Compiler/Runtime Framework for Dynamic Dataflow Parallelization of Tiled Programs

MARTIN KONG, The Ohio State University
ANTONIU POP, The University of Manchester
LOUIS-NOËL POUCHET, The Ohio State University
R. GOVINDARAJAN, Indian Institute of Science
ALBERT COHEN, INRIA
P. SADAYAPPAN, The Ohio State University

Task-parallel languages are increasingly popular. Many of them provide expressive mechanisms for intertask synchronization. For example, OpenMP 4.0 will integrate data-driven execution semantics derived from the StarSs research language. Compared to the more restrictive data-parallel and fork-join concurrency models, the advanced features being introduced into task-parallel models in turn enable improved scalability through load balancing, memory latency hiding, mitigation of the pressure on memory bandwidth, and, as a side effect, reduced power consumption.

In this article, we develop a systematic approach to compile loop nests into concurrent, dynamically constructed graphs of dependent tasks. We propose a simple and effective heuristic that selects the most profitable parallelization idiom for every dependence type and communication pattern. This heuristic enables the extraction of interband parallelism (cross-barrier parallelism) in a number of numerical computations that range from linear algebra to structured grids and image processing. The proposed static analysis and code generation alleviates the burden of a full-blown dependence resolver to track the readiness of tasks at runtime. We evaluate our approach and algorithms in the PPCG compiler, targeting OpenStream, a representative dataflow task-parallel language with explicit intertask dependences and a lightweight runtime. Experimental results demonstrate the effectiveness of the approach.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors

General Terms: Languages, Performance, Compilers, Task Parallelism

Additional Key Words and Phrases: Dataflow, point-to-point synchronization, auto-parallelization, polyhedral framework, polyhedral compiler, tiling, dynamic wavefront, dependence partitioning, tile dependences

ACM Reference Format:
Martin Kong, Antoniu Pop, Louis-Noël Pouchet, R. Govindarajan, Albert Cohen, and P. Sadayappan. 2014. Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. ACM Trans. Architec. Code Optim. 11, 4, Article 61 (December 2014), 30 pages.
DOI: http://dx.doi.org/10.1145/2687652

This work was supported in part by the European FP7 project CARP id. 287767, the French "Investments for the Future" grant ManycoreLabs, the U.S. National Science Foundation award CCF-1321147 and Intel's University Research Office Intel Strategic Research Alliance program.
Authors' addresses: M. Kong (corresponding author), L.-N. Pouchet, and P. Sadayappan, Computer Science and Engineering Department, The Ohio State University; email: kongm@cse.ohio-state.edu; A. Pop, School of Computer Science, The University of Manchester; R. Govindarajan, Department of Computer Science and Automation, Indian Institute of Science; A. Cohen, INRIA.

DOI: http://dx.doi.org/10.1145/2687652

1. INTRODUCTION AND MOTIVATION

Loop tiling and thread-level parallelization are two critical optimizations to exploit multiprocessor architectures with deep memory hierarchies [Allen and Kennedy 2002]. When exposing coarse-grain parallelism, the programmer and/or parallelization tool may rely on data parallelism and barrier synchronization, such as the `for` worksharing directive of OpenMP, but this strategy has two significant drawbacks:

—Barriers involve a global consensus, a costly operation on nonuniform memory architectures; it is more expensive than the resolution of point-to-point dependences unless the task-dependence graph has a high degree (e.g., high fan-in reductions) [Bosilca et al. 2012].
—To implement wavefront parallelism in polyhedral frameworks, one generally resorts to a form of loop skewing, exposing data parallelism in a wavefront-based parallel schedule.

An alternative is to rely on more general, task-parallel patterns. This requires additional effort, both offline and at runtime. Dedicated control flow must spawn coarse-grain tasks, that must in turn be coordinated using the target language's constructs for the enforcement of point-to-point dependences between them. A runtime execution environment resolves these dependences and schedules the ready tasks to worker threads [Planas et al. 2009; Budimlic et al. 2010; Bosilca et al. 2012; Pop and Cohen 2013]. In particular, runtimes that follow the dataflow model of execution and point-to-point synchronization do not involve any of the drawbacks of barrier-based parallelization patterns: tasks can execute as soon as the data becomes available (i.e., when dependences are satisfied) and lightweight scheduling heuristics exist to improve the locality of this data in higher levels of the memory hierarchy; no global consensus is required and relaxed memory consistency can be leveraged to avoid spurious communications; loop skewing is not always required, and wavefronts can be built dynamically without the need of an outer serial loop.

Loop transformations for the automatic extraction of data parallelism have flourished. Unfortunately, the landscape is much less explored in the area of task-parallelism extraction, in particular, the mapping of tiled iteration domains to dependent tasks. This article makes three key contributions:

—*Algorithmic*. We design a task-parallelization scheme following a simple but effective heuristic to select the most profitable synchronization idiom to use. This scheme exposes concurrency and makes temporal reuse across distinct loop nests (a.k.a. dynamic fusion) possible, and further partitions the iteration domain according to the input/output signatures of dependences. Thanks to this compile-time classification, much of the runtime effort to identify dependent tasks is eliminated, allowing for a very lightweight and scalable task-parallel runtime.
—*Compiler construction*. We implement this algorithm in a state-of-the-art framework for affine scheduling and polyhedral code generation, targeting the OpenStream research language [Pop and Cohen 2013]. Unlike the majority of the task-parallel languages, OpenStream captures point-to-point dependences between tasks explicitly, reducing the work delegated to the runtime by making it independent of the number of waiting tasks.
—*Experimental*. We demonstrate strong performance benefits of task-level automatic parallelization over state-of-the-art data-parallelizing compilers. These benefits derive from the elimination of synchronization barriers and from a better exploitation of temporal locality across tiles. We further characterize these benefits within and across tiled loop nests.

```
for (i = 1; i < N - 1; i++)
  for (j = 1; j < N - 1; j++)
S1: B[i][j] = (A[i][j] + A[i][j-1] + A[i][1+j]+
    A[1+i][j] + A[i-1][j] + A[i-1][j-1] +
    A[i-1][j+1] + A[i+1][j-1] + A[i+1][j+1])/8;

for (i = 1; i < N-2; i++)
  for (j = 2; j < N-1; j++)
S2: A[i][j] = abs(B[i][j]-B[i+1][j-1]) +
    abs(B[i+1][j] - B[i][j-1]);
```

Fig. 1. Blur-Roberts kernel.

Proc cores	ref ICC	pluto min fuse	pluto smart fuse	our work
opt-1	1.25	0.4	0.7	0.9
opt-8	1.25	2.7	3.9	4.7
opt-16	1.25	2.0	0.7	6.8
xeon-1	8.35	2.13	1.83	2.81
xeon-4	8.35	6.22	4.99	9.45
xeon-8	8.35	7.15	6.12	16.55

Fig. 2. Blur-Roberts kernel performance in GFLOPS/sec for AMD Opteron 6274 and Intel Xeon E5-2650, on 1, half and all cores.

We illustrate these concepts in a motivating example in Section 2 and introduce background material in Section 3. We present our technique in detail in Section 4, and evaluate the combined compiler and runtime task parallelization to demonstrate the performance benefits over a data-parallel execution in Section 5. Related work is discussed in Section 6.

2. MOTIVATING EXAMPLE

We use the blur-Roberts kernel performing edge detection for noisy images in Figure 1 as an illustrating example, with N = 4000 and using double precision floating point arithmetic. Figure 2 shows the performance obtained when using Intel ICC 2013 as the compiler, using flags `-O3 -parallel -xhost`; PLuTo variants compiled with -O3 -openmp -xhost; task variant corresponds to *task-opt* (See Section 5.3). The original code peaks at 3.9 GFLOPS/sec on an AMD Opteron 6274 (although with half the cores) and 8.35 GFLOPS/sec on an Intel Xeon E5-2650. (See Table II in Section 5 for complete description of machines used). This program is memory bound but contains significant data reuse potential, thus tiling is useful to improve performance. We leverage the power of the PLuTo compiler [Bondhugula et al. 2008] to tile simultaneously for coarse-grained parallelism and locality, considering the two fusion heuristics *minfuse* (cut nonstrongly connected components at each level) and *smartfuse* (fuse only loops of identical depth). For this example, the result of the third fusion heuristic of PLuTo, *maxfuse* (fuse as much as possible) gives an identical output code than the *smartfuse* heuristic.

The *minfuse* heuristic distributes all loops that do not live in the same strongly connected component (SCC). Thus, it tiles and parallelizes each loop nest independently, requiring 2 barriers, as shown in Figure 3.

Smartfuse performs a complex sequence of transformations such as multidimensional retiming and peeling to enable the fusion of program statements under a common loop nest. Skewing is used for the wavefront transformation to expose coarse-grain parallelism. The output exhibits an outermost sequential loop and a second outermost parallel loop followed by its implicit barrier (see sketch in Figure 4). The complete code of PLuTo variants can be found in Kong et al. [2014].

As observed in Figure 2, the performance does not increase linearly with the number of processors; instead, it either reaches a plateau with the Intel Xeon or drastically drops, as in the Opteron's case.

Figure 5 illustrates the nature of the data dependences in the blur-Roberts code and the parallelization options for static affine scheduling, as represented by state-of-the-art polyhedral compilers such as PLuTo, versus dynamic task dataflow. The left half of the figure shows the iteration spaces for an unfused form of the code, with the left square representing the first loop nest and the right square the second loop

```
parfor (ti=0; ti < ubti; ti++)
  for (tj=0; tj < ubtj; tj++)
   { /* tile body of S1*/ }
/* barrier */
parfor (ti=0; ti < ubti; ti++)
  for (tj=0; tj <= ubtj; tj++)
   { /* tile body of S2*/ }
/* barrier */
```

Fig. 3. Blur-Roberts kernel produced with PLuTo minfuse.

```
for (w = 0; w < wmax; w++) {
  parfor (ti=f1(w); ti < f2(w); ti++)
  { /* tile body */ }
  /* barrier */
}
```

Fig. 4. Blur-Roberts kernel produced with PLuTo smartfuse.

nest, along with 2D tiling of each loop nest, working on blocks of rows of the matrices. With PLuTo minfuse, a barrier is used between executions of tiles of the first and second loops. With smartfuse, the two loop nests are fused, but skewing is employed to expose coarse-grain parallelism using the parfor/barrier idiom supported in PLuTo. After fusion, only wavefront parallelism is feasible with 2D tiling, with barriers between diagonal wavefronts in the tiled iteration space. Thus, there is a trade-off: with minfuse, the tiled execution of each loop nest is load balanced, but interloop data reuse is not immediately feasible; with smartfuse, interloop data reuse is improved, but may lead to load imbalance due to the exploitation of wavefront parallelism. We remark that other tiling techniques such as split-tiling [Henretty et al. 2013] or diamond-tiling [Bandishti et al. 2012] can enable better load balance than wavefront-based rectangular tiling. It, however, still demands explicit barriers in the code between tile "phases." In this work, we focus exclusively on rectangular affine tiling.

With a task dataflow model, it is possible to get the best of both worlds: increased degree of task-level parallelism as well as interloop data reuse. This occurs by creating 1D parallel tasks for each loop, and point-to-point task-level synchronizations enable ready tasks from the second loop nest to be executed as soon as their input tasks (the ones corresponding to the same block row and the ones on either side) have completed. Thus, a "dynamic fusion" between the loop nests is achieved automatically, without the problem of load imbalance from the wavefront parallelism with a static affine tile schedule for the fused loop nests.

This problem has been recognized in the linear algebra community and specialized solutions have been designed [Bosilca et al. 2012]. We propose a solution for affine programs by leveraging properties of the schedule of tiles as computed by polyhedral compilers, and utilize it to determine at compile-time the interband (among disjoint loop nests) and intraband (within a single loop nest) dependences. Our approach produces a fine interleaving of instances of the first and second band (outer iterations of the tiled loop nests). Unlike classical approaches in automatic parallelization, these dependences will then be instantiated at runtime, and fed to a dynamic task scheduler. The three last columns in the table of Figure 2 show the performance obtained in GFLOPS/sec when using two of PLuTo's heuristics (see Figures 3 and 4) and comparing these to our generated task-parallel code. As one can see, by using point-to-point synchronization and statically mapping tile-level dependences, it is possible to greatly improve over state-of-the-art vendor and research compilers: on Intel's Xeon, we achieve nearly 2× latency improvement with regard to ICC and PLuTo's best; whereas on AMD's Opteron we obtain over 5× relative to the baseline and 1.5× over PLuTo.

