A Stream-Computing Extension to OpenMP

Antoniu Pop[1] and Albert Cohen[2]

[1] Centre de Recherche en Informatique, MINES ParisTech, France

[2] INRIA Saclay and LRI, Paris-Sud 11 University, France

Abstract

This paper introduces an extension to OpenMP3.0 enabling stream programming with minimal, incremental additions that seamlessly integrate into the current specification. The stream programming model decomposes programs into tasks and explicits the flow of data among them, thus exposing data, task and pipeline parallelism. It helps the programmers to express concurrency and data locality properties, avoiding non-portable low-level code and early optimizations. We survey the diverse motivations and constraints converging towards the design of our simple yet powerful language extension, and we present experimental results of a prototype implementation in a public branch of GCC 4.5.

1 Introduction

The performance of single-threaded applications is expected to stagnate with new generations of processors. Increasing performance requires changing the code structure to harness complex parallel hardware and memory hierarchies. But this is a nightmare for programmers: translating more processing units into effective performance gains involves a never-ending combination of target-specific optimizations. These manual optimizations involve subtle concurrency concepts and non-deterministic algorithms, as well as complex transformations to enhance memory locality. Optimizing compilers and runtime libraries no longer shield programmers from the complexity of processor architectures, and the gap to be filled by programmers increases with every processor generation.

High-level languages are designed to express (in)dependence, communication patterns and locality without reference to any particular hardware, leaving compilers and runtime systems with the responsibility of lowering these abstractions to well-orchestrated threads and memory management. In particular, stream programming has recently attracted a lot of attention. It is a widely applicable form of parallel programming that guarantees functional determinism, a major asset in the productivity race. It is also conducive to making relevant data-flow explicit and to structuring programs in ways that allow simultaneously exploiting pipeline, data and task parallelism. Stream computations also help reduce the severity of the memory wall in two complementary ways: (1) decoupled producer/consumer pipelines naturally hide memory latency; and (2) they favor local, on-chip communications, bypassing global memory. While many languages have been designed to exploit pipeline parallelism, we believe it is more interesting and sufficient to introduce minimal and incremental additions to an existing and well-established language.

This paper introduces an extension to the OpenMP3.0 language [20] to enable stream programming. It requires only minor additions that seamlessly integrate in the current language specification. We detail the motivation of our work and present the extension as well as the experimental results of an early implementation in a public branch of GCC 4.5.

The rest of the paper is structured as follows. Section 2 discusses our motivation and the related work. Section 3 details the considerations that have driven the design of our streaming extension. Section 4 presents the extension to OpenMP3.0 we propose. Section 5 defines the semantics of the new constructs and validates the execution model. Section 6 evaluates our implementation on realistic applications and benchmarks. We conclude by summarizing our contributions and discussing work in progress and future work in Section 7.

2 Motivation and related work

Many languages and libraries are being developed for the stream-computing model. Some are general purpose programming languages that hide the underlying architecture's specificities, while others are specifically targeted at accelerator architectures. While a complete survey is outside the scope of this paper, we present a selection of the most related efforts in this field. We also discuss the motivations and constraints that drive our proposal.

Data-parallel execution puts a high pressure on the memory bandwidth of multi-core processors. There is a well known tradeoff between synchronization grain and private memory footprint, as illustrated by performance models for bulk-synchronous data parallel programs [3]. But there are few answers to the limitations of the offchip memory bandwidth of modern processors. Pipeline parallelism provides a more scalable alternative to communication through main memory, as the communication buffers can be tailored to sit in the caches, effectively making cores communicate through a shared cache or through the cache coherence protocol. In addition, the choice of developing streaming languages for accelerator hardware [14] is partly due to the prohibitive latency of accessing the main memory.

Furthermore, a stream-programming model naturally exposes data, task and pipeline parallelism through its high-

level semantics, avoiding the loss in expressiveness of other parallel-programming models. A stream computation is divided in pipeline stages, or filters, where the producer-consumer relationships are explicit. Stages can be either sequential, if there is a dependence between successive executions of the stage, or parallel, in which case, the stage can be replicated at will for load balancing and/or exploiting data parallelism. The sequential filters are an issue that is shared with other forms of parallelism as it stems from the presence of state in the filter, or equivalently from a loop-carried dependence. Closely related to pipelining, doacross parallelization [6], can be used in such cases, but it is more restrictive and requires communication of the state between threads.

Because of this problem, there are applications where the use of pipelining is the only efficient solution for parallelization. As an example, the recent study [15] by Pankratius et al. of the parallelization of Bzip2 shows that this application is not only hard to parallelize, but more specifically that only pipelining allows it to be efficient and to achieve decent scalability levels. The authors of the study remark that OpenMP is not well suited for parallelizing Bzip2, but this was reversed by implementing FIFO queues to communicate between tasks, making it one of the best choices. In this paper, we want to show that it is neither necessary to develop a new language for streaming, nor to require developers to write the pipelining code by hand.

The StreamIt language [19] is an explicitly parallel programming language rooted in the Synchronous Data Flow (SDF) model of computation [12]. StreamIt provides three interconnection modes: the Pipeline allows the connection of several tasks in a straight line, the SplitJoin allows for nesting data parallelism by dividing the output of a task in multiple streams, then merging the results in a single output stream, and the FeedbackLoop allows the creation of streams from consumers back to producers. The channels connecting tasks are implemented either as circular buffers, or as message passing for small amounts of control information. Thanks to these static restrictions (periodicity, structure), a single StreamIt source can be compiled very efficiently on a variety of shared and distributed memory targets [9]. But we believe these expressiveness 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. In addition, when the compiler discovers that the SDF invariants are satisfied, it may trigger aggressive loop transformations to adapt the task parallelism to the target, matching the performance portability of StreamIt.

The Brook language [4] provides language extensions to C with Single Program Multiple Data (SPMD) operations on streams. Unlike StreamIt, it is control-centric, with control flow operations taking place at synchronization points. Streams are defined as collections of data that can be processed in parallel. For example: "`float s<100>;`" is a stream of 100 independent floats. User defined functions that operate on streams are called kernels. The user defines input and output streams for the kernels that can execute in parallel by reading and writing to separate locations in the stream. Brook kernels are blocking and isolated: the execution of a kernel must complete before the next kernel can execute. This is the same execution model that is available on graphics processing units (GPUs): a task queue contains the sequence of shader programs to be applied on the texture buffers.

