RICE UNIVERSITY

Integrating Stream Parallelism and Task Parallelism in a Dataflow Programming Model

by

Dragoş Dumitru Sbîrlea

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Houston, Texas

September 3rd, 2011

ABSTRACT

Integrating Stream Parallelism and Task Parallelism in a Dataflow Programming Model

by

Dragoş Dumitru Sbîrlea

As multicore computing becomes the norm, exploiting parallelism in applications becomes a requirement for all software. Many applications exhibit different kinds of parallelism, but most parallel programming languages are biased towards a specific paradigm, of which two common ones are task and streaming parallelism. This results in a dilemma for programmers who would prefer to use the same language to exploit different paradigms for different applications. Our thesis is an integration of stream-parallel and task-parallel paradigms can be achieved in a single language with high programmability and high resource efficiency, when a general dataflow programming model is used as the foundation.

The dataflow model used in this thesis is Intel's Concurrent Collections (CnC). While CnC is general enough to express both task-parallel and stream-parallel paradigms, all current implementations of CnC use task-based runtime systems that do not deliver the resource efficiency expected from stream-parallel programs. For streaming programs, this use of a task-based runtime system is wasteful of computing cycles and makes memory management more difficult than it needs to be.

We propose Streaming Concurrent Collections (SCnC), a streaming system that can execute a subset of applications supported by Concurrent Collections, a general macro data-flow coordination language. Integration of streaming and task models allows application developers to benefit from the efficiency of stream parallelism as well as the generality of task parallelism, all in the context of an easy-to-use and general dataflow programming model.

To achieve this integration, we formally define streaming access patterns that, if respected, allow CnC task based applications to be executed using the streaming model. We specify conditions under which an application can run safely, meaning with identical result and without deadlocks using the streaming runtime. A static analysis that verifies if an application respects these patterns is proposed and we describe algorithmic transformations to bring a larger set of CnC applications to a form that can be run using the streaming runtime.

To take advantage of dynamic parallelism opportunities inside streaming applications, we propose a simple tuning annotation for streaming applications, that have traditionally been considered with fixed parallelism. Our dynamic parallelism construct, the dynamic splitter, which allows fission of stateful filters with little guidance from the programmer is based on the idea of different places where computations are distributed.

Finally, performance results show that transitioning from the task parallel runtime to streaming runtime leads to a throughput increase of up to 40×.

In summary, this thesis shows that stream-parallel and task-parallel paradigms can be integrated in a single language when a dataflow model is used as the foundation, and that this integration can be achieved with high programmability and high resource efficiency. Integration of these models allows application developers to benefit from the efficiency of stream parallelism as well as the generality of task parallelism, all in the context of an easy-to-use dataflow programming model.

Acknowledgments

I would like to thank my advisor, Professor Vivek Sarkar, for his guidance during my graduate studies. His advice has always been insightful and realistic and helped keep this work on track. His management skills insured that non-research related problems did not cause delays. I am also grateful to him for the opportunity of working in such a wonderful group as the Habanero team.

I express my appreciation for my committee members: Prof. Keith D. Cooper, Prof. Lin Zhong and Jun Shirako, their careers have served as models and inspiration for my approach to research.

Jun Shirako has been always willing to help; it has been a pleasure to work with him on both personal and professional level.

My fellow graduate students from the Habanero group have enriched my research experience. Great ideas are born from interaction and I appreciate the discussions I had with all of them. I owe a lot of thanks to Sagnak Tasirlar who guided me through the tools used in this work.

Last but definitely not least, without my wife's support and patience this thesis would never have been possible.

Contents

Illustrations

Tables

Chapter 1

Introduction

As modern processors hit the power and frequency walls, multicore architectures are the solution to allow future processors to continue scaling. For the software developer to take advantage of the new various types of processing power available, new programming models are needed, that can the express the multiple types of parallelism that an application might have.

Two common paradigms for parallelism are task parallelism and stream parallelism. There is a large family of task-parallel programming languages and libraries currently available including OpenMP 3.0[1], Java Concurrency, Intel Threading Building Blocks[2], .Net Parallel Extensions[3], Cilk[4], and Habanero-Java[5]. Likewise, a number of stream-parallel programming languages have been proposed in the past, with StreamIt[6] being the most recent exemplar for the stream-parallel paradigm.

Applications for the streaming paradigm are common and becoming more and more prevalent. Up to 37% of Internet traffic is done by streaming video and have been estimated to take up to 90% of compute cycles as early as 2000s[7]. DSP applications, cell phone network call processing, database and classification algorithms, media streaming, HDTV video and audio processing and other compute-intensive applications are candidates for efficient parallel implementation using streaming languages. However, the expressiveness of streaming languages and programming models is usually limited to streaming parallelism and they are unable to express other forms of parallelism easily.

Macro dataflow programming languages such as the Intel Concurrent Collections (CnC)[8] are well suited for multicore execution of tasks because they separate the definition of the tasks from their scheduling, thereby making exploitation of different types of parallelism easier. The current CnC implementations only take advantage of the task based parallelism, like many other parallel programming models. As a result, the performance of applications following the streaming parallelism patterns suffers greatly if written in standard CnC. Integration of the two models would provide the best of both worlds: the generality and ease of use of task-based programming models, together with the performance streaming can offer to particular kinds of applications. The Streaming CnC extensions introduced in this thesis bring the benefits of streaming parallelism with some of the flexibility of task based parallelism through a dynamic split-join parallel construct; this construct allows dynamic creation of streaming filters in certain situations, bringing the parallelism to higher values to potentially match the parallelism of the machine.

Many previous streaming languages or frameworks do not offer either determinism or deadlock-freedom guarantees. This work preserves the determinism guarantees of CnC and provides an algorithm that can statically adjust the size of the buffers to ensure a deadlock equivalence between the streaming and task based execution: no extra deadlocks can happen with streaming compared to task based and if task based is deadlock free, so is the streaming execution.

The structure of the thesis is as follows. Chapter 2 looks at previous work on which this thesis builds, including streaming languages, the Concurrent Collections language and the Habanero Java language which is used to build both the task based and streaming runtime proposed. Chapter 3 describes the design and features of

Streaming CnC, the CnC subset that can be run using the streaming runtime. It describes interesting patterns that can be streaming-optimized and shows the dynamic parallelism feature that we propose. Chapter 4 describes how we can identify through analysis if an application conforms to the streaming restrictions and how we can obtain deadlock freedom guarantees if this is the case.

Chapter 5 and 6 describe the implementation of the streaming runtime and the performance results obtained on the set of benchmarks. In Chapter 7 we discuss and compare the related works and we conclude in Chapter 8 with future work directions.

Chapter 2

Previous Work

This work builds on past work on streaming languages (Section 2.1) and Habanero Concurrent Collections(Section 2.2), Habanero Java (Section 2.3), phasers, accumulators and streaming phasers (Sections 2.4 and 2.5). Streaming Concurrent Collections, the streaming system proposed in this thesis is related to streaming languages; as a notable example of such languages, StreamIt, has provided a rich source of streaming applications to test our work on.

2.1 Streaming Languages

Streaming parallelism is a type of parallelism encountered for applications that work over data that is structured as a "stream". Characteristics of such applications have been suggested [6], and the most important are:

- Processing large streams of data. The application has to execute operations on a large dataset, viewed as a sequence of data items that might not have a specific end point (unbounded size). However, each item must have a limited lifetime.
- Stream filters process the input sequence through specific operations that allow reading input items from the input stream and producing items to an output stream. The filters are connected to each other through the streams they process: the output stream of one filter can be the input to another, thereby

forming the streaming graph. Filters are relatively independent with few communications between them outside of the streams.

- The streaming graph structure does not change often.

The StreamIt language was designed to better express and take advantage of the structure of these applications. The original paper [6] allowed for static flow rates through single input and single output filters with special split and join nodes. It supported three main constructs that, combined, could allow concise descriptions of stream applications: the pipeline, split-join and feedback loop patterns. Listing 2.1 shows a StreamIt pipeline with 3 filters (lines 20-22), one of which is a finite impulse response filter (FIR Filter), defined through a class with an initialization function (lines 4-9) and a work function (lines 10-17). The processing of the filter is done in the work function, but the initialization is needed to set members to their initial values and also describe the type of data items contained by the stream.

```
 1 class FIRFilter extends Filter {
 2     float[] weights;
 3     int N;
 4     void init(float[] weights) {
 5         setInput(Float.TYPE); setOutput(Float.TYPE);
 6         setPush(N); setPop(1); setPeek(N);
 7         this.weights = weights;
 8         this.N = weights.length;
 9 }
10     void work() {
11         float sum = 0;
12         for (int i=0; i<N; i++)
13         sum += input.peek(i)*weights[i];
14         input.pop();
```

```
        output.push(sum);
    }
}
class Main extends Pipeline {
    void init() {
        add(new DataSource());
        add(new FIRFilter(N));
        add(new Display());
    }
}
```

Listing 2.1: Example of a StreamIt filter and its use when building a simple pipeline based application

An example of connecting filters using split and join nodes is shown in Listing 2.2. The filter presented consists of a splitter node that duplicates its input so that each child branch gets the same items (line 3) followed by a delay on each of the two branches (lines 4 and 5). The two delay filters feed into a join filter that takes input alternatively from the two branches (line 6). Together, the round robin join and delays with different amounts create an echo effect.

```
1 class EchoEffect extends SplitJoin {
2     void init() {
3         setSplitter(Duplicate());
4         add(new Delay(100));
5         add(new Delay(0));
6         setJoiner(RoundRobin());
7     }
8 }
```

Listing 2.2: Connecting stream filters through a split-join pattern

Listing 2.3 shows the use of the feedback loop pattern in building a Fibonacci string. The filter result stream is duplicated (line 15) after computing the sum of the previous values(line 12) and the value fed to its round robin join node that outputs only items from the feedback loop: 0 from the the normal edge, 1 from feedback (line 4).

```
1  class Fibonacci extends FeedbackLoop {
2      void init() {
3          setDelay(2);
4          setJoiner(RoundRobin(0,1));
5          setBody(new Filter() {
6              void init() {
7                  setInput(Integer.TYPE);
8                  setOutput(Integer.TYPE);
9                  setPush(1); setPop(1); setPeek(2);
10             }
11             void work() {
12                 output.push(input.peek(0)+input.peek(1));
13                 input.pop();
14             }});
15         setSplitter(Duplicate());
16     }
17
18     int initPath(int index) {
19         return index;
20     }
21 }
```

Listing 2.3: Building a Fibonnacci string with feedback loop in StreamIt

2.2 The Concurrent Collections Model

The Concurrent Collections(CnC) programming model[8] is a macro dataflow parallel programming system that uses components of three types to model programs: item collections, control collections and step collections. These collections and their relationship are defined statically for each application in a CnC specification file and the code of the application, split in tasks-like steps, can be written in any one of multiple languages for which there is a CnC runtime available.

Step collections are procedures in today's programming languages. Control collections drive the control flow of the program, by executing a procedure corresponding to a step collection when a control tag is "put" in the control collection(prescribed). The task that is executed is called step instance and receives the control tag as parameter. The step instance can then cause other step instances to run by putting new control tags in control collections.

Item collections play the role of variables in other programming languages and are sets of key-value pairs. Each item represents a value, which is put in an item collection with an assigned tag once during the execution of a program, respecting a single assignment rule. The tag can later be used to access that item (by the same step, or by another). The only restriction is that the step has to be registered as a producer on the control or item collection to which it puts tags.

Tag collections, also called **control collections** are the data that characterize the control flow of a step. A put into a tag collection leads to a step instance being prescribed. A prescribed step can start executing, but cannot finish executing until the items it reads become available through put operations performed by other step instances.

The **CnC graph** is a textual representation of the static relationship between the item, control and step collections in a CnC application. It is used by the runtime to ensure the access of the steps to the correct values in item collections, the execution of the correct steps when a tag is put into a control collection; the graph is also useful for the programmer, as an execution model that shifts the complexity of synchronization and communication between tasks from the programmer to the system,

The CnC graph is a directed graph whose nodes belong to the union of environment node, item, tag and step collections and edges consist of item-put edges (source: step, destination: item collection) representing producer relationship, item-get (source: item collection, destination: step) representing consumer relationship, tag-put (source: step, destination: control collection) representing control relationship, prescription edges (source : control collection, destination: step collection) and environment edges (from the environment node).

In this work, the following restrictions are implied for a CnC graph to be valid:

1. At least one tag collection is produced by the environment (There is at least an edge $X- > T$ where X is the environment node and T a control collection).

2. For each step collection , there is at least a possible execution that contains the execution of a step instance in that step collection.

Definition 2.1 The CnC control graph is the CnC subgraph restricted to only the environment node and control collections and step collections nodes and the tag-put and prescription edges.

Theorem 2.1

There is a path in the CnC control graph from the environment to any step collection.

Proof 2.1 Proof by contradiction. Presume there is a step collection SC0 for which there no path from the environment. According to the second restriction stated above,

Edge name	**Source**	**Destination**	**Meaning**
Item Put	Step Collection	Item Collection	At least one instance of the source step collection may put an item in the destination item collection
Tag Put	Step Collection	Tag Collection	At least one instance of the source step collection may put an tag in the destination control collection
Item Get	Item Collection	Step Collection	At least one instance of the destination step collection may get an item from the source item collection
Prescription	Control Collection	Step Collection	Any tag put into the source control collection leads to the execution of a step instance from the destination step collection.

Table 2.1 : Types of edges in a CnC graph

at least one step instance from that step collection has to be able to execute. For this to take place, there has to be another step collection SC1 that produces tags to execute steps in SC0 and has a path from the environment, so that it is executable. Thus, we discovered the path $Env \rightarrow ... \rightarrow SC1 \rightarrow SC0$, which contradicts our hypothesis. □

Theorem 2.2

The CnC control graph is weakly connected.

Proof 2.2 A directed graph is weakly connected if by replacing all its directed edges with undirected ones, the resulting (undirected) graph is connected. Proof by contradiction. I presume there is a step collection V and there is no path from it to another step collection T. But we know from Theorem 2.1 that for any node there is a path to the environment. For step collection V, this path would be: $Env \rightarrow N1 \rightarrow N2 \rightarrow .... \rightarrow V$ and for T, the path is: $Env \rightarrow M1 \rightarrow M2 \rightarrow ... \rightarrow T$. Thus, if we consider an undirected graph, V and T must be connected via Env, which contradicts our hypothesis. □

2.3 Habanero Java

The Habanero Java (HJ) [9] language is a programming language derived from X10 and developed in the Habanero Multicore research group which offers primitives for productive parallel programming. The base unit for parallel programming are tasks called **asyncs**, that are accompanied by a **finish** termination construct. Habanero Java supports a superset of Cilk's [10] spawn-sync parallelism. It eliminates the Cilk requirement that parallel computations should be fully strict: in HJ, join edges don't have to go to the parent in the spawn tree [11].

