopChain directive grammar

⟨nest annotation⟩ → **for** ⟨nest domain definition⟩ ⟨access definition⟩

⟨nest domain definition⟩ → **domain** (⟨expression⟩ : ⟨expression⟩ (, ⟨expression⟩ : ⟨expression⟩)* **)**

⟨access definition> → **with** (⟨id⟩ (,⟨id⟩)*) ⟨access atom⟩ (, ⟨access atom⟩)*

⟨access atom⟩ → (**read** | **write**) ⟨id⟩ **{**⟨iterator expression⟩ (,⟨iterator expression⟩)***}**

⟨iterator expression⟩ → **(** ⟨expression⟩ (,⟨expression⟩)* **)**

⟨loopchain annotation⟩ → **loopchain schedule** **(**(⟨schedule atom⟩ (, ⟨schedule atom⟩)*)?**)**

⟨schedule atom⟩ → **serial** | **parallel** | **wavefront**
| **fuse (**(**(**⟨int⟩ (, ⟨int⟩)***)**)*)**
| **tile (**(**(**⟨int⟩ (, ⟨int⟩)***)**,**
⟨schedule atom⟩, ⟨schedule atom⟩**)**

The loop chain information is communicated via pragmas. Pragmas allow for incremental changes to be made to existing applications. Much like OpenMP pragmas they can be ignored by a compiler that does not support the optimisations. For the purpose of this paper, we use the pragma label *omplc*, indicating that the loop chain transformations have the potential for inclusion within the OpenMP standard. The syntax of a loop chain pragma is similar to that of OpenMP pragmas.

```
#pragma omplc directive-name [clause [[ , ]clause]…]
```

See Figure 2 for the full loop chain directive grammar. There are two directives in this grammar: *for* and *loopchain*. The following sections describe each of the directives and clauses included in the loop chain abstraction as well as their semantics.

2.1 The for directive: domains and access patterns

The domain and access pattern annotations describe the iteration space of each loop nest and the accesses to data made within that space. These annotations only describe the behaviour of the corresponding code; they do not modify or operate on the nests that they describe.

A domain is specified at the top level of each loop nest in the chain. The domains among loop nests within the same loop chain must share dimensionality, but do not need to share bounds. Note that the domains specified are the domains that participate in the scheduling transformations specified for the loop chain. Any loops inner to that domain are treated as a single statement with respect to scheduling. In other words, it is possible for a sequence of loop nests that vary in dimensionality to be included in a loop chain, because it is the domain expressed in the directive that determines the dimensions considered for scheduling, not the code itself (see Figure 3 for an example of this).

Figure 3 An example of a sequence of loop nests of unequal depth and unknown access patterns that could be defined and optimised with the loop chain abstraction

```
for (int i = lb; i <= ub; ++i){
    A[i] = (B[i–1] + B[i] + B[i+1])*(1.0/3.0);
}
for (int i = lb; i <= ub; ++i){
    for (int j = lb; j <= ub; ++j){
        foo(A, B, i, j) ;
    }
}
```

The domain clause within the *for* clause of the pragma takes the form:

```
domain (d1_lb:d1_ub,d2_lb...)
```

The domain of a loop nest is specified as a list of *inclusive* ranges representing the lower and upper bound of each dimension of the loop nest. An N dimension loop nest defined by a k dimension domain (where $k \leq N$), indicates that the k outer loops will participate in the loop chain schedule and the $N - k$ inner loops of the nest will be treated as a single statement.

While a loop nest domain can often be retrieved through program analysis, it may be the case that the syntax of the loop nest does not reflect the domain of interest for loop chain scheduling. For example, the inner most loop(s) may be considered to be the body of the outer most loop(s), such as when iterating within the components of an array structure. This is common in computation fluid dynamics codes, where various physical components are stored at each mesh point in an array.

Each loop nest's data access pattern is expressed as a mapping between the iteration space and the accesses into abstract data space(s). The map can either be expressed as a read or a write access, depending on the action taken in the code. The data access pattern specification occurs after the domain specification leading with the keyword *with* followed by an ordered list of iterators: outer loop to inner loop. The iterator names do not necessarily have to match the actual loop iterators used in the loop nest.

Data space names do not have to match any actual variable names and can be used to aggregate and model accesses to a number of arrays. The data access clause takes the following form, where *f* and *g* are expressions using the available loop iterators:

```
with (i,j,…) read ID{(f(i,j,…)),
                     (g(i,j,…)),…}
             read ID2{…}
             write ID3{…}
             …
```

In the example in Figure 3, the access pattern for the first loop states that the iteration i writes to the data space A using i and that it reads from the data space B using $i - 1$, i, and $i + 1$. In this example, the access pattern in the first loop nest is obvious. However, the access pattern in the second loop nest is obfuscated with a function call. The `with` directive enables a programmer to declaratively indicate how data is being accessed in another function. Another example where program analysis would have difficulties analysing access patterns of interest is the following:

```
double* ptr_1 = buffer + mk_offset(...);
double* ptr_2 = buffer + mk_offset(...);
```

Without precise inter-procedural analysis (and in some cases even with it), it is impossible at compile time to know if `ptr_1` and `ptr_2` are the same or might result in overlapping accesses. For this reason, we have the programmer specify the access pattern explicitly.

*2.2 The **loopchain** directive: scheduling loop chains*

The *loopchain* directive, in addition to indicating the loop chain, communicates the scheduling transformations to be applied to the chain as a whole. This directive is placed at the beginning of the encapsulating scope. Currently, the schedule is specified by the programmer, but it has been designed so it can be used by an autotuner, compiler, or other automated tool when that capability is available.

Figure 1 shows an annotated input to the source-to-source translator. If the schedule command for Figure 1 was `schedule(fuse())` then the resulting code would be Figure 4. If the schedule command was `schedule(fuse(), tile((10), parallel, serial))`, then the resulting code would be Figure 5.

Figure 4 Expected form of transformed code from Figure 1 after loop fusion (*schedule*(*fuse*()))

```
for (int i = lb; i <= ub; i += 1){
    A[i] = (B[i–1] + B[i] + B[i+1]);
    A[i] = A[i] * (1.0/3.0);
}
```

Figure 5 Expected form of transformed code from Figure 1 after loop fusion and tiling by 10 (*schedule*(*fuse*(), *tile*((10), *parallel*, *serial*)))

```
#pragma omp parallel for
for (int tile = floord(lb, 10); tile <= floord(ub, 10);
tile = tile + 1 ){
    for (int i = max(10 * tile, lb); i <= min(10 * tile + 9, ub); i =
    i + 1){
        A[i] = B[i–1] + B[i] + B[i+1];
        A[i] = A[i] * (1.0/3.0);
    }
}
```

Note: `floord` is a C macro that does integer integer division.

A limited set of schedules are included in the design to balance the trade-off between ease-of-use by the programmer and potential performance gains. The initial set of available schedules is built prioritising those that have demonstrated performance impacts on scientific applications. The transformations performed on an application benchmark, mini-flux-div (Olschanowsky et al., 2014), motivated our choice of loop transformations. Transformations that are currently implemented in our prototype tools are *fuse* and *tile*. Additional transformations are currently under development. The following is a short description of each transformation specified by a schedule command.

2.2.1 Syntax and semantics of schedule operations

Currently there are five schedule operations included in the directive grammar: *serial*, *parallel*, *fuse*, *wavefront*, and *tile*. Syntactically, the schedule directive is a list of these schedule operations in the order they are to be applied (Section 2.2). The formal grammar for this portion of the directive can be found in the *loopchain annotation* production in Figure 2.

- *fuse*: *fuse*([$(s_1^1, ..., s_d^1)$, $(s_1^2, ..., s_d^2)$...]). All of the loops in the loop chain are fused using a loop fusion transformation to the depth indicated by their domains. The results are a single loop nest in the loop chain. How much shifting each loop requires will be determined based on the data dependences induced by the data accesses and constrained by the loop domains unless the programmer explicitly provides the *optional* shifting information (e.g., *fuse*([$(s_1^1, ..., s_d^1)$, $(s_1^2, ..., s_d^2)$...]). The value s_d^n tells the translator how far to shift the loop at depth d for loop nest n. The original order of the loops within the loop chain is the order of statements in the fused loop body.
- *tile*: *tile*((s_1, …, s_D), <*outer schedule*>, <*inner schedule*>) indicates that all the loop nests in the loop chain should be tiled. Specifically, the D outermost loops for each loop nest should be rectangularly tiled using a tile of size (s_1, …, s_D), where s_d indicates the tile size in dimension d of the loop nest. The <*outer schedule*> and <*inner schedule*> are the schedules over the tiles and within the tiles. The full grammar for this operation can be found in the schedule atom production in Figure 2.

  Currently, only constant-sized tiles are supported. Providing the schedule command *tile*(16) will result in tiling the outer loop of a loop nest with 16 iterations in each tile. Specifying the schedule command *tile*(16, 16, 16) will create cubic tiles of size 163. The dimensionality of the tiling specified cannot exceed that of the domain provided in the for loop pragmas in the loop nest.
- *wavefront*: This schedule command when at the outermost level of the schedule annotation indicates that all the loop nests in the loop chain should use a wavefront parallelisation strategy. A wavefront strategy turns a loop nest into one with an outer serial loop (no change) and then $D – 1$ inner parallel loops. The $D – 1$ inner parallel loops will be skewed enough to make the parallelism legal. The wavefront command can also be used as the schedule over and/or within tiles.
- *serial*: Typically only used in the context of tiling, the serial directive indicates that either the outer loop over tiles or within tiles should not be parallelised.
- *parallel*: indicates that the outer loop of all loop nests in the loop chain should be parallelised. Also used in a *tile* schedule to add parallelism over or within tiles.