Dynamic data-flow principles [21, 7, 1] have regained popularity as pragma-based extensions to imperative languages. Based on CellSs [2], StarSs [16] 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 [13]. TFlux follows a similar approach [18], focusing on data flow and targeting the Data-Driven Multithreading (DDM) execution model [11]. StarSs and TFlux are closely related to streaming languages, but they differ from our approach in two fundamental aspects:

- their design and implementation assume a short-lived task execution model, relying on data-driven scheduling of lightweight user-level threads and work stealing; this is excellent for load-balancing, but induces significant overheads for finer grain tasks, as our experiments confirm;
- they are not compatible with OpenMP, but introduce other pragmas with different semantics; StarSs handles distributed memory and heterogeneous targets, unlike OpenMP, but is not as expressive on shared-memory targets; TFlux is currently restricted to nested loops.

Closest to our work is the Streaming Programming Model of the ACOTES project [5]. This model takes its inspiration from the OpenMP3.0 tasks, but is not compatible with OpenMP. It adds decoupled communication channels and pioneered the *persistent interpretation of tasks*, among other contributions. Our proposal derives from this experimental platform, but it is more expressive, it achieves a complete and incremental integration within OpenMP, and it enables more efficient compilation methods.

3 Design goals of the extension

Our primary design goal is to enable OpenMP programmers to exploit pipeline parallelism without explicitly having to handle communication and synchronization, which is both error-prone and time-consuming. We also want to offer highly efficient decoupled pipelined executions to programmers with no experience in shared-memory concurrency. To achieve these goals, we propose extensions to the OpenMP language, exposing the necessary information for the generation of pipelined parallel code, while ensuring this additional expressiveness does not introduce excessive complexity and does not break the semantics of the current specification.

More specifically, we deem the three objectives of expressiveness, efficiency and simplicity to be the most important

to enable stream programming in OpenMP. In Section 4, we will show how to achieve these goals.

Expressiveness. The extension aims to provide a way for programmers to expose producer-consumer relationships. The current OpenMP specification lacks the capability to explicit the flow of data, as the existing sharing clauses only allow to distinguish between `shared` and `private` data. In order to use `task` constructs in non-embarrassingly parallel problems, manual synchronization is required to communicate through shared memory.

The convenience of "peek" operations and non-unitary production and consumption rates is often provided in streaming frameworks. The manual implementation is cumbersome enough to deserve a simplified mechanism at the language level, which in addition allows the compiler to generate in-place operations in stream buffers, avoiding the overhead of copying from streams to local buffers and back.

Productivity. One of the drawbacks of new stream-programming languages is that they come with a whole new programming environment, which lacks debugging support, interoperability and mature accompanying libraries. To this startup cost, one should often add the cost of target-specific tweaking required by the lower level languages (e.g., hard-wired offloading directives for accelerators). By extending a well-known parallel-programming language, OpenMP, with incremental, natural and target-independent constructs for streaming, the programmer's productivity is maximized. We believe a seamless integration with the current OpenMP specification is essential to avoid making the extension an additional burden on current developers. They should not change the way they are used to work with OpenMP for non-streaming applications. The semantics of the extension should therefore be incremental and any new interaction should, as far as possible, follow the existing rules. This practical constraint turns into a research challenge: building compositional data-flow constructs over an existing imperative, shared-memory semantics.

Efficiency. The execution model of OpenMP3.0 specification tasks is similar to that of coroutines or fibers. A task instance is created whenever the execution flow of a thread encounters a task construct, but the execution of the newly created task is contingent on the cooperative scheduling policy. No ordering of the execution of tasks can be assumed. Such an execution model is well suited for unbalanced loads, but the overhead of creating and scheduling tasks is significantly more expensive than synchronizing persistent tasks.[1] Indeed, when the load can be properly balanced, there is a strong case for relying on persistent tasks rather than on data-driven scheduling of short-lived tasks.

To sustain this claim, we compare the cost of our optimized synchronization algorithm to the cost of scheduling lightweight tasks on a synthetic benchmark, called `exploration`, consiting of a sequential producer task generating values and a consumer. In the persistent task version, the producer pushes the values in a stream, by groups of `burst` values at a time, and the consumer is a single task reading from that stream and synchronizing with the producer for every block of `burst` values. In the Cilk version the producer spawns a new task to process each block of `burst` values. The `burst` parameter allows to study the parallelization overhead as a function of the synchronization grain. The results, on an Intel Core2 Quad Q9550 with 4 cores at 2.83GHz, are presented on Figure 1. While the benefits of scheduling lightweight tasks for load-balancing are undeniable, the higher overhead of scheduling requires a significantly higher task granularity to amortize. In order to evaluate the granularity required to break even between persistent and short-lived tasks, we compare, on Figure 1, the execution time on the `exploration` synthetic benchmark. On one side we use persistent tasks, while on the other we have a Cilk implementation spawning short-lived user-level tasks [8]. Cilk is run with the `--nproc 4` option to generate parallel code, and with the `--nproc 1` option to specialize the code for sequential execution. The sequential Cilk version takes almost 7 s for the finest synchronization grain, and 5 s for larger ones. The parallel Cilk version with the finest synchronization takes $221.4\,s$ and the corresponding persistent task version takes $107.7\,s$. The performance gap widens significantly for bursts of intermediate size, and approaches $5\times$ when the persistent task version reaches its performance plateau. The most important figure, in practice, is that the persistent tasks break even for grain size $80\times$ smaller than Cilk. This demonstrates the need for data-flow interactions among long-lived, persistent tasks as an essential abstraction for scalable concurrency.

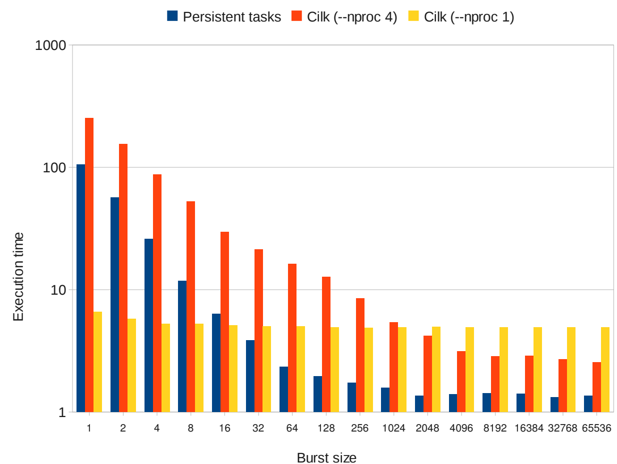


Figure 1: Exploration: persistent vs. short-lived user-level tasks.