2.4 Phasers

Phasers [12] are Habanero Java synchronization constructs that unify for point to point and collective synchronization for a dynamically variable number of tasks. The phaser registration mode models the type of synchronization required: signal-only and wait-only modes for producer and consumer synchronization patterns and signal-wait for barrier synchronization. In our work, we mainly use the producer consumer synchronization and only use collective synchronization (barriers) for the dynamic parallelism feature.

The Habanero Java implementation of phasers works by registering the phaser in the desired mode to each async that will use it. For the purposes of this work, one async will be the producer and one the consumer, so the code looks as shown in Listing 2.4. The producer task (line 5-8) creates an item (line 7) and then signals (line 9) the consumer task. The consumer task can then proceed past the wait call in line 15 on the same phaser used by the producer for signalling. Notice the phaser registrations that accompany the task creations(lines 6 and 13): signal mode for the producer and wait mode for the consumer.

An important detail is that this use of phasers - with explicit wait and signal operations - is, in general, not deadlock free. This desirable properly is offered by only using special next operations. Next operations are expanded to a sequence of signal and wait and in the absence of other signal and wait operations cannot deadlock [12] . Our choice of having multiple input streams per filter meant we have to wait for a variable number of times on each phaser, which is incompatible with the next operation.

The choice of using phasers for synchronization in this work was also supported by their ability of accommodate a dynamically varying number of tasks, unlike normal barriers. Their particular speed obtained by busy waiting in certain specific scenarios and have proved very efficient on current multicore processors.

```

final Item item = new ProducerConsumerItem();
// phaser declared with both signal and wait capabilities
final phaser ph1 = new phaser(phaserMode.SIG_WAIT);

// the producer task is registered in signal mode
async phased  (ph1<phaserMode.SIG>) {
  item.produce();
  // signal mode registration allows the signal operation
  ph1.signal();
}

// and the consumer in wait mode
async phased(ph1<phaserMode.WAIT>) {
  // wait mode enables the call to phaser.wait
  ph1.wait();
  // the consumer is blocked at the wait call
  // until the signaller performs the signal operation
  item.consumer();
}
```

Listing 2.4: Phasers used for producer-consumer synchronization

2.5 Phaser accumulators and streaming phasers

Phaser accumulators [13] are a reduction construct built over the synchronization capabilities of Habanero phasers. Each producer (which is registered in signal mode) sends a value to be reduced in the current phase and then, when all producers have signalled to the phaser, the consumer (which is registered in wait mode) can be unblocked and use the reduced value, as shown in Listing 2.5. An accumulator is associated with a phaser (ph) and needs to know the type of the values it is reducing (int) and what is the reduction operation (SUM). The consumers can *send* their values and then *signal* the phaser. The producer will get unblocked from its wait call after all signals have been received and it can access the reduced value through the accumulator *result* call.

```
final phaser ph1 = new phaser(phaserMode.SIG_WAIT);
accumulator acc = new accumulator(ph, int.class, SUM);
// multiple producers which reduce their produce values
for(int i=0; i< N; i++)
  async phased  (ph1<phaserMode.SIG>) {
    int val = produce();
    acc.send(val);
    ph1.signal();
  }

async phased(ph1<phaserMode.WAIT>) {
  ph1.wait();
  int reducedValue = acc.result();
}
```

Listing 2.5: Usage of accumulators for reduction

We use an extension of accumulators and phasers that is useful for streaming, called **bounded phasers** or **phaser beams** [14]. This extension eases the use of these constructs for streaming programs by adding support for bounded buffer synchronization in phasers and accumulators.

A bounded phaser is created with a given bound, k. In our work the bound is 1000. For phasers, the producer can proceed at most k phases ahead of the consumer. A bounded accumulator contains an internal circular buffer whose size matches the bound k that is used to store the additional items before they are consumed. Access to previously consumed elements is permitted, in the limits of the internal buffer, by providing an additional parameter to the result() call. The parameter is used as an offset from the current position in the buffer. These primitives provide the means for implementing our streaming runtime.

Chapter 3

Streaming extensions for Concurrent Collections

3.1 Streaming CnC

In this chapter, we introduce Streaming CnC (SCnC) as a subset of the CnC model (graph specifications, corresponding code generator and runtime library) that allows implementation and runtime support for building CnC applications that exploit streaming parallelism as opposed to task parallelism.

To make this possible, we need a mapping between CnC concepts and streaming concepts.We identified this mapping and it is shown in table 3.1. A subset of the CnC graphs where this mapping is valid and can be implemented efficiently has to be found. Theoretical characterization of this subset is presented in section 3.2 and the engineering considerations behind our choice is presented in section 5.1. A comparison between CnC, its SCnC subset and streaming graph shapes can be found in Section 3.3.

3.2 Well-formed Streaming CnC graphs

Only a subset of the graphs that are legal CnC graphs can be used as input for SCnC. This is because of the nature of streaming (not any application is a streaming application) and because of implementation considerations (underlying phaser beams reduction restriction). This section describes this subset in detail, but does not describe the restrictions on what item gets and puts are legal in SCnC.

The conceptual requirement on the shape of the CnC graph is that the CnC graph is well formed and its the CnC control graph is a directed tree. The analysis of the shape of this graph will prove useful when we try and formalize the requirements of streaming applications, specifically when we loop at streaming access patterns.

Definition A *well formed* CnC graph respects the following conditions:

1. Control collections have only one producing step collection and one prescribed step collection.

2. Item collections have only one producing and one consuming step collections.

3. The environment only puts tags into a single tag collection and has no other put edge (to any other tag or item collection). This tag collection whose tags are supplied by the environment is the root of the tree and has a single child, the entry step of the graph.

The data is provided through the control tags that get put from the environment; each tag can store also a data point of the stream.

Theorem 3.1

The CnC control graph of a well formed graph is a directed tree.

Proof 3.1 A directed tree is a directed graph with no cycles. We know from Th 2.2 the CnC control graph is weakly connected, need to prove the absence of cycles. As both step and tag collections have only one predecessor and at the same time the environment has none and it is connected to all nodes, this conclusion is obvious. No cycles and weak connectivity limply the desired conclusion. □

The root of the CnC control graph could be considered to be the environment. For uniformity, as the environment is not a standard step and because it is restricted to a single child, we can consider the root of the tree to be the sole child of the environment: the control collection of the entry node .

Theorem 3.2

The CnC control graph with the entry control collection as its root is an arborescence.

An arborescence is a directed, rooted tree in which all edges point away from the root. The CnC control graph with entry control collection as root is a directed tree, as Theorem 3.1 showed. We know that there is a directed path from the environment to each step collection (Theorem 2.1). As the entry control collection is the singular child of the environment all paths pass through it, so there must be a path from the entry control collection to every node.

The paths starting from the root start from tag collection and end in control collections, which is the correct orientation of the prescription edges in the CnC control graph, or they might go from step collection to tag collection, which is the correct orientation for control put edges. There is no other type of edges in the control graph.

3.3 Comparing SCnC to streaming and to CnC

The design of Streaming CnC started from the observation that some CnC concepts map naturally to streaming concepts: item collections can be viewed as streaming queues and steps as filters. Of course, there are differences such as the explicit control flow in CnC and the formalization of the environment. The mapping between streaming constructs and SCnC constructs is in table 3.1

Control collections support just a subset of the item collections operations (”put last” and ”get first” instead of ”put anywhere” and ”get from anywhere”). The restriction on their operations compared to item collections comes from the fact that

CnC name	**Streaming name**
Item collection	Queue between filters
Control collection	No exact match in streaming as control flow is not explicit.
Step collection	Filter
Environment	Not formalized (input stream)

Table 3.1 : Mapping between CnC concepts and streaming concepts

SCnC concept name	**Number of consumers**		**Number of producers**	
	CnC	SCnC	CnC	SCnC
Item collection	N	1(N)	N	1(N)
Control collection	1	1	1	1
Environment	N	1	-	-

Table 3.2 : Comparison between CnC and SCnC: the number of producers and consumers supported by different building blocks

in streaming applications there is a specific order in which the filters process data: the order in which the items are put. Realizing that control collections for streaming applications are in fact queues too queues,we mapped hem to the same primitives as item collections which are item queues.

A comparison of the number of producer / consumer edges supported by the different component types of SCnC and streaming and CnC is found in tables 3.2 and 3.3. Note that the number of consumers and producers of item collections is limited to 1 in SCnC. The restriction on multiple consumers of an item collection relaxed for dynamic parallelism; there, the consumers are ”synchronized” consuming the same items in the same order and and are prescribed the same number of times.

SCnC concept name	**Number of consumers**		**Number of producers**	
	Streaming	SCnC	Streaming	SCnC
Item collection	N	1(N)	N	1(N)
Control collection	-	1	-	1

Table 3.3 : Comparison between Streaming and Streaming CnC: number of producers and consumers supported by different building blocks

SCnC concept name	**Number of consumers**		**Number of producers**	
	Streaming	SCnC	Streaming	SCnC
Step collection	1	N	1	N

Table 3.4 : Comparison between Streaming and Streaming CnC: number of input and output streams for a step

The single consumer restriction for item collections does not necessarily decrease the number of programs that can be expressed, it just makes the distributions/duplications explicit in the SCnC graphs by split and join nodes. Distribution and collection (join operation) - the patterns of communication affected by the change - are actually operations themselves, it is natural for them to be explicit in a CnC based model. These operations are discussed in detail in Section 4.1.

Having join operations as explicit steps helps solve the determinism problems that might happen in a multiple producer/consumer scenarios otherwise, because the join step explicitly states the order of the gets and puts. Furthermore, a single SCnC step can operate on a number of inputs and output collections larger than one, as opposed to the limitation of StreamIt to a single input and output, as seen in Table 3.4.

The semantics of the item collection and streaming queues are similar, as Table 3.5

Table 3.5 : Comparison between Streaming and Streaming CnC item get semantics

SCnC operation	**Streaming operation**	**Description**	**Semantic difference**
collection.get(0)	pop()	remove the next element in the stream, and return it	none
collection.get(x)	peek(x)	get(x) returns the element that has been return by the x previous get(0) call; peek(k) return the item at offset k in the stream	get(x) is a reverse peek: item = peek(x); pop(); pop(); pop(); ...; item2 = pop() => item == item2; item3 = get(0); get(0); get(0);... item4 = get(x) => item3 == item4

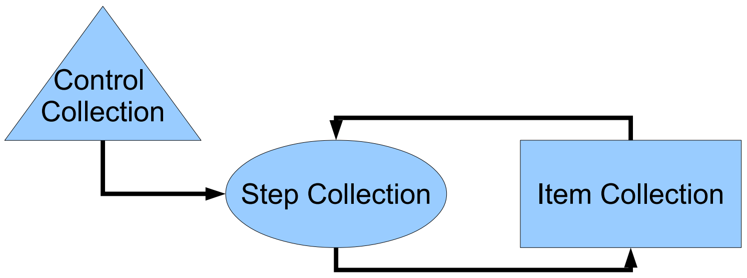

Figure 3.1 : A CnC Step-local item collection with its corresponding step collection

shows, with get(0) corresponding to a pop() operation (remove the first element in the stream). The peek operation is usually used to obtain read access to an item without removing it from the top of the queue (popping it). If the purpose of the operation is control flow related (control a different step collection), the CnC programmer would do a get(0) and send the value as tag to the step that needs it. If the purpose of peek is to allow reuse of the value in different step of the same collection, then the programmer can use the get(M), $M > 0$ operation provided by item collections. Get with a parameter different than 0 is similar to a "reverse peek" operation that allows access to the element obtained M pop operations ago.

3.4 Step-local collections

In many applications, one pattern that appears in the graph is the item collection - step collection cycle, as shown in Figure 3.1. This means that a single step collection is both producer and consumer of an item collection and for well formed graph the step collection, being the single producer and consumer, is the single entity to interact with the item collection. Such item collections are thus step-local item collections.

We have identified the cause of this pattern to be the restriction of CnC that the steps are stateless (that is, there is no state information preserved between different step instance executions). If the application access pattern is streaming, these collections can be transformed back to step-local variables as state is permitted in SCnC. The definition of streaming access pattern will be covered in Chapter 4.

3.5 Dynamic Parallelism support

Changing the structure of a streaming graph is rarely required by the semantics of streaming applications [6]. It might, however, be a feature that allows better performance for many applications, due to the dynamic adaptation. Streaming CnC offers a way of expressing a limited type of such changes through dynamic split-join nodes. This optimization is similar to the StreamIt fission optimization[15], only that in our case it is dynamic: the number of parallel branches of a split node can vary dynamically. In fact initially tehre does not need to be a split node.

We based the dynamic parallelism approach on the notion of places in X10 and HJ. A change to the meaning of CnC tags was performed: when a control tag is put, there one can supply an additional dimension for the tag, a place id. The code of a single filter runs in different places in parallel. When a new place id is used for the first time, the corresponding instance instance of the filter is instantiated and inserted in the graph with the same connections as the filter being prescribed , thus forming a dynamic split-join node. Each of the nodes maintain their own local data fields whose values can be used between iterations. This approach works well for situations when the programmer is aware of the additional parallelism, but does not need to write any low level synchronization or task management code. It proved useful for situations such as clustering applications or load balancing, as the Facility Location and Sieve applications described in sections 6.3.6 and 6.3.4 show.

For steps that do not use local state between step instances, we describe a compiler transformation that would make the parallelism transparent to the code inside the step for steps. In combination with adding automatic place distribution in the runtime, this approach has the potential of obtaining performance gains without the need for programmer-managed parallelism. As the algorithm involves knowledge of the implementation details, specifically of some phaser restrictions, it is presented later, in Section 5.2.

Chapter 4

Towards automatic conversion of Macro-dataflow Programs to Streaming Programs

This chapter deals with automatic transformation of a CnC application that follows the classic model to one that runs on the Streaming CnC runtime, when legal to do so. The process should be similar for any transformation of a macro dataflow application to exploit streaming parallelism. In order to implement such a transformation, we considered three major steps as illustrated in Figure 4.1.

The first step is transforming the graph shape of the CnC application to a form that can be supported by theSCnC model - this transformation might not even be possible and for this step the "success in converting" will return "No". The algorithm and detailed description for this step are located in Section 4.1 .

Then, we need to check the streaming access patterns, which filters out additional non-streaming applications. We show how to do this in Section 4.2. The approach assumes the availability of functions that identify the tags (keys) for item operations performed by steps. They are under development in the Habanero CNC system, but until their implementation is complete, the contents of this chapter remains in the algorithmic realm.

As a last step, we need to convert the tags of the collections from CnC to SCnC; our approach is described in Section 4.5. This is an integral step in the mapping from the CnC API to the streaming API;our implementation is discussed later.

