A

OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs

Antoniu Pop, INRIA and École Normale Supérieure
Albert Cohen, INRIA and École Normale Supérieure

We present OpenStream, a data-flow extension of OpenMP to express dynamic dependent tasks. The language supports nested task creation, modular composition, variable and unbounded sets of producers/consumers, and first-class streams. These features, enabled by our original compilation flow, allow translating high-level parallel programming patterns, like dependences arising from StarSs' array regions, or universal low-level primitives like futures. In particular, these dynamic features can be embedded efficiently and naturally into an unmanaged imperative language, avoiding the complexity and overhead of a concurrent garbage collector. We demonstrate the performance advantages of a data-flow execution model compared to more restricted task and barrier models. We also demonstrate the efficiency of our compilation and runtime algorithms for the support of complex dependence patterns arising from StarSs benchmarks.

Categories and Subject Descriptors: D.1.3 [**Concurrent Programming**]: Parallel programming; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures

General Terms: Languages, Design, Performance

Additional Key Words and Phrases: Stream computing, parallel programming, data-flow, code generation

1. INTRODUCTION

As multicores permeate through all consumer electronic devices, the need to provide productivity oriented programming models to exploit these architectures increases. High-level languages are designed to express (in)dependence, communication patterns and locality without reference to any particular hardware. Compilers and runtime systems are left with the responsibility of lowering these abstractions to well-orchestrated threads and memory management. Parallel programming languages based on data-flow principles have strong assets in the race for productivity and scalability:

— The data-flow model of computation guarantees functional determinism [Kahn 1974; Dennis and Misunas 1974].
— High-efficiency imperative languages can be extended to support thread-level data-flow concurrency, maintaining excellent single-thread performance and offering a path for incremental parallelization [Planas et al. 2009].
— Languages based on data-flow principles support the simultaneous exploitation of pipeline, data and task parallelism [Kim et al. 1996; Gordon et al. 2006].

This work is supported by the European Commission through the FP7 projects TERAFLUX id. 249013 and PHARAON id. 288307.
Author's addresses: A. Pop and A. Cohen, Département d'Informatique, École Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France.
Email: `antoniu.pop@inria.fr` and `albert.cohen@inria.fr`

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

— Data-flow execution reduces the severity of the memory wall in two complementary ways: (1) thread-level data flow naturally hides latency; and (2) decoupled producer-consumer pipelines favor on-chip communication, bypassing global memory.
— Enhancing data-flow concurrency with first class, unbounded streams of data also improves expressiveness for a variety of communication and concurrency patterns such as broadcast, delays, and sliding windows [Pop and Cohen 2011].

This paper builds on our previous proposal of a streaming data-flow extension [Pop and Cohen 2011] to the OpenMP3.0 language [OpenMP ARB 2008], presenting key semantic evolutions. We present OpenStream, a more expressive programming model to allow the composition of tasks communicating through first-class data-flow streams, as well as separate compilation. We also provide more general dynamic constructs to support complex data structures and unbounded fan-in and fan-out communications. We show that our model's expressiveness allows to efficiently encode high-level parallel language features such as the memory regions of StarSs [Planas et al. 2009].

Higher expressiveness may improve productivity, but it often comes with performance overheads, impacting the compiler optimizations and increasing the complexity of the necessary runtime support. However, it is also an important asset: general point-to-point synchronization alleviate scheduling constraints of simpler programming models like Cilk. We investigate this behavior on two extreme benchmarks: Fibonacci, where point-to-point and join synchronization are equivalent, is a clear winner for Cilk; and Gauss-Seidel, where barrier synchronized wavefronts are much more restrictive than point-to-point synchronization, where we significantly outperform Cilk.

The paper makes the following contributions:

— OpenStream: a stream-computing extension to OpenMP with vastly enhanced expressiveness. In contrast with our previous work [Pop and Cohen 2011], we introduce strongly typed, first-class streams that may be freely combined with recursive computations and dynamic data structures, while preserving modular (separate) compilation. We also add variadic stream clauses to construct arbitrarily complex, dynamic, possibly nested task graphs, and we provide syntactic support for broadcast operations and for synchronization with futures.
— A new compiler prototype and a new data-flow runtime dedicated to this extension, implemented in GCC 4.7.1 and freely available.
— A compilation technique for translating StarSs compiler directives with array region support to our streaming data-flow model.
— An experimental study of the performance tradeoffs between strict task models with join synchronization (e.g., Cilk), and more general data-flow tasks with point-to-point synchronization.
— Performance evaluation of our tools on a variety of benchmarks, comparing our approach with StarSs and Cilk.

The paper is structured as follows. Section 2 surveys the OpenStream language, highlighting its enhanced expressiveness and new constructs. Section 3 addresses the compilation of StarSs dependent array regions to our model. Section 4 details the code generation and runtime algorithms to map our expressive constructs to a lightweight, feed-forward data-flow execution model. Section 5 conducts an in-depth, comparative performance evaluation of the design choices in our proposal. Section 6 discusses related work, before we conclude in Section 7.

2. OPENSTREAM: A STREAM-PROGRAMMING MODEL FOR OPENMP

OpenStream relies on programmer annotations to specify the data flow between OpenMP tasks and to build the program task graph. Task graphs need be neither regular nor static, unlike the majority of the streaming languages. OpenStream programs allow dynamic connections between tasks, multiple tasks interleaving their communications in the same streams, and arbitrary and variable fan-in, fan-out and communication rates in a dynamically constructed task graph. The language also supports modular composition, separate compilation, and first-class streams (streams as arguments and return values). Despite this expressiveness, the model preserves functional determinism of Kahn networks [Kahn 1974] by enforcing a precise interleaving of data in streams derived from the sequential control flow of the main program.

2.1. Syntax and semantics

The syntactic extension to the OpenMP3.0 language specification consists in two additional clauses for `task` constructs, the `input` and `output` clauses presented on Figure 1. We provide an informal description of the programming model; see [Pop 2011] for a formal, trace-based operational semantics.

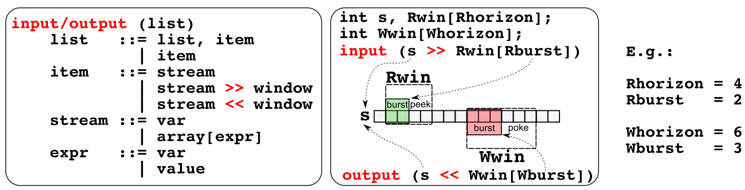

Fig. 1. Syntax for `input` and `output` clauses (left) and illustration of stream access through windows (right).

Both clauses take a list of items, each describing a stream and its behaviour with regard to the task to which the clause applies. If the item notation is in the abbreviated form `stream`, then the stream can only be accessed one element at a time through the same variable stream. In the second form, `stream >> window`, the programmer uses the C++-flavoured << >> stream operators to connect a sliding window to a stream, gaining access to multiple stream elements, within the body of the task.

Tasks compute on streams of values and not on individual values. To the programmer, streams are simple C scalars, transparently expanded into streams by the compiler. An array declaration (in plain C) defines the sliding window accessible within the task and its size, the *horizon*. The connection of a sliding window to a stream in an `input` or `output` clause allows to specify the *burst*, which is the number of elements by which the sliding window is shifted after each activation. In Figure 1 the input window `Rwin` would be shifted by two elements, while the output window `Wwin` would be shifted by three elements. The data-flow case corresponds to *horizon* $=$ *burst*. In the more general case where *horizon* $>$ *burst*, the window elements beyond the burst are accessible to the task; for an output window, the burst and horizon must be equal. Task activation is enabled by the availability, on each input stream, of all horizon elements on the input window, and is driven by the control flow of the main OpenMP program.

The example in Figure 2 illustrates the syntax of the `input` and `output` clauses. Task T1 uses the abbreviated syntax to produce one data element for stream `x`. The semantics of stream operations is to interleave accesses, as illustrated on Figure 3, in task creation order. This order is determined by the flow of control spawning tasks, called *control program*. In our example, T1 introduces a delay in stream `x`. Task T2 is also a producer, adding two elements to stream `x` at each activation. Tasks can be guarded by arbitrary control flow, as is the case for T3, which reads three elements at a time and discards two elements. T4 also reads from `x`, interleaving its accesses to the stream with the accesses from T3. This interleaving is entirely determined by the schedule of the control program, in this case it is a sequence (T4, T3, T4, T4, T3, ...).

```
#pragma omp task output (x)                     // Task T1
  x = ...;
for (i = 0; i < N; ++i) {
  int window_a[2], window_b[3];

  #pragma omp task output (x << window_a[2])    // Task T2
    window_a[0] = ...; window_a[1] = ...;
  if (i % 2) {
    #pragma omp task input (x >> window_b[2])   // Task T3
      use (window_b[0], window_b[1]);
  }
  #pragma omp task input (x)                    // Task T4
    use (x);
}
```

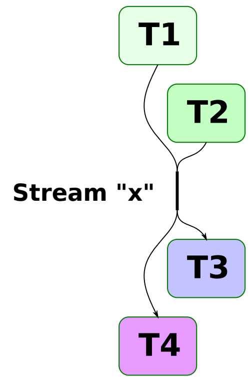

Fig. 2. Example of `input` and `output` clause uses (left) and the resulting task graph (right).

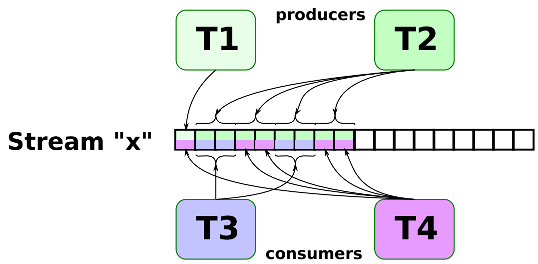

Fig. 3. Interleaving of stream accesses for the tasks on Figure 2.