2.3 Automatic determination of required shift distances

Stencil computations are common in scientific applications and cannot be directly fused. Fusing two stencil computations that share data in a producer/consumer pattern requires shifting at least one of the iteration spaces to preserve data dependences.

This can be observed in the Jacobi benchmark code in Figure 6. Figure 7 shows 16 iterations of loop 1 and of loop 2. Each iteration in loop 2 depends on its neighbouring iterations from loop 1. A direct fuse of these loops leads to an illegal dependence cycle between iterations in the fused loop. However, a shift of loop 2 in both iterator directions results in a legal loop fusion and an opportunity to expose wavefront parallelism.

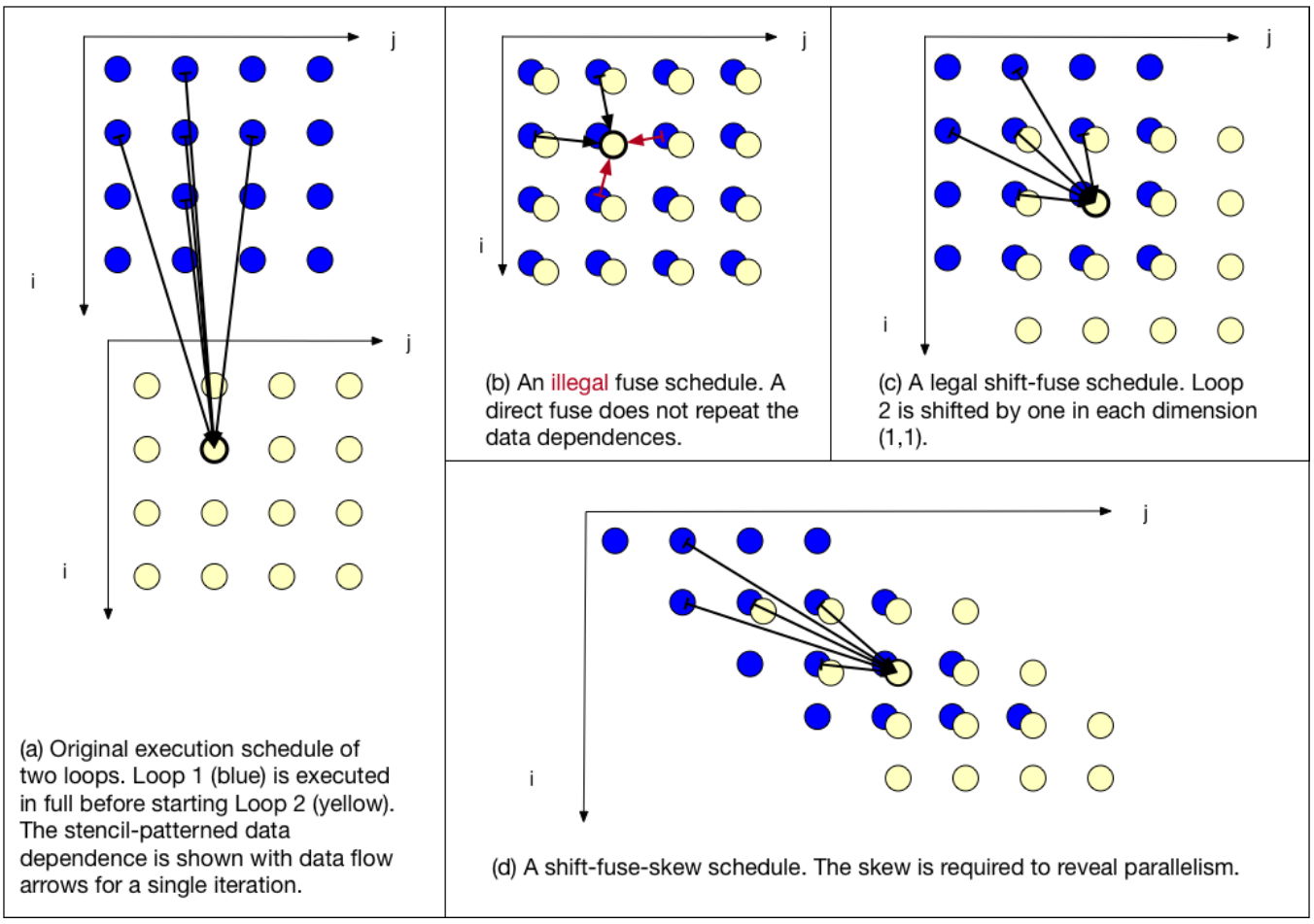
Figure 6 A Jacobi2D benchmark that uses ping pong storage is an example of a sequence of loops that could be defined and optimised with the loop chain abstraction

```
for (int t = 1; t <= T/2 ; t += 2){
    for (int i = 1; i <= N; i++){
        for (int j = 1; j <= N; j++){
            A[i][j] = (B[i–1][j] + B[i][j] + B[i+1][j]
                    + B[i][j–1] + B[i][j+1])*1/5;
        }
    }
    for (int i =1; i <= N; i++){
        for (int j = 1; j <= N; j++){
            B[i][j] = (A[i–1][j] + A[i][j] + A[i+1][j]
                    + A[i][j–1] + A[i][j+1])*1/5;
        }
    }
}
```

In this section, we present an algorithm for determining the required shifting for a loop chain to make a fuse legal. The challenge is to automatically identify the shift extent for each dimension of each loop nest in the loop chain. In this example, the stencil depth is the same for both loop nests and is 1 and this happens to lead to a shift of size 1 in each dimension for loop 2. However, in some scientific codes, the stencil depth varies among loops. Solving this challenge is one of the motivations for including the data access information in the *with* clause of the *for* directive.

The process for determining the shifts requires examining the data accesses of each loop nest compared with all of the loop nests that come before it in the loop chain. Each loop nest has associated with it the read and write patterns for each logical data space expressed using the access pattern grammar shown in Figure 2. Using this information, the shifts required to satisfy the dependences of each dimension of each loop nest can be computed as an integer linear programming (ILP) problem.

Figure 7 Data dependences in Jacobi2D require a shift before fuse (see online version for colours)

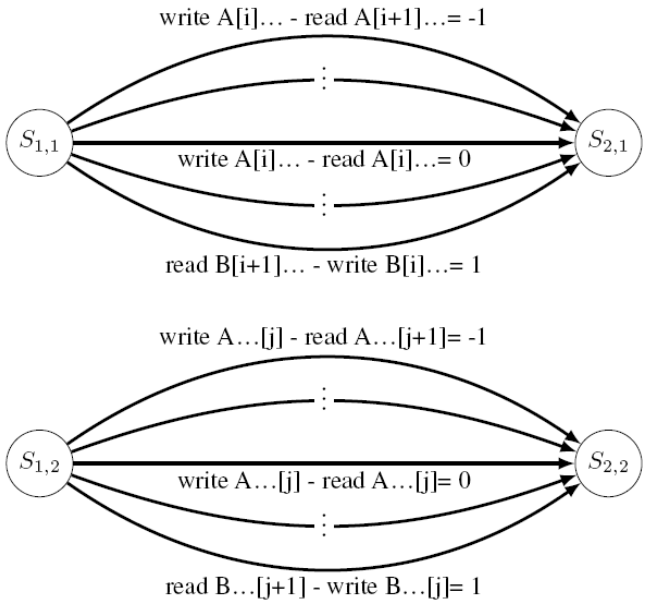


Notes: The original iteration order (on the left) shows the data dependences between the first and second loop nests. The centre iteration schedule is erroneous because there are data dependences that are not satisfied. The final ordering includes shifts of 1 in each direction and is correct.

We specify the ILP problem as a solution of the shift extents S_{ld}, where l indicates the position of the loop nest in the chain and d indicates the dimension of the loop nest. The objective function the ILP solver will minimise is the summation of all the shift extents, $\sum_{\forall l,\forall d} S_{ld}$. To synthesise the constraints of the ILP problem, we first form a weighted, directed multigraph. In this graph, each node represents some shift S_{ld}. If there are two loops x and y where $x < y$, who share a dataspace w that is either read in x and written in y, or written in x and read in y, or written in x and again written in y, then there exists an edge from S_{xd} to S_{yd} in the graph where the weight is difference between two accesses in terms of their constant offsets, $(i_{yd} + c_{yd}) - (i_{xd} + c_{xd}) = c_{yd} - c_{xd}$. There is a separate multigraph for each dataspace dimension. A legality constraint is formed with respect to S_{xd} and S_{yd}, such that $maxWeight(S_{xd}, S_{yd}) = S_{yd} - S_{xd}$, where $maxWeight(a, b)$ is the maximum weight between the nodes a and b. Additionally, all shift extents are individually constrained to be greater than or equal to zero, $\forall l, \forall d, 0 \leq S_{ld}$.

From the Jacobi-2D example in Figure 6, we can produce the multigraphs shown in Figure 8 and from it the constraints: $1 = S_{2,1} - S_{1,1}$, $1 = S_{2,2} - S_{1,2}$, and $0 \leq S_{1,1} \wedge 0 \leq S_{1,2} \wedge 0 \leq S_{2,1} \wedge 0 \leq S_{2,2}$. Our ILP solver produces the shifts $S_{1,1} = 0$, $S_{1,2} = 0$, $S_{2,1} = 1$, and $S_{2,2} = 1$.