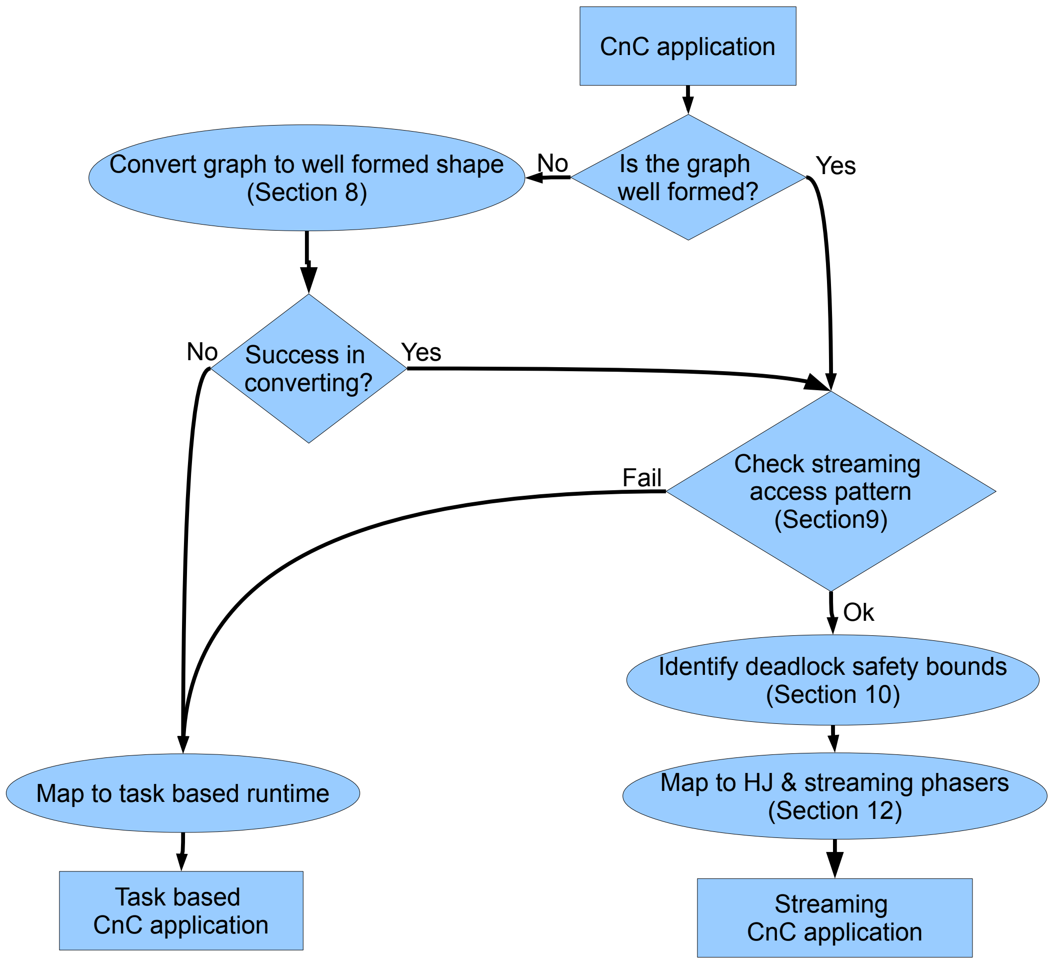
CnC application
Is the graph well formed?
No
Yes
Convert graph to well formed shape (Section 8)
Success in converting?
No
Yes
Check streaming access pattern (Section9)
Fail
Ok
Identify deadlock safety bounds (Section 10)
Map to HJ & streaming phasers (Section 12)
Map to task based runtime
Task based CnC application
Streaming CnC application

Figure 4.1 : The workflow of converting a CnC application to Streaming CnC

4.1 Transforming a CnC graph to a well formed shape

4.1.1 Possibility and Profitability of a CnC to SCnC transformation

Some programs written for the classic CnC runtime do not respect the restrictions of SCnC mentioned in the previous chapter on interaction with the environment, on the number of producers and consumers and the number of step collections prescribed by a control collection. If they were rewritten to a SCnC conforming (well formed) shape and found to respect some runtime behaviour restrictions, as the next sections show, some of these programs could run on the streaming runtime.

In some cases, it might not be profitable to run a CnC program using the streaming runtime if the graph needed alteration in order to conform to the well-formed shape. Although the overhead of streaming is less than the overhead of task based runtimes and there are memory management advantages too, the parallelism of the streaming runtime is usually fixed to the number of filters in the program, whereas the parallelism in task based runtimes can potentially approach the number of dynamic tasks in the program. We offer a solution for the limited parallelism exploited by classic streaming applications by the dynamic parallelism extension presented in later sections, but the parallelism in the classic CnC model could still be higher. At what point does the lower overhead become less profitable than simply using more parallelism is a matter of experience and practice. All our test applications benefit greatly from the streaming runtime, but it might not be always the case, depending on the parallelism available in the target hardware.

4.1.2 Algorithm for converting a CnC graph to well formed shape

The steps through which a CnC graph specification is rewritten to adhere to Streaming CnC well-formed shapes are the following:

1. Rewrite the graph by adding a new step collection and control collection for interaction with the environment. The instances of this new entry collection serve as sources for the items that would have been put from the environment before transformation. To perform the transformation, redirect all starting points of item put-edges from the environment to instead start from the entry node step collection. Redirect all the put-edges from the environment to end at this node and add a control-put edge from the environment to the control collection of entry step collection. Figure 4.2 illustrates this transformation.

2. Rewrite the graph by adding new control collections where there are multi-prescription control collections. Do this by replacing the control collection with N prescribed step collections with N control collections. Add prescription edges from each of the new control collections to one of the step collections and edges from the producer of the initial control collection to the new control collections.Figure 4.3 illustrates this transformation.

3. Reshape the graph to eliminate any multiple producer item collections. This is done by splitting the item collection and adding a step prescribed by one of the producer steps that functions as a custom join step: it gets items from all split collections and puts them into a single result collection. All the put-edges should be redirected to this step. Add a put-edge from the step to the item collection. This transformation requires also code to be inserted in the new step to perform the correct puts in the correct order, so as to obtain a custom join

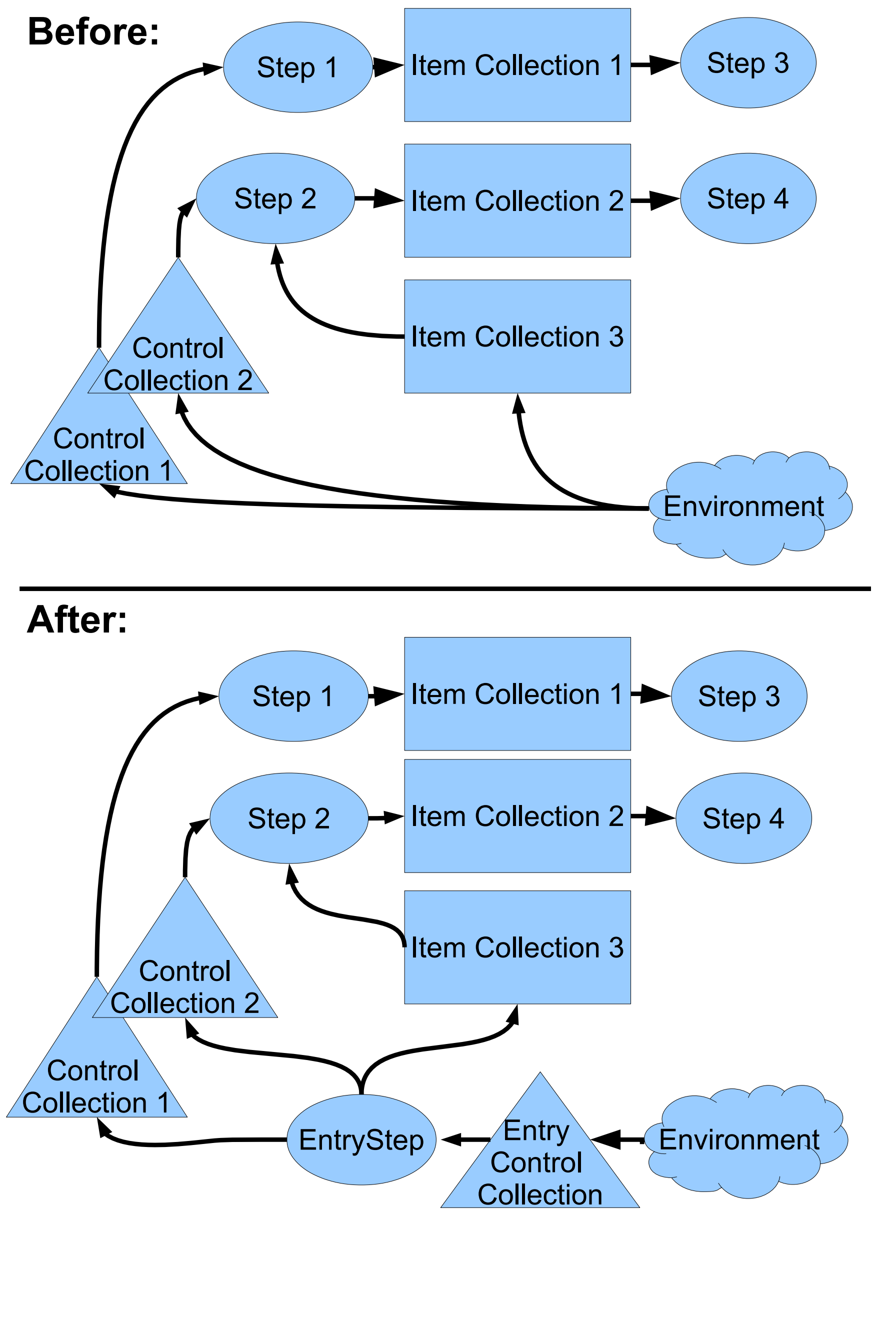

Figure 4.2 : Conversion of environment from multiple producer to single producer by adding an additional step

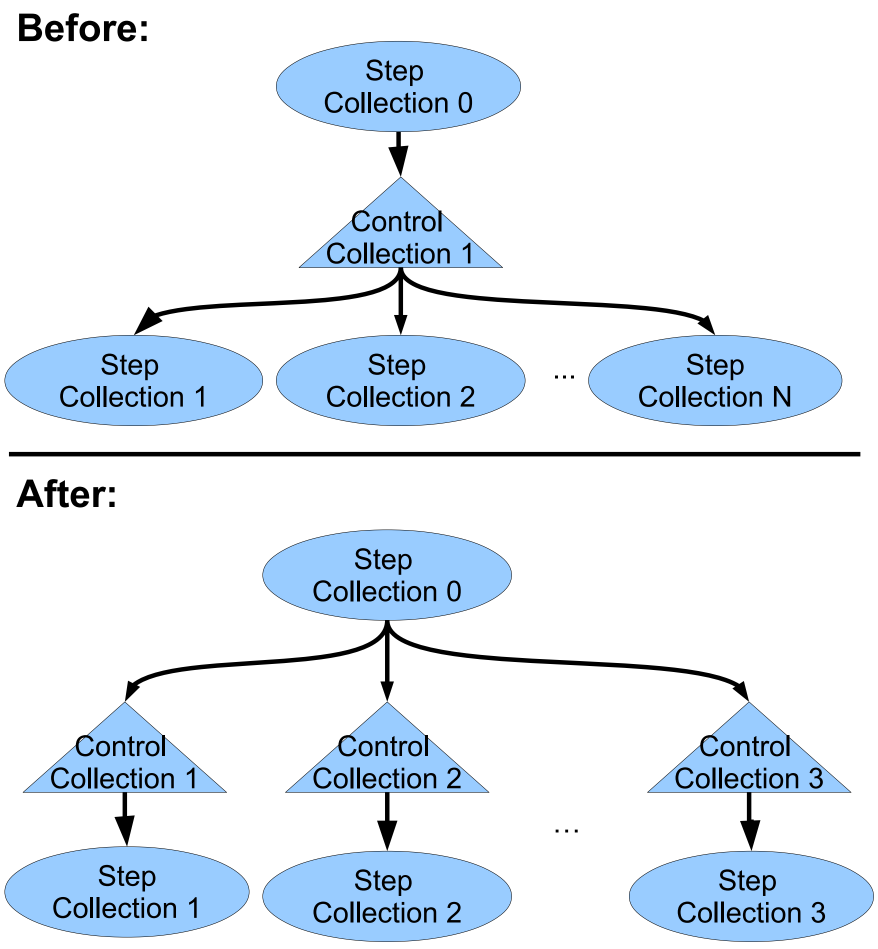

Figure 4.3 : Conversion of a control collection with multiple prescribed step collections to a control collections that prescribes a single step collection

that assures determinism. Figure 4.4 illustrates this transformation.

4. Duplicate any multiple consumer item collections such that they become single consumer, by duplicating the put-edges that produce items to each clone of the collection, and keeping a different single get-edge from each clone to a consumer. Figure 4.5 illustrates this transformation.