Our technique can be summarized as follows. We first compute tile-level constructs that are the input to an algorithm that selects stream idioms to be used for each tile dependence or to a partition routine that splits the loop nests into classes that share identical input/output dependence patterns. This algorithm chooses when to

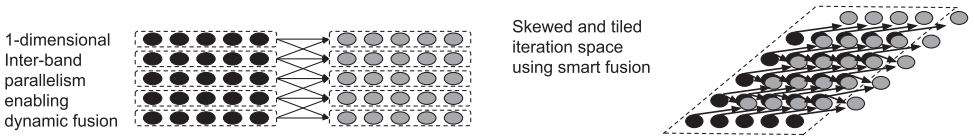

Fig. 5. Illustration of benefit of task dataflow over static affine scheduling.

extract parallelism across disjoint loops, while the partition routine allows creation of a dynamic wavefront of tiles. Then a static task graph is constructed to prune redundant dependences and to decorate it with dependence information. Finally, code is generated for the OpenStream runtime.

3. BACKGROUND

The principal motivation for research in dataflow parallelism comes from the inability of the Von Neumann architecture to exploit large amounts of parallelism, and to do so efficiently in terms of hardware complexity and power consumption. In dataflow languages and architectures, the execution is explicitly driven by data dependences rather than control flow [Johnston et al. 2004]. Dataflow languages offer functional and parallel composition preserving (functional) determinism. So-called *threaded* or *task-parallel* dataflow models operate on atomic sequences of instructions, or tasks, whereas early dataflow architectures leveraged data-driven execution at the granularity of a single instruction.

3.1. OpenStream

We selected the task-parallel dataflow language OpenStream [Pop and Cohen 2013], a representative of the family of low-level (C language) task-parallel programming models with point-to-point intertask dependences [Planas et al. 2009; Budimlic̀ et al. 2010; Cavé et al. 2011]. OpenStream stands out for its ability to materialize intertask dependences explicitly using streams, whereas the majority of the task-parallel languages rely on some form of implicit dependence representation (e.g., induced from memory regions in StarSs or OpenMP 4 [Planas et al. 2009]). The choice of an explicit dependence representation reduces the overhead of dynamically resolving dependences. For a detailed presentation, refer to Pop and Cohen [2013], and to the formal model underlying the operational semantics of OpenStream [Pop and Cohen 2012].[1]

Like the majority of task-parallel languages, an OpenStream program is a dynamically built task graph. Unlike the majority of streaming languages, OpenStream task graphs do not need to be regular or static. They can be composed of a dynamically evolving set of tasks communicating through dataflow streams. The latter are strongly typed, first-class values: they can be freely combined with recursive computations and stored in dynamic data structures. Programming abstractions and patterns allow construction of complex, fully dynamic, possibly nested task graphs with unbounded fan-in and fan-out communications. OpenStream also provides syntactic support for common operations such as broadcasts.

Programmer annotations are used to specify regions, within the control flow of a sequential program, that may be spawned as concurrent coroutines and delivered to a runtime execution environment. These control flow regions inherit the OpenMP task syntax. Without `input/output` stream annotations, OpenStream tasks have the same semantics as OpenMP tasks. For example, Figure 6 shows the OpenStream version of dgemm kernel. Each instance of the outer loop outputs a token to band_stream_ii. The

[1]The source code repository and web site for OpenStream can be found at http://www.openstream.info.

```
long band_stream_ii_size = (floor((15 + ni)/16));
int band_stream_ii[band_stream_ii_size] __attribute__((stream));
int read_window[W]; int write_window[W];
for (int ii = 0; ii <= floord(ni - 1, 16); ii += 1)
#pragma omp task  output(band_stream_ii[ii] << write_window[W])
 for (int jj = 0; jj <= floord(nj - 1, 16); jj += 1)
  for (int i = 16 * ii; i <= min(ni - 1, 16 * ii + 15); i += 1)
   for (int j = 16 * jj; j <= min(nj - 1, 16 * jj + 15); j += 1)
    C[i][j] *= beta;
```

```
for (int ii = 0; ii <= floord(ni - 1, 16); ii += 1)
#pragma omp task  input(band_stream_ii[ii] >> read_window[W])
 for (int jj = 0; jj <= floord(nj - 1, 16); jj += 1)
  for (int kk = 0; kk <= floord(nk - 1, 16); kk += 1)
   for (int i = 16 * ii; i <= min(ni - 1, 16 * ii + 15); i += 1)
    for (int j = 16 * jj; j <= min(nj - 1, 16 * jj + 15); j += 1)
     for (int k = 16 * kk; k <= min(nk - 1, 16 * kk + 15); k += 1)
      C[i][j] += ((alpha * A[i][k]) * B[k][j]);
```

Fig. 6. OpenStream example: gemm kernel tiled and parallelized with interband idiom.

second task, following the first loop nest, waits for tokens on band_stream_ii, and then proceeds. This pattern implements dataflow concurrency in OpenStream and avoids barrier synchronization.

While OpenStream is very expressive and can be used to express nondeterministic computations, the language comes with specific conditions under which the functional determinism of Kahn networks [Kahn 1974] is guaranteed by construction. These conditions enforce a precise interleaving of data in streams derived from the control flow of the program responsible for spawning tasks dynamically, hereafter called the *control program*. In the following, we will assume the control program is sequential, which is a sufficient condition to enforce determinism.

OpenStream Task Idioms. In this work, we leverage multiple programming patterns present in modern task-parallel languages.

(1) Input and output clauses: they extend the OpenMP task syntax and allow explicit specification of the point-to-point synchronizations between tasks via streams. By default, all streams are operated through a *window* of stream elements accessible to a given task. The horizon of the window is the number of stream elements accessible through it. The burst size of a window corresponds to the number of elements read/written from/to input/output streams at a given task activation. The default burst size is 1. The input clause can be specialized further into the two following clauses.
(2) Peek operation: Similar to the input clause, but it does not advance the stream's window, that is, the window's burst size is zero. It enables the reuse of stream elements across multiple task activations, which is particularly useful when implementing broadcast operations.
(3) Tick operation: A collection of peek operations may be followed by a tick operation, to advance the window to new stream elements.
(4) Soft-barrier, point-to-point barrier or data-flow barrier: it is an empty program statement that waits for a fixed or parametric number of elements from one or more streams. Once all its input dependences are satisfied, an element is written to each of its output streams. Then, the tasks waiting for these elements may read them with peek operations.

3.2. The Polyhedral Model

The polyhedral framework is a flexible and expressive representation for imperfectly nested loops with statically predictable control flow. Loop nests amenable to this algebraic representation are called *static control parts* (SCoP) [Feautrier 1992; Girbal et al. 2006], roughly defined as a set of consecutive statements such that all loop bounds and conditional expressions are affine functions of the enclosing loop iterators and variables that are constant during the SCoP execution (but whose values are unknown at compile-time). Numerous scientific computations exhibit those properties; they are found frequently in image processing filters (such as medical imaging algorithms), dense linear algebra operations, and stencil computations on regular grids.

Unlike the abstract syntax trees used as internal representation in traditional compilers, polyhedral compiler frameworks internally represent imperfectly nested loops and their data dependence information as a collection of parametric polyhedra. Programs in the polyhedral model are represented using four mathematical structures: each statement has an *iteration domain*, each memory reference is described by an affine access function, data dependences are represented using *dependence relations*/*polyhedra* and finally the program transformation to be applied is represented using a *scheduling function* or a *schedule map*. Since the operations and analyses performed in this work heavily rely on the map and set representation of the Integer Set Library (ISL) [Verdoolaege 2010], we briefly describe and give examples of these structures using the set and map notations. In addition, we also recapture two concepts that will be used in Section 4, *Bands of Tiles* and the *Polyhedral Reduced Dependence Graph*.

Iteration Domains. They represent the set of dynamic (runtime) executions of a syntactic statement, typically enclosed in a loop nest. Each program statement is associated with an iteration domain defined as a set of integer tuples, one for each dynamic execution of the statement. These tuples capture the value of the surrounding loop iterators when the associated statement instance is executed at runtime. For the two statements `S1` and `S2` in blur-Roberts we have:

$$\{S1[i,j] \in \mathbb{Z}^2 : 1 \leq i,j \leq N-1;\ S2[i,j] \in \mathbb{Z}^2 : 1 \leq i \leq N-2 \wedge 2 \leq j \leq N-1\}.$$

Access Relations. For a given memory reference (e.g., `B[i][j]`) access relations map statement instances with the set of memory locations of the array (e.g., `B`), which are accessed during the statement execution. Here are a few examples from the blur-Roberts kernel:

$$\begin{aligned}\{S1[i,j] \rightarrow B[i,j] : 1 \leq i,j \leq N-1;\ S1[i,j] \rightarrow A[i-1,j-1] : 1 \leq i,j \leq N-1;\\ S2[i,j] \rightarrow B[i+1,j-1] :\ 1 \leq i \leq N-2 \wedge 2 \leq j \leq N-1\}.\end{aligned}$$

Dependence Relations. Data dependences in ISL are represented as relations in between two iteration domains, possibly of the same statement for self dependences. The relation indicates the source and target of the dependence. Continuing with our example, two of the four dependences between $S1$ and $S2$ on array B are:

$$\begin{aligned}\{S1[i1,j1] \rightarrow S2[i2,j2] : 1 &\leq i1, j1 \leq N-1 \wedge 1 \leq i2 \leq N-2 \wedge 2 \leq j2\\ &\leq N-1 \wedge i1 = i2 \wedge j1 = j2;\\ S1[i1,j1] \rightarrow S2[i2+1,j2-1] : 1 &\leq i1, j1 \leq N-1 \wedge 1 \leq i2 \leq N-2 \wedge 2 \leq j2\\ &\leq N-1 \wedge i1 = i2+1 \wedge j1 = j2-1\}.\end{aligned}$$

Schedules. Reordering transformations are represented as relations from iteration domains to the logical time space. The relation assigns to each point in an iteration domain a multidimensional timestamp, which is later used to generate code. In the final code after transformation, each point in the iteration domains will be executed, according to the lexicographic ordering on the timestamps provided by the schedule function [Bastoul 2004]. For instance, the following relation permutes dimensions i and j of the first loop nest in the output code of our running example:

$$\{S1[i,j] \rightarrow [j,i] : 1 \leq i,j \leq N-1\}.$$

Bands of Tiles. When the computed schedule represents a tiling transformation, a band is a consecutive set of nonconstant dimensions (e.g., loop levels) in time space with the property that these dimensions are permutable. A band of tiles is a band that only considers some consecutive tile dimensions. Intuitively, one can think of these dimensions as the ones that will become the tile loops after code generation.

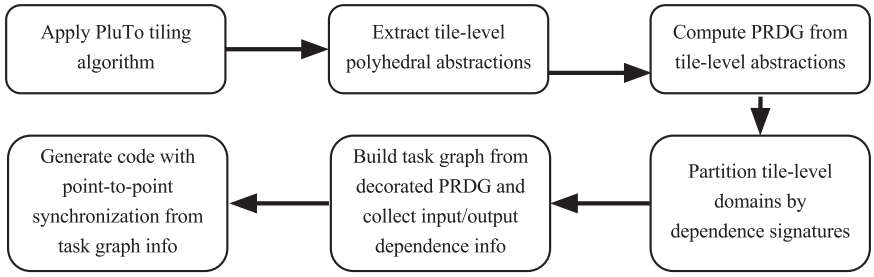

Fig. 7. Compiler stages from a sequential input code to a tile-level parallel version with point-to-point synchronization.

For instance, in Figure 3, only the tile loops are depicted. These correspond to the dimensions of the bands of tiles and would be generated by a schedule of the form:

$$\{S1[i,j] \rightarrow [0, ti, tj, i2, j2] : 1 \leq i, j \leq N-1 \ldots\};\ S2[i,j] \\ \rightarrow [1, ti, tj, i2, j2] : 1 \leq i, j \leq N-1 \ldots\}.$$

The leading constant value of 0 and 1 in the range of the relations for statements $S1$ and $S2$, respectively, indicate that it is a scalar dimension (e.g., not a loop) and that the generated code will consist of 2 bands of tiles, composed in both cases by the trailing tile loop dimensions ti and tj.

Polyhedral Reduced Dependence Graph (PRDG). It is a multigraph in which each node represents a program statement [Darte and Vivien 1997]. Edges in between nodes capture different dependences. Nodes are labeled with iteration domains and edges are labeled with dependence relations. In our case, a Tile-PRDG needs to be constructed from the statement-PRDG and the computed tile schedule, as shown to follow.