Broadcast operations. In addition to `input` and `output` clauses, we provide a convenience clause for performing pure peek operations (i.e., when a task reads on a stream without advancing the stream, through a 0-burst access window). The `peek` clause does not introduce any new semantics; it makes broadcast operations explicit.

```
int win[horizon];
// implicit 0-burst
#pragma omp task peek (x >> win)

// advance clock by ‘‘burst’’
#pragma omp tick (x >> burst)
```

$\equiv$

```
int win[horizon];

#pragma omp task input (x >> win[0])

#pragma omp task input (x >> win[burst])
  { ; }
```

Fig. 4. Syntax (left) and semantic equivalent (right) of the `peek` clause and `tick` directive.

In a streaming context, broadcasting can be performed without copying, by allowing multiple tasks to read the same data in a stream without advancing the reading index. The `peek` clause uses the same syntax as the `input` clause, except that the `burst` is disregarded, implicitly understood to be 0. To simplify the complementary operation of discarding elements from a stream, whether already read or not, we add a new `tick` directive. It has similar semantics to a code-less task using an `input` clause on a given stream: it advances the read index in streams, playing a similar role to advancing the logical clock represented by stream access indexes. Figure 4 presents the syntax of `peek` and `tick`, as well as the semantically equivalent code.

Variadic stream clauses. One of the main roles of our streaming annotations is to describe, in a compact way, how the dynamic task graph of an application is built. To generate arbitrary task graphs, it is necessary to allow connecting tasks to dynamically variable numbers of streams. However, this poses a challenge due to the static nature of compiler directives: the number of streaming clauses present on a task's `pragma` directive is inherently static. To specify a variable number of connections, we allow to simultaneously access multiple streams of an array through an array of windows.

```
int stream_array[N] __attribute__((stream));
int window_array[num_streams][num_elements];

#pragma omp task input (stream_array >> window_array[num_streams][num_elements])
  ... = window_array[0..num_streams-1][0..num_elements-1];
```

Fig. 5. Example of variadic `input` clause, connecting a task to `num_streams` simultaneously.

Figure 5 shows an example of an array of stream access windows connected, in a *variadic clause*, to multiple streams from an array of streams. The window `window_array` gives simultaneous access to the first `num_streams` streams in `stream_array`. The number of streams connected must be at most the size of the array.

2.2. Stream typing and modular compilation

We presented the declaration of stream variables as a plain C variable declaration. However, this poses problems for compiling streaming programs where streaming tasks occur in function calls, let alone programs divided in multiple translation units, and it makes type checking very difficult. To enable modular compilation, we need an interface to pass streams as parameters to functions and to store stream references in data structures, making streams first class entities which can be manipulated like any C variable. Consistently with our compiler directive approach to streaming, we add variable and parameter declaration attributes to type streams.

Streams are implicitly separated between a stack-allocated stream reference, which can be freely manipulated by the programmer, and the heap-allocated data structure used and managed by the runtime. In general, the user needs not know about the latter and can simply consider streams to behave like any stack-allocated variable.

```
// Declare a typed stream
int scalar_stream __attribute__((stream));

// Declare a typed array of streams (allocates runtime data on the heap)
int stream_array[size] __attribute__((stream));

// Declare a typed array of stream references (no allocation of runtime data)
int stream_ref_array[size] __attribute__((stream_ref));

// Function taking an array of streams as parameter
void foo (int x[] __attribute__((stream)));

// Call-site for a function taking an array of streams as parameter
foo (stream_array);
```

Fig. 6. Stream variable and parameter declaration.

Figure 6 shows the different types of stream declarations, as scalar variables, arrays of streams and arrays of stream references, and parameters to functions. Both streams and stream references can be manipulated in the same way, but stream references are not initialized and must be set by the programmer with an assignment. They are used for managing collections of streams, in particular for variadic streaming clauses.

The management of stream data structures is generally done entirely by the runtime, which transparently updates a reference counter to ensure timely deallocation. However, some advanced uses of stream references require programmer intervention in the form of runtime calls to increment and decrement the reference counter; specifically when they escape the current scope of the stream variable because the stream is returned by a function or it is stored in a heap allocated data structure. We purposefully chose this explicit approach to resource management to avoid relying on a general-purpose garbage collector. Indeed, a garbage collector would have to rule the whole heap memory. Since first-class streams allow to handle most situations automatically, programmer intervention is seldom necessary; so far, we only used explicit reference counting in complex, compiler generated codes that would be written in simpler ways by a programmer, without requiring explicit reference counting.

```
int temp __attribute__((stream_ref));
int streams[5] __attribute__((stream));

// Swapping streams in an array
temp = streams[1];
streams[1] = streams[3];
streams[3] = temp;
foo (streams, 4);
```

```
void
foo (int x[] __attribute__((stream)),
     int n)
{
  int win[n][horizon];
#pragma omp task input(x >> win[n][burst])
    // ... use win[0..n-1][0..horizon-1]
}
```

Fig. 7. Example of stream reference handling.

Figure 7 presents a simple example of manipulation of stream references, where the streams in the array are re-ordered before passing the array of streams to function `foo`. In this example, the function also takes as second parameter the number of relevant streams in the array and it accesses these streams through a variadic window.

2.3. Nested streaming

To allow recursion with concurrent tasks, and more importantly to enable the parallel execution of the control program, we add support for task nesting, valid for any arbitrary[1] nesting of streaming and non-streaming tasks.

As we target more than just structured nesting graphs, we need to be able to communicate streams to nested tasks, which allows them to further generate tasks accessing these streams. This is possible by passing streams by value to nested tasks, using the `firstprivate` clause. This clause copies the stream reference alone and issues the proper runtime calls to ensure proper management of the stream data structure (reference counting), without any further programmer involvement. To illustrate this, we present on Figure 8 the recursive implementation of Fibonacci, used in Section 5 for our experiments. The right side of the figure shows the main function of the program, which declares a `stream`, passed by copy to a first task that initiates the recursion and on which a second task will read the final `result`. The left side of the figure shows the recursive function taking as parameter a stream on which it writes its result. It further spawns two tasks to generate the remainder of the recursion.

We mentioned above that a restriction on this type of nesting is necessary to preserve determinism. Indeed, the problem comes from the fact that we rely on the total order on read (or, independently, write) accesses to each stream, which derives from the order of generation of tasks performing such accesses, to guarantee the determinism of the schedule of data in streams. If the *control program*, which is the thread of control that reaches a task construct, is not sequential, then concurrency between the generation

[1] With, however, a restriction necessary to preserve determinism, which we discuss below.

```
void stream_fibo (int n, int cutoff, int sout __attribute__((stream))) {
  int x;
  if (n <= cutoff) {
#pragma omp task output (sout << x)
      x = sequential_fibo (n);
    } else {
      int s1 __attribute__ ((stream));
      int s2 __attribute__ ((stream));

#pragma omp task firstprivate (s1)
      stream_fibo (n - 1, cutoff, s1);
#pragma omp task firstprivate (s2)
      stream_fibo (n - 2, cutoff, s2);
#pragma omp task input (s1, s2) \
                 output (sout << x)
      x = s1 + s2;
}}
```

```
// Main:
int stream __attribute__ ((stream));
int numiters = ...;
int cutoff = ...;
int result;

#pragma omp task firstprivate (stream)
  stream_fibo (n, cutoff, stream);

#pragma omp task input (stream >> result)
   printf ("Fibo␣result:␣%d", result);
```

Fig. 8. Recursive, nested implementation of Fibonacci.

of tasks performing the same type of access to the same stream will lead to non-determinism[2]. We must therefore ensure that the order of creation (not execution) of tasks producing to or consuming from each stream is preserved. This can either be achieved by ensuring that all tasks producing (or *independently* consuming) data in a stream are created by a single task, as is the case in the Fibonacci code in Figure 8, or that the order of creation is enforced through dependences in the task graph.

2.4. Implementing futures as streaming tasks

To illustrate the dynamic synchronization and parallel programming patterns captured by OpenStream programs, we show how to express general futures as streaming tasks [Pop and Cohen 2012]. Futures are a form of fine-grained synchronization, used to allow the asynchronous computation of an expression until the result is needed. This example also hints at how we encode complex dependence patterns from higher level programming models, introducing the next section.

We use a special stream access pattern, that decouples the read operations from the advancement of stream read indexes, because the number of times the value stored in the future is requested cannot be known in advance. This pattern consists of `input` clauses with null bursts, `peek` operations, followed by stream-advancing operations, `input` clauses with non-zero burst values or `tick` operations, once a given future no longer needs to be accessed. The programmer, or the compiler, needs to explicitly advance the read index in the stream, with an important side effect: no further reference to the past can be made. As a result, the number of consumers that have acquired a read handle on earlier futures is final, allowing to efficiently recycle memory. Figure 9 (left) shows how a future `x` can be implemented as a stream. The consumers of this future use 0-burst `input` clauses; a final code-less task advances the index in the stream for the next iteration.

The semantics of this kind of code-less task is similar to the `next` operator in the Lucid [Ashcroft and Wadge 1977] language, providing a notion of logical time when consumers do not advance the stream by themselves. Explicit advancement operations may be perceived as cumbersome, but do not reduce the generality of this approach.

[2]The proofs of the necessity and sufficiency of this total order on the generation of tasks performing either type of stream access, to guarantee determinism, can be found in [Pop 2011].

```
int x __attribute__((stream)), win[1];

while (...) {
#pragma omp task output (x)
   x = foo ();

  if (...)
#pragma omp task input (x >> win[0])
     ... = use (window[0]);
  if (...)
#pragma omp task input (x >> win[0])
     ... = use (window[0]);

#pragma omp task input (x >> win[1])
   { ; }
}
```

```
int x __attribute__((stream));

while (...) {
#pragma omp task output (x)
   x = foo ();

  if (...)
#pragma omp task peek (x)
     ... = use (x);
  if (...)
#pragma omp task peek (x)
     ... = use (x);

#pragma omp tick (x)
}
```

Fig. 9. Future implemented with OpenStream tasks (left) and with simplified syntax (right).

```
#pragma omp task [clauses]
  function-definition | function-declaration
```

```
input ([list of parameters])
output ([list of parameters])
inout ([list of parameters])
```

Fig. 10. StarSs task annotation (left) and clauses format (right).