4 Proposed OpenMP streaming extension

This section introduces the syntactic constructs we need to meet the design goals outlined in Section 3. We use a C syn-

[1] Except when the target architecture offers some support for very lightweight thread scheduling [11].

tax although everything can easily be transposed to Fortran.

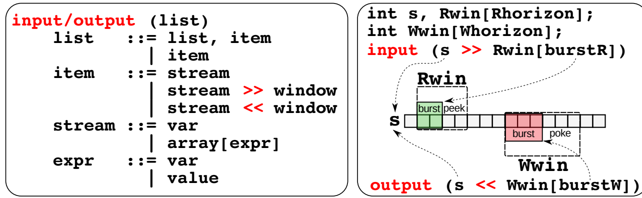


Figure 2: Syntax for `input` and `output` clauses.

Language extension. We propose to extend OpenMP3.0 with two additional clauses for `task` constructs, the `input` and `output` clauses presented on Figure 2. Both clauses take a list of items, each of which describes a stream and its behaviour w.r.t. the task to which the clause applies. In the abbreviated item form, `stream`, the stream can only be accessed one element at a time through the same variable `s`. In the second form, `stream >> window`, the programmer uses the C++ flavoured `<< >>` stream operators to connect a *sliding window* to a stream, gaining access, within the body of the task, to `horizon` elements in the stream.

Our programming model is more general than *scalar* data-flow: tasks compute *streams of values* and not individual values. To the programmer, streams are plain C variables, transparently expanded into streams by the compiler. An array declaration (in C) defines the sliding window accessible within the task and its size, the *horizon*. Connecting a sliding window to a stream in an `input` or `output` clause allows to indicate the *burst*, which is the number of elements by which the sliding window is shifted after each activation. In Figure 2 the input window `Rwin` is shifted by two elements, while the output window `Wwin` is shifted by three elements. Scalar data-flow corresponds to `horizon` = `burstR` (resp. `burstW`). In the more general case where `horizon` > `burstR` (resp. `burstW`), the window elements beyond the burst are accessible to the task; for an output window, the values of these elements will only be committed and made accessible to consumers in subsequent activations. Task activation is driven by the availability, on each input stream, of *all* `horizon` elements on the input window.

The examples on Figure 3 illustrate the syntax of the `input` and `output` clauses. In the first example on the left, the task reads from the stream `x`. It reads up to `horizon` values of `x` ahead of the current position in the stream, and consumes `burst` elements at each activation. In the second example on the left, the task reads from the stream `A[0]`, the first element of the array of streams `A`, and connects it to the "window" `z` for use within the task. In this degenerate form, the window is a scalar: the task can only access and consume one element at a time. In the third example on the left, the task reads from the stream of arrays `A` of 3 elements; depending on the task, the same array may interchangeably be used as an array of streams or a stream of arrays. Finally, the last example on the right shows a stream of arrays with parametric `horizon` and `burst` values; all combinations are possible.[2]

```
int x, z;
int X[horizon];
int A[3];

#pragma omp task \
        input (x >> X[burst])
  // task code block
  // horizon > 2
  ... = ... X[2];

#pragma omp task \
        input (A[0] >> z)
  // task code block
  ... = ... z ...;

#pragma omp task input (A)
  // task code block
  ... = A[0] + A[1] + A[2];
```

```
int y;
int Y[horizon];
int B[horizon][2];

#pragma omp task output (y)
  // task code block
  y = ...

#pragma omp task \
    output (y << B[burst][])
  // task code block
  for (int i=0; i<burst; ++i)
    {
      B[i][0] = ...;
      B[i][1] = ...;
    }
```

Figure 3: Example of `input` and `output` clauses.

Notice that the array used to access the stream does not need to be allocated. Its syntactic presence is motivated by compatibility reasons and to ensure the code can compile and execute even if the OpenMP annotations are omitted.

Following the semantics of OpenMP3.0, the enclosing context is considered to be an implicit task acting as a source and/or a sink for any streams that do not have other connection options. If a task has a unmatched `input` clause, the stream of data comes from the enclosing context and conversely, if a task has a unmatched `output` clause, the stream of data goes to the enclosing context.

Execution model. The OpenMP3.0 execution model states that, whenever a thread encounters a `task` construct, a task instance is generated from the code of the associated structured block. This task may either be scheduled immediately on the same thread or deferred and assigned to any thread in the team. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

Such a model is well suited for very unbalanced loads, but in most cases the overhead of creating and scheduling the tasks is significantly higher than synchronizing persistent tasks. While a modification of the execution model is not necessary for the correctness of our extension, this model is poorly suited from an efficiency perspective in the case of streaming tasks, which are naturally rather regular.

We propose to make streaming tasks persistent in our OpenMP extension. We emphasize the fact that this change only affects the *execution model*: the semantics of OpenMP programs is not impacted. This choice puts a heavier load on the compiler: it needs to convert the dynamic scheduling of new instances of a task into data-driven synchronization (i.e., based on the availability of data in the input streams). In this new model, all streaming tasks are created at the beginning of the enclosing OpenMP context and they can only execute when sufficient data is present on all input channels. In the particular case where a task has no input channels, but

[2]The size of the second dimension of `B` can be implicit in the clause.

has output channels (thus qualifying it for streaming), an implicit input stream is created to carry the control-flow, thus converted to data-flow.

5 Semantics of the streaming extension

In this section, we elaborate on the semantical interpretation of our proposed extension. The OpenMP specification provides many illustrative examples that help understanding of the semantics and we will also adopt this stance. For instance, because of its execution model, the `task` construct is mostly used within the scope of a worksharing construct[3]. As every thread encountering a `task` construct will create a dynamic instance of the task, it is necessary to be able to discriminate the different instances, in a context where tasks can be scheduled anytime, anywhere (barring tied tasks and "if" clauses). For this reason the `task` construct will actually be meaningful only in cases where threads are already differentiated, like e.g., within a worksharing construct.

```
#pragma omp parallel \
      num_threads (2) {

  for (i = 0; i < N; ++i) {
#pragma omp task
    work (i);
  }
}
```

```
#pragma omp parallel \
      num_threads (2) {
#pragma omp for
  for (i = 0; i < N; ++i) {
#pragma omp task
    work (i);
  }
}
```

Figure 4: Task instances need to be differentiable. Multiple undifferentiated instances (left) and properly differentiated (right).