Fusion Heuristics: minfuse and smartfuse. The loop structure produced by a transformation can be viewed as a partition of the program under the criterion of appearing fused in one or more outer loops. Smartfuse is the default PluTo heuristic for tiling where statements that do not share any data reuse are placed on different partitions. Minfuse attempts to maximize the number of partitions by setting statements into different classes, unless they are required to appear fused under some loop level (or same strongly connected component) [Bondhugula et al. 2008].

Macro statements. After applying a tiling algorithm such as PLuTo to an input program, statements are naturally fused under a sequence of loops. All statements that live under a common set of loops form a *macro statement*. The set of macro statements of a tiled program depends on the tiling heuristic used (e.g., smartfuse or minfuse).

4. EXTRACTING TASK PARALLELISM

The high-level flow that we follow to extract tasks at the granularity of a tile is shown in Figure 7. Its input is a sequential code that is first tiled with PLuTo's algorithm. Then the tile-level polyhedral representation (see Section 3.2) is extracted from the original statement representation and from the tile schedule computed in the previous stage. Next, the PRDG is obtained from this representation. The following stage partitions the tile domains according to the dependence signatures of all neighboring nodes. This produces a new set of domains for which a new PRDG is computed and input/output dependence patterns are associated to each new node/domain. Finally, code is generated and loops are decorated with OpenStream's task syntax that describes the input and output dependence instances for each task/tile.

The first stage of our method is to build a polyhedral representation of the *tiled* program, with one (macro-)statement per tile, each having an iteration domain (i.e., capturing the set of dynamic executions of each tile). A *Tile-PRDG* is then produced, which captures data dependences between tiles. Next, we introduce an algorithm to select the most profitable task-parallel idiom, in terms of number of required synchronizations (incoming tile dependences). We present a processing step that allows extraction of task parallelism from kernels that would otherwise use a static wavefront to expose tile-level parallelism, as well as handle cases in which two disjoint loop nests have more than one dependence in common. The following step consists of building a static task graph. Finally, we briefly discuss general details pertaining to polyhedral code generation.

4.1. Tile-Level Processing

The input to this compilation flow is a C program for which a tile schedule can be computed. In this work, we assume the PLuTo algorithm [Bondhugula et al. 2008] has been used to enable tiling; however, we are not limited to this particular tiling algorithm. We call *tile level* the outermost or outermost and next outermost tile dimensions. Two tile-level abstractions must be constructed from the program statement domains, dependences and schedule: the tile domains and the tile dependences. These are determined by first projecting the intratile dimensions $k..n$ of the schedule onto the tile dimensions $1..k-1$ in the range of the schedule map M^S of statement S:

$$M^S_{tile} = PROJ_{k,\ldots,n}(range(M^S)).$$

This yields a tile map M^S_{tile} for each statement S, where k is the starting dimension to be projected (2 for interband parallelism and 3 for intraband) and n is the number of output dimensions of the map. The modified map is then used to compute the tile domain by simply intersecting the map's domain with the set representing the statement domain. The range of the resulting map is the tile domain.

Tile dependences are constructed in a similar way. Given a dependence map $M^{S\to T}$ describing a dependence from statement S to statement T, and the tile maps M^S_{tile} and M^T_{tile} computed, the tile dependence $M^{S\to T}_{tile}$ is $(M^S_{tile})^{-1} \circ M^{S\to T} \circ M^T_{tile}$, where $\circ$ is the composition of maps. At this stage, we remove existent nonforward dependences, that is, those that have the exact same source and target tile coordinates (same tile instance). This type of dependence could emerge after the projection step, in which case the dependence was forward in some of the projected out dimensions. From the task-parallel dataflow perspective, these dependences are part of the task body of each tile instance, and do not represent any real flow of data. We do so by computing identity maps of the tile domain (e.g., from a domain $T\_0[tt, ii]$ we produce the map $T\_0[tt, ii] \to T\_0[tt, ii]$) and subtracting them from the original dependence maps, prior intersection with the tile domain. Nonforward dependences do not arise when considering interband dependences (due to the leading scalar dimension in the schedule). However, they do appear in single bands (e.g., Seidel kernel), and must therefore be pruned.

In addition, all tile dependences that share the same source and the same target can be further simplified by: (1) combining pairs of basic maps within a single map by using the map coalescing ISL functions (this simplifies the representation); (2) if a new tile dependence is a subset of a previous dependence, we do not include it; (3) conversely, if a new tile dependence is a superset of previous tile dependences, we remove the previous ones and add only the new one; (4) if a tile dependence partially intersects with some other tile dependence, we take their union.

Finally, each tile dependence represented by a map (in ISL terminology) is massaged by detecting equalities, removing redundancies, and making them disjoint (basic maps that constitute a map are not guaranteed to be disjoint unless invoking an explicit ISL

```
for (t = 0; t < tsteps; t++){
  for (i = 1; i <= n-2; i++)
    for (j = 1; j <= n-2; j++)
S1: B[i][j] = (A[i][j] +
    A[i][j-1] + A[i][j+1] +
    A[i+1][j] + A[i-1][j])*
    0.2;

for (i = 1; i <= n-2; i++)
  for (j = 1; j <= n-2; j++)
S2: A[i][j] = B[i][j];
}
```

Fig. 8. Jacobi-2d kernel.

```
[tsteps, n] -> { T_0[0, 0, i2, i3] -> T_0[0, 0, o2, o3] : i2 >= 0 and n >= 3
and o3 <= 1 + i3 and o2 <= 1 + i2 and 2o2 <= i3 and 16o3 <= 30 + n + 32i2 and
16i3 <= -5 + 2tsteps + n and 16o3 <= -4 + 2tsteps + n and 16i3 <= 29 + n + 32i2
and 16o2 <= -1 + tsteps and 16i2 <= -2 + tsteps and o3 >= i3 and o2 >= i2 and
16o3 <= 12 + n + 16i3 and 32o2 >= -27 - n + 16i3 and 16o3 <= 28 + n + 32o2;
T_0[0, 0, i2, i3] -> T_0[0, 0, o2, o3] : 16o3 <= -4 + 2tsteps + n and 16i2 <=
-2 + tsteps and 16i3 <= 28 + n + 32i2 and 16o3 <= 30 + n + 32i2 and i2 >= 0 and
o3 >= 2o2 and o2 >= i2 and n >= 3 and 32o2 >= -26 - n + 16i3 and o3 >= i3 and
16o3 <= 28 + n + 32o2 and 16i3 <= -6 + 2tsteps + n and o3 <= 1 + i3 and o2 <= 1
+ i2 and 16o2 <= -1 + tsteps and i3 >= 2i2; T_0[0, 0, i2, i3] -> T_0[0, 0, i2,
o3] : 16o3 <= -3 + 2tsteps + n and 16i2  <= -1 + tsteps and o3 >= i3 and 16o3
<= 29 + n + 32i2 and i2 >= 0 and 16i3 <= -4 + 2tsteps + n and  i3 >= 2i2 and n
>= 3 and 16i3 <= 28 + n + 32i2 and o3 <= 1 + i3; T_0[0, 0, i2, i3] -> T_0[0, 0,
o2, o3] : 16o3 <= -3 + 2tsteps + n and 16i2 <= -2 + tsteps and 16i3 <= 29 + n +
32i2 and 16o3 <= 31 + n + 32i2 and i2 >= 0 and 2o2 <= i3 and 16o3 <= 29 + n +
32o2 and n >= 3 and o2 >= i2 and o3 >= i3 and 32o2 >= -27 - n + 16i3 and 16i3
<= -5 +  2tsteps + n and o3 <= 1 + i3 and o2 <= 1 + i2 and 16o2 <= -1 + tsteps
}
```

Fig. 9. Jacobi-2d tile dependences.

function) [Verdoolaege 2010]. An additional pruning stage is later performed during the task-graph construction to remove dependence polyhedra that are already captured through transitive dependence relations.

Here we introduce the kernel Jacobi-2d (Figure 8), which we use to explain the partitioning steps. In Figure 9, we show 4 tile dependences computed from 9 statement dependences. These tile dependences will be further processed in order to make them uniform and proceed with the partitioning stage.

4.2. Partitioning

Domain partitioning is required in two scenarios: (1) when extracting interband parallelism, two nodes in the PRDG can share more than one normalized dependence (e.g., in blur-Roberts kernel), thereby needing different treatment for each dependence; (2) when the task graph consists of a single band of tiles, that is, a single node in the PRDG. We note that when partitioning domains for the former case, we only target the outermost tile dimension; for the latter case, we work on the two outermost dimensions if dependences are uniform. Our approach for exposing task-level parallelism from a polyhedral program representation is to implement a partitioning scheme that groups tiles of a loop nest's iteration domain into classes; each class is associated with a single signature of incoming and outgoing, intertile (or intratile) dependences. This single signature in turn enables the generation of OpenStream directives and tasks with well-defined input/output clauses.

Our partitioning scheme takes as input the bands of permutable dimensions resulting from the PLuTo algorithm. However, any tiling algorithm that outputs a similar structure can also be used instead. We make the two following assumptions:

(1) Tile dependences of all statements within a permutable band must be subsumed by that of a single (macro-)statement,
(2) All statements must live under a common and nondisjoint tile schedule space.

The first assumption is guaranteed by design of the PLuTo algorithm, which effectively groups statements in tiled parts of the schedule, where all dependences may be collected as if the band were formed of a single, fused, macrostatement. The second one allows us to propagate the partition generated from a leading statement domain into the follower domains in the band. This is discussed next.

Definition 4.1 (*Leading Domain and Follower Domains*). The iteration domain I^S of a statement S that belongs to a tiled band is called the *leading domain* if

(i) The depth of this domain is maximal within the band (at the chosen granularity, i.e., of one dimension for interband and two dimensions for wavefront).

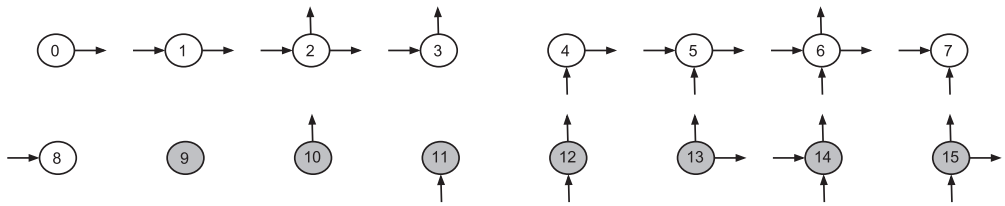

Fig. 10. Potential signatures generated by uniform dependences (gray denotes signatures that do not appear in the Jacobi-2d kernel).

(ii) The domain with maximal depth is unique within the band. This implies that all tile dependences within the band are subsumed by the leading domain.

Any domain that is not the leading domain is a *follower domain*.

When Definition 4.1 does not hold, we name any statement domain with maximal (tile) depth the leading domain, but consider its tile domain as the union of all tile domains within the band that have maximal depth. In our Jacobi-2d example (Figure 8) both statements have the same dimensionality. Therefore, we compute the tile domain of each individual statement, take their union, and name either of the statements as the leading domain. Definition 4.1, combined with our assumptions on the input schedule, guarantees that all dependences are subsumed by the tile domain of the leading domain. Partitioning can safely be applied to the *leading domain*, and then propagated onto the *follower domains*. In the case that the macro-statement (statements sharing a same band of tiles) has no self-dependence, then any statement can be chosen as the leading domain.

Each dependence relation is identified with a unique number $1..n$, for n dependence relations. The dependence signature lists the unique identifiers of dependence relations that are either to or from the domain.

Definition 4.2 (*Dependence Signature*). The dependence signature SIG^S of a domain I^S_{tile} is composed of two sets: the *IN* set and the *OUT* set. For each dependence relation k, k is put in *IN* (*OUT*, respectively) if $\mathcal{D}^{T\to S}$ ($\mathcal{D}^{S\to T}$, respectively) has at least one destination (source, respectively) in I^S_{tile}.

$$\text{SIG}^S = \{IN^S, OUT^S\}, IN^S = \left\{k : \text{Ran}\left(D^{k:T\to S}_{tile}\right) \cap I^S_{tile} \neq \varnothing\right\},$$
$$OUT^S = \left\{k : \text{Dom}\left(D^{k:S\to T}_{tile}\right) \cap I^S_{tile} \neq \varnothing\right\}.$$

It follows the definition of a disjoint partition of the iteration domain:

Definition 4.3 (*Domain Partition*). The partition P^S of a domain I^S is defined by the following conditions:

$$P^S = \left\{P^S_i\right\}, \quad I^S = \cup\left(P^S_i\right), \quad P^S_i \cap P^S_j = \varnothing \quad \wedge \quad \text{SIG}\left(P^S_i\right) \neq \text{SIG}\left(P^S_j\right), \quad i \neq j,$$

where an iteration domain I^S is divided into P^S_i disjoint domains such that they all have different dependence signatures.