The pseudocode for the transformation algorithm follows.

```
If (environment is multiple-producer) {
  insert new step EntryStep and prescription collection EntryTags
  add prescription edge EntryTags -> EntryStep
  redirect edges starting from environment to start from EntryStep
  add producer edge from the environment to EntryTags
}
// correct the control collections first
insert all control collections in the worklist
while (worklist not empty) {
   pop control collection crt from the worklist
   if (crt is multiple prescription) {
      add n control collections
      add a  edges from one collection to one of the step collections
      add edge from producer of crt to each new collection
      remove crt and its edges
   }
}
Insert all item collections in the worklist
while (worklist not empty) {
  pop item collection crt from worklist
  If (crt has multiple producers) {
    add an item collection $C_i$ for each producer edge
```

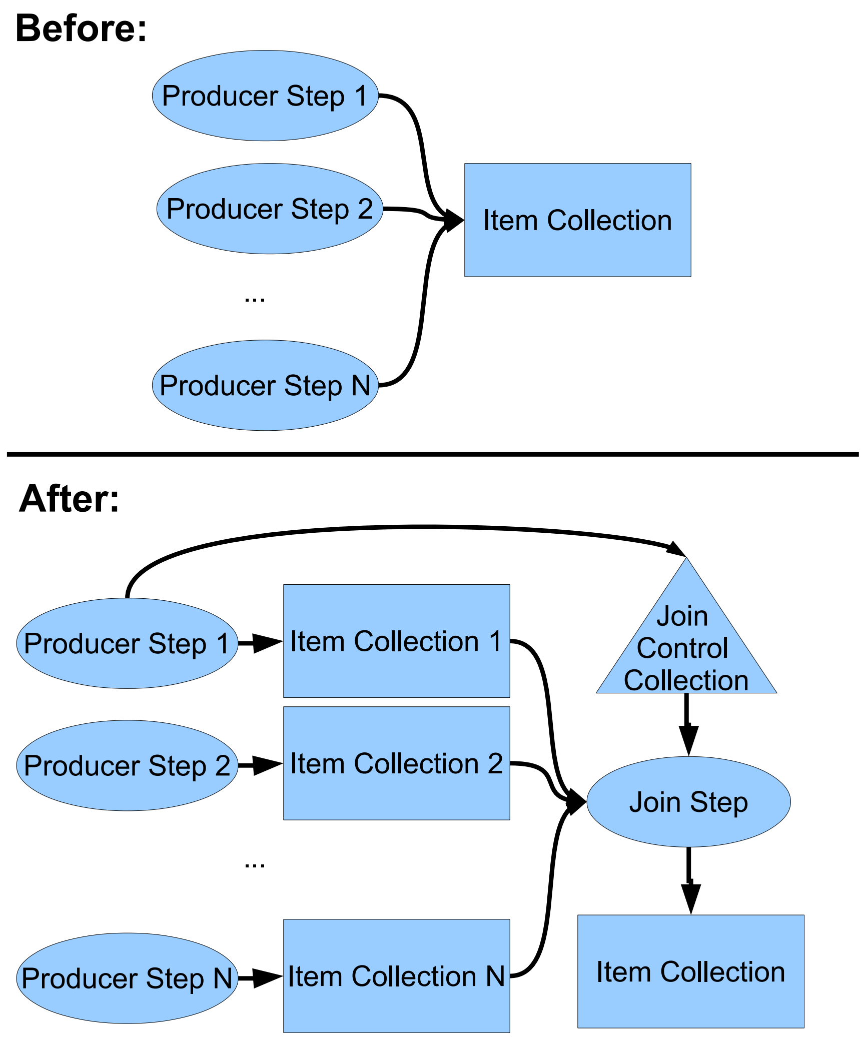

Figure 4.4 : Conversion of a collection with multiple producers to a collection with a single producer

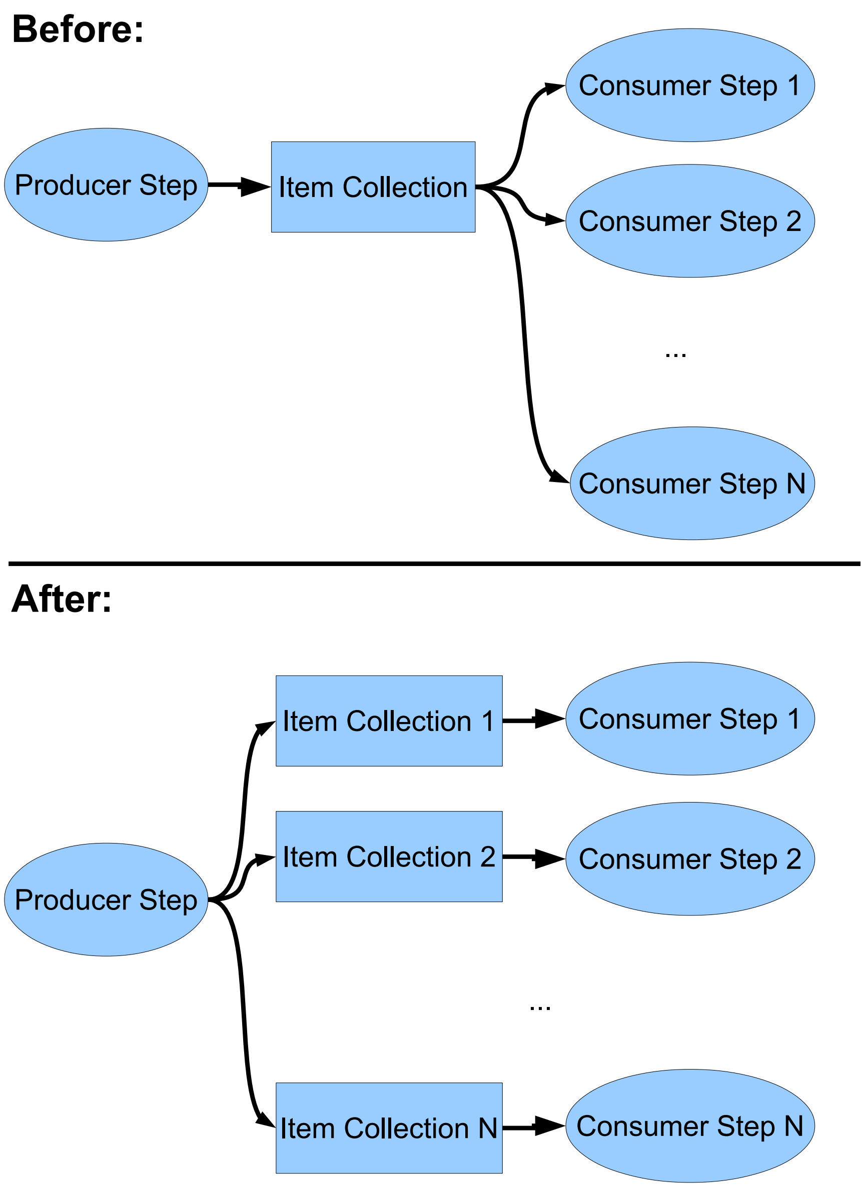

Figure 4.5 : Conversion of a collection from multiple consumers to a collection with a single consumer

```
    redirect each edge to one of the item collections
    add collections $C_i$ to worklist
    add tag collection TJ and join step collection SJ
    add prescription edge $TJ->SJ$
    add item consumer edges from each $C_i$ to $SJC_i->SJ$
    add item producer edge from $SJ$ to crt $SJ->crt$
  }
  If (crt has multiple consumers) {
    // crt has a single producer now
    remove crt
    add an item collection $C_j$ for each consumer crt had
    add each $C_j$ to the worklist
    add item consumer edges from each $C_j$ to a consumer
    add item producer edges from the producer step or crt P to each $C_j$
  }
}
```

4.1.3 Algorithm analysis

In this section, we analyze the complexity of the transformed graph relative to the input graph. Of course, if the input graph is already well formed,, then no further transformation is needed.

The addition of nodes and edges in the course of the transformations mentioned could lead to two sources of overhead: additional memory consumption because of the new item collections added and additional synchronization from the additional edges added. For example, a conversion from a multiple producer item collection to single producer adds N item collections, one for each separate producer, and a new step collection with associated control collections. The space requirements grows N+1 times (N item collections + 1 control collection), though this space requirement may be reduced in a later transformation when item collections are replaced with bounded buffers.

Synchronization requirements cannot be easily compared because the CnC synchronization is different from SCnC one. Let us assume that synchronization overhead is proportional to the number of edges (in SCnC, a pair of edges results in the use of a phaser; in CnC depending on the runtime, synchronization mechanisms vary, but the synchronization overhead remains proportional to the number of collections).

For the same example of multiple to single producer transformation for item collections, the number of edges in the figure increases from 4 to 9. In the general case of N producers, the number of edges increases from N to 2*N+2+1 edges, which leads to a doubling in the number of buffers.

Another limitation, caused by our use of explicit join nodes as opposed to implicit joins, is the inability of performing optimizations based on the relative flow rates as these are hidden inside user code. We considered the option of having the puts and joins of a step be part of its signature, but we chose not to do so - we would lose the flexibility of variable input output rates and thus not been able to support all well formed SCnC graphs.

4.2 Identifying streaming patterns in a well formed CnC graph

The Concurrent Collections model allows for the distinction between the domain expert (who writes the CnC graph and maybe the step code) and the tuning expert (who optimizes the application for best performance, by setting CnC scheduling parameters, adding scheduling restrictions and optimizing the step code).

As the streaming runtime is more restrictive than the task based one, additional checks have to be made before using it. First, the expert has to determine if the application can be rewritten to a streaming shape. To do this, we proposed in Section 4.1 an algorithm that can check for the structural graph requirements of the streaming parallel model. The current section deals with the required checks for the streaming access patterns on a well-formed graph, as in the output of the algorithm presented in Section 4.1.2. In this section, we take the well-formed shape of the application graph as a given and use the theorems 3.1 and 3.2 to support our analysis.

The proposed algorithm has two phases: graph analysis (computing auxiliary information) and streaming checks. The second phase can throw errors indicating that the application cannot be transformed to streaming form using our algorithm.

Any step that requires the computation of a function that cannot be solved (function does not exist) will fail and lead to early termination of the algorithm with an output of FALSE (application cannot be converted to streaming form using simple rewrite rules).

The graph analysis phase consists of the following steps:

1. Require the tags of the EntryStep collection (the $Env- > EntryTags$ edge) to be consecutive integers starting from 0 and the tags of all other control collections to be integers. It is possible to relax this restriction by allowing tags that contain an integer component.

2. Annotate each item-put edge (between a step T and an item collection O) with at least one put-function with domain the possible step prescription tags for step T and codomain the tags of the items that are put. There should be a

put-function for each tag that can possibly correspond to an item put by the step. If a step instance puts at least k items, there have to be at least k put-functions, to model the relationship between the tag of the step and the tag of the items produced.

3. Annotate each tag-put relationship with (at least one) function f^i_{tagPut} with the same eaning as in the previous step.

4. Annotate each item-get relationship with (at least one) function $f^i_{itemGet}$ similar to the previous functions, but for item get operations.

5. Label each prescription edge with the identity function $f_{tagGet}(x) = x$.

6. Do a traversal of the CnC control graph (which, for a well formed CnC graph, is an arborescence according to Theorem 3.2), labelling each step collection and attached item collection with the result of the composition of the functions through which the path from the root of the tree passes to reach that particular step. We call this label function a *producer function* for that step. $f^n_p = f_n(f_{n-1}(f_{n-2}(...f_1))))$ where the path from root EntryStep to $Step_n$ passes though Steps n-1, n-2, ..., 1 and $Step_1$ is EntryStep. The identify functions can safely be folded away in this chain. The traversal is easily done in a preorder traversal of the CnC control graph, thus incurring only a linear complexity cost. At each step collection node in the graph, label it with the same producer functions of its parent tag collection. At each tag collection node, label it with the composition of the producer functions of the parent and its incoming tag-put function. The producer functions for each step collection there will result in an associated set of producer functions, as control collections can have multiple incoming tag-put functions, depending on the producing step collection code.

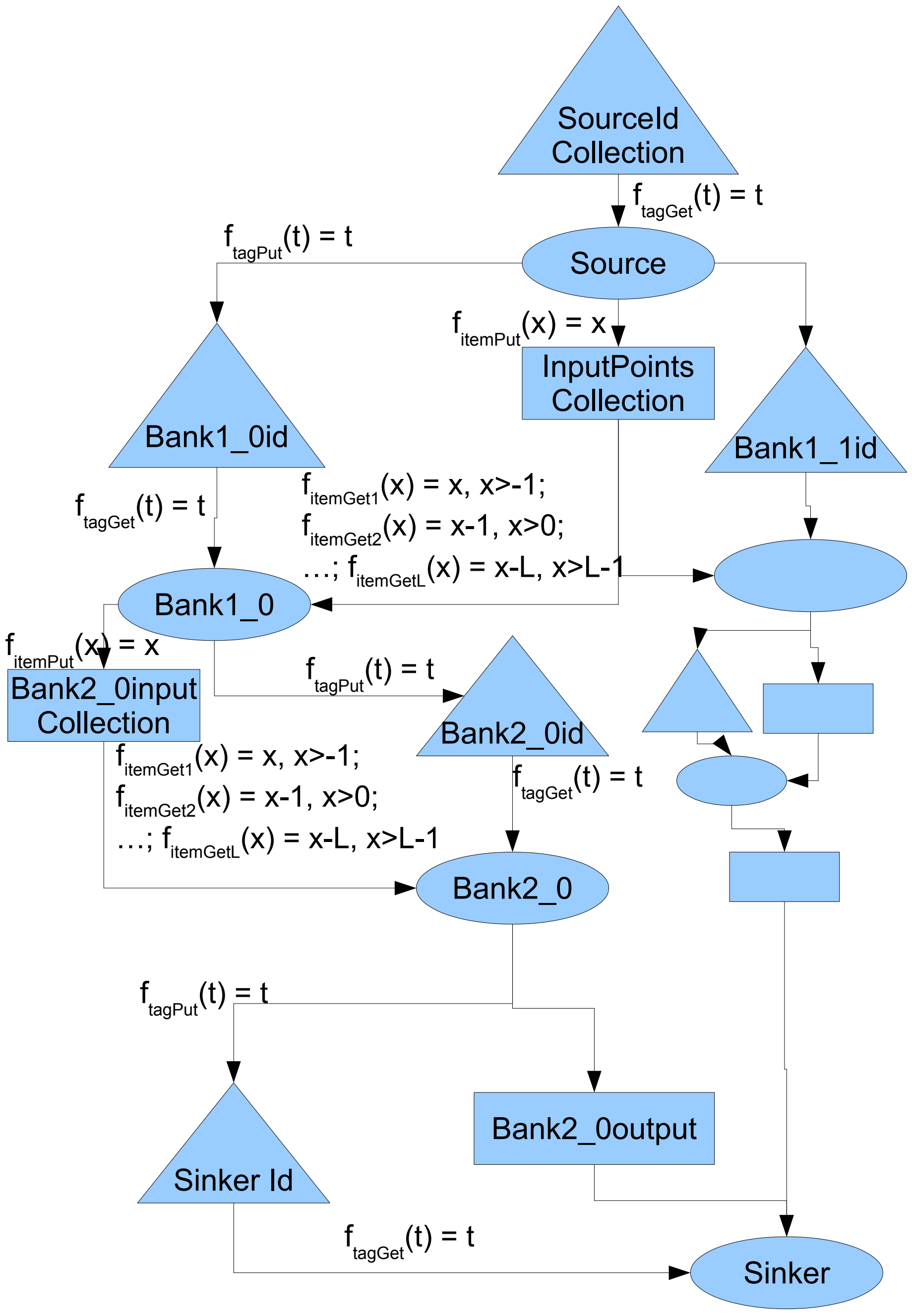

Figure 4.6 : The SCnC graph for the 2 branches of the FilterBank application, annotated with item-put, item-get and tag-put and tag-get functions (after step 5 of the algorithm)