Figure 4 illustrates this issue. On the left, the `task` construct is used directly within the `parallel` directive. There will be duplicate instances of the task, one for each of the two threads executing this parallel section, and the two instances will be impossible to differentiate. By contrast, on the right, the worksharing construct (the `omp for`) distributes loop iterations onto the available threads and ensures a single instance of the task will be activated for each value of `i`.

The same semantics will apply to streaming tasks. While we do not forbid the use of streaming tasks outside a worksharing construct, there is hardly any reason why such a behaviour would be used, except if the code preceding the `task` construct manually differentiates the threads, which is seldom portable and often considered bad coding practice.

Another important point is that, by nature, pipelining requires tasks to be part of a loop structure as filters are applied to a sequence of elements. It is possible to use streaming tasks outside of a loop nest, which then behaves as a loop with a single iteration. This is of little interest overall as it only incurs the overhead of creating streams for single elements.

In the remainder of this paper, we will only discuss the cases where streaming tasks are enclosed by a loop, which itself must be nested within a worksharing construct. However, as these constructs can be part of a caller function, the callee can possibly only exhibit freestanding tasks, which will sometimes be the case in our examples for brevity.

[3]OpenMP3.0 defines the following worksharing constructs: `loop`, `sections`, `single` and `workshare`.

5.1 Programming model

Though this is by no means an exhaustive list of the possible uses of streaming tasks or of their interaction with other OpenMP constructs, we will present in the following paragraphs how our extensions can provide the necessary building blocks for programming streaming applications.

Pipeline parallelism. To provide the fundamental basis for pipelining, we use the `single` worksharing construct for building a simple pipeline, as for example on Figure 5.

```
#pragma omp parallel
#pragma omp single
  for (i = 0; i < N; ++i) {
#pragma omp task output (x)
    x = ... ;
#pragma omp task input (x)
    ... = ... x ...;
  }
```

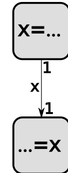


Figure 5: Simple pipeline using the `single` worksharing construct.

Parallel and sequential filters. Filters are naturally parallel, as the ordering of the elements in the stream is preserved by the stream library implementation, except in the case where a stream is both input and output to the same filter as in Figure 6. This denotes a loop-carried dependence, or equivalently that the filter has a state that is preserved between executions.

```
#pragma omp parallel
#pragma omp single {
  int counter = 0;
  for (i = 0; i < N; ++i) {
#pragma omp task input (counter)\
            output (x, counter)
     {
       counter++;
       x = ... ;
     }
#pragma omp task input (x)
     ... = ... x ...;
  }
```

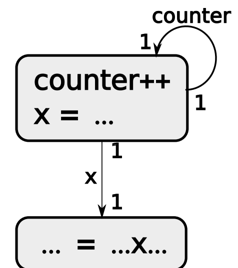


Figure 6: Pipeline with one sequential filter because of a self-loop.

In this case, the first filter must execute sequentially while the second could be data-parallelized. This case often arises when a filter reads or writes to a file as the file descriptor is both read and written at every access to the file. If a task does not properly specify inputs and outputs, the resulting behaviour is unspecified.

Multiple connections. It is possible to connect multiple filters to the same stream, both as input and as output. The semantics of multiple filters using the same stream as input is to broadcast the data: all the consumers have access to the all data. If multiple filters use the same stream as output, then the values produced by these filters are interleaved in the stream, according to the sequential schedule.

Delays. Among stateful filters, *delays* play a central role. A unit delay prepends an initial value to the stream it takes as input, effectively delaying input values by one activation of the

task. Delays are used to break instantaneous dependence cycles in the SDF model of computation [12]; they take the form of the “pre” operator (akin to a synchronous register) in synchronous data-flow languages [10]. Their role is paramount in the modeling of hardware circuits and control-dominated embedded systems. Strangely, delays have not met the same success for parallel stream programming (yet), as illustrated by their absence from the StreamIt benchmark suite [19].

Delays can be implemented through tasks producing the desired amount of data before the producer filters start executing. This is illustrated on Figure 7, where k initial elements are inserted in the stream x by the first task. Thanks to the multiple output semantics defined in the previous paragraph, this first task implements a k-delay operator. If $k \geq 1$, this is sufficient to break the instantaneous dependence cycle among the two following tasks. No internal state is required to implement delays, the state is hidden in the stream, by storing the delay values, and outside in the control flow. This guarantees that delays do not waste data-parallelism by inducing spurious serialization due to internal state.

```
#pragma omp task output (x << A[k])
 for (i = 0; i < k; ++i)
  A[i] = ...;

 for (i = 0; i < N; ++i) {
#pragma omp task input (y) output (x)
  x = ... y ...;
#pragma omp task input (x) output (y)
  y = ... x ... ;
 }
```

Figure 7: Introducing delays on streams.

Mixing data and pipeline parallelism. While the runtime may exploit data parallelism within a pipeline by executing multiple instances of parallel filters (starting with the heaviest ones for load balancing), the programmer can decide to explicit data-parallelism at the pipeline or individual filter level.

```
#pragma omp parallel
#pragma omp for shared (A)
   for (i = 0; i < N; ++i) {
#pragma omp task output (x)
      x = ... ;

#pragma omp task input (x) shared (A)
      A[i] = ... x ...;
   }
```

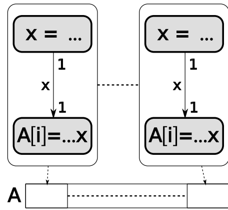


Figure 8: Parallel replicated pipelines with a worksharing construct.

For example the code on Figure 8 creates parallel pipelines of filters by using the loop worksharing construct instead of single. We must note that the correctness of this parallelization depends on the behaviour of the tasks. The user is responsible for ensuring no dependence is violated and for using proper synchronization if required. The programmer could even introduce a second level of data parallelism within a filter, by nesting a parallel loop within a streaming task as illustrated on Figure 9. But if the compiler does not take special provisions for the specialization of nested parallelism, it may result in unnecessary overheads.