Figure 10 shows all the possible signatures that could be extracted from a 2D tile domain having as input dependences that are parallel to the axes in the transformed space. The actual signatures that arise after partitioning are a function of the tile schedule as well as the parameters. In Figure 10, shaded circles represent signatures never applicable to jacob-2d's tiled code. Moreover, some partitions might not execute at runtime due to the actual parameter values (e.g., the partition associated to signature 7).

```
[tsteps,n]->{
  T_0[0,0,i2,i3]->T_0[0,0,i2+1,o3]: o3 >= i3 and 16i2 <= -17 +
  tsteps and 16i3 <= 29 + n + 32i2 and 16o3 <= 31 + n + 32i2 and i2 >= 0 and i3
  >= 2 + 2i2 and o3 <= 1 + i3;  T_0[0,0,i2,i3]->T_0[0,0,o2,i3+1]: 16o2 = -1 +
  tsteps and 16i2 = -1 + tsteps and 16i3 <= -19 + 2tsteps + n and 8i3 >= -1 +
  tsteps and tsteps >= 1; T_0[0,0,i2,i3]->T_0[0,0,i2,i3+1]: 16i3 <= -19 + 2tsteps
  + n and 16i2 <= -2 + tsteps and 16i3 <= 13 + n + 32i2 and n >= 3 and i2 >= 0
  and i3 >= 2i2; T_0[0,0,i2, 2i2+1]-> T_0[0,0,i2+1,2i2+2]: i2 >= 0 and 16i2 <=
  -17 + tsteps and n >= 3 }
```

Fig. 11. Jacobi-2d tile dependences prior to transitive reduction pruning.

```
[tsteps, n]->{ *T_0[0,0,i2,i3]->T_0[0,0,i2+1,i3]
 T_0[0,0,i2,i3]->T_0[0,0,i2+1,i3+1];
 *T_0[0,0,i2,i3]->T_0[0,0,i2,i3+1]
 T_0[0,0,i2,i3]->T_0[0,0,i2+1,i3+1]
 T_0[0,0,i2,i3]->T_0[0,0,i2,i3+1]
 T_0[0,0,i2,2i2+1]->T_0[0,0,i2+1,2i2+2] }
```

Fig. 12. Jacobi-2d tile (uniform) dependences. Dependences marked with * are the nonredundant ones and used for partitioning.

Making dependences uniform. Figure 11 shows the tile dependence map for the jacobi-2d kernel before making all dependences uniform. In order to apply our partitioning algorithm, we convert all the basic maps that constitute a tile dependence into their uniform shape. We do so by expanding the output dimensions that produce a range of values from the input dimensions, for example, the first basic map with constraints $o3 >= i3$ and $o3 <= 1 + i3$ is expanded into 2 basic maps. The result of this expansion is shown in Figure 12. This step allows further pruning of dependences by transitive reduction [Midkiff and Padua 1986, 1987] as well as removal of new covered dependences and duplicated dependences. At this point, all output sets are explicit functions of their input sets. This permits replacement of the tile dependence constraints by the constraints of the (leader) tile domain. The net effect of this is that dependences become broader, but also enables removal of redundant communication. After this process, only two nonredundant dependences are left, each parallel to one of the outermost axis on the transformed space. The only two required dependences are marked with an asterisk (*).

Algorithm 1 is used to perform domain partitioning (when required), based on Definitions 4.2 and 4.3, during the task-graph construction and after pruning transitively covered dependences. The domain of a band could be partitioned both according to the incoming and outgoing dependences; for example, domain S could be partitioned first when processing dependence $R \rightarrow S$ and again partitioned when processing dependence $S \rightarrow T$, and also due to multiple dependences between two nodes (since the PRDG is a multigraph).

Figure 15 shows the partitions generated by Algorithm 1, the respective dependence signatures, and the distribution of the signatures around the (tile) domain. It may be surprising to see signature 7 in this figure. However, this signature could be triggered depending on the parameter values, specifically for this condition:

$$(2 * ((tsteps - 1)\%16) + ((14 * tsteps + 15 * n + 2)\%16) <= 13 \&\& (tsteps - 1)\%16 <= 6).$$

Partial tiles. A band of tiles contains a partial tile when one or more of its intratile dimensions have a smaller cardinality than the selected tile size. Our algorithm does not distinguish partial tiles from full tiles (tiles that have their cardinality equal to the tile size), since we are only interested in the outer tile coordinates, and which are only dependent on the problem size and outer tile coordinates. Furthermore, partial tiles already have a different dependence pattern, and only appear at the end of some domain. As such, they are naturally placed into their own signature class. A similar

ALGORITHM 1: PartitionDomains (G, u, v)

Input: G: Tile-Level PRDG; u: source tile domain; v: target tile domain
Output: Updated G where nodes u and v have been decorated with their partitions

```
if parts(u) = ∅ then parts(u) ←I^u_tile;
if parts(v) = ∅ then parts(v) ←I^v_tile;
foreach tile dependence d from u to v do
    indom ←domain(d);
    outdom ←range(d);
    foreach p in parts(u) do
        s ←p ∩ indom;
        if s ≠ ∅ then
            diff ←p − s;
            SIG(diff) ←SIG(p);
            SIG(s) ←SIG(p) ∪{OUT^s = d};
            remove p from parts(u);
            insert s and diff in parts(u);
            indom ←indom − p;
        end
    end
    foreach p in parts(v) do
        s ←p ∩ outdom;
        if s ≠ ∅ then
            diff ←p − s;
            SIG(diff) ←SIG(p);
            SIG(s) ←SIG(p) ∪{IN^s = d};
            remove p from parts(v);
            insert s and diff in parts(v);
            outdom ←outdom − p;
        end
    end
end
```

handling applies to all tiles that appear in a domain border, that is, a first or last tile instance along some dimensions.

In the following two sections, we will discuss in more detail when partitioning is required.

4.3. Task-Graph Construction

The task-graph construction takes as input the decorated PRDG. We contemplate a single-dimensional tile band PRDG in order to limit the number of potential synchronizations, which increases as one considers more tile dimensions by factors that depend on the problem sizes as well as the chosen tile size. This still allows a sufficiently fine interleaving between the tile instances. Thus, this stage is only performed if we have multiple loop nests after tiling. The goal of this step is to traverse the PRDG and consider the minimum number of edges in the task graph so that all dependences are satisfied as well as being able to remove redundant synchronizations that can arise from transitively covered dependences, which often appear in PRDGs composed of several bands. Algorithm 2 shows the pruning steps performed for the dependences (edges of the PRDG) on each array (scalars are 0-dimensional arrays) and the addition of the relevant edges from the PRDG into the task graph. Transitively covered dependences are found by composing the dependence relations in between PRDG nodes. For example, given nodes R, S, and T, and dependences $R \rightarrow S$, $S \rightarrow T$ and $R \rightarrow T$, then dependence $R \rightarrow T$ can be removed if it is equal to the composition $R \rightarrow S \circ S \rightarrow T$.

ALGORITHM 2: ConstructTaskGraph (PRDG)

Input: PRDG: Tile-Level PRDG
Output: T: Task graph with signatures and partitioned domains
foreach *written array A* **do**
  Collect dependences on array A;
  Traverse the PRDG and remove transitively covered dependences on array A;
end
foreach *edge* $u \rightarrow v$ **do**
  if *u or v have more than 1 dependence* **then**
    PartitionDomain(PRDG,u,v);
  end
end
$T \leftarrow \varnothing$;
foreach *edge* $u \rightarrow v \in PRDG$ **do**
  $\pi_u \leftarrow$ Get Partitioned Domains of u;
  $\pi_v \leftarrow$ Get Partitioned Domains of v;
  Add a node to T for each domain in π_u and π_v;
  Add edges for dependences between π_u and π_v to T;
end
return T

These dependences can be of type RAW, WAR and WAW. Then, Algorithm 1 is invoked for nodes u and v of the PRDG when they have more than one dependence relation connecting them, that is, 2 or more dependences that differ in either source or domain, or both (such as in Blur-Roberts). Finally, the underlying task graph is constructed from the partitioned domains attached to each node of the PRDG.

4.4. Implementing Interband Parallelism

We now use the 3mm benchmark from Polybench/C to illustrate the various implementations of interband parallelism that can be achieved. Our algorithm automatically prioritizes the possible implementations based on data dependence features, as shown later.

In the following, the term *band* can be intuitively thought as a single loop nest. We need to decide whether two nodes connected by an edge in the task graph can be effectively parallelized with interband (across disjoint loop nests) task parallelism, if a soft barrier should be used to satisfy the dependences of the target node, or if a band should be considered as a single task. *We emphasize that this algorithm is designed for a single level of intraband parallelism. Thus, it focuses exclusively on 1-dimensional tile-level abstractions.* Consider the dependence between $S1$ and $S3$ on array A; each $S3(i)$ depends on an $S1(i)$. Intuitively, one can think of this as a "row-to-row" mapping. Consider now the dependence from $S2$ to $S3$ on array D. Each $S3(i)$ requires all the instances $S2(i, j, k)$, a many-to-1 relation. However, when considering all instances of $S3$, this becomes a many-to-many relation. Thus, depending on the desired granularity, this may lead to oversynchronization. At the very least, each $S3(i)$ will have to wait for the m instances of $S2(i)$, and could become as bad as $(l \times n \times m)$ instances of $S3$ each waiting for $(m \times n \times p)$ instances of $S2$.

This highlights that additional granularity of parallelism is not for free, as synchronization could increase in several orders of magnitude, and the additional benefits of parallelism will be mitigated. This granularity criterion also serves a second purpose, which is to exclusively restrict interband parallelism to the outermost dimensions of nodes/bands in a dependence relation. Finally, we systematically treat untiled bands

```
for (i=0; i<l; i++) // <-- Band 1
  for (j=0; j<m; j++)
    for (k=0; k<q; k++)
 S1:  A[i][j] += B[i][k] * C[k][j];
for (i=0; i<m; i++) // <-- Band 2
  for (j=0; j<n; j++)
    for (k=0; k<p; k++)
 S2:  D[i][j] += E[i][k] * F[k][j];
for (i=0; i<l; i++) // <-- Band3
  for (j=0; j<n; j++)
    for (k=0; k<m; k++)
 S3:  G[i][j] += A[i][k] * D[k][j];
```

Fig. 13. 3mm kernel (with removed initializations).

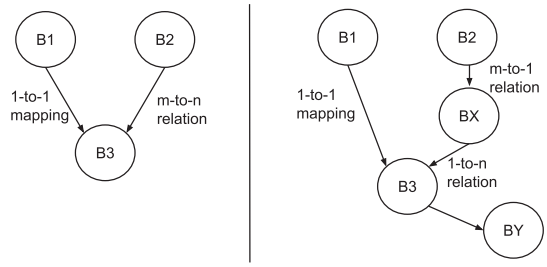

Fig. 14. 3mm 1-dimensional tile-PRDG (left) and task graph (right).

ALGORITHM 3: SelectTaskIdiom (G)

Input: G: Task-graph after partitioning
Output: G: Decorated task-graph
for *each edge* $e \in G$ **do**
  $S \leftarrow$ tile-level domain of dependence source; $T \leftarrow$ tile-level domain of dependence target;
  $m \leftarrow$ dependence map $S \rightarrow T$ of edge e; $m \leftarrow$ intersect domain of m with S;
  $m \leftarrow$ intersect range of m with T; $m \leftarrow$ decompose m into a union of basic maps in ISL;
  if S *is an untiled band* **then**
    Create a new stream *single*; Output a single dependence from S;
    Input dependence of T with peek operation from S; Insert a tick operation after band T;
  end
  else
    if *(Def.* 4.4 *is true for S and T)* **then**
      Create an array of streams D, one per distinct basic map in m;
      Decorate e with 1-to-1 inter-band parallelism on D;
    end
    else
      Create a stream *bar* of size 1 for barrier synchronization; Insert statement *barout* between nodes S and T;
      Output all dependences of S to *bar*; Input dependences of *barout* from S;
      Output a single dependence of *barout* to T; Input dependence of T from *barout* with peek operation;
      Insert statement *barin* after node T with tick operation on *bar*;
    end
  end
end

(e.g., those that result from multilevel reduction statements) as single tasks. The motivation comes from the exponential growth in synchronizations (a factor of tile size for each dimension) that can arise with such loops.