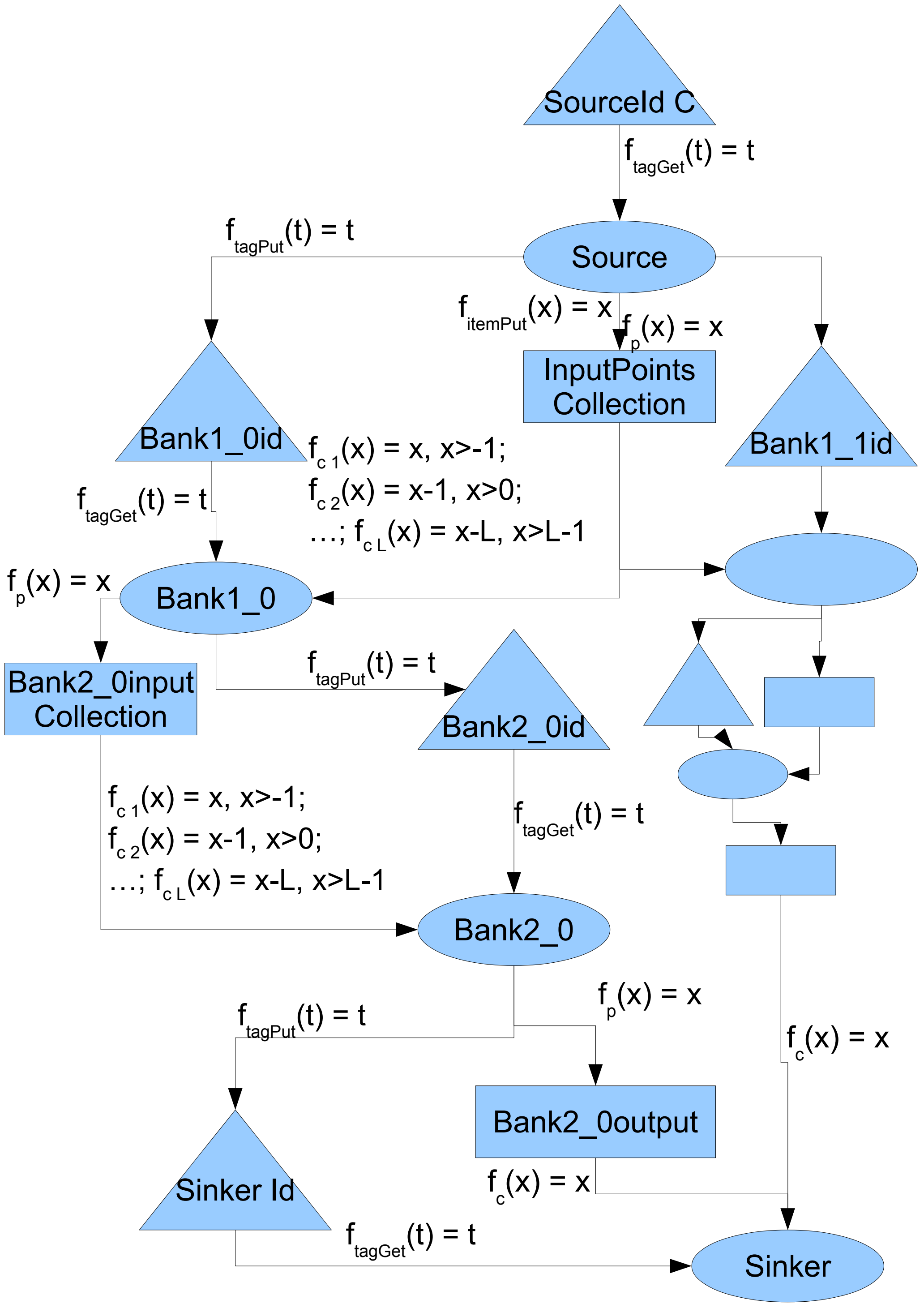

Figure 4.7 : The SCnC graph for the 2 branches of the FilterBank application, annotated with producer and consumer functions for item collections, after applying the algorithm

7. For each item collection, label it with a *consumer-function* f_c by composing the get function $f^i_{itemGet}$ of the item collection outward edge with the producer function of the consumer step.

8. For each item collection, label it with (at least one) *producer-function* by composing the get function $f^i_{itemGet}$ of the item collection producer edge with the producer function of the producer step. Both steps are made possible because the CnC graph we are working on has previously been reshaped to a well formed shape.

9. For each step compute the minimum consumer function, defined as the minimum of the values of all the consumer functions for each pair of (step, consumed item collection), $f_{cmin}(y) = min_x(f_{c_x}(y)), \forall y$.

All these functions will be used in the testing phase to ensure that the application access patterns are streaming. Note that, according to Theorem 3.2 if there are functions for steps 2 to 5, then the composition of functions required for step 6 exists (the set of producer functions that are attached to each node will have at least one element). The only way this algorithm can fail is if steps 2-4 in the testing phase fail to find a function.

The purpose of the test phase is to test the fact that the graph functions respect the streaming access restrictions to items. It consists of the following steps:

1. Using the consumer-functions and producer-functions of the item collections, we can test if the application is streaming or not. There may be multiple producer functions and multiple consumer functions for a single item collection and they will all have to be taken into consideration. Producer functions have to output consecutive increasing values for consecutive increasing inputs.

2. Test that the producer functions inverse and consumer functions for all item collections respect the following three conditions:

   a. "*producer precedence*" constraint, expressed through the following equation: $f_p^{-1}(y) \leq f_c^{-1}(y), \forall y \geq 0$. If there is no inverse for either producer or consumer functions for any item collection or if the previous relationship does not hold, the application is not a SCnC streaming application.

   b. "*bounded buffer*" constraint: there exists a constant N such that for any pair of consumer functions f_{c1} and f_{c2} of a step collection, the difference between the value of the consumer functions is smaller than N. The constraint is expressed though the equation where x the time iterations/sequence numbers put from the environment as tags in step 1 of the analysis phase: $|(f_{c1} - f_{c2})(x)| < N, \forall x \quad and \quad \forall f_{c1} \quad and \quad \forall f_{c2}$ consumer-functions of a single step collection.

   This is a restriction of the more general streaming requirement that once item i with tag t has been accessed, one can only access items with tags higher than t-N.

   c. "*sliding window*" constraint: For a single step collection, but different consecutive step instances tagged y and y+1, the minimum value of the tag that can be consumed by that step tagged y+1 is not lower than the minimum value that can be consumed by step instance tagged y. $f_{cmin}(y) \leq f_{cmin}(y+1)$

   The bounded buffer and sliding window constraints guarantee that we will never need a buffer size larger than N for an item collection.

   d. *"bounded lifetime"* constraint: For any item tagged t, produced in iteration t_1 and consumed in iteration t_2, there is N_2 constant such that $t_2 - t_1 < N_2$ Bounded buffer, sliding window and bounded lifetime assure that we will not need a buffer size larger than N_1 or N_2 to satisfy get calls on an item collection.

e. *"unique timestep"* constraint: Each step instance performs no more than a single put in each of its output control collections. This constrain assures us that, for a given step collection there will never be more than one step instance with the same iteration number (started by a single ancestor).

If the functions of all item collections respect the previous constraints, then the algorithm outputs TRUE. Otherwise it outputs FALSE.

Theorem 4.1

For an application with a well formed CnC graph, if the producer and consumer functions exist and respect the bounded buffer, producer precedence and sliding window rules and the CnC application terminates (with no suspended steps/deadlocks), then the corresponding SCnC application, if it terminates, terminates with the exact same state than the CnC application. The state of the CnC application consists of the items it has produced in each of the item collections.

Proof 4.1 In order to have item collections with the same items, the same steps should run and steps must have the same inputs and must produce the same outputs.

The first condition for this to happen is for the desired inputs to be available; the proof for this is as follows. The "bounded buffer" and "sliding window" constraints prohibit the access to items that are not in the streaming buffer of size N: bounded buffer means that a single step execution will need to access more items than the buffer has space for (accesses max N elements) and the sliding window rule shows that no step will need access to items that have already been removed (they can only access items that are "newer" than the oldest item consumed by the previous step). If neither the execution of a single step nor the sequence of two step executions lead to accessing an item that will not correspond to the CnC one, then, by induction, any execution will not lead to this situation.

The second condition is that the steps executed are the same in both SCnC and CnCand have identical inputs and outputs. They are, as no code changes are needed for well formed graphs, except the conversion of tags, but that transformation affects only the the tag keys, not the values accessed by them. Steps are executed on the same input identified as a subset from the codomain of the item put functions, by the step code, whose control flow is governed by the control tags which are explicitly and identically sent through the corresponding stream. As proved in the previous paragraph, the selected items are available. As the control flow is identical, then the items produced are identical. □

4.3 Deadlock

The question remains: can the SCnC version "hang" when the CnC application does not? We show that the SCnC application hangs only because of insufficient buffer-size problems that are common to all streaming programs.

First, let's look at when a CnC application can hang. A CnC application can hang if a step hangs. A step hangs if an item that is the target of a get is not produced in a finite amount of time. This can happen if the producer step hangs(reducing the problem to a previous step) or the producer step is not run because it is never prescribed. If we presume the CnC application does not hang, then none of these problems appear for the SCnC implementation.

A SCnC application can hang for any of the causes that a CnC application can hang, plus the following:

1. if a step blocks on a get on an item that cannot be produced because it requires

the current step to complete (because of the implicit serial execution of step iterations).

2. if a step performs a get on an item that is no longer in the streaming buffer.

3. the full-empty buffer problem [**?**, **?**]. Lets say one of the streaming buffer queues (called A) becomes full, blocking the producer and thus prevents him from producing items in another queue (B). If the consumer will block too waiting on B because of this (and cannot unblock A), then there is a deadlock. We describe a technique that finds a sufficiently large bound for the streaming buffers so that they never fill up. Note that the "bounded buffer" rule is not sufficient in this case, as the rule looks only at a single step and its data requirement from one item collection, whereas in this case the problem is inherently related to at least two item collections and the relative ordering of puts and gets from two steps.

For situation 1, we express the problem in terms of producer functions. The first point where the program hangs, some item could not be available because its producer did not complete and cannot complete, as it is waiting for some item that would only be produced later. If the producer function inverse value (representing an iteration number) is smaller than the consumer function inverse for that particular item, then we know that the producer can run independently from the consumer : if $f_p^{-1}(itemtag) = n$, the item will be produced after tags 1,2,3...,n are produced by the environment. The consumer, with $f_c^{-1}(itemtag) = m$ will run after some more tags are produced by the environment 1, 2, 3, ...,n, n+1,..., m, thus it cannot hang because the item was not produced. The "producer precedence" rule assures us that either $m > n$ which is sufficient, or m=n.

We still need to prove that if the inverses of the producer and consumer functions of an item collection are equal for an item, the SCnC program execution cannot hang. If the inverses are equal, then producer and consumer definitely reach the prescription stage. To hang, they each would have to hang on an item produced by the other (if we presume only one hangs, the other will finish, thus will produce the item which will allow the first one to continue). If they both hang waiting for an item produced by the other one, it means both of them block on get calls followed at some point by put calls that would unblock the other one. This situation would block the normal CnC implementation too, as any parallel execution of the producer and consumer steps would block, not just the streaming execution. The same argument applies identically for cycles of length more then two.

For situation 2, the "bounded buffer" rule ensures this does not happen. Situation 3 is dealt with in the following section.

4.4 Deadlock freedom

In order for SCnC to become a safe optimization to perform to CnC applications, we still need to make sure there is no possibility of deadlock. We first characterize the conditions that lead to deadlocks for SCnC applications and then present the restrictions that need to be respected in order for the application to be deadlock - free on the SCnC runtime.

First, it is important to notice that a program having only control flow (Control Collections and Step Collections) cannot deadlock, as the control graph is always a tree and there can be no other edges in this case (deadlocks appear as a cycle in the wait-for graph of the program). So, the deadlocks can appear as cycles that contain item get/put edges or both item/put edges and control edges.

We now express the restrictions needed for the SCnC execution to be deadlock free.

A step can block if an input buffer is empty or an output buffer is full. Deadlock for streaming applications can only occur after the full-buffer state is reached for at least one buffer. For a non-deadlocking(no suspended steps at the end) CnC application, a single empty buffer is not sufficient to cause deadlock in the SCnC version.

Let us look at the item collection buffers that can potentially be involved in a deadlock. For an item tag t produced with the restrictions of a well formed CnC application we have the equation: $t = f_p^i(t_1) = f_c^j(t_2)$ that shows the producer step instance that puts the item is tagged t_1 and the consumer is tagged t_2. If there are multiple possible producer and consumer functions, all combinations must be considered and the final buffer size should be the maximum of those identified through the following computation.

The required buffer size for item t is $(t_2 - t_1) * item_rate(producer)$. Where the item rate is the number of items produced in an iteration of the producer step,which is bounded above by a limit R, where R is less or equal to the the cardinality of the set of put functions corresponding to the producer and item collection.

Also, for most streaming applications (including all of those we tested) there is an integer constant k fixed such that $t_2 - t_1 < k$ which means that the items consumed by a step are at most produced a fixed number of timesteps before. Note that the $t_2 - t_1$ difference is always positive , as a restriction of SCnC. This does not mean there are no feedback loops in program; it is a restriction affecting the iteration numbers and not the structure of the graph.

The item collection buffer size is thus bounded above by $L = (t_2 - t_1) * R = k * R$. If the actual buffer size of the item collection buffer is larger than L, the buffer will never fill thus the producer and consumer edges cannot participate in a deadlock cycle.

The previous condition is not sufficient to guarantee deadlock freedom as, even though there is space in the item buffer, there might not be space in the control collection buffers somewhere on the path between the producer and the consumer (as shown previously, the control graph is a tree, so there is only one such path). To find an upper bound for the size of the buffers on this path, we should consider that each step can produce at most one control tag per iteration per destination control collection (otherwise, there will be multiple steps with the same timestep label). The maximum number of tags that need storage is thus $M = t_2 - t_1$ but this limit applies for all control collections on the path between the producer and consumer steps. As for the item buffer size, we need to consider all pairs t_1 and t_2 that can produce, respectively consume any item tagged t and take the maximum of the different M values obtained.

The combination of using sufficiently large buffers for item collections (L) and control buffers (M) insures that the SCnC program introduces no more deadlocks than the CnC one had. The additional restrictions imposed, except those implied by the previous chapter on streaming pattern identification, are: the existence of a constant k as outlined above and the single control-put per iteration per control collection.

4.5 Converting CnC tags to Streaming CnC tags

This section describes the conversion of item and control tags from CnC to SCnC. Tags used for item puts disappear completely, as streaming item collections allow puts only to the top of the buffer. Tags used for item gets have to refer to offsets instead of absolute tag values.

For item tag values, we present an algorithm only for consecutive integer tags$(1, 2, 3, ...)$, as these map to streaming application implementation naturally. For a get call, the offset from the top of the stack of the desired item will be the difference between the CnC tag t and the number of distinct items previously obtained from the stream. The CnC tag t will turn into $t' = t - |\bigcup_{i=0}^{consumerFunction^{-1}(t)}|$. If $t > 0$, as the implementation does not offer a classic peek operation, but a reverse peek one (access previously accessed items again), we have to do t' get operations without arguments and access the last obtained value. If it is less than 0, one can use t as a tag, offering it as parameter for the get function call.

For put calls, presuming a single monotonously increasing put function, the CnC tag parameter can safely be ignored. If there are multiple put functions, we will need dynamic checks that the streaming restriction holds, and small buffer to rearrange the put items efore they are sent in the correct order to the streaming phaser buffers.