Dynamic pipelines. In some cases, like the butterfly stages of an FFT, it is necessary to build a pipeline where the depth is parametric. We can build dynamic pipelines by using classical OpenMP3.0 tasks in addition to the worksharing construct, and using an array of streams to communicate through the pipeline, which we illustrate on Figure 10.

```
int X[k];
#pragma omp parallel
#pragma omp single
   for (i = 0; i < N; i+=k) {
#pragma omp task output (x << X[k])
     {
#pragma omp parallel for
       for (j = 0; j < k; ++j)
         X[j] = ... ;
     }
#pragma omp task input (x)
       ... = ... x ...;
   }
```

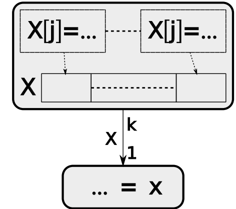


Figure 9: Parallel loop worksharing construct within a filter.

```
int A[K];
#pragma omp parallel
#pragma omp single
   for (i = 0; i < N; ++i) {
#pragma omp task output (A[0] << x)
      x = ... ;
   }
   for (j = 0; j < K-1; ++j)
#pragma omp task
       for (i = 0; i < N; ++i) {
#pragma omp task input (A[j] >> x) \
                 output (A[j+1] << y)
         y = ... x ...;
   }
```

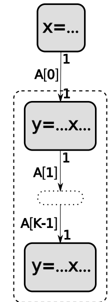


Figure 10: Dynamic pipeline of filters generated from a loop by using the task construct.

Hierarchical streaming. Figure 11 shows how pipelines can be organized hierarchically using nested streaming tasks. The enclosing task can be seen as a wrapper to factor the designation of inputs and outputs.

```
int sub_pipeline (int y) {
 int a, res;

#pragma omp task input (y) \
                 output (a)
   a = ... y ...;
#pragma omp task input (a) \
                 output (res)
   res = ... a ...;
 return res;
}
```

```
#pragma omp parallel
#pragma omp single

  for (i = 0; i < N; ++i) {

#pragma omp task output (x)
    x = sub_pipeline (...) ;
#pragma omp task input (x)
    ... = ... x ...;

  }
```

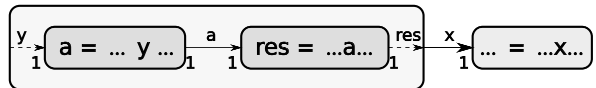


Figure 11: Hierarchically structured pipeline. The first streaming task on the right serves as a wrapper for the pipeline on the left.

Variable burst sizes. In C99, it is possible to declare arrays whose size is only known when entering a block, which means that our syntax can lead to variable-sized bursts and

horizons in streams. Variable horizons make it very problematic to determine the global buffer size for a stream variable, so the programmer should instead provide the maximal horizon across all iterations to make this computation possible. We thus disallow the usage of dynamically-sized arrays as horizons for `input` and `output` clauses. This does not result in a loss of generality: bursts can be dynamic, with each instance of a task consuming/producing a different number of elements, as illustrated on Figure 12.

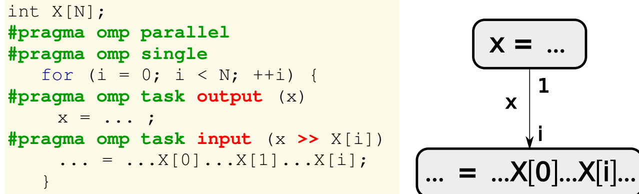


Figure 12: Restricting the horizon size to be a constant does not forbid a task to consume a variable number of elements.

Deadlocks and dependence cycles. The previous constructs can induce dependence cycles among tasks, through the `input` and `output` clauses. Delays can be used to break such cycles. Unfortunately, high expressiveness has a cost: with arbitrary control flow enclosing task activations and variable burst rates, detection of instantaneous (i.e., delay-free) dependence cycles is undecidable. We have to accept deadlocks as part of the semantics of the language extension.

At least, we know that if deadlocks occur, they will occur deterministically, independently of the number of threads or scheduling policy. This means that traditional test and debugging procedures for sequential programs are still applicable.

Although no complete method to avoid deadlocks can exist, conservative approaches have been very successful for embedded system design; they are based on control-flow and burst rate restrictions [12] (also adopted by StreamIt), or they rely on type systems of synchronous clocks [10]. Integrating some of these principles in the compiler could provide debugging help and support more aggressive optimizations; these research directions are left for future work.

`firstprivate` vs. `input`. These two clauses are semantically very close. They both represent a privatizing copy-in of a value. The main difference is that the `input` clause will in priority try to connect to an `output` clause while the `firstprivate` clause will copy the variable directly from the enclosing context. We promote `firstprivate` clauses to `input` clauses in streaming tasks as our choice of a persistent-task execution model means the expansion of the clause needs a stream to forward copy-in values to the thread in charge of executing the successive instances of the task.

5.2 Execution model

Following are a few important considerations on the execution model underlying these language constructs.

Persistent tasks. To improve performance, we propose to adjust the execution model to make streaming tasks persistent. Instead of having one instance of the task for each point in the iteration space of the enclosing OpenMP context (worksharing construct or any other OpenMP construct), we will have a single instance that will traverse the full iteration space, consuming data on the input streams and producing on the output streams. We emphasize the fact that this modification of the execution model is not a requirement of the extension we propose. Given the right circumstances, in particular w.r.t. the target architecture support for lightweight scheduling [11], the compiler may still generate code that fits the current execution model for tasks.

To prove the validity of this transformation, let us consider the acceptable schedules of the task instances. In the old schedule, no ordering, no exclusion and no thread locality could be assumed. All schedules were therefore acceptable (without explicit locking). In the new execution model, the persistent task traverses the iteration space in a statically-defined partial order, thus restricting the possible schedules to a subset of the acceptable schedules. The particular case where a parallel filter is replicated to benefit from data parallelism means that the iteration space has been strip-mined and that the different instances will impose a local order for the execution. The resulting set of possible schedules is still a subset of the acceptable schedules in the old execution model.