Algorithm 3 prioritizes the 3 types of communication from less communication (single task), to 1-to-1 mappings, to m-to-n communication patterns that require soft barriers. Along the process, nodes are decorated with the type of synchronization and the name of the stream to be used; new nodes are inserted to handle barriers and tick operations. The latter are OpenStream operations responsible for consuming data tokens that were previously broadcasted. Figure 14 shows the 1-dimensional tile-PRDG (left) and the result of applying our algorithm to it (right).

Interband parallelism is possible and profitable between bands B1 and B3 (akin to a row-to-row mapping). However, the algorithm deems unprofitable the number of synchronizations between bands 2 and 3. Thus the task graph is modified to reflect the insertion of a node BX that collects the output dependences from B2, and outputs a single dependence that is reused via *peek* operations in B3. This is OpenStream's broadcast stream idiom. After B3, it is also necessary to insert a new node BY to consume the token that has been used by all instances of B3 (a tick operation). Experimental results show that for this 3mm example, 1D interband parallelism (only one barrier synchronization) outperforms by 41% on AMD Opteron a 2D interband parallelism approach (no barriers, three 1-to-1 2D mappings and two many-to-many synchronizations with peek operations). The performance between the two schemes is comparable on Intel's Xeon, and also in favor of 1D interband parallelism for AMD Phenom.

We now formalize the requirements for interband parallelism. Definition 4.4 establishes that a fine-grain interleaving of loop iterations between two bands can be legally executed. Once an iteration of the source band is executed, the dependent iteration of the target band can initiate execution.

Definition 4.4 (*Interband Parallelism*). Given two distinct bands A and B. Barrierless interband parallelism is exploitable if:

(1) There is at least one point in band B that does not depend on all the points of band A.
(2) Neither band A nor band B have dependence cycles.

Definition 4.4 is a general definition for interband parallelism. Condition 1 enforces that some subset of the target band (possibly exhibited after domain partitioning) can initiate early execution, that is, a barrierless behavior. Condition 2 essentially states when a point-to-point synchronization is possible. Some of the cases that this definition covers are:

—When all points from band A have exactly one outgoing dependence instance and all points from band B have exactly one incoming dependence instance (e.g., between bands B1 and B3 in Figure 14);
—When the number of dependence instances outgoing from band A are bounded by some fixed number K_A, and the number of incoming dependence instances to band B are bounded by a fixed number K_B (e.g., in blur-Roberts kernel, $K_A = K_B = 3$). Note also that not all points in both bands are required to have the same number of incoming or outgoing dependence instances;
—With no particular relation between the cardinalities of band A and B (e.g., $card(A) = card(B)$, $card(A) = 2 \times card(B)$, $card(A) = card(B) + K$).

Definition 4.5 is used in Algorithm 3 to select when a soft barrier must be inserted in the task graph. We recall that a soft barrier is a barrier with point-to-point synchronization, that is, it does not affect all bands, but only the one that requires output data of the source band.

Definition 4.5 (*Soft-barrier synchronization*). Given two bands A and B, if no barrierless interband parallelism can be found according to Definition 4.4 then a soft barrier must be inserted between both bands.

4.5. Extracting Dynamic Tile-Level Wavefronts

The static partitioning scheme presented in Section 4.2 allows extraction of dynamic wavefronts of tiles, that is, all intertile instance dependences are made explicit to the runtime, which in turn determines the actual wavefront as tiles go executing

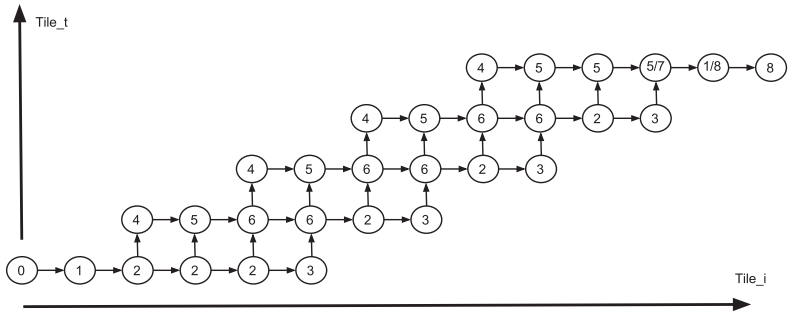

Fig. 15. Partition generated for Jacobi-2d with tile schedule (tt,2 * tt + ti). Circles represent tile instances and the number within each is the signature ID (see Figure 10).

and completing. Figure 15 displays the partition produced for a 2-dimensional (tiled) iteration domain corresponding to the Jacobi-2d running example. The arrows depict the direction in which the data flows, in this case, one token for each tile instance. The x-axis represents space, while the y-axis represents time. Partitioning yields 9 (tile) domains, each identified by its signature number. The appearance of a specific 2-dimensional signature depends directly on the domain's shape. For Jacobi-2d, the outer space dimension (tile-i) is skewed w.r.t. the time dimension and has schedule (tt,2*tt + ti), where tt is the time tile coordinate and ti is the space tile coordinate. Figure 10 shows the possible dependence signatures on a given 2D partitioning. The shaded ones are signatures that do not arise in Jacobi-2d.

4.6. Code Generation

During polyhedral code generation, special care must be taken to replicate the structures of partitioned domains as well as pretty-printing OpenStream clauses from the annotated information in the task graph. This includes the generation of input/output clauses from the dependence signature of each node. Streams are allocated and declared before the SCoP. The appropriate scalar values on the tile schedule must be set to represent the correct interleaving of bands.

—Separating dimensions are schedule dimensions, which hold scalar values. Their purpose is to keep separated certain domains. In general, polyhedral code generation will attempt to fuse all possible dimensions allowed by the schedule. Thus, in order to keep the generated partitions separated (and their code signature unique), we insert an additional scalar dimension at position 1 (offset to 0) for each band parallelized via dynamic wavefront. In standard polyhedral techniques, this is not common practice. However, since the dependences of the outer dimensions will be dynamically resolved, inserting an arbitrary and distinct scalar value for each partition remains legal.
—Stream declarations: The current implementation of OpenStream only supports 1-dimensional arrays of streams. For each interband stream, we compute the volume of the outermost dimension and use it as the stream array size. For dynamic-wavefront, we assume a 2-dimensional linearized array of dimensions $D_0 \times D_1$, where D_0 and D_1 are the volumes obtained by projecting out all dimensions except dimension 0 for the former and dimension 1 for the latter. However, this results in a nonaffine parametric product. Thus, the product is computed out of the polyhedral code generation process.
—Stream indexation: Stream arrays are indexed by the outer tile coordinates, which is either the source or target of the tile dependence: all input streams read from their local tile coordinate and write to the tile coordinate determined by the outset of a tile dependence. Here is where the property of having output sets that are

```
for (int tt = 1; tt < (tsteps-1) / 16; tt++)
 for (int ii = 2*tt+2; ii <= 2*tt + (n-3) / 16; ii++)
  #pragma omp task input(stream_tt[(tt) * S_1_3 + ii] >> token1_1,\
      stream_ii[(tt) * S_1_3 + ii] >> token2_1)\
    output(stream_tt[(tt+1) * S_1_3 + ii] << token1_2,\
      stream_ii[(tt) * S_1_3 + 1+ii] << token2_2)
  for (int jj=ii; jj <= ii+(n-3)/16+1; jj++)
   for (int t=max(16*tt,-n+8*jj+2); t <= 16*tt+15; t++)
    for (int i=max(16*ii,-n+16*jj+2); i<=min(min(16*ii+15,16*jj+14),n+2*t-1); i++)
     for (int j=max(i+1,16*jj); j <= min(16*jj+15,n+i-2); j++) {
      if (n+2*t >= i+2) B[-2*t+i][-i+j] = 0.2*(A[-2*t+i][-i+j] + A[-2*t+i][-i+j-1]
        + A[-2*t+i][-i+j+1] + A[-2*t+i+1][-i+j] + A[-2*t+i-1][-i+j]);
      A[-2*t+i-1][-i+j] = B[-2*t+i-1][-i+j];
     }
```

Fig. 16. Jacobi-2d code snippet for partition with signature 6 (see Figure 15).

explicit functions of the input set in a tile dependence becomes essential. In addition, since the dynamic wavefront requires a 2-dimensional access to the stream array, which commonly results in a nonaffine expression (e.g., *iterator* × *parameter*), the linearization process is performed in a pragmatization pass after polyhedral code generation.

—Simplification: To avoid some corner and degenerate cases, such as having only one tile instance on any dimension, we set the parameter context (e.g., problem sizes) as having at least 4 tiles along a dimension. The partition shown in Figure 15, as well as the code in Figure 16, have this simplification.

—Dealing with One-Time-Loops (OTLs): OTLs are loops that execute only once. Stream indexation requires all tile coordinates in order to map tile instances to individual streams in an array. Currently, PPCG (which uses CLooG-ISL) does an overoptimization and removes all OTLs. In this case, we resort to regenerating the explicit tile coordinates. However, other code generators, such as the non-ISL based CLooG [Bastoul 2004], allow the explicit generation of OTLs. This simplifies the stream mapping process.

—Finally, a global *taskwait* is inserted after the SCoP so that the program does not terminate early. It is particularly useful to use unscaled tile iterators during code generation in order to facilitate the mapping between tile coordinates and stream arrays. The generated code for one of the partitions is shown in Figure 16. More examples of the generated code can be found in Kong et al. [2014].

4.7. Putting It All Together

Algorithm 4 brings together our approach for extracting both interband and intraband tile-level parallelism, which allows description of concurrent tasks with explicit, point-to-point dependences. Based on the number of tile bands produced, one of our two main techniques is applied, that is, interband parallelism in the presence of more than one band, and our partitioning scheme, which enables a dynamic wavefront of tiles.

4.8. Handling Complete Applications

The techniques described in this article can be applied to a variety of real-world applications. To do so, the input program needs to be affine or to have affine-friendly portions. Also, a number of preprocessing transformations would normally be required to make a program tilable and to further expose parallelism (e.g., array/scalar expansion, renaming, privatization, and scalar expression inlining).

Partitioning scalability. Our partitioning technique could be easily extended to handle more than the two outermost dimensions. In general, partitioning a domain

ALGORITHM 4: High-level algorithm for data flow task-level parallelization

```
Input: Statement level PRDG and tile schedule
Output: Tile level task graph and PRDG with decorated nodes and edges
if (nbr.bands > 1) then
    if (no dependences) then
        Bands fully parallel and independent;
    end
    else
        Tile_PRDG ← build 1-dimensional tile-level PRDG;
        T ← ConstructTaskGraph (Tile_PRDG);
        T ← SelectTaskIdiom (T);
    end
end
else
    if (outer dimension is parallel) then
        Band is fully parallel;
    end
    else
        if (all dependences are uniform) then
            Tile_PRDG ← build 2-dimensional tile-level PRDG;
            T ← ConstructTaskGraph (Tile_PRDG);
            T ← SelectTaskIdiom (T);
        end
    end
    return {Tile_PRDG,T};
end
```

according to dependence signatures yields $O(2^{2\times nbr.deps})$ parts. This, for a 2D domain, represents up to 16 parts per program statement. In practice, a few signatures are impossible to generate for a particular input and tile schedule. However, for higher-dimensional domains this number grows fast. A 3D domain could potentially generate up to 64 parts and a 4D domain 256. At this point, the polyhedral code generation could become a bottleneck, especially for complete applications, which can have hundreds of statements.

5. EXPERIMENTAL RESULTS

We now demonstrate the effectiveness of our approach both analytically and experimentally. In Section 5.1, we present an analysis of certain program properties that allow analysis of performance. Section 5.2 describes our experimental setup. We conduct three sets of experiments. In Section 5.3, we first show the performance achieved by our framework and discuss its relation to the analytical features presented previously. In Section 5.4, we conduct a workload distribution study for three benchmarks while comparing it to PLuTo's tiled variants. In Section 5.5, we compare our dynamic-wavefront + partitioning technique to diamond tiling in terms of performance scalability and code features. Finally, Section 5.6 discusses the profitability of tiling and fusion, and provides guidelines for the class of programs that best suit our framework.