3. HIGH-LEVEL COMPILATION

To highlight the expressiveness of our streaming extension, we propose a translation path for higher-level languages with rich constructs to define inter-task dependences. We selected StarSs [Planas et al. 2009] and its array region mechanism to express such dependences, and we provide a method to automate the translation to OpenStream.

3.1. The StarSs programming model

StarSs [Planas et al. 2009] relies on compiler directives for blocking asynchronous operations into coroutines similar to OpenMP tasks. It provides additional clauses to describe the memory accesses of each task, from which inter-task dependences are inferred. In its latest evolution [Perez et al. 2010], accesses are specified with *dynamic array regions*, providing a lot of flexibility to programmers and an incremental path to parallelize existing programs. The price for this rich, implicit dependence abstraction is paid through the need for a sophisticated runtime algorithm. A *runtime dependence resolver* detects the effective overlaps between the memory accesses of different task instances and the ordering constraints deriving from the task creation order.

We briefly recall the syntax and informal semantics of the StarSs programming model. We also analyze the workings of the array region support it provides. For more information, we encourage the readers to peruse the two detailed references above.

Figure 10 presents the syntax of task directives and their clauses as defined for the OpenMP incarnation of StarSs. The task directive is identical to that of OpenMP and OpenStream; the clauses allow specifying three types of accesses (read, write or read/write) taking a list of parameters that define the memory regions where accesses occur. The parameters of these clauses are of the form `A[lo1:up1][lo2:up2]`, which means that memory accesses occur within the region delimited by the lower and upper bounds (both inclusive) on each dimension of array `A`.

To better understand how the StarSs programming model works, Figure 11 shows an implementation of the Gauss-Seidel kernel. It performs a heat transfer simulation over a rectangular plane, computing a 5-point stencil over a tiled array, `data`. We also provide, in Figure 12, a graphical representation of the regions described by the StarSs

```
for (iter = 0; iter < numiters; iter++)
  for (i = 1; i < N-1; i += B)
    for (j = 1; j < N-1; j += B) {
#pragma omp task inout (data[i:i+B-1][j:j+B-1])        \
     input (data[i-1;1][j:j+B-1], data[i+B;1][j:j+B-1]) \
     input (data[i:i+B-1][j-1;1], data[i:i+B-1][j+B;1])
        {
          for (k = i; k < i + B; ++k)
            for (l = j; l < j + B; ++l)
              data[k][l] = 0.2 * (data[k][l] + data[k-1][l] + data[k+1][l]
                                  + data[k][l-1] + data[k][l+1]);
        }
      }
```

Fig. 11. Gauss-Seidel implementation with StarSs region annotations using tiles of size B.

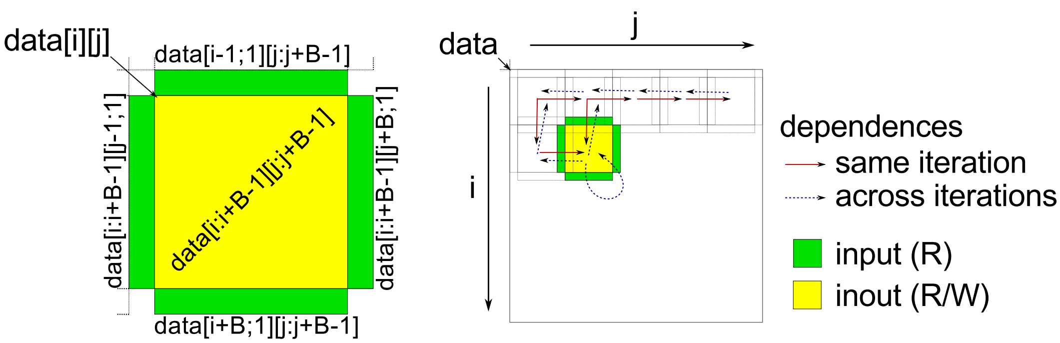

Fig. 12. Data dependences, within one single iteration, for the Gauss-Seidel application in Figure 11.

annotations and the data dependences present in this code. The task annotation, in Figure 11, uses five access regions on array `data`, one in read/write mode, using the `inout` clause, and four in read mode, using the `input` clause. The read/write region corresponds to the body of the tile, represented in yellow on Figure 12, while the read regions correspond to the accesses that overlap neighboring tiles, in green on Figure 12. The syntax used for read regions in this example is slightly different from that presented on Figure 10, the semicolon notation defines a region as a starting index and a length: `data[i-1;1][j:j+B-1]` represents a region one element wide at index `i-1` on one dimension and spanning between `j` and `j+B-1` on the other dimension.

The dynamic dependence resolution uses the declared access regions and evaluates possible overlaps between regions to provide the set of dependences that need to be enforced to preserve the semantics of the program. In order to efficiently compute these overlaps, the dependence resolver relies on a linearized representation of memory regions based on the actual addresses of elements that belong to the region. The representation consists of a number in base 3, where each digit is encoded as 0, 1 or X. The length of the region representation depends on the binary width of the architecture. The intuition here is that an address belongs to a region iff each digit of the address' binary representation is either equal to the corresponding digit in the region's representation or the region's digit is X. The details on the way the region representation is constructed to ensure an efficient detection of collisions, as well as the way regions are organized in a ternary sort tree, can be found in [Perez et al. 2010].

The key property is that the runtime implementation of this dependence resolver provides precise dependence information at the region level. This information enables the compilation-time translation of StarSs directives to our streaming directives.

3.2. Translating StarSs into OpenStream

In this section, our objective is to show that OpenStream constructs can be used to capture the dependences between tasks working on shared data, using the dependence information provided by the StarSs resolver. We will show that such an embedding can be implemented at compilation time, generating the adequate synchronizations with data-flow streaming constructs. Our choice of the translation of StarSs directives, to showcase the expressiveness of our programming language, was primarily motivated by the closeness of StarSs annotations and programming style with OpenMP, which makes this translation easier to understand. However, this process applies more generally to any higher-level language (HLL) for parallel-programming that handles dynamic dependences between tasks.

Importantly, the resolver of dynamic dependences expressed *implicitly* in HLLs is a necessary component, provided by any language where dependences are not specified by programmers. We do not address the design and optimization of such resolvers, but rather we use this particular example to show that task graphs can be built dynamically. This allows expressing the semantics of the dynamic constructs found in HLLs. We show how the *output* commonly available from such dependence resolvers can be used to lower HLL constructs to OpenStream directives.

The key insight behind our translation scheme is that StarSs array regions, or any memory location, can be encoded by a stream as a sequence of versions, enforcing a form of *dynamic single assignment* on each version. To comply with the in-place update policy of StarSs, we restrict the live range of each stream to one single version/element: a single instance of the data is alive at any time in shared memory. For example, the degenerated case where array regions are guaranteed (e.g., by programming language semantics) to always either fully overlap or be disjoint, which means that each region can be assimilated to a scalar w.r.t. its dependences, can be directly handled without additional support from a dependence resolver. Indeed, in our scheme, this case only requires of the resolver to perform an identity function.

The StarSs dependence resolver is marginally modified to attach a stream to each StarSs region and to return two sets of streams for each dynamic task instance T:

— The set of streams attached: (1) to any region that overlaps with the write regions of task T (`output` and `inout`); or (2) to any write region that overlaps with the read regions of task T (`input` and `inout`); or (3) to any of the own access regions of task T. We will call this set `streams_peek(T)`.
— The set of streams attached to the regions of task T, irrespectively of their type. We call this set `streams_out(T)`.

Implicitly, each stream attached to a StarSs region is initialized with an element representing the initial state of the region.

Figure 13 illustrates the translation scheme for the Gauss-Seidel example presented on Figure 11.

Figure 13 presents the result of our translation of the Gauss-Seidel algorithm from StarSs to OpenStream annotations. We first need to invoke the StarSs dependence resolver, passing a set of region descriptors built in the same way as in the StarSs compilation framework, and obtaining in return the two sets of streams `streams_peek` and `streams_out` as defined above, as well as the number of streams in each set. We then replace the original StarSs task annotation with our own task annotation with two clauses: (1) a variadic `peek` clause for all streams in the `streams_peek` array; and (2) a variadic `output` clause for all streams in the `streams_out` array. Finally, we issue a `tick` directive for each stream in `streams_out`.

```
for (iter = 0; iter < numiters; iter++)
  for (i = 1; i < N-1; i += B)
    for (j = 1; j < N-1; j += B) {
        starss_resolve_dependences (region_descriptors, &streams_peek, &streams_out,
                                    &num_streams_peek, &num_streams_out);

#pragma omp task peek (streams_peek >> peek_view[num_streams_peek][0])          \
                 output (streams_out << out_view[num_streams_out][1])
        {
          for (k = i; k < i + B; ++k)
            for (l = j; l < j + B; ++l)
              data[k][l] = 0.2 * (data[k][l] + data[k-1][l] + data[k+1][l]
                                  + data[k][l-1] + data[k][l+1]);
        }

        for (k = 0; k < num_streams_out; ++k) {
#pragma omp tick (streams_out[k] >> 1)
          }
      }
```

Fig. 13. Gauss-Seidel OpenStream code translated from the StarSs implementation in Figure 11.

The semantics of the code we generate is quite natural: the `peek` clauses request reading the current live version of a region, which enforces the order of the current task's execution after the last task that invalidated that region, and the `output` clause generates a new version for each invalidated regions, followed by a `tick` directive to prevent any subsequent tasks from accessing the old version of the region.

3.3. Correctness of the translation

Let us now verify that all dependences are properly enforced in our resulting code. Consider two regions A and B, accessed by two different tasks T_A and T_B created in this order by the control program, such that the two regions overlap: $A \cap B \neq \varnothing$. Relying on the classical definition of dependences [Bernstein 1966], there is a dependence $A \delta B$ if at least one of the accesses is a write, so if either of the regions is `output` or `inout` on its respective task. We distinguish flow-, anti- and output-dependences (respectively read-after-write, write-after-read and write-after-write dependences). As StarSs does not expand shared data into private copies, we enforce all of these dependences.