Correctness is of course the burning question at this point. Overall, the transformation is always possible and correct when the only scheduling constraints are the data-driven ones enforced by `input` and `output` clauses. Obviously, introducing atomic sections within tasks will not interfere with task-level scheduling constraints. However causality problems may arise when combining our streaming extensions with arbitrary locking mechanisms, if the acquisition of a lock escapes outside task boundaries. In real applications, locking may be legitimate to handle other forms of concurrency unrelated with the parallelization itself (e.g., I/O or user interfaces). Conversion to persistent tasks forces the ordering of successive task instances. Whereas valid schedules may exist for independent, freely schedulable tasks, it is possible that none of them be compatible with the sequential execution of dynamic instances of a given task. Without further precautions, conversion to persistent tasks may thus result into deadlocks (of the evil, target-dependent kind). Because of the critical performance advantage of the persistent-task execution model, and because of the importance of compiler optimizations to tune the grain of task and pipeline parallelism [9], we choose to (minimally) constrain the usage of cross-task locking mechanisms. Since OpenMP encourages programmers to make the sequential execution a subset of the legal schedules of the parallel program [20], one may require cross-task locking to be compatible with the serial execution of tasks. When generating persistent tasks, the compiler can safely assume that the original schedule of task instances is deadlock-free. Regarding debugging and test, one only has to compile the

program for serial execution to make sure it is deadlock-free.

Nesting. For all nesting purposes, we consider that the nesting of a streaming task within any OpenMP construct behaves in the same way standard `task` constructs behave. The iteration space taken into account for a streaming task is relative to the nearest enclosing OpenMP construct. However, the visibility of `input` and `output` clauses and therefore the visibility of the resulting task graph spans across all constructs within the nearest enclosing parallel region.

Data parallelism. Data parallelism is typically exploited automatically at runtime as all tasks that do not partake in a dependence cycle (induced by `input` and `output` clauses) are fully data-parallel. The programmer is free to mix pipelining with other data-parallel OpenMP constructs: the compiler will generate broadcast, splitter and selector patterns to handle synchronization and stream buffer indexing.

Streaming tasks are data-parallel by nature. This comes from the fact that they only read from and write to private memory (including their stream horizon). Unless the task uses shared memory with explicit synchronization or it is part of an inter-task dependence cycle, it will be deemed parallel.

5.3 Example: FFT

In order to illustrate the programming model this extension enables, let us show how this all works together on the implementation of FFT presented on Figure 13. This example was chosen as it illustrates most of the constructs we have introduced so far. The global structure is a linear pipeline of filters using two dynamic pipelines with an array of streams, `STR[]`. The horizons and rates are constant, but they vary across the different filters in the dynamic pipelines. This accounts for varying degrees of available data parallelism within the filters, which is a well-known issue for FFT.

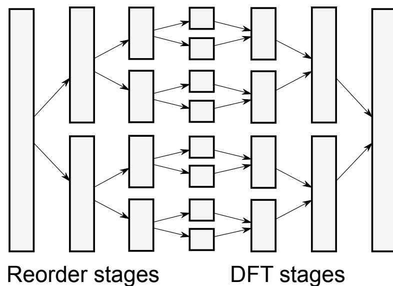


Figure 14: Data-flow graph for FFT.

For this FFT implementation, data-parallelism is available in each stage or vertical slice of the data-flow graph presented on Figure 14. Pipelining allows to relax the synchronization and enables wavefront parallelization. We control the granularity by varying the depth of the pipeline, thus changing the number of times the data is split. The maximal degree of parallelism available varies through the execution, and the optimal achievable speedup (on a PRAM) is in between $\log_2(\text{size})/2$ and $(\log_2(\text{size})+1)/2$.

6 Implementation and experiments

The implementation of this extension is under way in a public branch of GCC. This early implementation already provides full support for generating (coarse-grain) pipelined code from our OpenMP extension. The current stable state has reached the point where programs requiring simple pipelines can be compiled with support for constant burst and horizon sizes. The automatic exploitation of mixed pipeline- and data-parallelism as well as variable burst rates and dynamic pipelines are still under development. The streaming library takes advantage of the memory hierarchy by aggregating communication in reading/writing windows. The windows' size is a multiple of the size of a L1 cache line, which reduces false sharing and improves performance [17].

```
#pragma omp parallel
#pragma omp single
{
  while (16 == fread (readbuf, 4, 16, in_file)) {
#pragma omp task input (readbuf) output (qd_buf)
     fm_quad_demod (&qd_conf, readbuf, &qd_buf);
#pragma omp task input (qd_buf) output (band11)
     ntaps_filter_ffd (&lp11_conf, 1, qd_buf, band11);
#pragma omp task input (qd_buf) output (band12)
     ntaps_filter_ffd (&lp12_conf, 1, qd_buf, band12);
#pragma omp task input (qd_buf) output (band21)
     ntaps_filter_ffd (&lp21_conf, 1, qd_buf, band21);
#pragma omp task input (qd_buf) output (band22)
     ntaps_filter_ffd (&lp22_conf, 1, qd_buf, band22);

#pragma omp task input (band11, band12) output (res_1)
     subctract (band11, band12, res_1);
#pragma omp task input (band21, band22) output (res_2)
     subctract (band21, band22, res_2);
#pragma omp task input (res_1, res_2) output (ffd_buf)
     multiply_square (res_1, res_2, ffd_buf);

#pragma omp task input (qd_buf1) output (band2)
     ntaps_filter_ffd (&lp2_conf, 8, qd_buf, band2);
#pragma omp task input (ffd_buf) output (band3)
     ntaps_filter_ffd (&lp3_conf, 8, ffd_buf, band3);

#pragma omp task input (band2, band3) output (out1, out2)
     stereo_sum (band2, band3, &out1, &out2);

#pragma omp task input (out1, out2) private (output_short)
     {
       output_short[0] = trunc_and_norm (out1);
       output_short[1] = trunc_and_norm (out2);
       fwrite (output_short, sizeof(short), 2, out_file);
     }
}}
```

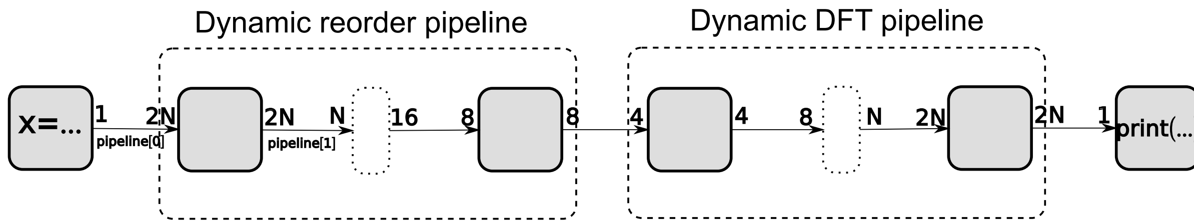
Figure 15: Annotated kernel from the GNU radio project.