5.1. Benchmark Properties

We evaluate numerous benchmarks with different reuse and barrier patterns, comparing the performance obtained using our PPCG/OpenStream-based approach versus a PLuTo/OpenMP-based one. It is well known that for certain benchmark types, typically dense linear algebra with $\mathcal{O}$(n) reuse and $\mathcal{O}$(1) barriers such as dgemm, that

Table I. Benchmark Features

Configuration	gemm	syrk	2mm	3mm	gemver	Jacobi-2d	fdtd-2d
Data reuse	$\mathcal{O}$(n)	$\mathcal{O}$(n)	$\mathcal{O}$(n)	$\mathcal{O}$(n)	$\mathcal{O}$(1)	$\mathcal{O}$(t)	$\mathcal{O}$(t)
Tile footprint (KB)	24	24	24	24	8	4	6
DF Par. Method	I	I	I	I	I	P+DW	P+DW
#Minfuse OMP Barriers	2	2	4	6	4	$\mathcal{O}$(t+n)	$\mathcal{O}$(t+n)
#Smartfuse OMP Barriers	2	2	2	6	4	$\mathcal{O}$(t+n)	$\mathcal{O}$(t+n)
#DF Barriers	0	0	0	1	3	0	0

Configuration	blur-Roberts	Seidel-1d	Seidel-2d	corre-lation	cova-riance	segmen-tation	ExpCNS
Data reuse	$\mathcal{O}$(1)	$\mathcal{O}$(t)	$\mathcal{O}$(t)	$\mathcal{O}$(n)	$\mathcal{O}$(n)	$\mathcal{O}$(1)	$\mathcal{O}$(1)
Tile footprint (KB)	16	8	2	8-24	8-24	32	32
DF Par. Method	P+I	P+DW	P+DW	I	I	I	P+I
#Minfuse OMP Barriers	2	$\mathcal{O}$(t+n)	$\mathcal{O}$(t+n)	14	7	20	29
#Smartfuse OMP Barriers	$\mathcal{O}$(n)	$\mathcal{O}$(t+n)	$\mathcal{O}$(t+n)	8	7	$\mathcal{O}$(n1+n2)	$\mathcal{O}$(n)
#DF Barriers	0	0	0	10	4	6	0

(I:Inter-band, DW:Dynamic Wavefront, P:Partitioning; n,t: problem sizes)

Table II. Experimental Testbed (left) and Benchmark Description (right)

	AMD Opteron 6274	Intel Xeon E5-2650 v2
Freq	2.2GHz	2.6GHz
Cores	16	8
L1	16KB	32KB
L2	8 x 2MB	256KB
L3	6MB	20MB
RAM	32GB	16GB

Benchmarks	Category	Problem Size	Tile Size
2mm, 3mm, gemm, syrk	linear algebra	2000^3	32
gemver	linear algebra	8000^2	32
correlation, covariance	data mining	2000^2	32
Seidel-1d	stencils	200000^2	1024
blur-Roberts	image processing	4000^2	32
fdtd-2d, Jacobi-2d, Seidel-2d	stencils	2000^3	16
segmentation	3D MRI processing	$760^2 \times 130$	4×4×32
ExpCNS	DoE mini-app	32^3	4×4×32

the PLuTo/OpenMP approach is already very effective, implementing the available reuse via tiling and exposing load balancing via a simple OpenMP parallelization. For these benchmarks, our objective is to show that PPCG/OpenStream matches the performance of PLuTo/OpenMP, that is, we do not suffer performance degradation through our approach. On the other hand, for benchmarks with $\mathcal{O}$(1) data reuse and/or $\mathcal{O}$(n) barriers generated using PLuTo/OpenMP, our technique for barrier removal and dynamic fusion has the potential to significantly outperform the PLuTo/OpenMP approach, as demonstrated later in this section.

Table I summarizes several of the properties of the benchmarks we evaluate. The DF Par. method shows the type of dataflow parallelism implemented by our framework. The #DF barriers shows the number of "dataflow barriers" that are generated by our framework, and compares to the number of OMP barriers generated by two relevant fusion heuristics implemented in PLuTo used in our experiments. Additional code features can be found in Kong et el. [2014].

5.2. Experimental Methodology

The machines used for the experiments as well as the benchmarks are described in Table II. We test our approach on a subset of Polybench-3.2 [Pouchet 2012] and two applications. The compilers used were GCC 4.8.1, ICC 2013-update5, and PoCC-1.3. Our framework was built over PPCG [Verdoolaege et al. 2013]; experiments were performed with two OpenStream versions, the public version [Pop and Cohen 2013] and one in development for trace capabilities. The FLOPS for each benchmark are computed statically.

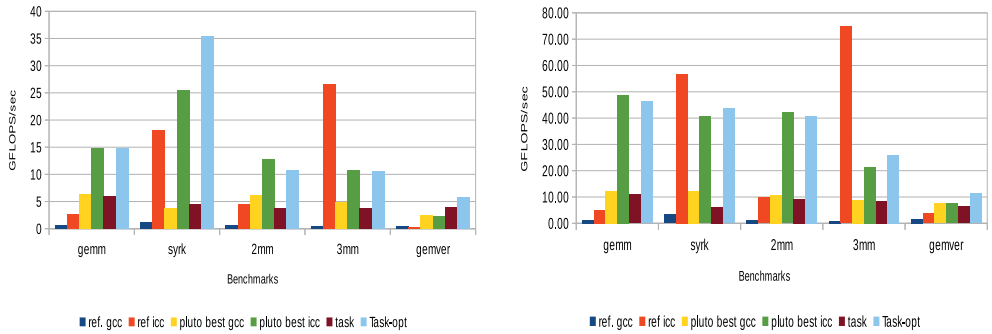

Fig. 17. Performance on AMD Opteron (*left*) and Intel Xeon (*right*) of linear algebra kernels.

We report the performance (GFLOPS/sec) on double-precision floating point for all Polybench benchmarks and ExpCNS, and single-precision for *segmentation*. All available cores are used. For each benchmark, we report baseline performance on GCC and ICC (-O3 -ffast-math for GCC and -O3 -parallel -xhost for ICC), PLuTo's best performance between tile with minfuse and tile with smartfuse compiled with both ICC and GCC (generated using PoCC's parallel, minfuse/smartfuse, prevector, pragmatizer and vectorizer flags, compiled with -O3 -openmp -xhost), and the performance of two OpenStream variants: one, Task, exclusively compiled with OpenStream's modified GCC (v4.7.0) and the second one, Task opt exporting the task bodies into separate compilation units that are (1) processed by PoCC using pragmatizer, vectorizer, and past-hoist-lb flags to have similar intratile loop order as with; and (2) compiled with Intel ICC -O3 -xhost for further optimization, and then linked with OpenStream's GCC modified compiler.

5.3. Performance Results

We remark that the achievable performance for the same code could vary significantly between GCC and ICC. This is why we report numbers not only for OpenStream's native compiler (GCC), but also for (*task-opt*), a hybrid compilation scheme relying on ICC for task bodies and OpenStream's GCC for task creation and scheduling. GCC does not always implement as many optimizations as ICC for these numerical codes, implementing less effective automatic vectorization and less optimizations in the low-level generated code. In addition, the intratile (i.e., task) loop order with *task-opt* is subsequently optimized to match PLuTo's generated intratile loop order, leading to additional performance improvements.

5.3.1. Linear Algebra Kernels. 2mm, 3mm, gemm, syrk, and gemver fall into this group. Figure 17 shows that we achieve performance comparable to PLuTo for the variant codes with $\mathcal{O}$(n) reuse. These codes are already balanced in their OpenMP/PLuTo variant, and tiling achieves the $\mathcal{O}$(n) reuse independently of the parallelization method. Concurrent start is possible with PLuTo's barrier synchronization method as well as our interband. PLuTo is expected to perform very well on these codes, and no significant gain is expected with our task-based parallelization scheme.

Regarding gemver, the relative workload per barrier is much lower than for the other codes, as illustrated by the $\mathcal{O}$(1) reuse, exacerbating the cost of barriers. In both PLuTo- and PPCG-generated codes, there is one inner loop with high-stride memory access. Still, there is some reuse of data between phases of the computation; this reuse is fully implemented through OpenStream's default task's firing policy. Through trace analysis, it was determined that for this benchmark, task execution followed a zig-zag

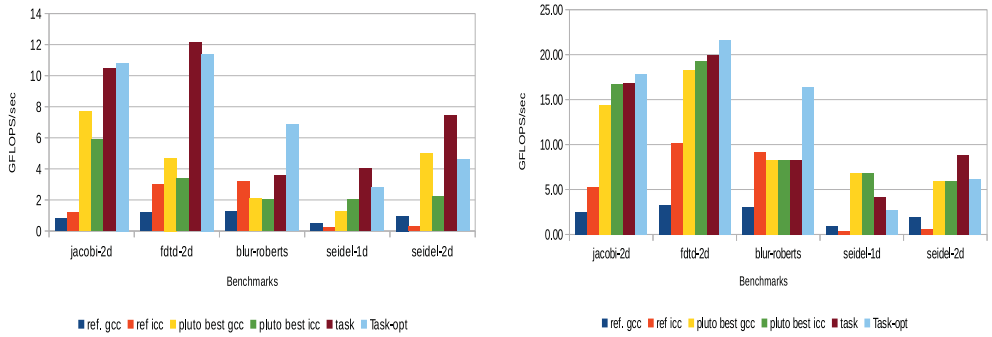

Fig. 18. Performance on AMD Opteron (left) and Intel Xeon (right) of stencil kernels.

pattern between tasks of adjacent bands, that is, the first band executes tasks mostly in ascending order, the second band in descending order, the third, again in ascending order, and so on. This also enables dynamic fusion by reusing the data of the last task.

We also note that kernel 3mm exhibits an unusually high baseline. Further inspection revealed that ICC was able to pattern-match matrix-multiply kernels in the input code, replacing all three loop nests in 3mm by MKL calls, and one of the two matrix-multiply of 2mm by an MKL call. ICC was, however, unable to recognize and pattern-match the gemm kernel or its occurrence in 2mm, meaning that the only two kernels for which ICC was able to use MKL instead of the original code were 2mm (in part) and 3mm (in full). Regarding syrk, ICC was unable to vectorize PLuTo variants, while partially vectorizing the reference code. syrk *task-opt* was vector-array scalarized, but allowing ICC to decide whether it is vectorizable or not (i.e., no SIMD/vector pragmas were inserted). Finally, no significant differences were observed between PLuTo's fusion heuristics.

5.3.2. Stencil Kernels. Here we have Jacobi-2d, fdtd-2d, Seidel[1-2]d and blur-Roberts. All five kernels update neighboring points on a dense grid. blur-Roberts performs a single sweep on the image, whereas the other are iterative stencils that are repeated T time steps, thereby showing a higher order of data reuse. These first four iterative stencils can benefit from time tiling + wavefront parallelism and show $\mathcal{O}(t)$ reuse, while only standard tiling is meaningful for blur-Roberts. The iterative stencils have $\mathcal{O}(\mathrm{t}+\mathrm{n})$ barriers when parallelized with PLuTo (the exact value depends on the selected tile sizes as well as the skewing factors). The number of barriers with PLuTo could be reduced by selecting larger tile sizes, but there is a limit on the maximal tile size to preserve L1 data locality and it would also decrease the parallelism. In contrast, our framework leverages point-to-point dependences to address load imbalance, and tile sizes can be selected in a more independent fashion. In particular, smaller tile sizes benefit our framework by decreasing the task footprints and increasing the possibility of further temporal reuse via dynamic fusion, whereas such options will simply add overhead to static wavefronts.

Figure 18 shows that our partitioning technique combined with dynamic wavefront can achieve speedups ranging from 1.1× to above 3×. Moreover, for these benchmarks the performance gap between *task* and *task-opt* is negligible or favors GCC, that is, ICC does not provide additional benefit and most performance gains are due to increased parallelism and barrier removal. Finally, we note that PLuTo produces a 3-dimensional wavefront for Jacobi-2d, fdtd-2d, and Seidel-2d, whereas our approach exposes a 2-dimensional wavefront.