Chapter 5

Efficient Implementation of SCnC

5.1 Use of streaming phasers

The implementation of streaming item and control collections is based on the streaming extensions to phaser accumulators, as discussed in Section 2.5. Each item or control collection has a phaser and accumulator pair that allow synchronization and communication between the producer and consumer for item collections and controller and for control collections.

The code generator creates a class with these two members. The generated collections also contains an *init* function that serves as source for item collections that are produced by the environment, as opposed to being produced by some CnC step collection. For ease of use, it is legal in our implementation to populate more than one item collection from their *init* functions, if they do not have a producer step within the graph. Because of the generation of environment produced streams inside the item collection classes, it becomes feasible to generate both the graph and the main program for the CnC application, which can be modified by the user.

The difference from the classic CnC semantics are in the put and get operations on item collections. Put operations always put the the next item in the item collection (stream) and get operations take as parameter, instead of a tag, an offset relative to the position of the item produced by the last wait operation on the phaser of the item collection. Access to elements not produced yet is not permitted except by waiting for each element up to the desired one.

The essential operations of the functions are found in listing 5.1

```
public abstract class SCnCObjectItemCollection
{
     public phaser ph;
     public accumulator a;

    public SCnCObjectItemCollection()
    {
         ph = new phaser(m, cfg);
         a = accumulator.factory.accumulator(accumulator.Operator.ANY,
             Object.class, ph);
    }

    public Object Get (int no) {
       Object value = null;
        if (no == 0) {
          ph.doWait();
          value = a.objResult();
       }
       else {
          value = a.objResult(no);
       }
       return value;
      }

    public void Put (Object p) {
       a.send(p);
         ph.signal();
     }

}
```

Listing 5.1: Item collection implementation code fragment

As described in Section 2.4, the phaser synchronization construct needs to be registered on the task that uses it. In our Habanero Java implementation, a step collection is modelled as a single *async* task containing a loop, whose iterations correspond to step instances. The implementation detail is hidden from the user through autogenerated code by the using object oriented class hierarchy. The translator creates a base abstract class for each step collection and the template for the actual user step class. The user only works with the user step class, in which he inserts code in only one function, as shown in Listing 5.2 and 5.3.

```
...
public void start(WrappedInt tag ) {
  final Tag ftag = tag;
  async phased(
    prescribingControlCollection.ph<phaserMode.WAIT>,
    producedItemCollection1.ph<phaserMode.SIG> ,
    producedControlCollection1.ph<phaserMode.SIG> ) {
    run(ftag);
  }
}
public void run(WrappedInt ptag) {
  WrappedInt tag = null;
  // if the step was started with an initial control tag,
  // use that, otherwise
  // get a new tag
  //  from the prescribing control collection
  if (ptag!=null)
    tag = ptag;
  else {
    tag = prescribingControlCollection.Get() ;
  }
  while (tag.value != prescribingControlCollection.endStream) {
    // the step function is written by the user
    step(tag);
    // get the next control tag used in the next iteration
    tag = prescribingControlCollection.Get() ;
  }
}
```

Listing 5.2: Code fragment of the abstract base class for a step

```
import Collections.*;
public class ConcreteStep extends AStep {
    ConcreteStep (SourceIdCollection prescribingControlCollection,
        SCnCIntTagCollection producedControlCollection,
        SCnCDoubleItemCollection producedItemCollection) {
         super(prescribingControlCollection, producedControlCollection,
             producedItemCollection);
    }

    public void step( WrappedInt tag ) {
      // the code in this function is written by the user
    }
}
```

Listing 5.3: User editable class for a step

5.2 Implementation of Dynamic Parallelism

In traditional streaming models, multiple iterations of a single filter do not execute in parallel. This limitation might reduce the performance unnecessarily if there are more processors available than filters. We have extended the streaming model towards an integration of streaming with task parallelism by enabling a filter to dynamically create new filters so as to allow each step to have multiple iterations executing in parallel. This behaviour is controlled through a “place” dimension of the control tags, which now become pairs of the form (placeId, old tag value). The implementation then creates a separate async for each placeId of a given control collection. As before, each of these async tasks has a loop that does the actual step computation and each of them receives all tags. The dynamic nature of phasers is helpful here, as phasers allow the number of tasks waiting at a barrier to vary dynamically. However, phasers also have the restriction that, if multiple tasks are registered as producers (signal mode), in order for the consumer to unblock from its wait state, all the producers must perform the signal call, which complicates the code generation.

Let us consider the case in which there are only 2 placeIds, 0 and 1. When there are 3 control tags (in order 1, 2 and 3), if all are assigned to a placeId 0 with tag pairs (0,1), (0,2) and (0,3), their steps will execute serially. If instead tag 2 is assigned to place 1 with tag pairs (0, 1), (1,2), (0,3), the get operation for the filter with tag 2 will succeed before the iteration corresponding to tag 1 is finished and will execute in parallel with it. Any item produced by siblings filters in different places will wait for corresponding signals from its siblings before being accessible to the consumer steps, as the underlying accumulator reduction operation needs to know all reduced items. Because the tag place id is read by all siblings and they realize that the placeid is different and the computation does not belong to them, the siblings will just signal, without computing or producing any value. This approach allows us to overcome the requirement that in a multiple producer situation, for an item to be available for consumption all producers must produce some item and the final result is the reduction of all items produced. In our case, because only one actual item is produced and the rest of the siblings produce null items, the reduction result is always trivially equal to the single item. This process is shown in Figure 5.2.

This would allow the reduction operation on the produced item to complete and the item to be consumed by its consumer step. Figure 5.2 shows how the situation would look in this case, with an initial step collection split into two sibling syncs, each processing step instances with different placeIds, but both having the same phaser registrations. Listings 5.4 and 5.5 show the structure of the SCnC step code for the

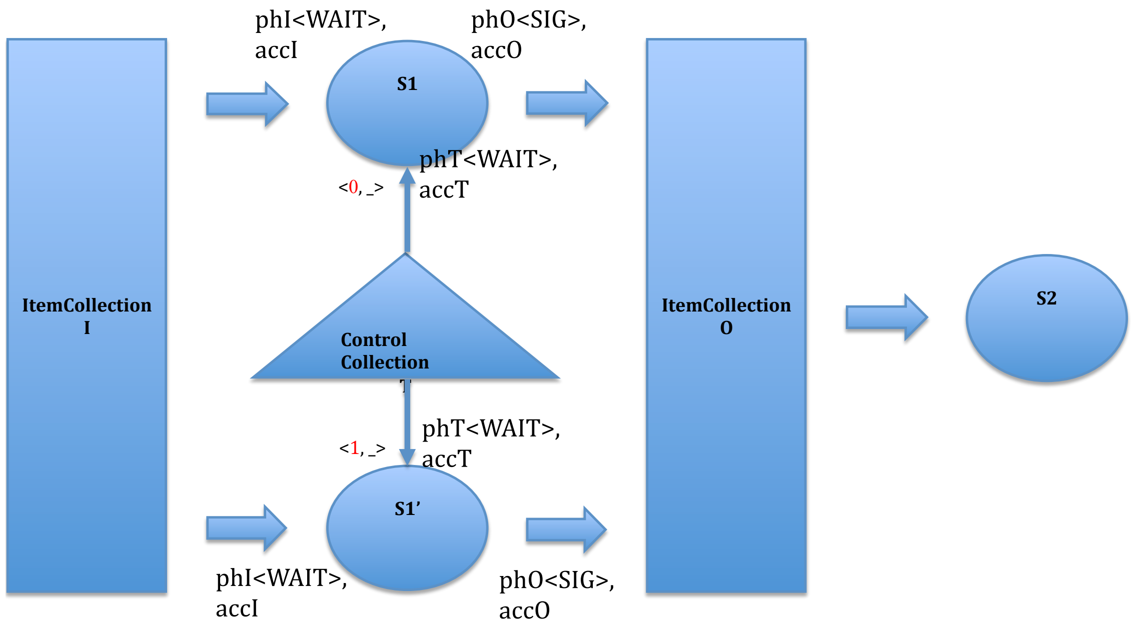

Figure 5.1 : The Streaming CnC graph with dynamic parallelism: step S1 and $S1'$ correspond to a single step collection, $S1$ processing the class of tags with placeId=0 and $S1'$ processing the class with placeId=1.

cases without and with dynamic parallelism.

To better understand how the dynamic parallelism implementation works , considering that items get produced only after all asyncs registered on the phaser perform the signal, we can analyse figure 5.2. The fake steps mentioned are obtained from the user step code by replacing the get and put with similar functions that do not perform the accumulator.put call and by removing the actual work of the step, while keeping the control flow in place, so the puts get performed only if the real user written code performs them too. These version of the step code can be automatically generated, but currently we rely on the user writing the code himself.

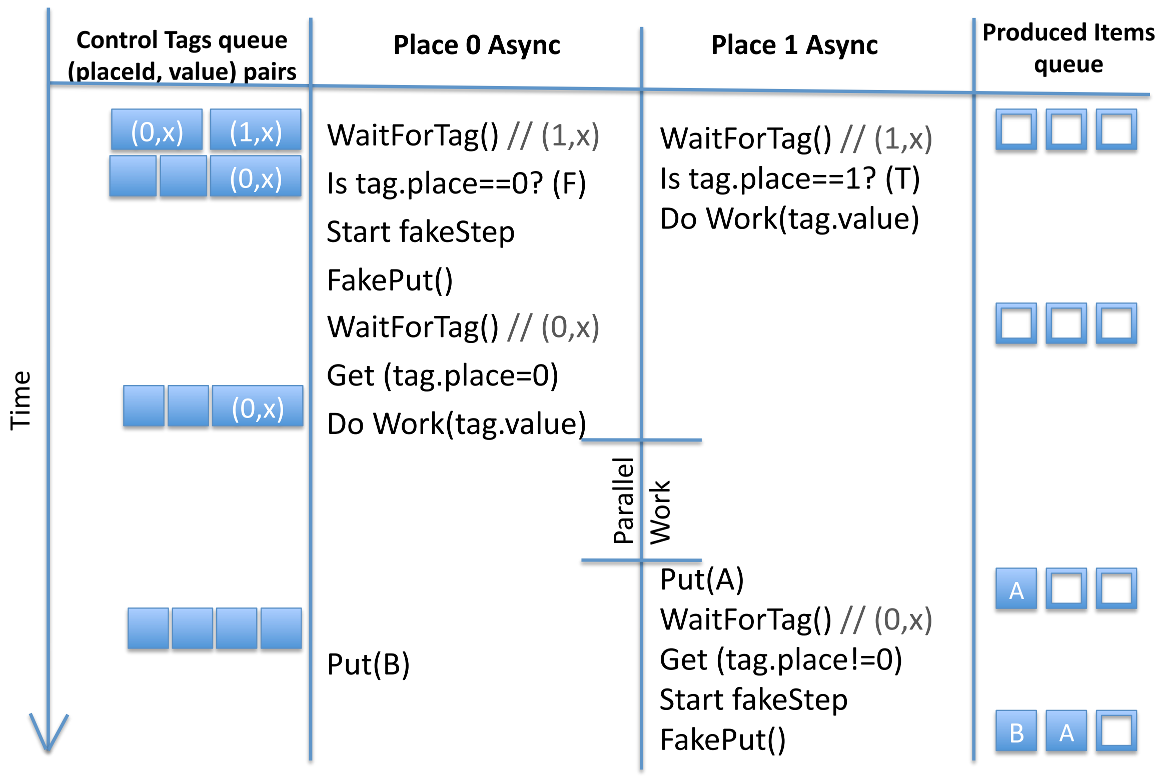

Figure 5.2 : The Streaming CnC dynamically parallel execution of a step that consumes one item and produces one item. The item is produced only after both places have signaled, but parallelism can still be exploited in this situation.

```
...
While (!end)
{
    phT1.doWait();
    tag t = accT1.get();

    // *** USER STEP CODE ***
    item i1 = phI1.get();
    result r = DoCondition(i1);
    if (cond(r) )
    {
        DoWork();
        accI2.put();
        phI2.signal();
    }
    // *** END USER CODE ***
}
```

Listing 5.4: Example code for execution of a step collection. The part between the comments is written by the user in SCnC non-dynamically parallel application. The next figure will show the additional code that will need to be generated for dynamic parallelism.

```

while (!end) {
   phT1.doWait();
   tag t = accT1.get();
   if (t.spaceTag==MySpacetag) {
      // *** USER CODE ***
      item i1 = phI1.get();
      result r = DoCondition(i1);
      if (cond(r) )  {
         DoWork();
         accI2.put();
         phI2.signal();
      }
      // *** END USER CODE ***
   }
   else {
      // *** GENERATED CODE AFTER USER CODE ***
      // DO the reads
      item i1 = phI1.get();
      result r = DoCondition(i1);
      if (cond(r) )  {
         // DO NOT  do work, DO NOT put anything
         phI2.signal();
      }
   }
}
```

Listing 5.5: Example code for execution of a step collection in a dynamically parallel SCnC application. The initial tag comparison and the code in the else branch are needed for dynamic parallelism support.

In the current implementation, the place 0 async is responsible with cloning itself to create the new async places, whenever it gets a tag whose placeId it did not previously see.

In an ideal model, the user shouldn't need to take into consideration the parallelism between the step instances that execute in different places; he could write the code ignoring the control placeIds and the compiler would do generation of the proper code for each place. This requires analysis and transformation on the step code similar but more complicated than dead code elimination, as we need to keep the feature that each step might have a variable number of produced items. The analysis would consist of labelling the code that computes the value of the items and tags produced by a step with the label COMPUTATION and identifying the code required to decide if the *put* operations will be performed and labelling it with CONTROL. The transformation phase would remove the code that is labelled COMPUTATION but is not labelled CONTROL and replace the put operations with signal operations.

Furthermore, to maintain support for local step fields whose value is accessed between iterations, it would be required to also label with CONTROL any code that decides the execution of instructions that update fields; synchronization for the fields would have to be added.

Right now, the implementation supports explicit place management: steps that support places have to make sure they do not execute work that is assigned to steps in other places (that happens if the placeId of the input tag does not correspond to the placeId of the step) and they have to make sure they signal to the proper output collection when this happens. Note that new tasks are spawned automatically when a new placeId is encountered.

A further extension to the model would be automatic tag generation: the runtime could detect that cores are underutilized and assigned placeIds automatically to take advantage of the possible parallelism.

Chapter 6

Results

6.1 Implementation status

The complete workflow for a CnC programer who wants to take advantage of SCnC is presented in Figure 6.1. The initial CnC graph has to be transformed to the SCnC well formed shape, generating a SCnC graph description. The CnC code has to be adapted to the semantics of SCnC get and put operations, thereby obtaining SCnC step code. Both these transformation steps, corresponding to algorithms presented in this thesis, have not been implemented as yet and were performed manually to obtain the results in this thesis. Their output is the complete SCnC application. After compiling the hand-transformed code, we can run the application using the Habanero Java infrastructure and the SCnC runtime. The runtime, as well as the code generator from an SCnC graph and SCnC runtime were implemented as part of this thesis.