Figure 8 Graph forming the constraints of the ILP problem solving for shift extents to perform fusion



In the situation where a tile schedule has already been applied, we convert the original dependences between individual iterations into dependences between tiles by applying the function

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

to all the weights in the graph. The constraints for the ILP problem can be synthesised as usual. We make the assumption that no access $i_d + c_d$ will have a constant c_d greater than the tile size in that dimension. This forces dependencies to be at most between adjacent tiles.

2.4 Determining a skew factor for wavefront parallelism

After loops have been shifted and fused, dependences that once went between loops will now be carried by loops in the fused loop nest. This may have eliminated some parallelism. An example of this can be seen in Figure 7, where after shifting and fusing, both dimensions of the new loop nest carry a dependence thus preventing parallelisation. By skewing the loop nest, we can force the outermost loop to carry the loop dependencies and expose parallelism in the inner dimensions.

To review, a dimension d of a loop nest carries a dependence when the first non-zero in a dependence vector $(k_1, \ldots, k_D)$ occurs in that dimension. After a shift by $(S_1, \ldots, S)$, the dependences are shifted to create the new dependence vector $(k_1 + S_1, \ldots, k_D + S_D)$. After the shift has been applied the fused loop is permutable and therefore all of the entries in the dependence vectors are non-negative, $\forall d, k_d + S_d \geq 0$. Additionally, since we are only considering dependence vectors that cause one of the loops to carry a dependence, for each dependence vector at least one of the entries in the dependence vector will be at least one non-zero, $\exists d, k_d + S_d > 0$.

A conservative solution for skewing would be to skew i_1 by all i_d for $d > 1$, using the transformation map: $(i_1, \ldots, i_D) \rightarrow \left(\sum_{d=1}^{D} i_d, i_2, ..., i_D\right)$. Applying this transformation to each of the dependence vectors results in dependence vectors of the form $((k_1 + S_1) + (k_2 + S_2) + \ldots + (k_D + S_D), k_2 + S_2, \ldots, k_D + S_D)$. Since at least one of the subexpressions in the first dimension of the dependence vectors is non-zero, the outermost loop will now carry all of the dependences.

In the Jacobi 2D example from Figure 6, the dependence vectors in the illegal fuse are {(–1, 0), (0, –1), (0, 0), (0, 1), (1, 0)}. After the (1, 1) shift has been applied, the new, legal dependence vectors are {(0, 1), (1, 0), (1, 1), (1, 2), (2, 1)}. The dependence vectors {(1, 0), (1, 1), (1, 2), (2, 1)} are carried by the outer loop (dimension 1), and the dependence vector (0, 1) is carried by the inner loop (dimension 2). To skew for wavefront parallelism, the mapping would be $(i_1, i_2) \rightarrow (i_1 + i_2, i_2)$. This changes the dependence vectors to be {(1, 1), (1, 0), (2, 1), (3, 2), (3, 1)}. The outer loop (dimension 1) now carries all the dependencies and so the inner dimensions of the loop can be parallelised.

2.5 Ordering and nesting of the scheduling commands

The scheduling commands can be listed in order or nested in a tiling command. A command in the list, not nested in another command, affects all loop nests in the loop chain. For example, the fusion command turns a list of N loop

nests into a list with only a single loop nest in it. Any commands after fusion will be operating on that new list with only a single item. This is the conceptual organisation. To handle shifts and/or loop nest domains in general not lining up, there will be extra edge loops in the code that is generated.

Tiling adds outer loops to the loop nest domains that iterate over tiles and then modifies the loop nest specified domains so that they iterate within a single tile. The tiles will cover the original loop domain for the loop nest.

Thus, tiling splits each loop nest domain into an outer domain over tiles and an inner domain over the points within a single tile. Each of those domains can be scheduled with any of the scheduling commands excluding fusion. Fusion over a single domain does not apply. Fusion is only relevant on a loop chain where there is a list of loop nest outer domains.

Figures 9 and 10 illustrate the impact ordering have in the schedule. In Figure 9, fusion happens before tiling thus resulting in a single loop nest that has been tiled. In Figure 10, tiling is specified before fusion thus leading to only the tile loops being fused.

Figure 9 Original loops with loop chain schedule and corresponding scheduled loops

```
#pragma omplc loopchain schedule(fuse (), tile ((3, 4), serial,
serial))
{
    #pragma omplc for domain(lb:ub, lb:ub) …
    for i1
        for i2
            loopbody_1()
    #pragma omplc for domain(lb:ub, lb:ub) …
    for i1p
        for i2p
            loopbody_2()
}
```

Result

```
for t1
    for t2
        for i1
            for i2
                loopbody_1()
                loopbody_2()
```

3 Legacy code preparation for loop chaining

Chombo (Adams et al., 2014) is a library for solving partial differential equations on rectilinear grids with adaptive-mesh-refinement (AMR). Solutions are implemented on Cartesian grids and exhibit many stencil computations. Applications that use Chombo include land ice sheet modelling, shallow water flows, plasma simulations, Navier-Stokes solvers, and combustion modelling. Block-structured AMR is used to locally refine the mesh, improving accuracy near strong gradients and reducing memory and run time. The block-structured mesh provides many nested loops with well-defined domains that would be candidates for loop chains. However, like many legacy applications, the loop chain abstraction can not be directly applied to Chombo applications. In this section, automatic transformation of a legacy Chombo application to satisfy loop chaining constraints is presented. Impressive results are obtained, but the translator proved to be extremely brittle. Overall, the cost of maintaining the translator may outweigh rewriting the application in a domain-specific language more amenable to loop chaining.

Figure 10 Original loops with loop chain schedule and corresponding scheduled loops

```
#pragma omplc loopchain schedule(tile ((3, 4), serial, serial),
fuse())
{ #
pragma omplc for domain(lb:ub, lb:ub) …
    for i1
        for i2
            loopbody_1()
#pragma omplc for domain(lb:ub, lb:ub) …
    for i1p
        for i2p
            loopbody_2()
}
```

Result

```
for t1
    for t2
        for i1
            for i2
                loopbody_1()
        for i1p
            for i2p
                loopbody_2()
```

Note: This time the tiling is done before the fuse.

3.1 Preparing a Chombo application for loop chaining

Existing legacy applications in Chombo often have loops sunken into Fortran functions, various code snippets between loop nests, and a significant amount of conditional code within the loop to handle boundary conditions. While there are countless opportunities for loop chaining, the legacy applications need to be transformed in preparation for incorporating the loop chaining abstraction. A typical set of transformations and the challenges associated with automating them are presented in an application called Chord, a fourth-order finite volume compressible Navier-Stokes solver with adaptive-mesh refinement and combustion. Use of the library implies some

domain-specific knowledge that can be used to help with the transformations.

Chord is written with the Chombo framework and follows a consistent coding pattern common to Chombo applications. The higher-level data structures are written in C++, and Fortran subroutines are called to iterate over rectangular portions of the domain, called boxes, and perform mathematical calculations, often in the form of stencils. The targeted loops of the loop chain are the loops iterating over these boxes. Therefore, the code targeted for transformation into a loop chain could span several calls to Fortran subroutines.

A source-to-source translator was developed that automates the required transformations using the ROSE compiler framework (Quinlan and Liao, 2011). Transformations required include: translating Fortran subroutines to C, in-lining function calls, relocating code that occurs between loop nests, and control flow consolidation. The original code is annotated with the required loop chain directives and an additional set of helper directives that are specific to the code preparation translator. The transformations are described below along with the challenges associated with their implementation.

3.2 Fortran to C translation

The majority of the compute intensive code in Chombo applications is written in dimension-independent Chombo Fortran to gain the efficiency of Fortran multi-dimensional arrays. This is translated with Perl to dimension-dependent Fortran, i.e., the number of spatial dimensions is selected at compile time. The contents of the Fortran file are translated to C using an existing Fortran to C translation tool developed with the Rose compiler. To accomplish this while keeping a valid AST, the translator creates a child process to translate the Fortran to C. The parent then parses theC file into its AST to have access to the functions. The challenge with this step is not the language translation, but locating the relevant Fortran code.

Figure 11 This is example pseudocode of the C++ code that needs to be prepared for loop chaining

```
#pragma pre-lc begin file (filename) schedule(schedule type)
#pragma pre-lc inline
    call to a Fortran subroutine
    statements that are not part of the loop chain
#pragma pre-lc inline
    call to a Fortran subroutine
#pragma pre-lc end
```

Recall that the calling code is written in C++. This necessarily means that the Fortran code does not exist in the same file as the driving code. During a typical build, the Fortran and C++ object code are combined during the linking phase. During compilation, where the code is transformed, there is no access to Fortran. Figure 11 shows the `pre-lc` annotations used by the 'pre-loopchain' tool to indicate where the Fortran code is located.

The need to indicate the location of the Fortran file is the other limitation of this approach. This places an extra burden on the developer and, perhaps more problematic, introduces an opportunity for failure if the code is refactored and the Fortran code is moved.

3.3 Inlining

Once the contents of the Fortran code have been translated into C, the function calls are replaced with inlined functions. The Rose compiler framework automates this step. However, several steps are taken to modify the inlined block. Some unnecessary variables declared as part of the inlining are replaced. The inlined blocks are flattened to get the loops from multiple function calls into the same block. Therefore, any variable declarations that are not removed get a unique name to avoid naming conflicts. The renamed variables include the variables referenced in the domain directive, and the pragma has to be modified to use the new names. There is a problematic side effect involving the creation of the data access portion of the loop chain directive. The loop chain loop directives are included in the definition of the Fortran subroutine. At that point, there is no context for naming the data spaces. The data space names used do not have a reference to be consistent across multiple Fortran subroutines. The current tool renames the data spaces after inlining based on data flow analysis. This is a challenging step and not always possible. One solution is to put the original pragma at the calling location rather than at the definition of the subroutine. This may be easier for the compiler infrastructure, but if a subroutine is called multiple times in the code it necessitates duplicating information. It also places the data access directives away from the source code it is representing. This challenge is not related to crossing language boundaries, but should be considered for all cases that may include function calls within a loop chain.