Let s_A and s_B be the two streams attached to regions A and B. Streams inherently enforce flow dependences between their producers and consumers for each element, so in order for the dependence $A \delta B$ to be properly enforced, it is necessary and sufficient for T_A to be producer of an element, of a stream s, consumed by T_B.

If T_A writes to region A, then it generates a new version on stream s_A and as $A \cap B \neq \varnothing$, we deduce that the dependence resolver must return $S_A \in \texttt{streams\_peek}(T_B)$, irrespectively of the nature of the access to B, and therefore T_B reads that element from the stream s_A, preventing the execution of T_B before T_A completes. This means that both flow- and output-dependences are properly synchronized.

If T_A reads from region A, then we only need to enforce the dependence if T_B writes to B. As by definition $s_A \in \texttt{streams\_out}(T_A)$, T_A produces a new version on stream s_A. Similarly to the previous case, the resolver must return $s_A \in \texttt{streams\_peek}(T_B)$ because T_B writes to B and we deduce that T_B reads the element written by T_A in s_A, therefore synchronizing the anti-dependence.

We can also verify that we do not over-synchronize read-after-read dependences by noting that $s_A \notin \texttt{streams\_peek}(T_B)$, with one notable exception: if $A = B$. Indeed, in that case $s_A = s_B$ and we always have $s_B \in \texttt{streams\_peek}(T_B)$. This exception is necessary to enforce anti-dependences transitively when successive read accesses to the same region discard older versions. Consider, for example, three task instances T_A, T_B and T_C such that all access the same region A, T_A and T_B read from A and T_C writes to A. In this case, enforcing the order T_A happens before T_B allows guaranteeing that T_A happens before T_C, which cannot be guaranteed otherwise as the version created by T_A in s_A is discarded by T_B's new version.

This over-synchronization can be avoided by creating a new region for T_B, that overlaps with A, but with its own stream. However, our performance results show no significant degradation without this optimization.

3.4. Implementation

This translation scheme is much simpler than one would anticipate given the semantic gap between dynamic array regions and data-flow streams. As we show in Section 5, it is also reasonably efficient. We did not automate this scheme, but instead applied it by hand to StarSs applications. Indeed, the StarSs dependence resolver proved quite difficult to extract due to the need to generate region descriptors, and we implemented our own mock-up of an array region dependence resolver for our benchmarks.

4. LOW-LEVEL COMPILATION

We present here an entirely new compilation flow, implemented as a front- and middle-end extension to GCC 4.6.1, expanding streaming task directives into data-flow threads and point-to-point communications. It is the first complete, fully automatic compilation framework for OpenStream. In this section, we briefly present the feed-forward data-flow execution model we are targeting, and discuss some constraints this model imposes on our stream programming model. Then we detail the code generation algorithm and the main features of the implementation.

4.1. Feed-forward data-flow model and interface

Our code generation pass targets an abstract data-flow interface, designed after the DTA data-driven execution model and the T* ISA [Giorgi et al. 2007; Giorgi 2012]. The interface defines two main components: *data-flow threads*, or simply *threads* when clear from the context, together with their associated *data-flow frames*, or *frames*.

The frame of a data-flow thread stores its input values, and may also store local variables or thread metadata. The address of this data-flow frame also serves as an identifier for the thread itself, to synchronize producers with consumers. Communications between threads are single-sided: the producer thread knows the address of the data-flow frames of its dependent, consumer threads. A thread writes its output data directly into the data-flow frames of its consumers.

Each thread is associated with a *synchronization counter* (SC) to track the satisfaction of producer-consumer dependences: upon termination of a thread, the SC of its dependent threads is decremented. A thread may execute as soon as its SC reaches 0, which may be determined immediately when the producer decrements the SC. The initial value of the SC is equal to the amount of data that needs to be externally written to its frame plus the number of consumer threads to which it connects. In our implementation, the producer responsible of the last decrementation on a thread directly schedules the consumer for execution. This token-less driven execution is one of the strengths of this form of point-to-point synchronization.

In contrast, token-based approaches [Arvind and Nikhil 1990; Ottenstein et al. 1990; Najjar et al. 1994] require checking the presence of the necessary tokens on incoming edges. This means that either (1) a scanner must periodically check the schedulability of data-flow threads, or (2) data-flow threads are suspendable. The former poses performance and scalability issues, while the latter requires execution under complex stack systems (e.g., cactus stacks) that may introduce artificial constraints on the schedule. The SC aggregates the information on the present and missing tokens for a thread's execution, allowing producer threads to decide when a given consumer is fireable.

Four primitives manage threads and frames. They are implemented as compiler builtins, recognized as primitive operations of the compiler's intermediate representation. We implement them in software, but they can also be efficiently implemented in hardware [Giorgi et al. 2007; Giorgi 2012].

— `void *df_tcreate(void (*func)(), int sc, int size);` Creates a new data-flow thread and allocates its associated frame. `func` is a pointer to the argument-less function to be executed by the data-flow thread, `sc` is the initial value of the thread's synchronization counter, and `size` is the size of the data-flow frame to be allocated. It returns a pointer to the allocated frame. Once created, a thread cannot be canceled. Collection of thread resources is triggered by the completion of the thread's execution.
— `void df_tdecrease(void *fp, int num);` Marks the thread designated by frame pointer `fp` to be decremented by `num` upon termination of the current thread.
— `void df_tend();` Terminates the current thread and deallocates its frame.
— `void *df_tget_cfp();` Returns the frame pointer of the current thread.

4.2. Programming model restrictions

Due to the underlying data-flow model of execution, and its semantic requirement of writing stream data directly in the consumer's data-flow frame, we need to impose two simple restrictions on our programming model for this compilation path: (1) on a given stream, the horizon of all consumer tasks must be an integer multiple of the bursts of producer tasks (i.e., a producer's output window on a stream cannot be split between multiple consumer input windows); and (2) the burst of a consumer is always either 0 or equal to its horizon (i.e., when a task peeks on a stream, it cannot simultaneously advance the stream's read index).

The purpose of these restrictions is to ensure that any given output window on a stream cannot be split among multiple consumer windows. If a producer's output window must be split dynamically between multiple consumers, then each write access to the output window must be guarded by a conditional expression or made through an indirection. This would prevent us from generating optimized code where the producer writes its outputs directly in the data-flow frame of the consumer.

These constraints can be relaxed, but not without a performance overhead, or extending our target abstract data-flow interface and execution model. Other compilation paths have been explored and evaluated, where these restrictions do not apply [Pop and Cohen 2011; Pop 2011], but we preferred to focus on the complete automation of a compilation flow, supporting all the features of the programming model, even the most dynamic ones, and delivering an efficient execution. Furthermore, these restrictions only bear on some advanced stream-oriented features of the language, like the ability to compute over sliding windows on a stream of data. We plan to support these features, and remove any restrictions, in future work.

4.3. Compiling streaming tasks to data-flow threads

The data-flow compilation path for OpenStream does not rely on streams for communication, but rather as a meeting point for producers and consumers of data. Streams record the production and consumption schedules, matching each producer with its consumer(s). *Before* it can start executing, a producer must acquire the locations, within its consumers' data-flow frames, where it needs to write its output data. While this adds some overhead, it is an essential part of our execution model that provides outweighing benefits, as we and discuss in Section 5. To illustrate the compilation process, we rely on some trivial examples of streaming tasks that exhibit the key characteristics required to explain the important parts of the code generation algorithm.

For each data-flow thread (or task instance), we keep track of the information required for the stream matching scheme and for the synchronization algorithm with a metadata block embedded within the thread's frame. Figure 14 shows an example of two streaming tasks and the two key data structures we use: the *frames* and the *views*. The former hold the metadata and the input data required for executing a data-flow thread, the latter are used to implement stream access windows.

```
int x __attribute__((stream));
#pragma omp task output (x)            // T1
  x = ...;
#pragma omp task input (x) output (y) // T2
  y = foo (x);
```

```
struct view {
  // pointer to the data
  // accessed through the window
  void *data;
  // pointer to owner frame
  // (always the consumer thread)
  frame_p owner;
} view_t, *view_p;
```

```
struct frame {
  int synchronization_counter;
  view_t view_x;
  view_t view_y;
  view_t ...

  void *data_block;
} frame_t, *frame_p;
```

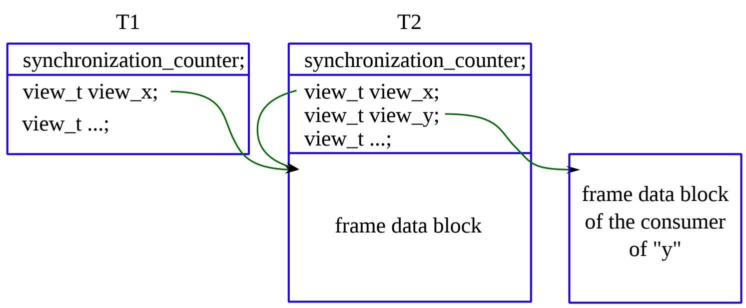

Fig. 14. Data structures used for generating data-flow code: simple example of streaming tasks (top) with the frame and view data structures (middle) and frame layout and chaining (bottom).