6.2 Testing methodology

The SCnC translator and runtime have been tested on three of the applications from the StreamIt project, in particular BeamFormer, FilterBank and FMRadio, as well as a clustering application, FacilityLocation and the well known mathematical algorithm Sieve of Eratosthenes algorithm.

The initial implementation of the applications was done in CnC. Transferring this

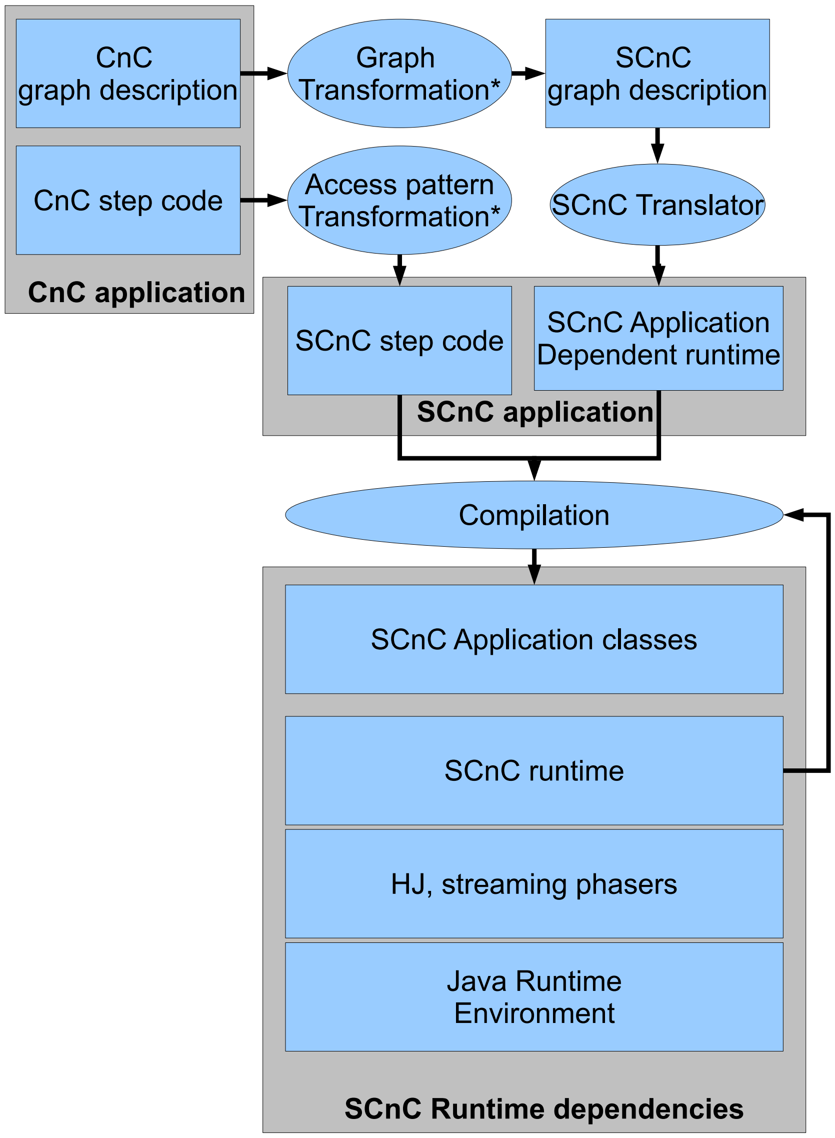

Figure 6.1 : The workflow for using a CnC application for SCnC. The only manual transformations are marked with a star, the rest are automatic.

implementation to SCnC helped to validate our algorithm for transforming a CnC graph to a SCnC graph, and to test if the resulting graph satisfies the constraints of a streaming application. The experimental results have been encouraging for SCnC, with increases in throughput of up to 40× compared to the CnC performance. In addition, SCnC showed it can support larger problem sizes compared to the CnC implementation. The Habanero Java implementation for CnC used a work sharing scheduling policy, and the number of workers for all CnC results was set to match the numer of cores of the machine. The performance results were also helpful in evaluating the performance overhead of the SCnC wrapper over the optimized streaming phasers implementation, which are included in the performance comparisons as well.

The tests have been performed on 2 different systems: a dual core Intel i5 2.6GHz system with 4GB RAM, and a system with 4 quadcore Xeon processors and 16GB RAM. The performance analysis focuses on throughput comparison of SCnC and CnC.

6.3 Applications

6.3.1 FilterBank

The FilterBank application implements a filter bank for signal processing [6]. On each parallel branch, delay, filter, and downsample steps are performed and followed by upsample, delay, and filter steps, but the implemented versions merged many of the individual components in higher level tasks. The application needed no modification of the CnC graph to be applied in order to run on the streaming runtime and offer increased performance. A partial graphical representation of the graph is in Figure 6.3.1 (some names have been omitted for lack of space). As seen in Table 6.1, on

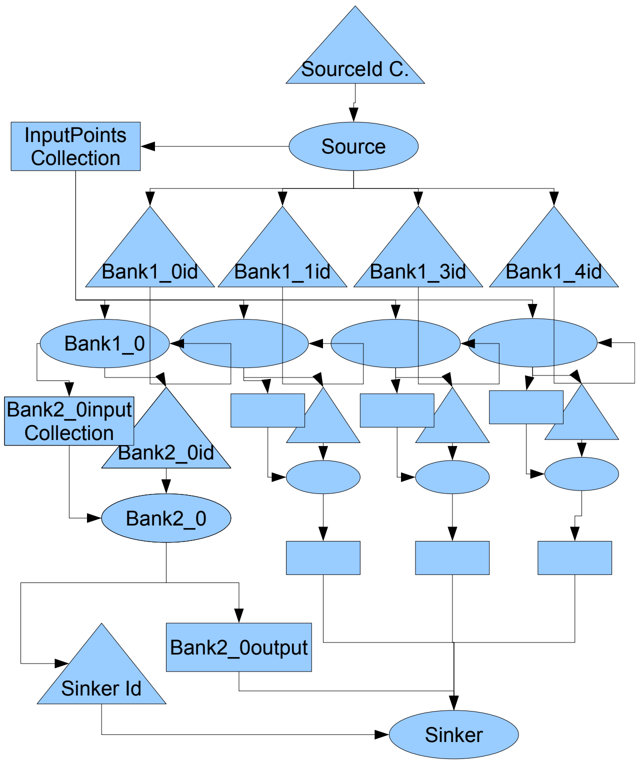

Figure 6.2 : The SCnC graph for the FilterBank application is identical for SCnC

the i5, the throughput increase compared to CnC is around 4.89x and the overhead compared to streaming phasers is under 3x. On the Xeon, the throughput increase compared to CnC is 10x but the streaming phasers throughput is 18 times better.

6.3.2 BeamFormer

The BeamFormer application performs beam forming on a set of inputs[6]. The version we implemented has deterministic output ordering and 4 parallel beams. For

Input size	**Model**	**Execution Time (s)**	**Throughput (items/s)**
50,000	CnC	45	1111
1,000,000	CnC	OOM	OOM
1,000,000	SCnC	184	5434
1,000,000	Streaming phasers	76	13157

Table 6.1 : SCnC, CnC and streaming phasers performance for FilterBank (Core i5)

Input size	**Model**	**Execution Time (s)**	**Throughput (items/s)**
50,000	CnC	44	1136
50,000	SCnC	9	5555
5,000,000	CnC	OOM	OOM
5,000,000	SCnC	400	12500
5,000,000	Streaming phasers	34	147058

Table 6.2 : SCnC, CnC and streaming phasers performance for FilterBank (Xeon)

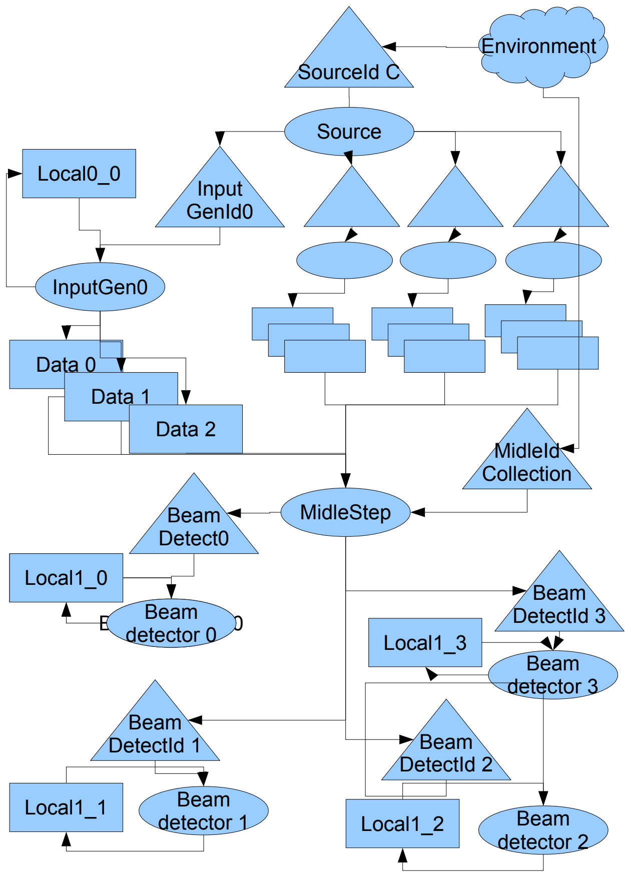

Figure 6.3 : The CnC graph for BeamFormer application

the conversion to streaming, we had to make changes to the environment, because it interacted with the graph using more than one tag collection. Also, local data collections from the CnC implementation, which had been modelled in the StreamIt version with local filter state, were returned to local step state, as Figure 6.3.2 and 6.3.2 show (some names have been omitted for lack of space).

The performance results on Xeon (Table 6.3) showed an increase in throughput of 75x compared to CnC and a 2.4x slowdown of throughput compared to streaming

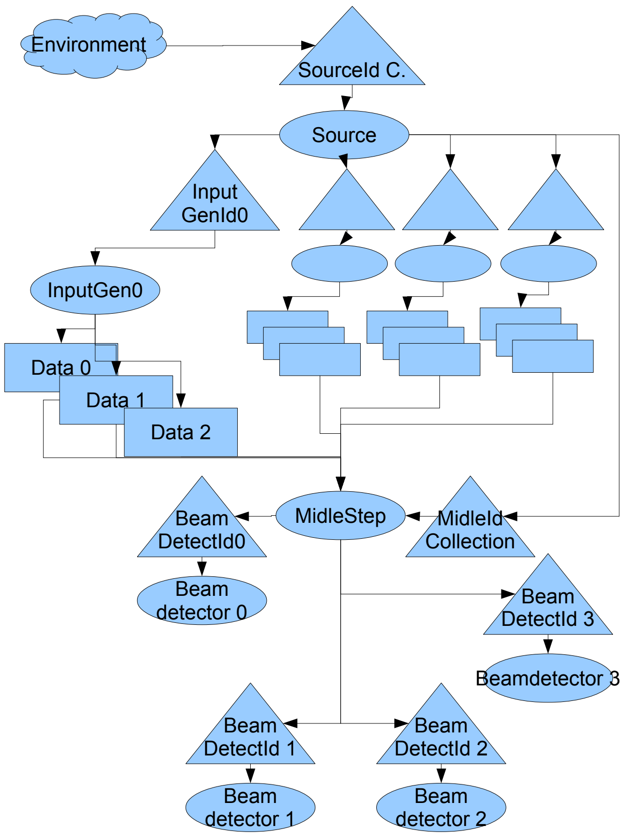

Figure 6.4 : The StreamingCnC graph for BeamFormer application

Input size	**Model**	**Execution Time (s)**	**Throughput (items/s)**
30,000	CnC	60	500
30,000	SCnC	11	2727
3,000,000	CnC	OOM	OOM
3,000,000	SCnC	140	20270
3,000,000	Streaming phasers	51	58823

Table 6.3 : SCnC, CnC and streaming phasers performance for Beamformer (Xeon)

Input size	**Model**	**Execution Time (s)**	**Throughput (items/s)**
30,000	CnC	67	447
3,000,000	CnC	OOM	OOM
3,000,000	SCnC	215	1395
3,000,000	Streaming phasers	41	7317

Table 6.4 : SCnC, CnC and streaming phasers performance for Beamformer (Core i5)

phasers.

6.3.3 FMRadio

The FMRadio application is another application from the StreamIt benchmark suite. The SCnC performance results on the Xeon machine for this application are shown in Table 6.5.

Input size	**Model**	**Execution Time (s)**	**Throughput (items/s)**
100,000	CnC	102	980
100,000	SCnC	3.4	29411
1,000,000	CnC	OOM	OOM
1,000,000	SCnC	29	34482
1,000,000	Stream phasers	5	200000

Table 6.5 : SCnC, CnC and streaming phasers performance for FMRadio (Xeon)

6.3.4 Facility Location without dynamic parallelism

The facility location application is a clustering application that solves the problem of optimum placement of production and supply facilities depending on an input stream of customer locations.

Formally[16], we are given a *metric space* and a *facility cost* for each node as well as a stream of *demand points*. The problem is finding the minimum number and positioning of nodes such that it minimizes a metric space expression dependent on which demand points are assigned to a node. This problem occurs in several fields such as strategic placement of production facilities, networking and communication, document classification.

For example, consider the creation of a network, where servers have to be purchased and clients assigned to the servers in the order they arrive by purchasing cables. Once the demand gets too high, new servers have to be purchased and at the same time the costs should be kept as close to the minimum as possible. A similar example is the webpage clustering problem: pages can have to be assigned to clusters according to some attributes. As the web grows rapidly, new pages have to be classified and servers brought in to handle the load.

As the problem is relevant to many fields, different formulations and approaches for solving it exist: two level [17], various hierarchical approaches [18], online and incremental [19]. There is also an offline formulation of the FacilityLocation problem [20]. Both these versions are in fact streaming problems because of their dynamic nature in which new data arrives constantly (online) a working solution is expected at every point in time (incremental).

The online, incremental (streaming) approaches to solving this problem do not find an optimal solution, but instead offer at any point in time a solution that is at most a constant factor worse than the best one when points come in random order, guarantee due to probabilistic reasoning. Against an adversarial opponent, no online solution can be O(1) away from the optimum [16]. We have implemented the randomized algorithm in [16] for its simplicity. Our implementation takes advantage of the dynamic parallelism feature of SCnC in the sense that each place represents a cluster and the async corresponding to a place updates the metrics for points assigned only to that cluster only.

The results on the Core i5 system are summarized in Table 6.6 and show throughput increases of 5x compared to CnC. On the 16 core Xeon, Table 6.7 shows that the speedup obtained is 3.6x of the CnC performance. For higher CnC input sizes the garbage collection time starts to dominate the execution time, so the speedup listed is expected to increase on larger inputs. Also, buffer sizes for these experiments have been kept under 1/1000 of the input size(1000) and the speedup grows as the buffer size increases.