3.4 Control flow consolidation

A common pattern in Chombo Fortran applications is to check for boundary conditions within each function as seen in Figure 12 line 7. During the majority of the execution these conditions are false and bypassed. The programmer can designate the statement to be removed with a `strip` directive as shown in Figure 12 line 6. The `strip` directives are expected to be on *if* statements, but it is not necessary. The conditionals are combined with *or* operations and moved outside the loop chain to maintain correctness. The conditional of the *if-else* loop chain statement is replaced with this new combination of conditionals from the loop chain block. In the case that they are true, the code generated for the loop chain is bypassed and the original code is called instead. The structure of the resulting code can be seen in Figure 13.

Figure 12 This is example pseudocode of a Fortran subroutine that needs to be prepared for loop chaining

```
subroutine func(parameters)
    implicit none
    declaration statements
!$ pre-lc loop (loop directives)
    loop over box
!$ chaining pre-lc strip
    if at boundary then
        treat boundary
    endif
    return
end
```

Figure 13 An if-else statement replaces the statements between pre-lc pragmas with begin and end directives

```
if (stripped conditionals) {
    original statements
} else {
    {// loop chain block
        prospective loop chain
    }
}
```

Notes: If any conditionals evaluate to true, the loop chain is avoided. The example of Figure 12 illustrates conditionals that must be removed from the loop chain.

3.5 Relocating intermediate code

Any statement that is not part of the loop chain, but is still within the loop chain boundary must be moved. Much of this code includes declarations and space allocation that can safely be moved before the loop chain. This knowledge is specific to the domain-specific style of Chombo applications.

Figure 14 This is the structure of source code after a pre-loop chain transformation

```
if (stripped conditionals) {
    original code
} else {
    lifted statements
    #pragma omplc loopchain schedule(schedule type)
    {
        #pragma omplc for …
        loop from inlined function
        #pragma omplc for …
        loop from inlined function
    }
}
```

The statements between *for* loops need to be moved. With the functions inlined, the translator can check for dependences. The translator traverses the immediate children of the loop chain block for any statement that has not been designated with a loop directive. When such a statement is found, the statement needs to be removed from the loop chain block. If the statement is a pragma, it is removed and deleted. Otherwise the statement is compared to previous statements in the loop chain block for dependences. If there are any read-after-write, write-after-read, or a write-after-write dependences, the statement cannot be moved before the loop chain. The dependency checking is overly conservative; for example, function parameters are assumed to be read and written. An exception is a member functions that are assumed to not change the object they operate on. This is because of Chombo's extensive use of access functions to pass objects to the Fortran subroutines. If a statement cannot be lifted, this loop chain can not proceed.

After the pre-loop chain transformation has the code in a structure conducive for the loop chain transformation, the loop chain pragmas are inserted. The pragmas corresponding to the loops need to be treated first. The text for the pragma is copied from the original loop location.

3.6 Results

To obtain performance results, the translator was applied to the fourth-order PPM limiter written in Chombo and a simulation of shock dynamics was executed in two spatial dimensions with three levels of AMR. In gas dynamics solvers, limiters are used to suppress numerical oscillations near discontinuities by reducing the order of accuracy of interpolations in an effort to obtain a monotone solution. The PPM limiter features two calls to Fortran subroutines and each Fortran subroutine has a loop nest. The loop nests have a three-dimensional data domain with loops over the slopes, the cells in the y-direction, and the cells in the x-direction. The first loop nest has a read of a dataspace with a three-point stencil and a write to another dataspace. The second loop nest has reads of both the previous dataspaces, and each read has a five-point stencil. All of the stencils are written in one-dimension and then evaluated in both the x and y-directions.

For the shock-dynamics case with 3 levels of adaptive mesh refinement, the coarsest grid is 64 × 64, and each AMR level has a refinement ratio of 2. The maximum box size is set to 32 cells per direction. The adaptive mesh refinement increases the number of boxes and the total number of cells as the simulation progresses. The AMR shock box case was run with three schedules for the PPM limiter: the original version with calls to Fortran subroutines, the loop chain with an empty schedule, and a loop chain with a fuse schedule.

These cases were run on a single core of an Intel Xeon E5-2670 v2 CPU at 2.50 GHz clock frequency. The cores include a 32 KB L1, 256 KB L2, and 25,600 KB L3 caches. GCC g++ version 4.9.2 was used to compile all cases with

the `-O2`, `-ftree-vectorize`, and `-funroll-loops` optimisation flags passed to the compiler.

The performance results are given in Table 1. The timings are the total time the simulation was in the PPM limiter function for each AMR level. Converting the original code into a loop chain with an empty schedule reduced the total time in the PPM limiter. Fusing the loop nests further reduced the time for an overall reduction of approximately 10%.

Table 1 The total time spent in the PPM limiter on each AMR level for different schedules is shown in seconds

	Level 3	*Level 2*	*Level 1*
Original	7.46	2.27	0.57
No schedule	6.89	2.19	0.52
Fuse	6.66	2.08	0.5

4 Loop chain source-to-source translator infrastructure

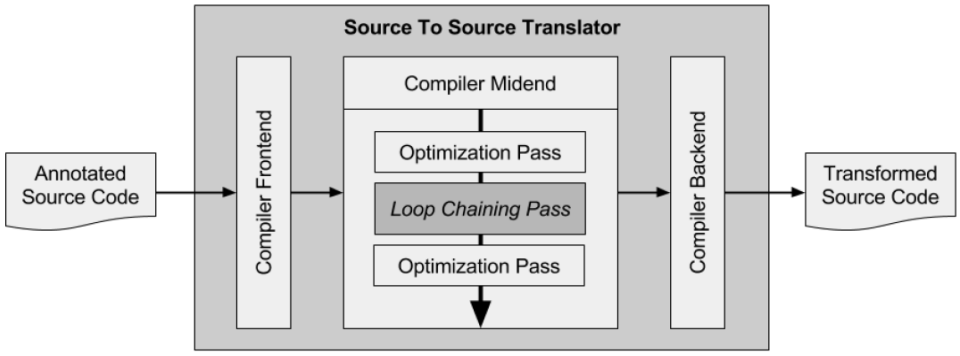
This section summarises how we translate C code annotated with loop chain pragmas into C code that has been transformed as specified in the loop chain pragma schedule. The key contributions of our work are the loop chain intermediate representation and the approach for converting loop chain pragma schedule commands into affine loop transformations. We first give an overview of the loop chain translation pass, and then present the approach for composing loop chain scheduling commands, and how it is enabled by the LoopChainIR.

Two main components comprise the code transformation infrastructure. The first, LoopChainIR, is an internal representation specifically designed to represent and support loop chain transformation. The second is a prototype implementation of a transformation pass written in the Rose compiler framework (Quinlan and Liao, 2011) that utilises the LoopChain directives and LoopChainIR to perform loop chain transformations. The transformation has been designed as a single, discrete pass that could be used in conjunction with other transformation passes within a compiler.

4.1 Loop chain transformation pass

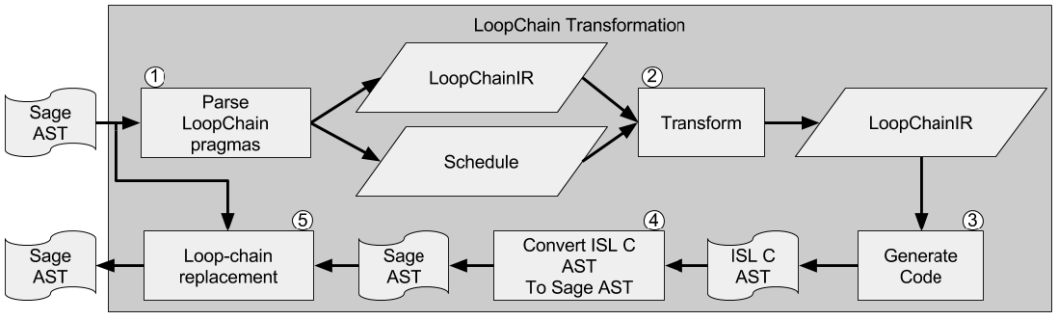
The transformation pass is implemented as a visitor in the Rose framework and performs the actions required to go from loopchain directives to transformed and integrated code. The call to transform source code based on loop chain pragmas can be inserted at any point during the optimisation stage of compilation. The transformation results in a valid AST and, therefore, can be followed by additional transformations if desired. Figure 15 shows a larger overview of process from frontend to backend. Figure 16 gives an overview of the LoopChain transformation process and the flow of different ASTs and placement of various components. The steps are numbered here to indicate their relationship to Figure 16.

Figure 15 A high-level overview of the usage scenario of this prototype infrastructure



Note: The loop chaining pass is designed to fit seamlessly into the optimisation pipeline.