We present results on three full applications: `FFT` from the StreamIt benchmarks [19], `FMradio` from the GNU radio package[4] and a `802.11a` production code from Nokia.[5] These applications are complex enough to illustrate the expressiveness of our extension. The annotated kernels of `FFT` and of `FMradio` are presented on Figures 13 and 15. The main loop of `802.11a` could not fit in this paper.

At this time, the fully automated stream code generation from extended OpenMP annotations is only possible for `FMradio` and `802.11a` and it can only exploit pipeline parallelism. It achieves a speedup of $3.1\times$ on `FMradio` and a

[4] `http://gnuradio.org/trac`

[5] From the ACOTES FP6 European project.




```
#pragma omp parallel
#pragma omp single
{
  float x, STR[2*(int)(log(N))];

  // Generate some input data (or read from a file)
  for(i = 0; i < 2 * N; ++i)
#pragma omp task output (STR[0] << x)
    x = (i % 8) ? 0.0 : 1.0;

  // Reorder
  for(j = 0; j < log(N)-1; ++j) {
    int chunks = 1 << j;
    int size = 1 << (log(N) -j + 1);
#pragma omp task
    {
      float X[size];
      float Y[size];
      for (i = 0; i < chunks; ++i) {
#pragma omp task input (STR[j] >> X[size])\
                 output (STR[j+1] << Y[size])
        for (k = 0; k < size; k+=4) {
          Y[k/2] = X[k];
          Y[k/2+1] = X[k+1];
          Y[(k+size)/2+1] = X[k+2];
          Y[(k+size)/2+2] = X[k+3];
        }
  }}}
```

```
  // DFT
  for(j = 1; j <= log(N); ++j) {
    int chunks = 1 << (log(N) - j);
    int size = 1 << (j + 1);
#pragma omp task
    {
      float X[size], Y[size];
      float *w = compute_coefficients (size/2);

      for (i = 0; i < chunks; ++i) {
#pragma omp task input (STR[j+log(N)-2] >> X[size]) \
                 output (STR[j+log(N)-1] << Y[size]) shared (w)
        for (k = 0; k < size/2; k += 2) {
          float t_r = X[size/2+k]*w[k] - X[size/2+k+1]*w[k+1];
          float t_i = X[size/2+k]*w[k+1] + X[size/2+k+1]*w[k];
          Y[k] = X[k] + t_r;
          Y[k + 1] = X[k+1] + t_i;
          Y[size/2+k] = X[k] - t_r;
          Y[size/2+k+1] = X[k+1] - t_i;
        }
  }}}

  // Output the results
  for(i = 0; i < 2 * N; ++i)
#pragma omp task input (STR[2*log(N)-1] >> x, stdout) \
                 output (stdout)
    printf ("%f\t", x);
}
```

Figure 13: FFT implementation using dynamic task pipelines and the corresponding task graph.

speedup of 2.9× on `802.11a` on an Intel Core2 Quad Q9550 with 4 cores at 2.83GHz. Similar results are achieved on the other platforms for the pure (coarse-grain) pipelined version.

To get an idea of what could be achieved once the implementation is complete, we manually parallelized the three applications using the low-level stream programming extensions supporting the expansion of OpenMP pragmas. We conducted performance evaluations on a 4-socket AMD quad-core Opteron 8380 (Shanghai) with 16 cores at 2.5 *GHz* and a 4-socket Intel hexa-core Xeon E7450 (Dunnington), with 24 cores at 2.4 *GHz*, both with 64 *GB* of memory. These targets are respectively called Opteron and Xeon in the following.

`FMradio` presents a high amount of data-parallelism and is fairly well-balanced; annotating the code with the extended OpenMP directives requires little effort and provides up to 12.6× speedup on Opteron and up to 18.8× speedup on Xeon. `802.11a` is more unbalanced and complicated to parallelize. A lot of code refactoring is necessary to expose data parallelism as the original version extensively uses global and static variables. After this step, annotating the program is straightforward and the parallelized code achieves up to 13× speedup on Opteron and 14.9× speedup on Xeon.

Speedups for `FFT` are presented on Figures 16 and 17. The baseline is an optimized sequential `FFT` implementation used as a baseline within the StreamIt benchmark suite. Combined pipeline- and data-parallelism achieve the best results, compared to pure data-parallelism or pure pipelining. We report two sets of results for each target: the single configuration results (on the left) correspond to speedups obtained when using the same tuning parameters (number of threads, granularity, etc.) for all the data sizes and for all the code versions; the best configuration results correspond to the optimal performance achieved with the best configuration for each individual data size point. The size of the machines and the associated cost of inter-processor communication set the break-even point around vectors of 256 `doubles` and more.

These results validate our approach and constitute a strong incentive to complete the development of this streaming framework in GCC. We are also proposing to include this extension in upcoming revisions of the OpenMP specification.

7 Conclusion

We presented an incremental extension to enable stream programming in OpenMP. Our work is motivated by the quest for increased productivity in parallel programming, and by the strong evidence that has been gathered on the importance of pipeline parallelism for scalability and efficiency. We discussed the design principles necessary to maximize the expressiveness and performance benefits of our extension, while preserving backward compatibility. One key choice was to