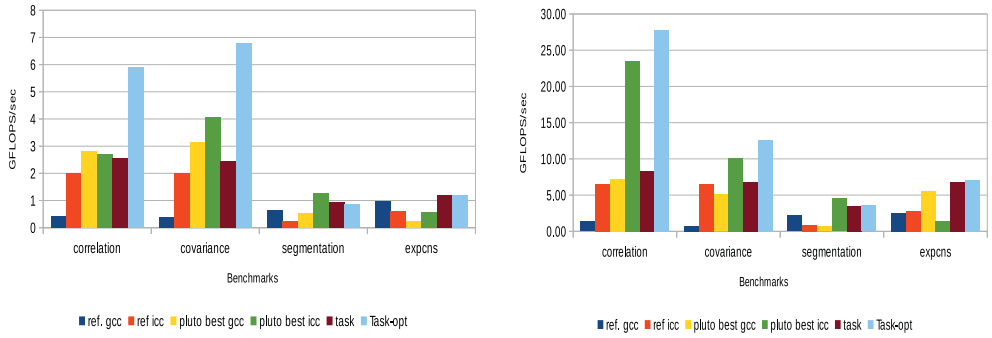

Fig. 19. Performance on AMD Opteron (*left*) and Intel Xeon (*right*) of applications group.

Regarding blur-Roberts, partitioning allows pipelining tasks between two bands. The reuse order is $\mathcal{O}(1)$ and the reuse distance 1 (one stream is induced by a flow dependence). OpenStream's firing policy favors immediate triggering of tasks, enabling dynamic fusion. PLuTo variants trade-off locality and parallelism: minfuse has 2 barrier synchronizations and achieves the steady state in $\mathcal{O}(1)$ time, but has poor interband locality, whereas smartfuse enhances locality but strongly diminishes parallelism by inducing a wavefront-like parallelism ($\mathcal{O}$(n) barrier synchronizations). Furthermore, smartfuse also affects negatively the vectorizability of the code.

5.3.3. Applications. Figure 19 show our results for correlation, covariance, segmentation, and expcns.

Correlation and covariance. Both applications show, as for gemver, a low ratio of workload per barrier for the most part of the application, increasing the gain in removing barriers. The $\mathcal{O}$(n) reuse happens in the next to last band on both applications. This avoids a potential early data flush that could hinder performance. We also note that even when not removing all barriers (see Table I), our barriers are dataflow point-to-point barriers, for example, tasks only wait on their specific input streams and a single task can wait on more than one dataflow barrier.

Segmentation. Segmentation is one of the five stages (the most time-consuming one) of the Computer-Aided Diagnostic pipeline for lung cancer screening on 3D MRI images developed by the Center for Domain-Specific Computing. It performs 3D image segmentation, using an iterative solver that typically converges in two time iterations. Only the spatial loops have a static control flow, thus we limit to optimizing a single iteration of the algorithm. We evaluated multiple versions of PLuTO and different tile sizes. Tiled and untiled minfuse variants outperform their smartfuse counterparts. PLuTo minfuse outperforms interband by approximately 20%. This is due to the large program footprint, the DAG's shape and the dependence distance. Essentially, this program can be divided into 2 stages: the first has a high number of parallel bands with a dependence distance of 1 (immediate intertask reuse); the second is a more "ordered" stage, in which most bands require the result of one or more of the previous bands. The tile sizes selected in the space allow for a fair number of task footprints to fit in L1. However, the ordered stage induces a complete flushing of data, killing locality. It is expected that a better performance could be achieved with our framework if we had some control over the task scheduling with OpenStream.

Exp_CNS_NoSpec. Exp_CNS-NoSpec is a mini-application to integrate the Compressible Navier Stokes (CNS) equations, from the DoE Exact center (Center for Exascale Simulation of Combustion in Turbulence). It is written primarily in Fortran.

Various C-functions are provided to replace the native *hypterm* and *diffterm* routines, which amount to 90% of the execution time. Three SCoPs are extracted from the array-based variant. ifort (ICC v13) and gfortran (GCC 4.8.1) compiler settings were used. The default dataset was used, performing 100 time iterations on a 32^3 grid. Full dynamic fusion through interband parallelism is not achieved due to long dependence chains, but it is still able to exploit a better balance between parallelism and locality than PLuTo variants. PLuTo minfuse experiences a lack of locality, whereas smartfuse loses in terms of parallelism (complex loop structure and parallel loops at depth 1 with regard to the SCoP's root). PLuTo untiled variants were also evaluated, yielding an average 50% slowdown. An extended study of *segmentation* and *expcns* can be found in Kong et al. [2014].

5.4. Workload Distribution

We now analyze the degree of parallelism and load balance achieved by our method enhanced with further ICC optimization (*task-opt*) and contrast it to PLuTo's heuristics, minfuse and smartfuse, which are parallelized with OpenMP's *for* work-sharing construct with default scheduling policy. We focus the analysis on the AMD Opteron platform, and select 3mm, Seidel-2d, covariance and gemver as case studies. For each benchmark, we decompose its execution time into 3 parts: the sequential time, the parallel balanced time, and the parallel unbalanced time. The breakdown of the execution time has been obtained through the following methodology:

—We use a version of OpenStream capable of generating traces. In particular, we consider the time spent in task creation (serial), task initialization, task execution and task-seeking states, of which the last three are total time across all cores. The breakdown shown was converted from cycles to seconds, considering the number of executing cores and their frequency. We consider the task execution time as parallel balanced, and the initialization and seeking times as unbalanced.
—PLuTo's variants are compiled with ICC v11, using the same flags as in the previous section, with the exception of -openmp, which is replaced by -openmp-profile. The output obtained includes sequential time and per-core minimum, average and maximum parallel time, both balanced and unbalanced. Here we define $time^{unbalanced}$ as $\max(time^{unbalanced}_{core\_id})$ and $time^{balanced}$ as $\max(time^{parallel}_{core\_id}) - time^{unbalanced}$.

Although we use different compiler versions for this set of experiments than in the previous section, we note that no significant difference was observed in the performance.[2]

3mm. A dense linear algebra kernel such as this is not expected to benefit much from the dataflow parallel model; minfuse and smartfuse produce 6 tiled loop nests, but with different loop structure. In both cases, these are mapped to 6 parallel regions. In contrast, the inherent task parallelism is limited to the initializations of each of the 3 product matrices, followed by the computation of the left and right side matrices required for the final product. Figure 20 shows that *ref icc* and PLuTo's variants saturate at 8 cores. This is due to the 8×2MB distributed structure of Opteron's L2 cache, which forces many threads to access data from a distant core. Variant *task*, being fully compiled with GCC, is compute bound, but after further optimizing the task bodies with ICC (see the *task opt* graph) this is no longer the case, achieving linear scaling with the number of cores. Regarding the impact of barrier removal, we show 3mm's time breakdown in Figure 21. The performance achieved is slightly superior to both of

[2]The OpenMP profiling option has disappeared from recent versions of ICC, thus the use of v11 in this specific experiment.

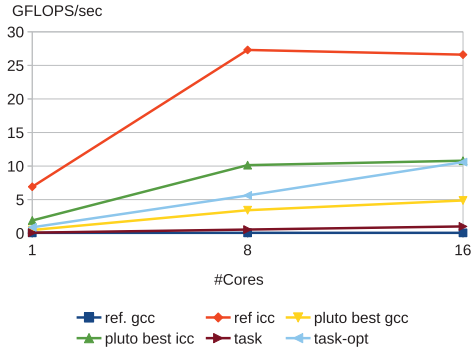

Fig. 20. Performance scalability for kernel 3mm.

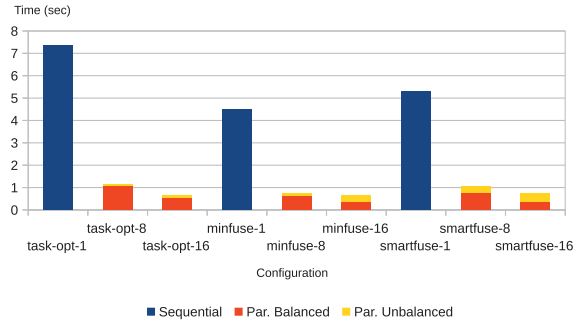

Fig. 21. Time breakdown of 3mm kernel.

PLuTO's. *task opt* suffers from load imbalance due to initialization and task creation overhead, as these steps run on a single core. This phenomenon is visible in most of our experiments. The fraction of the unbalanced execution time for OpenStream represents approximately 8% on 8 cores and 17% on 16 cores, whereas for PLuTo it varies between 15% and 50%.

Seidel-2d. This kernel becomes highly unbalanced when applying PLuTo's tiling heuristics combined with the static wavefront technique. It consists of a single statement, and both tiling heuristics produce the same fusion structure. Figure 22 shows the lack of scalability of this benchmark when compiled with GCC and ICC. PLuTo minfuse, on the other hand, scales well with GCC, but saturates at 8 cores with ICC. Further inspection revealed that task variant incurred in 50% less L1 accesses when scaling from 8 to 16 cores, whereas minfuse-ICC experienced a 15% reduction and minfuse-GCC only a 10% reduction. L1 misses were also reduced by 50% for task variant, 23% for minfuse-ICC, and minfuse-GCC remained the same. Regarding the L2 cache behavior, task variant showed 25% less accesses than minfuse for ICC and GCC for 8 cores. For 16 cores, the L2 accesses difference between task variant and minfuse variant varied between 4% and 6%. L2 misses were also reduced by 54% for task variant from 8 to 16 cores, but only 13% for minfuse-ICC, and 30% for minfuse-GCC. When using all 16 cores, the 3 variants exhibit the same number of L2 misses. Figure 23 shows that the unbalanced fraction of time for minfuse-ICC is approximately 50% on 8 and 16 cores. Task variant (compiled only with GCC) spends 5% of its time in an unbalanced state for 8 cores and increases to 10% on 16, while the balanced time is reduced to half as there are twice as many cores for computing.

Covariance. PLuTo's heuristics and parallelization method limit the potential performance of this kernel. Both smartfuse and minfuse heuristics yield 7 parallel regions. While PLuTo variants saturate at 8 cores (Figure 24) and reach a maximum of 4× speedup, *task-opt* continues scaling, although at a lower rate, achieving a final speedup of 7× with regard to single-threaded execution time. Figure 25 shows that approximately 96% of the total time is spent in unbalanced parallel execution with PLuTo-generated variants, while this is reduced by a factor of 4 with task-parallel execution.

5.5. Dynamic Wavefront vs. Diamond Tiling

We now compare our dynamic wavefront technique to diamond tiling [Bandishti et al. 2012], which aims at allowing concurrent start and improving the load imbalance in stencil computations that are parallelized with doall/barriers. It allows concurrent

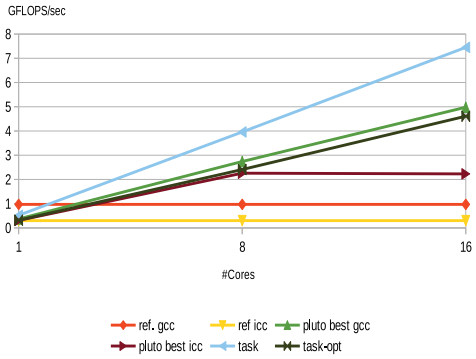

Fig. 22. Performance scalability for kernel Seidel-2d.

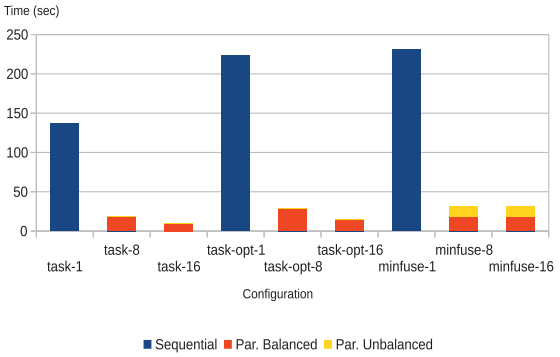

Fig. 23. Time breakdown of Seidel-2d kernel.

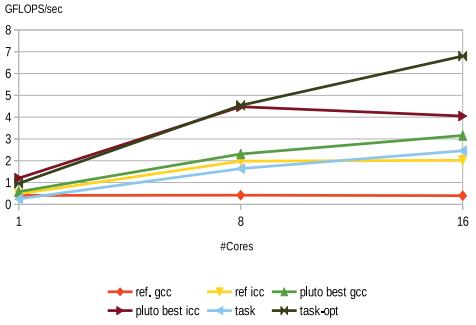

Fig. 24. Performance scalability for covariance kernel.

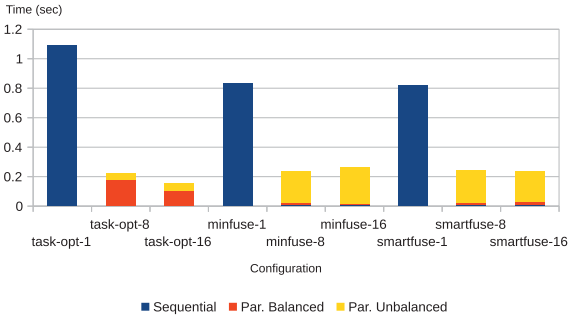

Fig. 25. Time breakdown of covariance kernel.

start along (at least) one of the faces of the iteration domain. It assumes that all tile dependences (in the transformed space) are unit vectors, thus similar to our usage of uniformizing dependences prior to the domain partitioning stage. A sufficient condition for diamond tiling is that the tile schedule must have the same direction as a face of the iteration domain.