Figure 16 Overview of the process within the LoopChain transformation pass



The first step (1) is to parse the loopchain directives. Parsing the `nest annotation` portion of the grammar produces a series of LoopChainIR objects representing loop nests and their domains, with associated iterator symbols and loop bodies. Parsing the `loopchain annotation` portion of the grammar collects those loop nests into a list, ordered by their position within the chain. The next step (2) is to take the list of loop nests and the scheduling directives and generate code. Once the scheduling directives are brought together with the entire loop chain, a series of LoopChainIR transformation objects are created to implement each of the scheduling directives. These transformation objects create functions that, along with representations of the loop nests, are passed to the polyhedral code generation tool to generate the transformed code. At this point, our code generator can generate its C AST representing the transformed loop chain (3).

The last step is to take that AST, convert it into Rose's Sage AST (4) and inject it back into the source AST (5). Importantly, the loop bodies from the original code must be injected into the bodies of the generated code, and iterator symbols from the original code matched to the iterators in the generated loop code.

4.2 Forming affine transformations from scheduling directives

LoopChainIR utilises integer set library (ISL) (Verdoolaege, 2010, 2016) to perform code transformation and code generation. To accomplish the transformations, LoopChainIR needs to provide ISL with representations of loop nests' initial iteration space, and the transformation functions that will be applied.

Iteration spaces are represented as sets of integer tuples, where each tuple is a single iteration of the whole loop nest. Each position in the tuple is a dimension of the loop nest, and the values are constrained by the loop iterator constraints. For example, the iteration space of the first loop nest from Figure 6 is $\{[i, j] \mid 1 \leq i \leq N \wedge 1 \leq j \leq N\}$. When code is generated to represent an iteration space, it is generated so that the sequence of iterations the code produces preserve the lexicographic ordering of the iteration space. Iteration $[i_1, i_2, \ldots, i_D]$ is lexicographically before iteration $[j_1, j_2, \ldots, j_D]$ if $(i_1 < j_1) \vee (i_1 = j_1 \wedge i_2 < j_2) \vee \ldots \vee (i_1 = j_1 \wedge i_2 = j_2 \wedge \ldots, \wedge i_D < j_D)$ (where $\vee$ represents a Boolean 'or' and ^ represents Boolean 'and').

For ISL, LoopChainIR provides the initial iteration spaces (or domains) as strings. For example, the strings provided to ISL to represent the nests of Figure 6 are:

```
[N]->{ statement_1[i,j] : 1 <= i <= N and
1 <= j <= N };
[N]->{ statement_2[i,j] : 1 <= i <= N and
1 <= j <= N };
```

The syntax '`[N]->`' tells ISL that there exists a symbolic constant N in the constraints. Additionally, the tuples of these sets are given a statement macro that will be generated to the code to represent all the statements in the loop body. It appears similar to a function call: `statement_0(c0,c1)`.

Transformations are functions that map from one iteration space to another. These are also provided to ISL as strings. The first function that is created embeds all the independent iteration spaces of the loop nests into a single iteration space that comprises the entire loop chain:

```
{
  statement_1[i,j] -> [loop_idx, i, j,
  stmt_idx] :
    loop_idx = 1 and stmt_idx = 1;
  statement_2[i,j] -> [loop_idx, i, j,
  stmt_idx] :
    loop_idx = 2 and stmt_idx = 1;
}
```

Because of lexicographical ordering, all the iterations of the first nest (`statement_1`) will come before any iteration of the second nest (`statement_2`).

After this initial transformation, the iteration space matches the input loop chain and new transformations can be applied. These transformation functions do not need to reference the statement macro. They only need to take in an iteration space of the same dimensionality as the iteration space output by the previous transformation function. All the transformation functions are then composed together by ISL and applied as one onto the original input iteration spaces. No intermediate C code is created during the transformation process.

An important concept in our framework is that an iterator tuple is a concatenation of one or more subspaces. A subspace itself is a tuple of zero or more 'variable' iterators, and one 'constant' iterator. A variable iterator is an iterator whose constraints may have more than one solution (e.g., `1 <= i <= N` and `1 <= j <= N`). A constant iterator is an iterator whose constraints have exactly one solution (e.g., `loop_idx = 1`). The constant iterator is always the last iterator of the subspace tuple. This constant iterator indicates the position of the statement within the variable iterators, and is how statements of fused loop bodies maintain their order in the new loop.

In the running example, there are two subspaces [`loop_idx`] and [`i, j, stmt_idx`]. In the [`loop_idx`] subspace, there are no variable iterators, and `loop_idx` is the constant iterator. In the [`i, j, stmt_idx`] subspace, `i` and `j` are the variable iterators, and `stmt_idx` is the constant iterator. The complete iterator tuple is the concatenation of these two: [`loop_idx`] + [`i, j, stmt_idx`] = [`loop_idx, i, j, stmt_idx`].

While subspaces are not provided to ISL, they are tracked and used in LoopChainIR while synthesising the ISL transformation functions. Subspaces allow LoopChainIR to perform nested transformations, such as nested tiling. All base level transformations are synthesised with respect to one of the default subspaces, [`loop_idx`] or [`i, j, …, stmt_idx`]. A base level tiling directive (such as `schedule(tile(…)…)`) will create a transformation

function with respect to [i, j, …, stmt_idx], and produces a new tiling subspace, [t_i, t_j, …, t_stmt_idx].

For example, a 10 × 20 tiling of the running example would be:

```
{
  [loop_idx, i, j, stmt_idx] ->
    [loop_idx, t_i, t_j, t_stmt_idx, i, j,
    stmt_idx]:
      loop_idx = 1 and t_stmt_idx = 1 and
      T_i * 10 <= i_1 < (t_i + 1) * 10 and
      T_j * 20 <= i_2 < (t_j + 1) * 20;
  [loop_idx, i, j, stmt_idx] ->
    [loop_idx, t_i, t_j, t_stmt_idx, i, j,
    stmt_idx]:
      loop_idx != 1 and t_stmt_idx = 1 and
      t_i = 1 and
      t_j = 1;
}
```

Any nested transformations, in this case, over or within tiles, will be synthesised with respect to the appropriate subspace. In this case, transformations over tiles will be synthesised with respect to the new tiling subspace, [t_i, t_j, t_stmt_idx], and transformations within the tile will be synthesised with respect to the tiled subspace, [i, j, stmt_idx].

4.3 Loop chain intermediate representation

LoopChainIR is a C++ library that provides abstractions, as classes, of loop nests, loop domains, loop chains, code generation ready schedules, and transformations to apply these schedules. This library serves to encapsulate a loop chain, provide developers with a way to package a loop chain optimisation, and provide compiler developers with a way to add loop chain optimisations into their optimisation toolbox, while also being simple and independent of any one compiler framework.

Figure 17 demonstrates the simplicity of using LoopChainIR, building a representation of the code from Figure 6, and then performing the equivalent of schedule(fuse((0, 0), (1, 1)), tile((10, 20), serial, serial)). The output of this demo is Figure 9. This code is stand alone, requiring only the LoopChainIR and ISL libraries, allowing it to be used by any compiler infrastructure.

LoopChainIR was designed so that the task of describing a loop chain, and the task of transforming it, are largely done by different components of the library. The classes LoopChain, LoopNest, and RectangularDomain, are used to describe the loop chain. The classes schedule, subspace, and the transformation class hierarchy are used to describe transformations on the loop chain. The schedule class takes in a constructed LoopChain and produces the initial ISL iteration spaces, subspaces, and transformation function that are required for subsequent transformations to be allowed. Transformation classes themselves can be created separately from a schedule or from a LoopChain. Until it is used by the schedule class, a transformation class does not require any more information than the parameters needed to describe the transformation, such as shift factors and tiling extents. This decoupling allows different portions of a compiler to synthesise the loop chain representation, and the transformation representations separately.

Figure 17 In this demonstration of LoopChainIR, the schedule fuse((0, 0), (1, 1)) tile((10, 15), serial, serial) is applied to the Jacobi code in Figure 6

```
// Create a loop chain. LoopNests will be appended to it.
LoopChain chain;

// Lower and Upper bounds, in dimensional order.
string lb[2] = {"1", "1"};
string ub[2] = {"N", "N"};
// Symbolic constants present in those bounds.
set<string> symbols = {"N"};

// Create a loop nest from a rectangular domain those bounds
form.
chain.append(LoopNest(RectangularDomain(lb, ub, 2,
symbols)));
// Create an identical loop nest.
chain.append(LoopNest(RectangularDomain(lb, ub, 2,
symbols)));

// LoopChain has been created, transformations can now be
done.

// Create a schedule object from that chain.
Schedule sched(chain);

// Create a shift transformation for loop 1, shifting both
dimensions by +1
Transformation* shift = new ShiftTransformation(1, {"1",
"1"});

// Fuse loops 0 and 1 together.
Transformation* fuse = new FusionTransformation({0, 1});

// Tile both loops
TileTransformation :: TileMap extents;
// In LoopChainIR dimensions numbering begins at 0
// Size of tile in dimension 0.
extents [0] = "10";
// Size of tile in dimension 1.
extents [1] = "20";
```

Figure 17 In this demonstration of LoopChainIR, the schedule `fuse((0, 0),(1, 1)) tile((10, 15), serial, serial)` is applied to the Jacobi code in Figure 6 (continued)

```
// In LoopChainIR loop numbering begins at 0
// Make tile transformation for loop 0.
Transformation* tile_0 = new TileTransformation(0, extents);

// Take the transformations and apply them, in order, to the schedule.
sched.apply({tile_0, tile_1, shift, fuse});

// Hand-off to ISL to perform transformations and code generation.
sched.codegenToFile("output.c");
```

Figure 18 Output found in "output.c" after running Figure 17

```
for (int c1 = 0; c1 <= floord(N + 1, 10); c1 += 1){
    for (int c2 = 0; c2 <= (N + 1) / 20; c2 += 1){
        for (int c4 = max(max(1, 10 * c1), –N + 20 * c2 + 1);
                c4 <= min(N + 1, 10 * c1 + 9); c4 += 1){
            for (int c5 = max(max(1 , 20 * c2), –N + c4 + 1);
                    c5 <= min(min(N + 1, 20 * c2 + 19), N + c4 –
                    1); c5 += 1) {
                if (N >= c4 &&N >= c5)
                    statement_0(c4, c5);
                if (c4 >= 2 && c5 >= 2)
                    statement_1(c4 – 1, c5 – 1);
            }
        }
    }
}
```

The schedule class does the managerial work during the transformation process. It keeps track of the subspaces that transformation classes use to help construct their constraints, and provides the input and output iterator tuples that form the rest of the function.