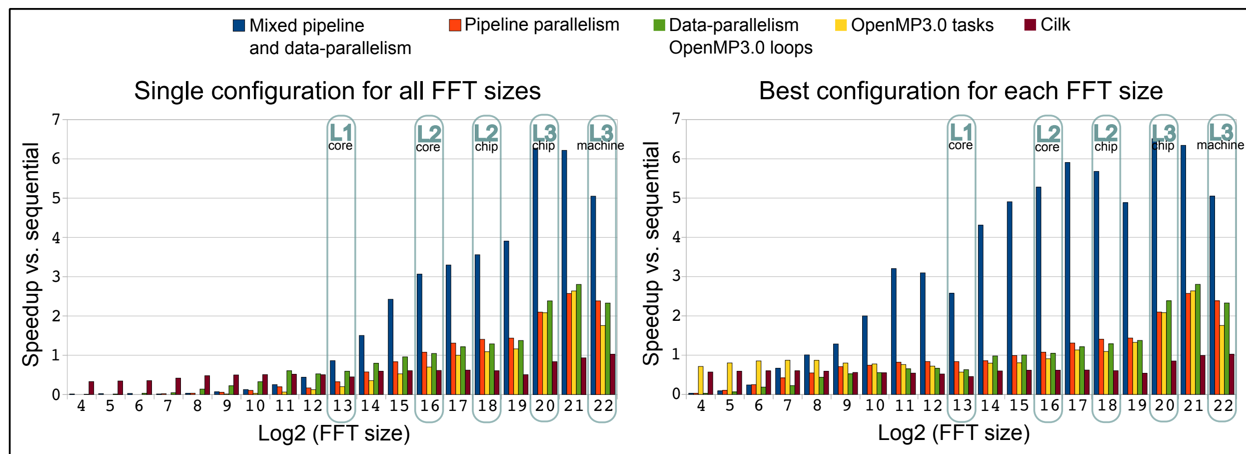


Figure 16: FFT performance on Opteron.

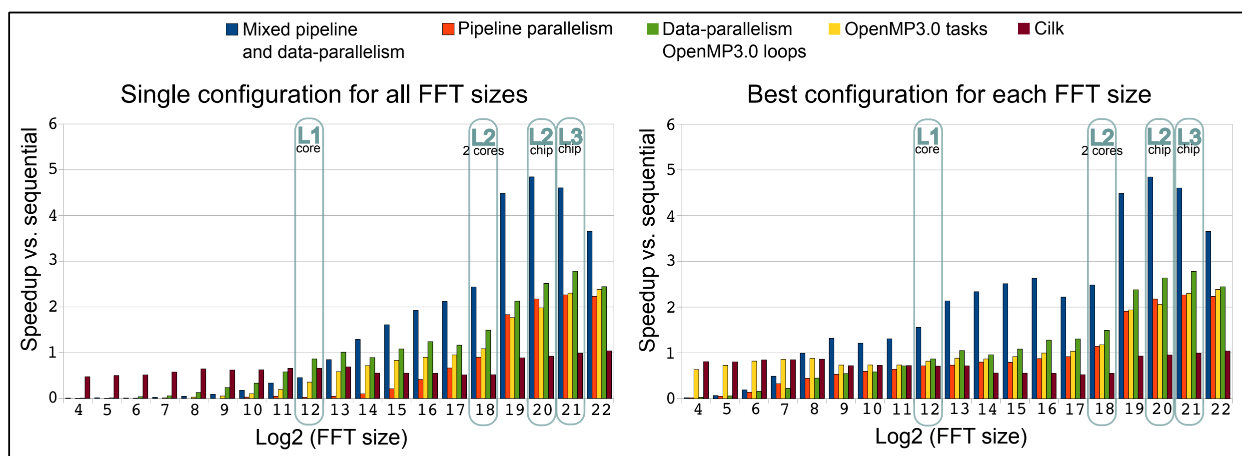


Figure 17: FFT performance on Xeon.

favor an execution model where the tasks are persistent: this choice allows static scheduling and lightweight, lock-free implementations for streaming communications. We demonstrated the expressiveness and performance advantages of our design on real-world applications.

The only down-side of persistent tasks is the lack of native support for load balancing. In the future, we plan to complement our implementation with lightweight threading and work-stealing mechanisms to address this limitation. We believe that data-driven scheduling at coarse grain can be combined with persistent tasks and streaming communications at finer grain to offer the best of both worlds in terms of load balancing and synchronization overhead.

References

[1] Arvind, R. S. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. on Programming Languages and Systems*, 11(4):598–632, 1989.

[2] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *SC*, 2006.

[3] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, Mar. 2004.

[4] The Brook Language. `http://graphics.stanford.edu/projects/brookgpu/lang.html`.

[5] P. M. Carpenter, D. Ródenas, X. Martorell, A. Ramírez, and E. Ayguadé. A streaming machine description and programming model. In *SAMOS*, pages 107–116, 2007.

[6] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Intl. Conf. on Parallel Processing (ICPP)*, Saint Charles, IL, 1986.

[7] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Supercomputing (SC'88)*, pages 368–373, 1988.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM Symp. on Programming Language Design and Implementation (PLDI'98)*, pages 212–223, Montreal, Quebec, June 1998.

[9] M. Gordon, W. Thies, and S. Amarasinghe. 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, Oct 2006.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[11] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-driven multithreading using conventional microprocessors. *IEEE Trans. on Parallel Distributed Systems*, 17(10):1176–1188, 2006.

[12] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–25, 1987.

[13] V. Marjanovic, J. Labarta, E. Ayguadé, and M. Valero. Effective communication and computation overlap with hybrid MPI/SMPSs. In *PPOPP*, 2010.

[14] A. Munshi. The opencl specification, v. 1.0, revision 29. www.khronos.org/registry/cl/specsopencl-1.0.29.pdf, 2008.

[15] V. Pankratius, A. Jannesari, and W. F. Tichy. Parallelizing bzip2: A case study in

multicore software engineering. *IEEE Softw.*, 26(6):70–77, 2009.

[16] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *Intl. J. on High Performance Computing Architecture*, 23(3):284–299, 2009.

[17] A. Pop, S. Pop, H. Jagasia, J. Sjödin, and P. H. J. Kelly. Improving GNU compiler collection infrastructure for streamization. In *Proceedings of the 2008 GCC Developers' Summit*, pages 77–86, 2008. http://www.gccsummit.org/2008.

[18] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso. Tflux: A portable platform for data-driven multithreading on commodity multicore systems. In *Intl. Conf. on Parallel Processing (ICPP'08)*, pages 25–34, Portland, Oregon, Sept. 2008.

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

[20] The OpenMP Architecture Review Board. OpenMP Application Program Interface. http://www.openmp.org/mp-documents/spec30.pdf.

[21] I. Watson and J. R. Gurd. A practical data flow computer. *IEEE Computer*, 15(2):51–57, 1982.