The top of Figure 14 shows an example where two tasks communicate through stream `x`. Task `T1` is the producer and `T2` the consumer, both using an implicit window to access the stream. The middle section of the figure shows (on the left) the `view` data structure. It contains a pointer to the actual data that a data-flow thread, which is an instance of a given task, is allowed to access within a conceptual stream through a window. The `view` further contains a pointer to the data-flow frame of the owner of the data, which is *always* the consumer thread. Indeed, for an output window, the view's "data" pointer gives access to a location within another thread's frame, while for an input window, this pointer points within the thread's own frame. On the right, the `frame` data structure shows a skeleton of what a frame might look like. Depending on a thread's inputs and outputs, each frame has a possibly unique structure, but respecting this layout: it always contains the synchronization counter, a set of views corresponding to the different stream access windows the task annotation uses and a `data_block`. The latter is not a pointer to a separately allocated buffer, but is just used as a marker for the beginning (offset) of the data block. The bottom of this figure shows the way the data-flow frames of the threads created for tasks `T1` and `T2` will be chained at runtime, through the `view` metadata. The frame of `T1` contains a view for the (implicit) output window “x”, which points to the data block of its consumer, within the frame of `T2`. The frame of `T2` contains a view for the input window “x”, which points to its own data block, but also a view for the output window “y” which will point to its consumer’s frame data block once it is determined.

```
int x __attribute__((stream)), prod_window[prod_burst], cons_window[cons_burst];
while ( ... ) {
  #pragma omp task output (x >> prod_window[prod_burst]) // Task T1
    prod_window[0..prod_burst-1] = ...;

  #pragma omp task input (x << cons_window[cons_burst]) // Task T2
    ... = cons_window[0..cons_burst-1];
}
```

↓ *Dynamically matching producers to consumers and chaining frames through views* ↓

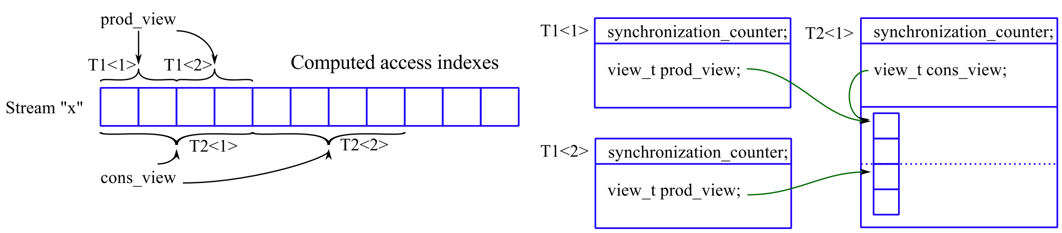

Fig. 15. Matching producers with consumers in a stream for an example of two streaming tasks (top) communicating through stream “x”, conceptually represented as an infinite sequence of indexed data elements and the resulting frame chaining (bottom). This illustrates the `stream_match_views` function in Figure 17.

The following step, presented in Figure 15, shows how the matching of producers and consumers is orchestrated by the runtime, illustrating the work performed by the `stream_match_views` runtime function. We rely here on a slightly more complex set of streaming tasks (top) which communicate through a stream `x` with explicit stream access windows, where the producer and the consumer bursts are non-trivial. We conceptually represent the stream and the stream accesses of both tasks on the bottom part of the figure, instantiating for the sake of illustration[3] with $prod\_burst \leftarrow 2$ and $cons\_burst \leftarrow 4$. As shown on the right side (bottom) of the figure, two instances of the producer task, represented by the two frames `T1<1>` and `T1<2>` are necessary to produce the data for one instance of the consumer task. The stream matching not only sets the `owner` field of each view, but it also computes the appropriate offset in the frame of a consumer to ensure that the producer’s view always points directly to the adequate memory location. For instance, thread `T1<2>`, which is the second instance of the producer task `T1`, produces the second half of the data accessed by the first instance `T2<1>` of the consumer task `T2` through its window.

The code generation itself is illustrated on Figures 16 and 17. First, the work function of a task is generated, as shown on Figure 16. The top of the figure shows a streaming task consuming data on a stream `x` and producing data on an output stream `y`, both accessed through their respective windows. The work function (bottom) consists of the body of the task annotation, outlined to a new function with no arguments. The input parameters are all stored within a thread’s frame, which can be accessed through the frame pointer returned by a call to the `df_tget_cfp` runtime function. Within the

[3]And also keeping in mind that any instantiation must abide by the restrictions described in Section 4.2.

```
int X[x_burst], Y[y_burst];

#pragma omp task input (x >> X[x_burst]) output (y << Y[y_burst])
   foo (X, Y);
```

↓ *Work function code generation* ↓

```
void work_function (void) {
  frame_type *fp = df_tget_cfp ();

  foo (fp->view_X.data, fp->view_Y.data); // Typically inlined work-function

  df_tdecrease (fp->view_Y.owner, y_burst); // Owned by the consumer of stream ‘‘y’’
  df_tend ();
}
```

Fig. 16. Code generation for the work function of a data-flow thread.

body of the original task, each stream access window is replaced with an indirection through the `data` field of the corresponding view. Then, a call to `df_tdecrease` is issued, at the function's exit, for each output frame. This call is used to implement the synchronization algorithm: it atomically decrements the consumer's (owner of the view) frame by a number which represents the amount of data effectively produced and written for this consumer. This call further contains a test that schedules the consumer thread on the ready (work-stealing) queue if the synchronization counter reached zero. Finally, each work function contains a last call to the `df_tend` function to deallocate the frame and perform any necessary cleanup operations.

```
fp = df_tcreate (work_function, x_burst + 1, sizeof (frame) + x_burst);

// input (x >> X[x_burst])
fp->X_view.data = &fp->data_block;
fp->X_view.owner = fp;
stream_match_views (&fp->X_view, x, READ);

// output (y << Y[y_burst])
fp->Y_view.data = NULL; // Unknown for now: data stored within the consumer's frame
fp->Y_view.owner = NULL; // Unknown for now: stream matching will determine this
stream_match_views (&fp->Y_view, y, WRITE);
```

Fig. 17. Code generation for the control program (task creation site).

Finally, the code generation for the control program is presented on Figure 17. It shows, on the same example used on Figure 16 (top), the code generated at the site of the original `pragma` annotation to allocate and prepare the data-flow frame. We first issue a call to `df_tcreate`, which allocates a data-flow frame for one instance of the task, passing a pointer to the work function, the initial synchronization counter and the size of the frame. The initial synchronization counter corresponds to the amount of input data required for a thread's execution, in our example `x_burst`, plus the number of output views of this thread, here `1`. This additional synchronization value is decremented, by the `stream_match_views` function, every time a consumer view is matched to one of this thread's output views. The size of the frame is computed by adding the size of the application data stored in the frame, which is the amount of input data, to the size of the frame's metadata. After this, we generate, for each streaming clause, initialization code for the views created to implement the stream access windows. Finally, we issue, for each stream accessed by the task, a call to `stream_match_views`, which implements the matching algorithm, setting the metadata for output views and finally decrementing the frame's synchronization counter by one for each output view that has been properly matched to a consumer.

4.4. Implementation

A full implementation of the code generation pass used for lowering OpenStream annotations to the data-flow runtime is publicly available,[4] supporting all features of the stream-computing extension presented in this paper. This implementation builds on top of the GCC compiler's OpenMP expansion pass and targets a separate runtime library, which implements stream dynamic matching and the point-to-point synchronization scheme detailed above.

The compiler's front-end is modified to parse streaming annotations, as well as stream attributes, and lower them to GCC's intermediate representation, preserving stream typing information. This typing information is used both to enable modular compilation, with a clean interface between translation units, and to perform type checking providing compile-time feedback when stream types are incompatible.

Frame and view data structures are fully constructed and typed to facilitate debugging. This allows to dump the intermediate representation, using the classical GCC `-fdump-tree-*` flags, in a human-readable format where each structure's field accesses are easily identifiable rather than just an offset. This mitigates part of the drawback of not relying on a source-to-source compiler where the output can be directly checked.

The code generation is integrated in the OpenMP expansion pass in the middle-end, and is activated with the same compilation flag, `-fopenmp`. The generated code does not target GCC's `libGOMP` OpenMP runtime but our own runtime library, `libWStream_DF`.

5. EXPERIMENTAL EVALUATION

We evaluate our data-flow implementation of OpenStream, comparing its performance with Cilk and StarSs implementations on a representative selection of benchmarks. All performance figures reported in this paper have been obtained, using the latest publicly available versions of the respective toolchains: Cilk v5.4.6; Mercurium compiler v1.3.5.8 and libNanox v0.7a for StarSs, targeting a dual-socket AMD Opteron Magny-Cours 6164HE machine with 2×12 cores at 1.7GHz and 16GB of RAM. The performance figures reported here correspond to the average over the third quintile (20% median values) out of a minimum of 30 runs.

We present this evaluation in four parts. First, we evaluate the tradeoffs between Cilk's simple, lightweight tasks with join synchronization, and our more complex data-flow tasks, communicating and synchronizing through streams, but offering more precise, point-to-point synchronization. Second, we compare the performance with StarSs; we demonstrate that the translation scheme presented in Section 3—which converts StarSs programs to OpenStream programs—achieves performance similar to a hand-written streaming version. Third, we use StarSs again to evaluate the performance of a sparse matrix computation with irregular dependences. Finally, we present additional performance comparisons for a classical LAPACK kernel.

5.1. Point-to-point vs. barrier synchronization

One of the main performance benefits of our work lies in the higher expressiveness of streaming annotations, allowing to avoid over-synchronization. In order to evaluate this advantage, as well as the overhead incurred by our approach, we compare against Cilk on two applications, each testing the limits of both approaches.

[4] `http://www.di.ens.fr/StreamingOpenMP`

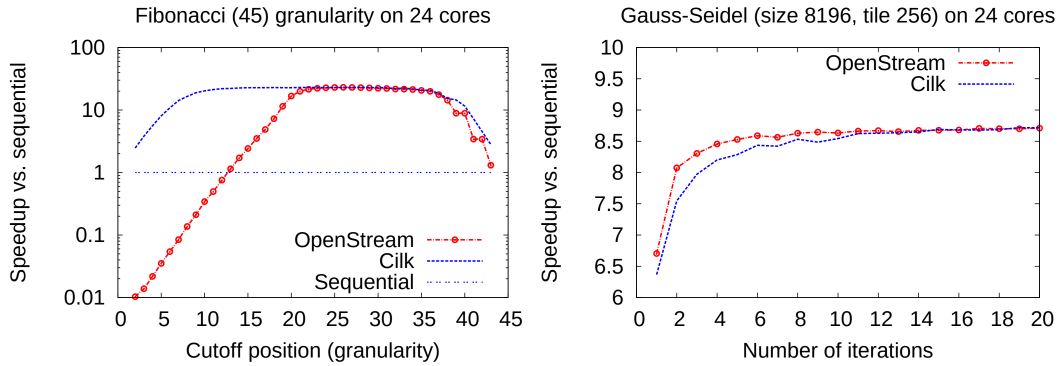

Fig. 18. Performance comparison with Cilk on Fibonacci (left) and Gauss-Seidel (right).