5 Experimental results

The goal of our evaluation was to verify that the translation tool produced correct results and that the generated source code obtained similar performance to a more manual approach. This was achieved by adding loop chain pragmas to an application benchmark, miniflux-div. The automatically generated code's performance matched that of the manual transformations. Additionally, previous work demonstrated that large performance gains were achieved through a combination of execution schedule optimisations and space optimisations. The loop chain pragmas only enable the execution schedule optimisations.

5.1 Application benchmark

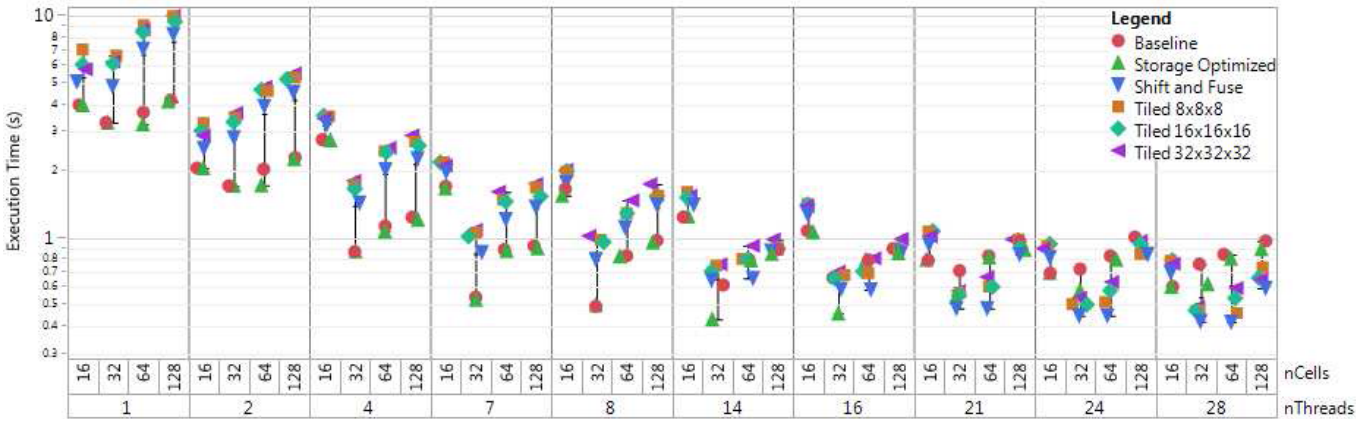
The application benchmark selected for this experiment was mini-flux-div, a partial implementation of the stencil computations used in the PDE solvers of CFD simulations. The original benchmark was developed to emulate the behaviours typically found in applications developed within the Chombo framework (Olschanowsky et al., 2014). The current benchmark has been modified to remove dependencies on Chombo, easing installation. The direct software dependencies have been removed, however, the pattern similarities were maintained. The mini-flux-div benchmark is available as part of the *Variations on a Theme* (Strout et al., 2017) benchmark repository.

A finite volume method is applied to discretise the domains of the flux calculations. The fluid volume is approximated by a number of boxes, each containing cells arranged in the *x*, *y*, and *z*-directions. Each cell contains five component values: density, energy, and the velocities in the *x*, *y*, and *z*-directions. The flux is calculated on each face of each cell, with the approximate solution accumulated from the values of adjacent cells. A layer of ghost cells surrounds each box.

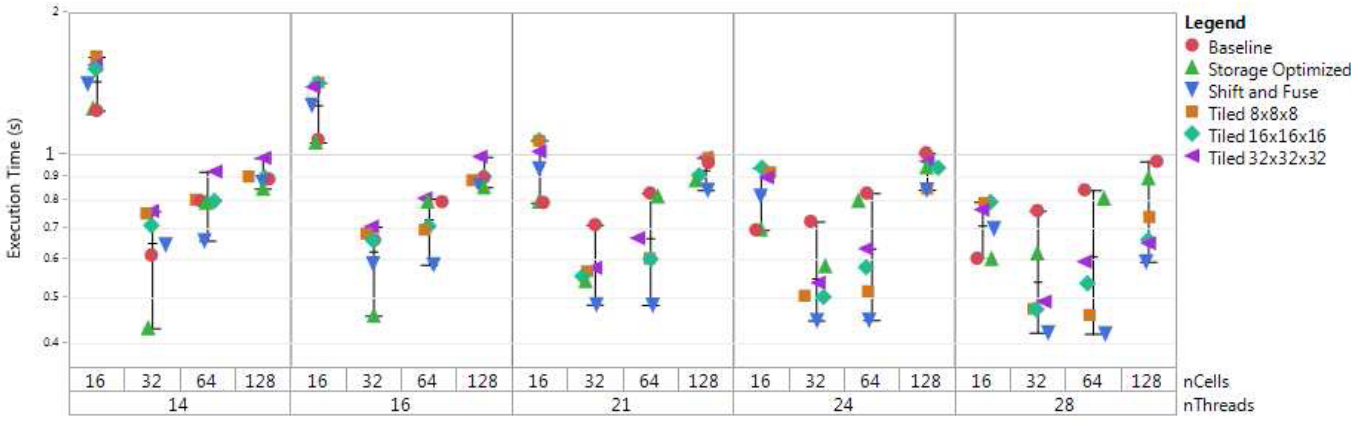
The calculation is expressed as a series of nine loop nests. The flux is calculated in the first two loop nests of each group of three. In the first loop nest the calculations for each dimension consist of three passes, first the initial values of the five components are produced from previous results (*Flux*1), in the next the values are updated with the velocity component (*Flux*2), and finally, the results are accumulated into adjacent cells (*Diff*). Each of the three passes must iterate over each cell in all three dimensions, for a total of nine loop nests. The five flux components are calculated individually in the initial baseline program, such that each nest contains only three loops rather than four. No data dependences exist between boxes, allowing them to be executed in parallel using OpenMP (box level parallelism).

The benchmark was tested over a 3D data domain, sweeping both the number of boxes and the number of cells per dimension in each box. The total cell count of 58,720,256 is constant for each experiment. The number of threads was also swept from 1 to 28, i.e., the number of cores in the experimental machine, to evaluate the effects of box level parallelism. The baseline code has only box level parallelism, with no inner loop transformations, and contains a single velocity cache shared by the three dimensions. A full cache, one that is written only once at each location during execution is used in the experimental program to enable loop transformations, otherwise prohibited by data dependences.

Figure 19 Experimental results of the mini-flux-div micro-benchmark, (a) overall performance results (b) zoomed view of lower-right of plot to highlight improvements (see online version for colours)



(a)



(b)

Table 2 Descriptions for each of the execution schedules presented

Legend label	*Schedule*	*Description*
Baseline	Series of loops	Original implementation with expanded storage allocation to enable transformations.
Storage optimised	Series of loops	Original implementation with storage allocation reduced.
Fuse	Shifts align Diff operations	Shift and fuse are applied to baseline implementation.
Tiled 8 × 8 × 8	Fuse then tile	
Tiled 16 × 16 × 16	Fuse then tile	
Tiled 32 × 32 × 32	Fuse then tile	

The loop chain pragma was added around the loop nests within the primary box loop, instructing the tool to apply the various schedules. Additional loop chain pragmas specifying domain and data dependence descriptors were added to the nine inner loops for the three dimensions calculated. For example, the *Flux*1 calculation reads data from the previous time step, and writes to the flux cache. *Flux*2 both reads from, then writes to the cache. In the accumulation step (), data are read from cache, and written to the current time step.

Several fused schedules are possible given the data dependencies and domains involved in the mini-flux-div benchmark. The domains of each pair of loop nests calculating the flux are rectangular and the same shape. Additionally, the data dependencies are point-to-point, making a direct fuse possible. The domains for the *Diff* operation are square and shifting the iteration space is required for correctness. The first four loop domains are flattened to be two-dimensional. There are two different legal shift-fuse configurations. We manually confirmed that the transformation tool produced the better performing of the two configurations, with all of the writes to a single data location contained within the same iteration and less control flow overhead. This is the 'shift and fuse' example in Figure 19. Three other schedules were evaluated, fused and tiled, with three different sets of tile dimensions, 8 × 8 × 8, 16 × 16 × 16, and 32 × 32 × 32. The overall results are summarised in Figure 19(a) and the different execution schedules are summarised in Table 2.

5.2 Experimental setup

All experiments were run on a single node of a multi-node cluster, R2 at Boise State University. Each node is composed of a dual socket, Intel Xeon E5-2680 v4 CPU at 2.40 GHz clock frequency with 28 cores (14 per socket). The cores include a 32 KB L1, 256 KB L2, and 35,840 K L3 caches. The system contains 192GB of RAM split over 2 NUMA domains. GCC g++ version 4.8.5 was used to compile all the benchmarks, with the `-O4` optimisation flag passed to the compiler.