We compare the scalability properties as well as the code characteristics of both approaches using the Jacobi-2d kernel, a 5-point stencil that meets diamond tiling's requirements. Diamond tiling variants were generated with PLuTo v.0.10, using flags –partlbtile (1-dimensional load balance tiling) and –parallel for OpenMP pragmatization, considering the default fusion heuristic (smartfuse). Experiments were conducted on the AMD Opteron and Intel Xeon (see Table II). The diamond tile variants were compiled with GCC 4.8 using flags -O3 -fopenmp and -ffast-math. We use the same tile sizes as the experiments reported by Bandishti et al. [2012].

On the performance aspect, our dynamic wavefront outperforms diamond tiling on the two machines (Figure 26). Both techniques scale very well on the Intel processor. However, as in the previous experiments, diamond tiling does not scale beyond 8 cores on the Opteron, whereas the dynamic wavefront achieves a speedup of 11× with regard to its sequential performance. As the Opteron has an L1 data cache of 16KB, a tile of 32×32 on double precision and two arrays occupies most of it. Increasing the tile size to 64 does not improve performance for the task variant, but represents a 3GF/s increase for diamond tiling. On Intel Xeon, we see a similar behavior, but without the

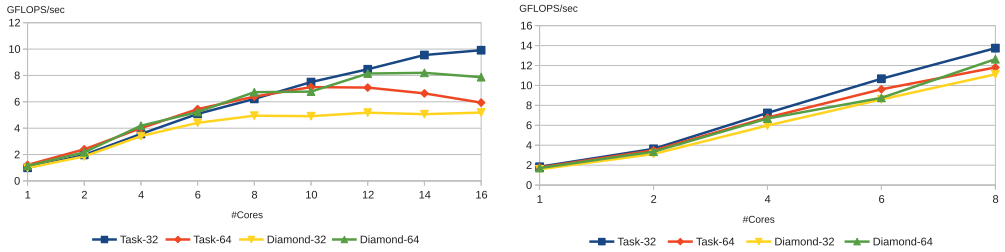

Fig. 26. Dynamic wavefront versus diamond tiling, tile sizes 32 and 64 on AMD Opteron (*left*) and Intel Xeon (*right*).

Opteron's plateauing, and both techniques scale almost linearly. In general, our task variants will benefit more from having tasks with tiles that are slightly smaller than the L1 cache.

Regarding the code structure, diamond tiling's code still suffers from a large number of barriers due to having a doall loop nested within a sequential loop. Furthermore, Bandishti et al. [2012] also state that, in practice, partial concurrent execution (only the outermost dimension) yields better performance than exploiting parallelism across all faces of the iteration domain due to code complexity, manifested as code explosion and numerous modulo conditions along the code (about 8,000 lines and 400 modulo conditions for full-dimensional concurrent start, but about 900 lines and 7 modulo conditions when using the 1-dimensional option). Two more features hinder the performance of this technique: conditions nested within the innermost loops and the tile shape. Both inhibit vectorization in many cases. On the contrary, partitioning for generating dynamic wavefronts produces a more compact code, about 160 lines for the same kernel and the modulo conditions appear mostly on the outermost dimensions.

Finally, we note that diamond tiling is not applicable to kernels such as Seidel-2d, because of the nature of their dependences on the tiled (transformed) program. On the other hand, our framework seamlessly handles these kernels, as long as the processed tile dependences used for partitioning are uniform.

5.6. The Role of Tiling and Loop Fusion

While not all affine programs are tilable, a strength of the polyhedral compilation framework is to dramatically increase the applicability of tiling by automatically computing a program transformation to make the code tilable [Bondhugula et al. 2008]. In our framework, we choose tiling as a mechanism to expose atomic tasks with a variable granularity (tile size), allowing amortization of the runtime overhead and better control of data locality opportunities by enforcing a partial order on the operations. We note that while tiling is not needed for programs without significant data reuse potential, it is not a detrimental transformation either, as long as the changes needed to create a tiled implementation do not prevent optimizations that were possible on the original code, such as prefetching or SIMD vectorization.

Loop fusion is often considered in conjunction with tiling, to increase data locality and create tile bodies containing more statements. Excessive fusion, however, can be detrimental to performance as it may saturate prefetch streams or prevent SIMD vectorization by reducing the dependence distance. Complementary transformations after tiling to restore SIMD vectorization capabilities, for example, Kong et al. [2013], were not considered in this work, nor the exploration of different coarse-grain fusion/distribution structures to form tiles [Pouchet et al. 2010]. In future work, we will consider evaluating the impact of these task creation methods.

In this article, we restrict our analysis to tiled variants only; the complete study can be found in Kong et al. [2014]. To conclude, the metrics displayed in Table I help in

understanding the suitability of a program for our framework, mostly to distinguish between already-efficient OpenMP implementations (large data reuse available, load balance achieved, limited number of barriers and large workload per barrier); and unbalanced/oversynchronized programs with or without data locality potential. We remark that our work precisely exposes data locality opportunities between tiles, an information that could be used to compute an affinity for the tasks. Unfortunately, the current OpenStream implementation we used does not allow provision of scheduling guidelines and therefore locality opportunities were often lost due to the task firing policy of OpenStream. It is expected that the performance of codes generated with our framework would improve further when using such affinity information.

6. RELATED WORK

A number of runtimes have been proposed for dynamic task parallelism with intertask dependences. In particular, the DAGuE runtime [Bosilca et al. 2012], the University of Delaware codelet model [Suettlerlein et al. 2013] and the related SWARM environment [ETI International 2014], and the T* runtime of OpenStream [Pop and Cohen 2013] are all "feed-forward" or "argument fetch" data-driven models. In such runtimes, tasks notify their successors upon completion. While this increases the programming or compilation burden—continuations may have to be exposed and passed explicitly, the dependence resolution algorithm can be fully distributed and its complexity is generally independent of the number of blocked tasks [Bosilca et al. 2012; Pop and Cohen 2013]. This contrasts with a family of runtime systems in which a dynamic dependence resolver identifies the ready tasks, such as the current runtimes supporting the StarSs [Planas et al. 2009] and CnC [Budimlic̀ et al. 2010] languages.

DAGuE is unique in that it uses a symbolic representation of the acyclic dependence graph to avoid building it in extension. This approach is reminiscent of context-based dataflow execution and architectures, in which the iteration space structure is directly embedded into the data-driven execution mechanics [Watson and Gurd 1982; Kyriacou et al. 2006]. While such symbolic approaches are more local memory-efficient than explicit graph representations, as in Star-PU, the exact benefits remain largely unknown compared to the fully dynamic dependence graphs of SWARM or T* (OpenStream). Nevertheless, the evaluation is orthogonal to our work, which can make use from both approaches.

Our technique relies heavily on the partitioning of the iteration domain of a loop nest. Griebl et al. proposed the technique known as Index Set Splitting (ISS) [Griebl et al. 2000] and Pugh and Rosser proposed Iteration Set Slicing (also ISS) [Pugh and Rosser 1997]. Both techniques partition iteration domains based on dependence patterns (e.g., when a dependence changes direction, or based on transitive closure computations). Their technique is applied as a preprocessing step that allows extraction of more parallelism, or application of more aggressive affine transformations. We devised a form of ISS that operates—on tile dependences—performed once the schedule has been computed.

Baskaran et al. [2009] develop a framework to extract tile-level task-parallelism but with full dynamic dependence resolution. Diamond tiling [Bandishti et al. 2012] and split tiling [Henretty et al. 2013] aim at reducing the load imbalance derived from static wavefronts, that is, the ramp up/ramp down. However, the wavefront itself is only the first part of the problem. The second part is the oversynchronization induced by barriers, and in that sense, both techniques still suffer from this issue. Furthermore, split tiling's two-phase execution (per dimension) increases the number of synchronizations exponentially in the number of dimensions, which are more expensive in nonuniform architectures. To conclude, our interband approach could also be combined with split-tiling to remove unnecessary synchronizations in between phases.

7. CONCLUSIONS AND FUTURE WORK

We presented a systematic approach to compile tilable affine loop nests into concurrent, dependent tasks. We formulated a partitioning algorithm based on the tile-to-tile dependences represented as affine polyhedra. This algorithm takes out much of the burden of runtime dependence enforcement, while preserving its load balancing and lightweight synchronization benefits. We implemented the algorithm in the PPCG research compiler, targeting the OpenStream dataflow language. Our results confirm the advantage of task-parallel execution over barrier-based, data-parallel patterns.

These results push for the generalization of the partitioning algorithm to more complex, irregular control flow, and for further efforts to reduce code size, implementing a hybrid approach in which some of the partitioning occurs offline, while less performance-sensitive dependence patterns are deferred to conditional dataflow and implicit dependence resolution at runtime.

REFERENCES

Randy Allen and Ken Kennedy. 2002. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, CA.

Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 40.

Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2009. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. *ACM Sigplan Notices* 44, 4, 219–228.

Cedric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 7–16.

Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices* 43, 6, 101–113.

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack Dongarra. 2012. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Comput.* 38, 1–2, 37–51.

Zoran Budimlic̀, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Taşirlar. 2010. Concurrent collections. *Sci. Program.* 18, 3–4, 203–217. http://portal.acm.org/citation.cfm?id=1938482.1938486

Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. ACM, New York, NY, 51–61.

Alain Darte and Frédéric Vivien. 1997. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming* 25, 6, 447–496.

Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem, part II: Multidimensional time. *Intl. J. of Parallel Programming* 21, 6, 389–420.

Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations. *International Journal of Parallel Programming* 34, 3, 261–317.

Martin Griebl, Paul Feautrier, and Christian Lengauer. 2000. Index set splitting. *International Journal of Parallel Programming* 28, 6 (2000).

Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A stencil compiler for short-vector SIMD architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ACM Press, New York, NY, 13–24.

ETI International. 2014. SWARM (SWift Adaptive Runtime Machine). Retrieved November 17, 2014 from http://www.etinternational.com/index.php/products/swarmbeta.

Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in dataflow programming languages. *Comput. Surveys* 36, 1, 1–34. DOI:http://dx.doi.org/10.1145/1013208.1013209

Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *IFIP'94*, North Holland (Ed.). 471–475.

Martin Kong, Antoniu Pop, R. Govindarajan, Louis-Noël Pouchet, Albert Cohen, and P. Sadayappan. 2014. *Compiler/Run-Time Framework for Dynamic Data-Flow Parallelization of Tiled Programs*. Technical Report OSU-CISRC-7/14-TR14. Department of Computer Science and Engineering, The Ohio State University.

Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. *ACM SIGPLAN Notices* 48, 6, 127–138.

Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. 2006. Data-driven multithreading using conventional microprocessors. *IEEE Trans. on Parallel Distributed Systems* 17, 10, 1176–1188.

Samuel P. Midkiff and David A. Padua. 1986. Compiler generated synchronization for do loops. In *ICPP*. 544–551.

Samuel P. Midkiff and David A. Padua. 1987. Compiler algorithms for synchronization. *IEEE Transactions on Computers* 36, 12. 1485–1495.

Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. 2009. Hierarchical task-based programming with StarSs. *International Journal on High Performance Computing Architecture* 23, 3, 284–299.

Antoniu Pop and Albert Cohen. 2012. *Control-Driven Data Flow*. Technical Report RR-8015. INRIA.

Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO),*

Louis-Noel Pouchet. 2012. PolyBench: The Polyhedral Benchmark suite. http://web.cse.ohio-state.edu/~pouchet/software/polybench.

Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. 2010. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Conference on Supercomputing (SC'10)*. IEEE Computer Society Press, New Orleans, LA.

William Pugh and Evan Rosser. 1997. Iteration space slicing and its application to communication optimization. In *Proceedings of the 11th International Conference on Supercomputing*. ACM, New York, 221–228.

Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. 2013. An implementation of the codelet model. In *Euro-Par*. 633–644.

Sven Verdoolaege. 2010. ISL: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*. Springer, New York, NY, 299–302.

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4, 54.

Ian Watson and John R. Gurd. 1982. A practical data flow computer. *IEEE Computer* 15, 2, 51–57.

Received June 2014; revised November 2014; accepted November 2014