The first application is the recursive Fibonacci computation, implemented with a cutoff to switch to a sequential version once the recursion reaches the intended depth. We use this very simple application to study the overhead associated with our framework and the data-flow execution model. Indeed, the structured parallelism available in Fibonacci is a perfect match for Cilk's join synchronization as only two tasks are synchronized at a time. This removes any advantage of point-to-point synchronization, and as the computation load is negligible for low cutoff points, it exhibits the raw overheads of our implementation and allows to assess the amount of granularity required to amortize such overheads, in a context where the application's needs for memory bandwidth and cache are close to none. The results are presented on the left side of Figure 18, where we compare the execution times of three versions of Fibonacci of 45 varying the cutoff position to increase the granularity of the computation. The key figures on this graph are: (1) for cutoff 2 (equivalent to no cutoff for Fibonacci), the execution time of the OpenStream version is $240\times$ greater than Cilk's, which directly equates to the amount of overhead each task incurs; (2) peak performance is reached for a cutoff of 12 for Cilk and 22 for OpenStream, showing that we need workloads with a granularity roughly three orders of magnitude larger than Cilk in order to effectively amortize the tasks' overhead.

The second application is the Gauss-Seidel kernel presented in Section 3, computing a heat transfer simulation by iteratively computing a 5-point stencil over an array. This application presents distinguishing features that allow testing the other side of the tradeoff between expressiveness and overhead. The dependence pattern of this application, see Figure 12, gives a clear advantage to point-to-point synchronization over barriers: the Cilk version can only exploit parallelism within one wavefront—or hyperplane—at a time, whereas our OpenStream version does not suffer from this over-synchronization and the associated load imbalance. The performance results, on the right side of Figure 18, show the speedups, against sequential execution, for both OpenStream and Cilk versions, when computing on a matrix of 8192×8192 double precision floating point numbers, tiled in 256×256 blocks, and varying the number of iterations of the algorithm. We recall that iterations are not independent, but that some parallelism can be exploited between iterations. We observe that the maximum speedup achieved in this configuration is similar for both versions, reaching up to $8.7\times$ speedup, yet at lower numbers of iterations, where Cilk's wavefronts have less parallelism, our streaming version is more efficient. Cilk reaches the maximal speedup only once the number of iterations is large enough, such that wavefronts contain sufficient parallelism to occupy all 24 cores. This occurs roughly around 10 iterations, which means a maximum of 50 independent tasks per wavefront.

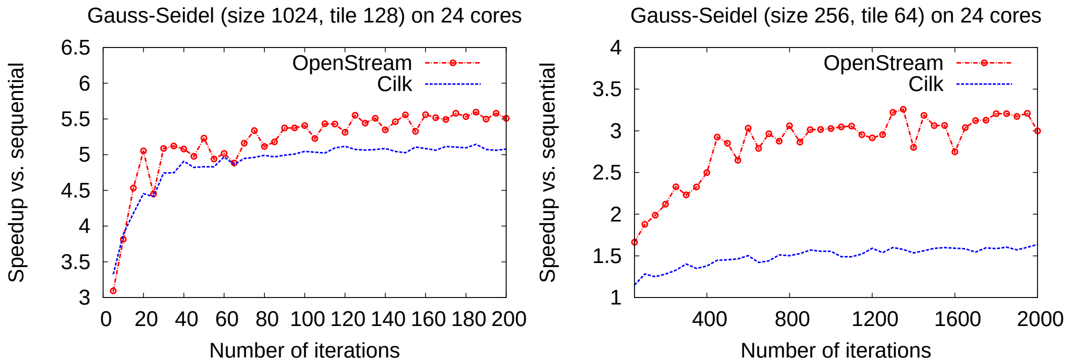

Fig. 19. Speedups obtained on smaller resolutions of Gauss-Seidel, highlighting the advantages of OpenStream point-to-point synchronization.

To complete this analysis, and to stress the performance advantage of point-to-point synchronization, we present additional speedup results for our Gauss-Seidel application on Figure 19. These graphs show the speedups obtained for smaller resolutions of the application, with a matrix of 1024×1024 and a tile size of 128×128 on the left and a smaller yet matrix of size 256×256 tiled by 64×64 on the right. It is important to note that Cilk's wavefronts have limited parallelism which amounts to a maximum of 32 for the graph on the left and 8 for the graph on the right, however these block sizes have been picked because they provide the *highest speedups for the Cilk version*: reducing the block size to provide more parallelism results in higher barrier overhead (more wavefronts and barriers), increasing execution time. Point-to-point synchronization gives our version the edge to reach $3.25\times$ speedup even for the smallest sized matrix.

5.2. Task-level scalability

In this section, we compare the results of OpenStream applications against StarSs implementations for two applications: Gauss-Seidel and Sparse LU, performing a sparse matrix decomposition. We also present the results of our translation technique, introduced in Section 3, for converting StarSs programs into OpenStream programs. In the remainder, we refer to this latter version as "translated".

Our first application is the same Gauss-Seidel algorithm presented above. Figure 20 shows the speedups against sequential execution for three parallel implementations: (1) a hand-coded OpenStream version, which is the same version used in our previous comparison with Cilk; (2) an OpenStream version obtained by translating the StarSs implementation of this algorithm; and (3) a StarSs implementation. The left side of the figure shows the results for a matrix of 8192×8192 double precision floating point numbers, tiled in 1024×1024 blocks, which corresponds to the best parameters for the StarSs version. Our hand-coded version (1) reaches a maximum speedup of $7.6\times$, while the translated version (2) achieves the best performance with a speedup of $7.7\times$, both in front of the StarSs version (3) reaching only a $6.6\times$ speedup.

However, the tile size used for this graph is sub-optimal for our streaming version, which earlier achieved a higher $8.7\times$ speedup for a tile size of 256×256 elements. The amount of parallelism is not optimal if the matrix is split in 8×8 tiles, each having more than enough work to be further divided without worrying about granularity. We present our results for this smaller tile size on the right side of Figure 20. The results of both OpenStream implementations are very similar, with a small difference due to the coding patterns used for the manual implementation, favoring a smaller number of streams, but the StarSs version yields unexpected results.

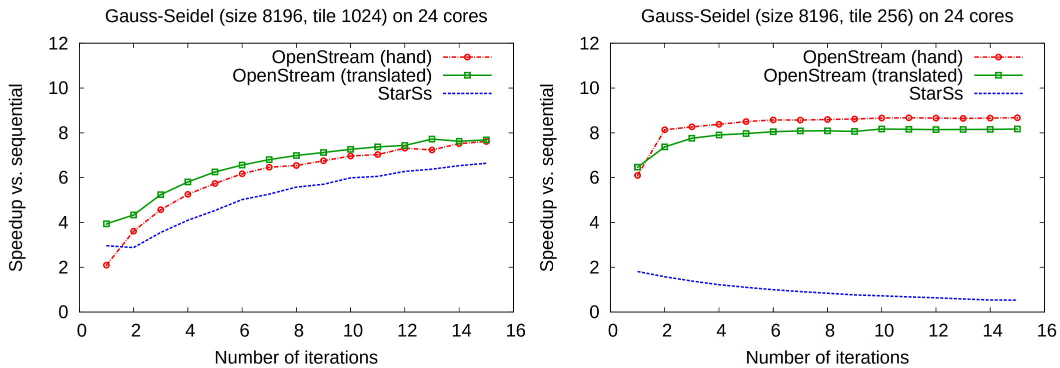

Fig. 20. Gauss-Seidel performance comparison with StarSs and our translation of StarSs to OpenStream. The optimal tile size for StarSs is 1024×1024 (left) and 256×256 for OpenStream (right).

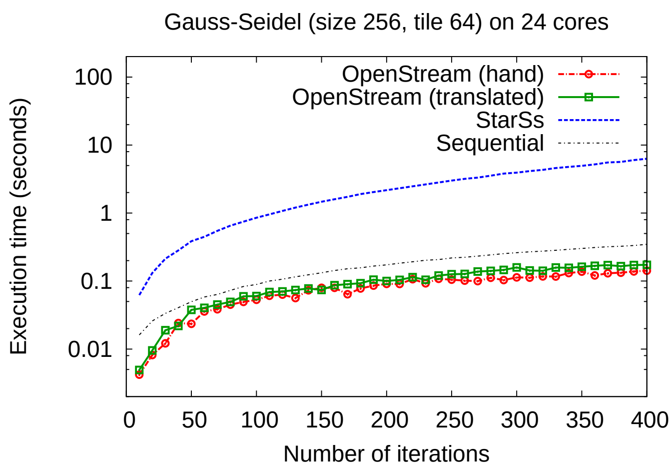

Fig. 21. Task-level scalability of OpenStream and StarSs. For a $B \times B$ tiles matrix and I iterations, the number of tasks id $I * B^2$, here spanning from 640 (10 iterations) to 25600 (400 iterations) tasks.

We were intrigued by this behaviour and, at first, suspected an error in the implementation of the StarSs version, especially due to the need for programmers to pay close attention to data padding and alignment. Indeed, StarSs approximates the matching of array regions (as discussed in Section 3), leading to over-synchronization when the dependence resolver detects false overlappings between regions. Data must be properly aligned and padded, to avoid serialization of the execution, possibly leading to as much as doubling the memory allocation size. However, we determined that this was not the problem and task dependences were correctly inferred. After careful investigation, it appears that the problem is due to the complexity of the algorithm detecting the readiness of tasks. Analysing the execution with *Oprofile* shows that when the number of tasks created by a program increases, this has a superlinear effect on the time spent in the scheduler, scanning for ready tasks.