5.3 Results

The shifted and fused code generated by the loop chain tool outperforms the baseline code in a number of experimental cases as the thread (core) count increases, most notably when the number of cells is 32 or 64. This is an expected result because the increased amount of storage required for the baseline schedule. Figure 19(b) contains a zoomed view of the performance results. The three fused and tiled schedules did not outperform the baseline in any of the test cases as in previous work (Olschanowsky et al., 2014), because the temporary data footprints have not been optimised.

The overall trend for the generated code to start out slower than the baseline at low core counts and close the gap as more cores are added is expected and worthy of further explanation. The baseline code has a temporary space optimisation that reuses the same data to communicate values between the producer loop nests (the *Flux*1 and *Flux*2 loops) and the consumer loop nest (the *Diff* loops). The generated schedules use a maximal temporary data scheme that will support any generated schedule. This translates to a 3X increase in temporary data usage. The baseline code was modified to include the full cache access pattern that shares the same memory overhead, baseline in Figure 19). The optimised schedules mostly outperform this version for higher thread counts.

6 Related work

The need to manage data locality and parallelism in tandem was first tackled by Kennedy and McKinley (1992). Their work focused on trading off data locality and parallelism within individual loop nests. In the 1990s the main issue was false sharing and creating enough coarse grain parallelism. Today's machines demand a wider variety of scheduling strategies, therefore the programmer might best be helped by providing orthogonal abstractions for specifying different schedulers either manually or with an autotuner.

Other approaches to balancing data locality and parallelism include domain-specific languages and compilation passes, aggregation of computation in tasks, and lower-level programmer-guided program transformation. Loop chaining differentiates itself from most of the previous work by removing complex tasks from the domain of the user. For example, *task aggregation* requires users to rewrite large portions of the code. Loop chaining depends on the user annotating existing code with summary data access information. This interface allows the optimising compiler to make decisions not always possible through data analysis. *Optimisation scripting languages* also enable programmer control over the schedule as the loop chain schedule commands do, however, the scripting interfaces are lower level and more complex to enable the expression of a broader range of transformations. The advantage of loop chaining is that it enables the programmer to have some control over the schedule in a loop chain while still hiding the complexity, thus making transformations across sequences of loops that share data more practical.

6.1 Domain-specific languages and transformations for stencil computations

The loop chain abstraction is most relevant to pipelines of stencil computations. Stencil computations are prevalent in image processing pipelines and in partial differential equation solvers. Thus there have been a number of domain-specific languages developed for specifying stencil computations (Tang et al., 2011; Christen et al., 2011; Henretty et al., 2013; Stone and Strout, 2013). The main issue with applying such approaches to existing code is that they require significant rewriting to target the new DSL.

Embedded DSLs such as Halide (Ragan-Kelley et al., 2013) in C++, PolyMage (Mullapudi et al., 2015) in Python, and TiDA (Unat et al., 2016) in C++, make it more practical to rewrite codes written in the embedded language. Additionally, PolyMage and Halide were originally developed for image processing pipelines, and the PDE solvers we have been targeting have difficult issues such as complex boundary conditions that make fitting the DSL programming model difficult.

TiDA (Unat et al., 2016) is used to implement PDEs. In TiDA, scientific computations such as the ones we target are written in terms of computations over tiles of data. This enables the compiler and runtime system to handle the data layout, communication overlap, and tuning of tiling sizes. This approach does require some rewriting, but modularises the codes in a similar way to what many PDE solvers already use: iteration over blocks of data. Their focus is on the compile time and runtime scheduling of those blocks while our focus is on the compile-time optimisation across those blocks as shown in Olschanowsky et al. (2014).

In Zhou et al. (2012), compiler strategies for fusing and hierarchically tiling across loops are presented. The hierarchical tiling across loops transformation is one we plan to incorporate into the loop chain schedule pragma. The loop chain abstraction provides just the information needed to apply this transformation in the compiler.

6.2 Programmer-guided code transformation

The idea of providing optimisation hints to the compiler through directives is not a new one. The Intel compiler (among others) offers a range of pragmas to aid in

optimising applications. For example, the following Fortran code uses the *loop count* directive.

```
!DIR\$ LOOP COUNT (10000)
do i =1,m
b(i) = a(i) +1
enddo
```

It is likely that with this information the compiler will schedule the code differently than it would without. The directives available through the Intel compiler that are most related to our work are the loop optimisation pragmas: nofusion, unroll, and nounroll. There is not, however, a pragma available that will simplify the data flow analysis necessary for loop fusion and shifting in complex scientific applications.

Other frameworks have been developed that allow the programmer to apply more complex optimisations. Frameworks such as Orio (Norris et al., 2007) involve annotating the source code with instructions for optimisations. POET (Yi, 2012), CHiLL (Hall et al., 2010), and URUK (Girbal et al., 2006) each provide a scripting language for optimisation. The optimisation scripts (or recipes) can be placed within the source code or associated with the source code from an external file.

6.3 Improving data locality by aggregating computation into tasks

Various approaches have been developed to navigate the trade-off between parallelism and locality. We leverage the concept that developing a static aggregation or tiling strategy followed by dynamic execution of a task graph results in improved data locality within each tile and concurrency, load balancing, and memory latency tolerance between tiles. The key difference between previous work and loop chaining is that the programmer's responsibilities are less while still having feasible program analysis requirements.

The problem with having the programmer aggregate computations into tasks is that the programmer has to make some decision about task granularity across loops, and that decision might not be portable. There are various ways to aggregate computations into tasks: using an OpenMP pragma and specifying the chunk size (Dagum and Menon, 1998), tiling the loop and having tile iterations be tasks (Baskaran et al., 2009), iteration space slicing (Pugh and Rosser, 1999; Beletska et al., 2011), and encapsulating tasks within functions that have parameters indicating the task granularity. The OmpSs work (Perez et al., 2008, Duran et al., 2011) has the programmer indicate tasks by placing pragmas on C function definitions with in/out information about parameters and whether a function should be considered higher priority. Many new programming models (Duran et al., 2008; Chan et al., 2008; Andrade et al., 2009; Huang et al., 2010; Chandramowlishwaran et al., 2010; Cicotti and Baden, 2011; Augonnet et al., 2011) provide a task graph abstraction and suggest that programmers rewrite existing code with sequences of parallel loops in the form of task graphs instead. Iteration space slicing techniques (Pugh and Rosser, 1999; Beletska et al., 2011) that find sets of iterations across loops by doing transitive closure with data dependence relation information help automate task aggregation but depend heavily on precise and interprocedural data dependence analysis.

Once a task graph has been created, there are various ways of optimising the performance of the task graph. In Vydyanathan et al. (2009), the authors provide algorithms for scheduling task graphs using a mix of task and data parallelism. Within each task have data parallelism. In Virouleau et al. (2016), the authors propose a new tag for OpenMP that allows the user to provide locality information through an affinity tag. Other work determines the data locality between threads when scheduling tasks and uses this to control thread affinity (Terboven et al., 2008; Song et al., 2009; Meng et al., 2010).

The loop chain abstraction can complement any and all of these approaches by providing the needed information for creating tasks to the compiler and then using the appropriate task graph-based system as a backend. An issue we do not address in the proposed work is generating distributed memory code. Some aggregation approaches do provide distributed memory implementations (Cicotti and Baden, 2011; Schlimbach et al., 2013).We are tackling the problem of providing effective shared memory parallelism for individual MPI processes.

7 Limitations

There are currently some conceptual and practical limitations of this work.

7.1 Parameterising transformations

A significant amount of effort has gone into parameterising code generation for transformations such as tiling (Renganarayana et al., 2012; Hartono et al., 2009; Renganarayanan et al., 2007). The foundation has been set to be able to move forward with this work, however, it is not supported in our current tool chain. A specific example of this is tiling. It is common to sweep through a set of tile sizes to determine the best performing configuration. However, with our current configuration each tile size needs to be determined at compilation time. It is preferable to change this at runtime.

7.2 Wavefront transformation and automatic skew determination

A method for determining the skew factor required to legally parallelise loops after fusion is described in Section 2.4. However, this is not currently implemented in our transformation framework.

8 Conclusions

There exist programming models, languages, and abstractions that can expose and exploit parallelism in applications. However, exploiting maximum parallelism without respecting data locality results in poor performance through excessive memory traffic. We have presented:

1 a novel programming abstraction though OpenMP style pragmas

2 a software framework to describe and transform loop chains.

These tools can provide developers of new applications, and maintainers of legacy applications, with the ability to identify and transform loop chains in order to increase arithmetic intensity by simultaneously increasing both parallelism *and* data locality.

Further, we have created a prototype code transformation pass and used it to demonstrate the potential of these tools to effectively transform simple benchmarks. Our performance results are encouraging. We believe that this programming abstraction can work reasonably well for developers looking to increase the performance of their application without requiring them to overhaul their codes.

Acknowledgements

This work was supported by a National Science Foundation grant NSF CCF-1422725.

References

Adams, M., Colella, P., Graves, D.T., Johnson, J.N., Johansen, H.S., Keen, N.D., Ligocki, T.J., Martin, D.F., McCorquodale, P.W., Modiano, D., Schwartz, P.O., Sternberg, T.D. and Straalen, B.V. (2014) *Chombo Software Package for AMR Applications – Design Document*, Technical Report LBNL-6616E, Lawrence Berkeley National Laboratory.

Andrade, D., Fraguela, B.B., Brodman, J. and Padua, D. (2009) 'Task-parallel versus data parallel library-based programming in multicore systems', in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, IEEE Computer Society, Washington, DC, USA, pp.101–110.

Augonnet, C., Thibault, S., Namyst, R. and Wacrenier, P-A. (2011) 'StarPU: a unified platform for task scheduling on heterogeneous multicore architectures', *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, Vol. 23, pp.187–198.

Baskaran, M.M., Vydyanathan, N., Bondhugula, U.K.R., Ramanujam, J., Rountev, A. and Sadayappan, P. (2009) 'Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors', *PPOPP*, Vol. 44, No. 4, pp.219–228.

Beletska, A., Bielecki, W., Cohen, A., Palkowski, M. and Siedlecki, K. (2011) 'Coarsegrained loop parallelization: Iteration space slicing vs. affine transformations', *Parallel Computing*, Vol. 37, No. 8, pp.479–497.

Chan, E., Van Zee, F.G., Bientinesi, P., Quintana-Orti, E.S., Quintana-Orti, G. and van de Geijn, R. (2008) 'Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks', in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), PPoPP '08*, ACM, New York, NY, USA, pp.123–132.

Chandramowlishwaran, A., Knobe, K. and Vuduc, R.W. (2010) 'Performance evaluation of concurrent collections on high-performance multicore computing systems', in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

Christen, M., Schenk, O. and Burkhart, H. (2011) 'Patus: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures', in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

Cicotti, P. and Baden, S. (2011) 'Latency hiding and performance tuning with graph-based execution', in *2011 First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*, pp.28–37.

Dagum, L. and Menon, R. (1998) 'Openmp: an industry-standard API for shared-memory programming', *IEEE Computational Science & Engineering*, Vol. 5, No. 1, pp.46–55.

Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X. and Planas, J. (2011) 'Ompss: a proposal for programming heterogeneous multi-core architectures', *Parallel Processing Letters*, Vol. 21, No. 2, pp.173–193.

Duran, A., Perez, J.M., Ayguadé, E., Badia, R.M. and Labarta, J. (2008) 'Extending the OpenMP tasking model to allow dependent tasks', in *Proceedings of the 4th International Conference on OpenMP in a Newera of Parallelism, IWOMP'08*, Springer-Verlag, Berlin, Heidelberg, pp.111–122.

Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M. and Temam, O. (2006) 'Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies', *International Journal of Parallel Programming*, Vol. 34, No. 3, pp.261–317.

Grosser, T., Cohen, A., Kelly, P.H., Ramanujam, J., Sadayappan, P. and Verdoolaege, S. (2013) 'Split tiling for GPUS: automatic parallelization using trapezoidal tiles', in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ACM, pp.24–31.

Hall, M., Chame, J., Chen, C., Shin, J., Rudy, G. and Khan, M.M. (2010) 'Loop transformation recipes for code generation and auto-tuning', in *Languages and Compilers for Parallel Computing*, Vol. 5898, pp.50–64, Springer, Berlin, Heidelberg.

Hartono, A., Baskaran, M.M., Bastoul, C., Cohen, A., Krishnamoorth, S., Norris, B., Ramanujam, J. and Sadayappan, P. (2009) 'PrimeTile: a parametric multi-level tiler for imperfect loop nests', in *Proceedings of the 23rd International Conference on Supercomputing*, 8–12 June, IBM T.J. Watson Research Center, Yorktown Heights, NY, USA.

Henretty, T., Veras, R., Franchetti, F., Pouchet, L-N., Ramanujam, J. and Sadayappan, P. (2013) 'A stencil compiler for short-vector simd architectures', in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13*, ACM, New York, NY, USA, pp.13–24.

Huang, M., Narayana, V.K., Simmler, H., Serres, O. and El-Ghazawi, T. (2010) 'Reconfiguration and communication-aware task scheduling for high-performance reconfigurable computing', *ACM Trans. Reconfigurable Technol. Syst.*, Vol. 3, No. 4, pp.20:1–20:25.

Kennedy, K. and McKinley, K.S. (1992) 'Optimizing for parallelism and data locality', in *Proceedings of the 6th International Conference on Supercomputing, ICS '92*, ACM, New York, NY, USA, pp.323–334.

Krieger, C.D., Strout, M.M., Olschanowsky, C., Stone, A., Guzik, S., Gao, X., Bertolli, C., Kelly, P.H., Mudalige, G., Straalen, B.V. and Williams, S. (2013) 'Loop chaining: a programming abstraction for balancing locality and parallelism', in *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*.

Meng, J., Sheaffer, J. and Skadron, K. (2010) 'Exploiting inter-thread temporal locality for chip multithreading', in *IPDPS*.

Mullapudi, R.T., Vasista, V. and Bondhugula, U. (2015) 'Polymage: automatic optimization for image processing pipelines', in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, ACM, New York, NY, USA, pp.429–443.

Norris, B., Hartono, A. and Gropp, W. (2007) 'Annotations for productivity and performance portability', in *Petascale Computing: Algorithms and Applications, Computational Science*, pp.443–462, Chapman & Hall/CRC Press, Taylor and Francis Group, Preprint ANL/MCS-P1392-0107 [online] http://www.mcs.anl.gov/uploads/cels/papers/P1392.pdf (accessed 10 July 2017).

Olschanowsky, C., Strout, M.M., Guzik, S., Loffeld, J. and Hittinger, J. (2014) 'A study on balancing parallelism, data locality, and recomputation in existing PDE solvers', in *The IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Perez, J., Badia, R. and Labarta, J. (2008) 'A dependency-aware task-based programming environment for multi-core architectures', in *Proceedings of the IEEE International Conference on Cluster Computing*.

Pugh, W. and Rosser, E. (1999) 'Iteration space slicing for locality', in *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, LNCS*, Springer-Verlag, London, UK, Vol. 1863, pp.164–184.

Quinlan, D. and Liao, C. (2011) 'The rose source-to-source compiler infrastructure', in *Cetus Users and Compiler Infrastructure Workshop, in Conjunction with PACT 2011*.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F. and Amarasinghe, S. (2013) 'Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines', in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, ACM, New York, NY, USA, pp.519–530.

Renganarayana, L., Kim, D., Strout, M.M. and Rajopadhye, S. (2012) 'Parameterized loop tiling', *Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 34, No. 1, pp.3:1–3:41.

Renganarayanan, L., Kim, D., Rajopadhye, S. and Strout, M.M. (2007) 'Parameterized tiled loops for free', in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Schlimbach, F., Brodman, J. and Knobe, K. (2013) 'Concurrent collections on distributed memory theory put into practice', in *2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp.225–232.

Song, F., Moore, S. and Dongarra, J. (2009) 'Analytical modeling and optimization for affinity based thread scheduling on multicore systems', in *IEEE International Conference on Cluster Computing and Workshops, 2009, CLUSTER '09*, pp.1–10.

Stone, A. and Strout, M.M. (2013) 'Programming abstractions to separate concerns in semiregular grids', in *Proceedings of the 27th International Conference on Supercomputing (ICS)*.

Strout, M., Olschanowsky, C., Bertolacci, I., Willer, S., Gurses, E. and Patil, R. (2017) *Variations on a Theme* [online] https://github.com/CompOpt4Apps/VariationsOnATheme (accessed 10 July 2017).

Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C-K. and Leiserson, C.E. (2011) 'The pochoir stencil compiler', in *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '11*, ACM, New York, NY, USA, pp.117–128.

Terboven, C., Mey, D., Schmidl, D., Jin, H. and Reichstein, T. (2008) 'Data and thread affinity in OpenMP programs', in *Proceedings of the 2008 Workshop on Memory Access on Future Processors (MAW)*, ACM, New York, NY, USA, pp.377–384.

Unat, D., Nguyen, T., Zhang, W., Farooqi, M.N., Bastem, B., Michelogiannakis, G., Almgren, A. and Shalf, J. (2016) 'Tida: high-level programming abstractions for data locality management', in *ISC High Performance*.

Verdoolaege, S. (2010) 'ISL: an integer set library for the polyhedral model', *Mathematical Software – ICMS 2010*, pp.299–302 [online] http://link.springer.com/chapter/10.1007/978-3-642-15582-6_49 (accessed 10 July 2017).

Verdoolaege, S. (2016) *Integer Set Library* [online] http://isl.gforge.inria.fr/ (accessed 10 July 2017).

Virouleau, P., Roussel, A., Broquedis, F., Gautier, T., Rastello, F. and Gratien, J-M. (2016) 'Description, implementation and evaluation of an affinity clause for task directives', in *'IWOMP 2016', IWOMP 2016 – LLCS*, Nara, Japan, Vol. 9903 [online] https://hal.inria.fr/hal-01343442 (accessed 10 July 2017).

Vydyanathan, N., Krishnamoorthy, S., Sabin, G.M., Catalyurek, U.V., Kurc, T., Sadayappan, P. and Saltz, J.H. (2009) 'An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 20, pp.1158–1172.

Yi, Q. (2012) 'Poet: a scripting language for applying parameterized source-to-source program transformations', *Software: Practice and Experience*, Vol. 42, No. 6, pp.675–706 [online] http://dx.doi.org/10.1002/spe.1089 (accessed 10 July 2017).

Zhou, X., Giacalone, J-P., Garzarán, M.J., Kuhn, R.H., Ni, Y. and Padua, D. (2012) 'Hierarchical overlapped tiling', in *Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12*, ACM, New York, NY, USA, pp.207–218.