The StarSs scheduler behaves similarly to a token-based data-flow model: it needs to scan for ready tasks, which becomes very cumbersome as soon as the number of active tasks becomes large. More specifically, the StarSs scheduler has a large shared data structure, the tree of regions, which is used to determine whether tasks are ready to execute. When the number of tasks increases, the time spent looking for ready tasks increases, as well as the contention on this data structure. In contrast,

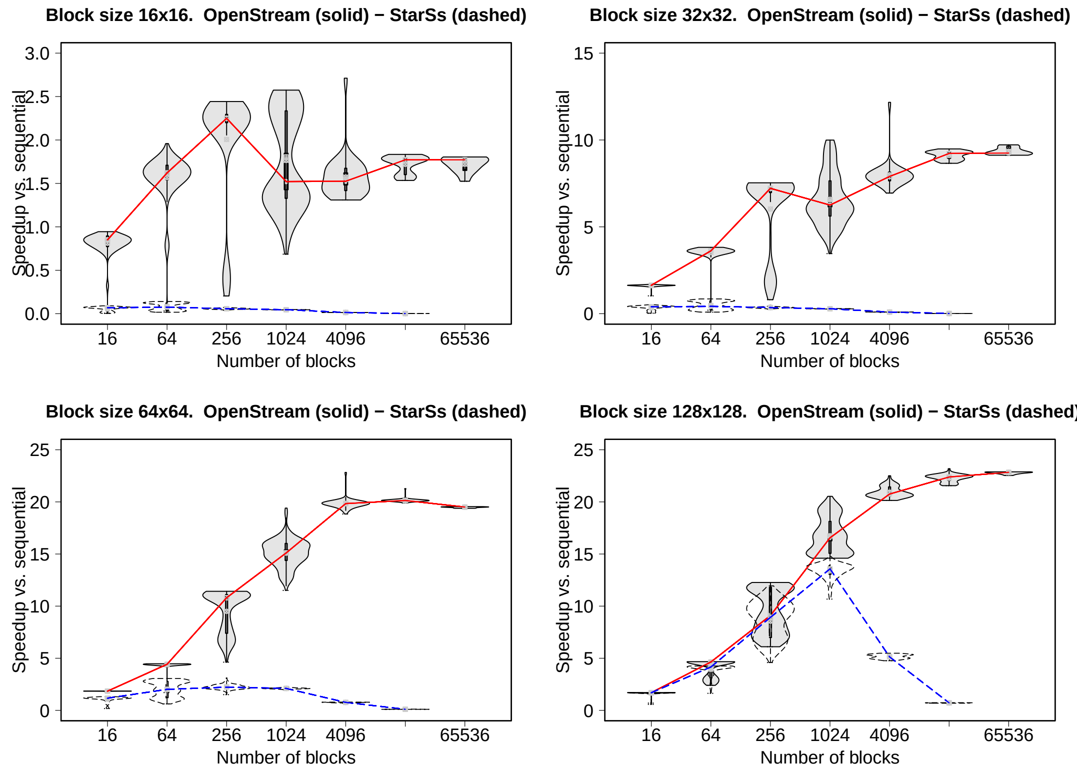

Fig. 22. Sparse LU factorization for 4 different configurations, speedup vs. sequential.

our scheduler uses local ready queues, one per core, which are load-balanced with a work-stealing algorithm [Chase and Lev 2005]. This is possible because we spend the extra time at task creation to find the producer-consumer matchings, which means that producers will know when consumer tasks become ready, without any polling operation or any lookup in a shared data structure. Once a producer determines that one of its consumers has become ready, it adds it to its *local*, work-stealing ready queue.

To give a clearer idea of the impact of the number of tasks on the overhead incurred by the two frameworks, we present in Figure 21 the execution times on a small instance of Gauss-Seidel on a matrix of 256×256 with 64×64 tiles, and varying the number of iterations. The granularity is small enough to show the impact of task-level overhead when increasing the number of tasks. The results show that the OpenStream versions are not impacted by the additional tasks, benefiting from the increased amount of parallelism as the number of iterations increases, whereas the StarSs version sees a degradation of performance as the number of tasks increases, despite the additional parallelism. While at 10 iterations StarSs is only $3.85\times$ slower than the sequential execution, at 400 iterations it is $18.1\times$ slower.

5.3. Evaluation on a sparse matrix computation with irregular dependences

We also report performance results for a block-sparse matrix LU factorization algorithm, which we implemented by hand in OpenStream, and compare to the StarSs implementation. This application is the perfect example for StarSs as it has entirely dynamic dependences which are easy to capture with region annotations. The matrices used are tiled with $1/8$th of the blocks full and the diagonal blocks always full. The performance results, on Figure 22, show speedups against sequential execution for four different block sizes, ranging from 16×16 (top left of the figure) to 128×128 (bottom right). We rely on *violin* plots. The width of the "violins" represent the density of data points for a given value, with a logarithmic bias. The median speedup values of each configuration are connected with a solid red line for OpenStream and a dashed blue line for StarSs. We further represent the 95% confidence intervals for the median as solid black boxes — which are often not wide enough to be visible on our graphs, even for cases where a few singular points induce large extremal variations. Some configurations show wider confidence intervals, due to the presence of two distinct modes, clearly identifiable by the shape of such distributions.

For each granularity, we evaluate speedups on different sizes of matrices, expressed in the number of blocks. Our streaming implementation achieves a median speedup of $2.25\times$ even at the smallest granularity, $9.2\times$ for blocks of size 32×32, $20.1\times$ for size 64×64 and $22.8\times$ for size 128×128. For StarSs, the picture is similar to the Gauss-Seidel application: lower granularities require more blocks to obtain sufficient workload to amortize the overheads, but more blocks mean more tasks, which overloads StarSs' scheduling scheme. However, once block sizes become large, StarSs shows similar speedups as OpenStream for low block counts, reaching a maximum median speedup of $13.6\times$ in a configuration, on the bottom right of the figure, with 1024 blocks of size 128×128. For this same configuration, OpenStream achieves a slightly better $16.6\times$ speedup. We note that the confidence intervals of OpenStream and StarSs are entirely disjoint in this configuration, while they overlap at lower block counts. [Frigo et al. 1998]

5.4. Parallelization of a highly tuned LAPACK kernel

We conclude this experimental evaluation on the Cholesky factorization algorithm, a benchmark with very high computational intensity and temporal locality. Figure 23 shows the speedups obtained when parallelizing Cholesky using OpenStream constructs and calls to the sequential LAPACK implementation within tasks. The figure presents the results for six different input matrix sizes, ranging from 256×256 up to 8192×8192, using different tiling factors, which we express as the number of blocks per matrix to accommodate for multiple input sizes per graph. The graph on the left contains the plots for the smaller sizes of inputs, showing that we break even for matrices of size 256×256, where we get a speedup of $1.08\times$ for 16 blocks of size 64×64. Our best speedup, reaching a superlinear $27.4\times$ for a matrix of size 4096×4096, is due to cache effects: the parallel version computes on 256 blocks of size 256×256 each, precisely fitting within each core's private 512K L2 cache. Similarly, the best speedup for matrix size 2048×2048 is achieved for 64 blocks of size 256×256 and for matrix size 4096×4096, it occurs at 1024 blocks of the same size.

6. RELATED WORK

The principal motivation for research into data-flow models comes from the limitations of the von Neumann architecture to exploit massive amounts of fine grained parallelism. The early data-flow architectures [Dennis and Misunas 1974; Davis 1978; Watson and Gurd 1982] avoid the von Neumann bottlenecks by only relying on local memory and replacing the global program counter by a purely data-driven execution model, executing instructions as soon as their operands become available. More recent data-flow architectures rely on the same principles, albeit at a coarser grain, executing sequences of instructions, or data-flow threads, instead of single instructions.

While many data-flow programming languages have been proposed [Johnston et al. 2004], no one achieves the level of expressiveness and dynamicity combined with

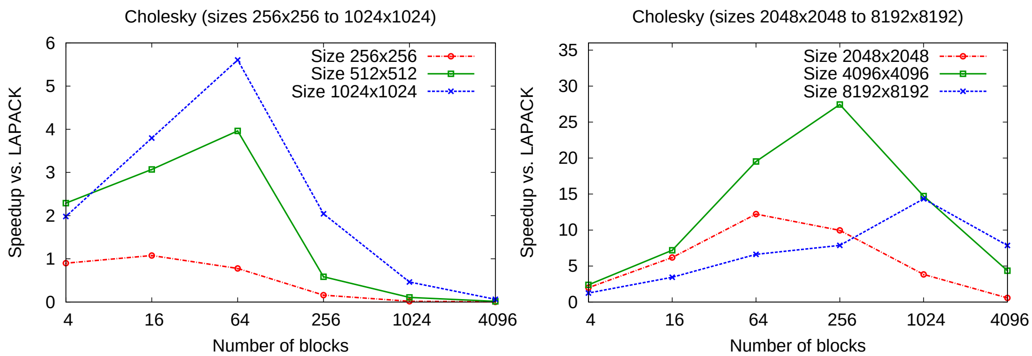

Fig. 23. Cholesky factorization speedups of OpenStream vs. LAPACK.

the efficiency of our proposal. Based on CellSs [Bellens et al. 2006], StarSs [Planas et al. 2009] defines a complete set of pragmas to program distributed-memory and heterogeneous architectures; it supports both data-flow and control-flow programming styles. SMPSs is one of the StarSs incarnations for shared-memory targets [Marjanovic et al. 2010]. TFlux follows a more data-flow centric approach [Stavrou et al. 2008], focusing on pipeline and data parallelism in nested loops and targeting the Data-Driven Multithreading (DDM) execution model [Kyriacou et al. 2006]. Closest to our approach, Swan [Vandierendonck et al. 2011] is an extension of Cilk that complements strict task parallelism with region-induced task-to-task dependences. It only considers situation where dependences are tracked between the children of a single parent procedure. It supports nested task creation, like our proposal, and unlike StarSs.

Both Habanero Java (HJ) [Cavé et al. 2011] and CnC [Budimlìc et al. 2010] provide constructs enabling data-flow computations. HJ supports dynamic task creation and relies on phasers, which generalize join and point-to-point synchronization, and on *data driven futures* to enable data-flow interactions in the same way as I-structures. CnC is a declarative coordination language for deterministic parallel programming. The program is structured in functional blocks, or steps, written in a sequential programming language. Steps communicate through tagged collections of data items (generalizing I-structures beyond integer indexes). CnC express static task graphs only, the precise communication patterns being explicitly constructed by a serial control program. Compared to CnC and HJ, we focus on high-efficiency parallel computations and our experiments use reference baselines (such as LAPACK), making direct performance comparisons irrelevant. We also exploit a wider range of granularity in parallel computations,i.e., taking advantage of finer grain parallelism when available.

The family of stream-computing languages is closely related with data-flow principles, as we illustrated in our earlier work, enhancing data-flow concurrency with unbounded streams of data [Pop and Cohen 2011]. First-class streams of data improve expressiveness for a variety of communication and concurrency patterns such as broadcast, delays, and sliding windows. This was observed by data-flow computing pioneers, who designed I-structures as unbounded streams of futures to alleviate some of the overheads of a pure data-flow execution model [Arvind et al. 1989]. Recently, we showed that dynamic control flow and futures could be expressed as streaming data-flow programs [Pop and Cohen 2012]. In this paper, we generalize this result to arbitrary inter-procedural control and data flow, separate compilation, and to richer dependence patterns such as StarSs dynamic array regions.

The StreamIt language [StreamIt] is another stream-computing language, representative of the cyclo-static data flow model of computation [Lee and Messerschmitt 1987; Bilsen et al. 1995]. Our language proposal is much more expressive than

StreamIt and any CSDF-based programming model; in particular, it does not have any of StreamIt's static and periodic restrictions. StreamIt leverages these restrictions to enable aggressive compilation-time optimizations, targeting a variety of shared and distributed memory targets [Gordon et al. 2006]. But in most cases, these restrictions are not necessary to achieve excellent performance, assuming the programmer is willing to spend a minimal effort to balance the computations and tune the number of threads to dedicate to each task manually. This is the pragmatic approach OpenMP has successfully taken for years. On top of this, the performance portability of StreamIt may still be achieved, relying on complementary static analysis to identify regular parts of the task graph automatically.

7. CONCLUSION

We presented OpenStream, a data-flow extension of OpenMP and the associated compilation and runtime algorithms. We designed new code generation algorithms to support high-level dependence patterns and to provide a path for the modular compilation of dynamic data-flow programs to a very efficient execution model. Our experimental results confirm the excellent behavior of data-flow execution w.r.t. the latency and bandwidth walls. Despite the slight overhead of general data-flow threads over simpler, strict task models, the aforementioned benefits and the load balancing advantages largely dominate. Our language and tool flow provide enhanced expressiveness in an unmanaged imperative language, while retaining all the expected efficiency and scalability benefits.

These results open many questions regarding the intrinsic costs of data-flow, point-to-point synchronization, pushing for an investigation of hardware mechanisms. The availability of a robust compilation flow also opens the door to the study of larger, more complex applications, where the expressiveness benefits should shine. Finally, numerous compiler optimizations are yet to be designed and implemented, aiming to completely eliminate the abstraction penalty of all but the most general communication patterns captured by our data-flow streaming extension.

Acknowledgments. We would like to thank Rosa M. Badia and Guillermo Miranda, from the Barcelona Supercomputing Center, for helping us improve the comparison with StarSs, and Feng Li, from INRIA, for contributing to the implementation of our data-flow runtime. This work was partly supported by the European FP7 projects PHARAON id. 288307 and TERAFLUX id. 249013.

References

ARVIND, NIKHIL, R. S., AND PINGALI, K. 1989. I-structures: Data structures for parallel computing. *ACM Trans. on Programming Languages and Systems 11,* 4, 598–632.

ARVIND, K. AND NIKHIL, R. S. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Computers 39*, 300–318.

ASHCROFT, E. A. AND WADGE, W. W. 1977. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM 20,* 7, 519–526.

BELLENS, P., PÉREZ, J. M., BADIA, R. M., AND LABARTA, J. 2006. CellSs: a programming model for the Cell BE architecture. In *SC*.

BERNSTEIN, A. 1966. Program analysis for parallel processing. *IEEE Transactions on Computers 15*, 757–762.

BILSEN, G., ENGELS, M., R., L., AND PEPERSTRAETE, J. A. 1995. Cyclo-static data flow. In *Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP'95)*. Detroit, Michigan, 3255–3258.

BUDIMLIĆ, Z., BURKE, M., CAVÉ, V., KNOBE, K., LOWNEY, G., NEWTON, R., PALSBERG, J., PEIXOTTO, D., SARKAR, V., SCHLIMBACH, F., AND TAŞIRLAR, S. 2010. Concurrent collections. *Sci. Program. 18*, 203217.

CAVÉ, V., ZHAO, J., SHIRAKO, J., AND SARKAR, V. 2011. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. ACM, New York, NY, USA, 51–61.

CHASE, D. AND LEV, Y. 2005. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. SPAA '05. ACM, New York, NY, USA, 21–28.

DAVIS, A. L. 1978. The architecture and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the 5th annual symposium on Computer architecture*. ISCA '78. ACM, New York, NY, USA, 210–215.

DENNIS, J. B. AND MISUNAS, D. 1974. A preliminary architecture for a basic data flow processor. In *ISCA*, W. K. King and O. N. Garcia, Eds. ACM, 126–132.

FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. 1998. The implementation of the Cilk-5 multithreaded language. In *ACM Symp. on Programming Language Design and Implementation (PLDI'98)*. Montreal, Quebec, 212–223.

GIORGI, R. 2012. TERAFLUX: exploiting dataflow parallelism in teradevices. In *Proceedings of the 9th conference on Computing Frontiers*. CF '12. ACM, New York, NY, USA, 303–304.

GIORGI, R., POPOVIC, Z., AND PUZOVIC, N. 2007. DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems. In *SBAC-PAD*. 263–270.

GORDON, M., THIES, W., AND AMARASINGHE, S. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA.

JOHNSTON, W. M., HANNA, J. R. P., AND MILLAR, R. J. 2004. Advances in dataflow programming languages. *ACM Comput. Surv. 36*, 1–34.

KAHN, G. 1974. The semantics of a simple language for parallel programming. In *Information processing*, J. L. Rosenfeld, Ed. North Holland, Amsterdam, Stockholm, Sweden, 471–475.

KIM, C., GAUDIOT, J.-L., AND PROSKUROWSKI, W. 1996. Parallel computing with the sisal applicative language: Programmability and performance issues. *Software, Practice and Experience 26,* 9, 1025–1051.

KYRIACOU, C., EVRIPIDOU, P., AND TRANCOSO, P. 2006. Data-driven multithreading using conventional microprocessors. *IEEE Trans. on Parallel Distributed Systems 17,* 10, 1176–1188.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers 36,* 1, 24–25.

MARJANOVIC, V., LABARTA, J., AYGUADÉ, E., AND VALERO, M. 2010. Effective communication and computation overlap with hybrid MPI/SMPSs. In *PPOPP*.

NAJJAR, W., ROH, L., AND WIM BHM, A. 1994. An evaluation of medium-grain dataflow code. *Intl. J. of Parallel Programming 22*, 209–242. 10.1007/BF02577733.

OPENMP ARB. 2008. OpenMP Application Program Interface. `http://www.openmp.org/mp-documents/spec30.pdf`.

OTTENSTEIN, K. J., BALLANCE, R. A., AND MACCABE, A. B. 1990. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. of the ACM SIGPLAN 1990 Conf. on Programming Language Design and Implementation*. PLDI '90. ACM, New York, NY, USA, 257–271.

PEREZ, J. M., BADIA, R. M., AND LABARTA, J. 2010. Handling task dependencies under strided and aliased references. In *Proceedings of the 24th ACM International Conference on Supercomputing*. ICS '10. ACM, New York, NY, USA, 263–274.

PLANAS, J., BADIA, R. M., AYGUADÉ, E., AND LABARTA, J. 2009. Hierarchical Task-Based Programming With StarSs. *Intl. J. on High Performance Computing Architecture 23,* 3, 284–299.

POP, A. 2011. Leveraging Streaming for Deterministic Parallelization: an Integrated Language, Compiler and Runtime Approach. Ph.D. thesis, MINES ParisTech.

POP, A. AND COHEN, A. 2011. A stream-computing extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. HiPEAC '11. ACM, New York, NY, USA, 5–14.

POP, A. AND COHEN, A. 2012. Work-streaming compilation of futures. In *5th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'12, associated with ETAPS)*.

STAVROU, K., NIKOLAIDES, M., PAVLOU, D., ARANDI, S., EVRIPIDOU, P., AND TRANCOSO, P. 2008. Tflux: A portable platform for data-driven multithreading on commodity multicore systems. In *Intl. Conf. on Parallel Processing (ICPP'08)*. Portland, Oregon, 25–34.

StreamIt. The StreamIt language. http://www.cag.lcs.mit.edu/streamit/.

VANDIERENDONCK, H., TZENAKIS, G., AND NIKOLOPOULOS, D. S. 2011. A unified scheduler for recursive and task dataflow parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. PACT '11. IEEE Computer Society, Washington, DC, USA, 1–11.

WATSON, I. AND GURD, J. R. 1982. A practical data flow computer. *IEEE Computer 15,* 2, 51–57.