Input size	**Model**	**Execution Time (s)**	**Throughput (items/s)**
30,000	CnC	71	8450
3,000,000	CnC	OOM	OOM
3,000,000	SCnC	69	43478
3,000,000	Streaming phasers	21	142854

Table 6.6 : SCnC, CnC and streaming phasers performance for Facility Location (Core i5 system)

Input size	**Model**	**Execution Time (s)**	**Throughput (items/s)**
300,000	CnC	54	5454
3,000,000	CnC	OOM	OOM
3,000,000	SCnC	150	20000
3,000,000	Streaming phasers	65	46154

Table 6.7 : SCnC, CnC and streaming phasers performance for Facility Location (Xeon)

6.3.5 Facility Location with dynamic parallelism

The facility location application is interesting because if shows the potential for dynamic parallelism. We created a SCnC implementation to take advantage of this feature and we studied the speedup that we obtained as a result.

As a academic benchmark, we did not add any code that computes per cluster statistics such as a real world application might (total length of cabling needed for a server, total cost of cabling, average distance of consumers from the server, etc). Because of this, the clustering time is similar to the statistics computation time. In such a producer/consumer example, speedup by running multiple consumers in parallel is not possible, as they would block waiting for input to consume. We modelled the computation of such statistics by adding artificial wait times for consumers: several runs were performed with increasing time intervals up to 1ms of delay added to every 12th point of the input stream. The additional time is small, but it is enough to show some scalability of the parallel implementation of FacilityLocation. Higher values might correspond better to real world implementation but we decided to be conservative in our analysis. The results for input of size 10,000,000 on the 16 core Xeon are presented in Table 6.8 and get us an additional speedup of 3.4 for a consumer delay of 0.83 ms on average.

6.3.6 Sieve of Eratosthenes with dynamic parallelism

The Sieve of Eratosthenes is an algorithm for finding the prime numbers, attributed to the ancient Greek mathematician Eratosthenes. Our implementation is a dynamic split-join with feedback loop. There is one producer that streams in consecutive number starting at 2; the numbers are then sent to several parallel filters that check if the number is divisible with any of the prime numbers that each filter stores. If a

Delay	**SCnC time (s)**	**SCnC dynamic parallelism time (s)**	**Speedup**
none	90	94	0.95
1ms every 50th	212	101	2.1
1ms every 25th	414	131	3.16
1ms every 12th	857	250	3.4

Table 6.8 : SCnC dynamic parallelism execution time, compared to the SCnC implementation, 16 core Xeon

Variant	**SCnC time (s)**	**SCnC Dynamic parallelism time (s)**	**Speedup**
M=N	238	40	5.95
M=2*N	863	80	10.78

Table 6.9 : SCnC dynamic parallelism execution time compared to SCnC without dynamic parallelism on the 16 core Xeon system, N= 1,000,000

filter finds a divisor, it sends to the join node a 1, if not, it sends 0. The join node performs an accumulator reduction with the operation SUM on the results and if the result is 0, the number is prime. It then sends back to the filters the id of the filter that should add the newly discovered prime number to its prime number store. The CnC graph of the application is in Figure 6.3.6.

If in Facility Location the algorithm decides how many clusters there are, for Sieve we can tune the number of dynamic filters to the number of cores in the machine. Performance results are found in Table 6.9 for the 16 core Xeon machine, 15 filters and a cyclic distribution of primes to the filters.

Another possible implementation is the dynamic pipeline, shown in figure 6.3.6, in

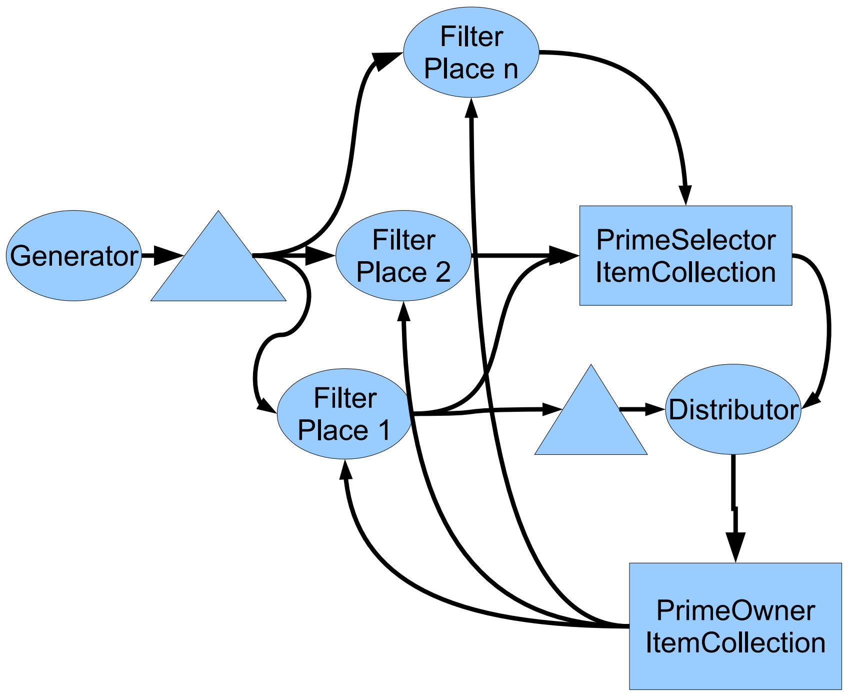

Figure 6.5 : The dynamic StreamingCnC graph for Sieve application.

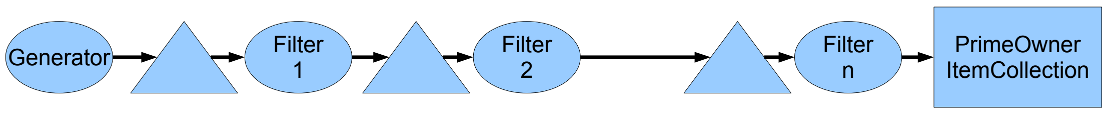

Figure 6.6 : The dynamic StreamingCnC graph for Sieve application pipeline version.

which filters are connected serially to one another. Each filter accumulates numbers that are not divisible with any prime number in its store up to a maximum, after that it spawns another filter that attaches to the end of the pipeline and from the on receives all numbers that are prime relative tot he numbers stored by the previous filters. This approach has a dynamic parallelism that is not easily controlled for an unknown number of primes: if we want to never have more stages in the pipeline than cores and an equal distribution of primes to asyncs, there is no easy way to accomplish this even if the cyclic distribution in the dynamic split-join case does this naturally. Crude control can be had through the constant that controls the maximum number of primes per filter.

However, SCnC is not able to express pipeline dynamic parallelism, as only dynamic split-joins are supported. This version offers simpler implementation with only a single wait call per input, compared to for the split-join implementation that needs one wait to get the input, one to receive the answer if it should add the number to the store or not. This lower latency of the filter should lead to an increased throughput compared to the split-join implementation. Comparing the two is difficult in the general case, as the pipeline length might not be in the parallelism sweet-spot of the machine(ie equal to the number of cores). On the other hand, when there are few primes already discovered, the split-join might not have enough work to justify the existence of all filters, and furthermore the latency should be higher.

We compared the two implementations, both using handcoded streaming phasers. We also implemented an extension of the Sieve that not only finds the prime numbers up to N, but also counts the numbers between N and M that are not divisible by any prime number less than N. This extension allows us to analyse the speedup of the split-join pattern without the overhead of variable granularity and added feedback synchronization. The results, published in [14] confirm that the speedup for the pipeline version of Sieve are consistently better than the split-join, specially if using the extension (it reaches 9.8× when compared to a optimized sequential execution). This motivates us to continue this work by incorporating support for dynamic pipline execution, that would offer complete dynamic parallelism support for both streaming basic blocks.

The speedup of the SCnC split-join implementation on the 16 core Xeon obtained for M = 2*N, N = 1,000,000 increases up to 10.8x, when compared to the streaming SCnC version without dynamic parallelism and overhead under 10%. For this application the overhead is smaller as the there was no need for a separate control tag stream, as opposed to the other applications. It is a good result, but, considering the behaviour of the streaming phasers implementation, we are confident the pipeline implementation will offer more in the future.

Chapter 7

Related work

StreamIt [15, 21] is the most notable recent exemplar of streaming languages. Its contribution was an efficient implementation of streaming for programs that can be expressed using a simple set of streaming primitives. StreamIt is a compelling alternative for writing streaming applications that were otherwise written using general purpose programming languages such as C: it replaced the error-prone low level time expensive programming process with an efficient portable readable and robust higher level streaming language. The project lead to the publication of a streaming benchmark set and a characterization of the streaming applications identified during the research [22].

StreamIt programs use basic operations push, pop, peek that are also available in SCnC (put, get, get(k)), but lack a meta-description of tasks as in the CnC specification; thus, StreamIt lacks the ability to execute applications that are partially streaming. Note that streaming parallelism in general is a combination of pipeline, data and task parallelism, so StreamIt does take advantage of task parallelism in structures such as split-join, but there are no features that can express task parallelism in which data does not respect the streaming paradigm that value only have a lifetime; CnC and SCnC offer the same model for task-parallel and streaming applications. In fuuture work we plan to integrate both runtimes so that the task parallel and streaming components can run on the joint CnC and SCnC runtime. The same lack of meta-description of task interaction might make larger Streamit programs difficult to understand. The likely appeal for SCnC by programmers is that by learning one language (CnC), they can implement both streaming and non-streaming applications.

As mentioned in Section 3.3, the split-join distributions in SCnC are more general than StreamIt, which only offers round robin, weighted round-robin and duplicate/-combine as distributions and join operations. This makes sense, as StreamIt expects filters to have a fixed input/output rate and such fixed distributions are sufficient. In SCnC, we allow variable flow rate and join and split nodes are explicit filters themselves and are able to perform any custom join or distribute operation.

As we decouple the control and data dependencies in control tag (control collection) streams and data (item collection) streams, we can more easily express some streaming shapes that StreamIt does not allow. We relax the restriction that filters have one input stream through the existence of both control and data streams and through our ability of receiving multiple data streams as input for any filter. The decoupling between the data and control can be used to emulate StreamIt's message passing [23], by making the signaler the control producer of the filter.

The StreamIt approach is based on static analysis and program transformation, whereas SCnC is runtime based. The SCnC implementation does not contain any of the static analysis/optimizations that StreamIt performs such as granularity coarsening, data parallelism exploitation and software pipelining- though SCnC has other features that perform similar roles.

We rely on the batching optimizations performed by the streaming phasers primitive instead of doing a StreamIt-like scheduling optimization. Our dynamic parallelism feature has the same goal as StreamIt filter fission, but the StreamIt approach allows only for a fixed number of branches for spit joins. If we know the number of available cores in the system ahead of time, the StreamIt approach is sufficient. However, the StremIt approach does not work well statically, because the same executable may be invoked on different machines. Also, the StreamIt approach only allows parallelization of stateless filters, whereas SCnC allows parallel copies of stateful filters to keep individual state.

StreamIt's orchestration of filters is performed using greedy scheduling algorithms that are guaranteed to be deadlock-free [24]. More recent work by Manjunath Kudlur [25] shows integer linear programming might be an effective alternative to greedy schedulers.

Other projects work towards adjusting the streaming model to work with new architectural features or accelerators: GPUs [26] and FPGAs [27]. Scratchpad memories and their use in streaming is the subject of [28] where the authors use integer linear programming to balance computation. Implementations of StreamIt for the Cell BE processor have been shown to be efficient [29]. For multicore, software pipelineing is used to generate streaming-like programs [30]. Efficient usage of the task, data and pipeline parallelism models on multicore architectures by means of streaming is shown in recent work by Michael Gordon [31].

Brook [32, 33] takes a different approach to managing granularity: if StreamIt uses fission and fusion to get to a steady schedule starting from fine grained operations, Brook exposes only coarse grained multi-dimensional data structures (called streams) to the programmer who is expected to process them through predefined operators. Using stream shape analysis they end up performing kernel fusion and optimizations similar to loop interchange. They target both multiprocessors and GPU systems.

The integration between the streaming model and architectures sometimes reached the point in which architectures are built for particular streaming applications. The tool proposed by Nikolaos Bellas et al[34] automatically generates the design of accelerators used in conjunction with system on chips to run streaming applications.

The problem of finding a better task mapping for stream programs has been tackled for years [35]. Static mapping together with dynamic adjustments for load balancing have been implemented and have shown good performance on a Cell BE system [36]. Recently, dynamic approaches have become possible because of work performed on execution time prediction for streaming tasks. The dynamic parallelism approach we propose complements several projects that aim to offer tailored load balancing for streaming applications. Farhana Aleen et al. [37] use taint analysis and simulation to identify pipeline delays as a function of input data. They could add dynamic split-join patterns to complement their analysis to get dynamic load balancing optimization. The difficulty in such orchestration of streaming programs is maintaining accuracy while keeping a low overhead,but the results are encouraging — an improvement of up to 38% with dynamic load balancing compared to static load balancing. At least for large graphs, Sardar Farhad was able to show [38] that approximate algorithms might offer better performance compared to integer linear programming techniques for large graphs. Machine learning techniques hve also shown good results for partitioning streaming graphs [39].

The technique of sending special messages through the streams is used in other projects, to obtain deadlock freedom. A current area of research is lowering the overhead of such messages by identifying the frequency or moments when they should be sent, as Pend Li et al.[40] show.

Comparing the dataflow model performance with streaming, and comparing the performance of using the streaming versus task based implementations of dataflow was started in [41]. Their work relies on a special language and the comparison with data-flow can only be taken as a guideline, as their dataflow implementation is not not based on a special streaming or data-driven implementation, relying on the general Cilk model for short-lived tasks. For a single synthetic benchmark, their use of different input language representations shows there is a lot of room for improvement. Furthermore, the results for the benchmark they propose are not entirely positive for their system. With SCnC, we show that consistently better results are possible for a larger number of real applications, even without using a custom-built language, compiler and intermediate representation while retaining a determinism guarantee. Furthermore, we start with the general CnC model with a task-based implementation that can offer best-of-breed performance [42].

Automatic "streamization" is usually used in projects targeting new architectures such as GPUs or scientific stream processors, such as FT64 [43]. Steps towards our goal of automatic streamization of programs for multicore processors have been taken by GCC [44] by "transforming loops into concurrent pipelines of concurrent tasks that use streams to communicate and synchronize". The differences between this work and our project include their compiler based approach, their restrictive use cases and finer granularity and their approach of using serial code as input compared to parallel code.

Recent extensions to the OpenMP model show how some programs, such as BZip are difficult to parallelize efficiently using a task based model [45], unless exploiting pipeline parallelism. Using additional annotations to allow the communication through FIFO queues between tasks, the performance can be drastically improved for several programs [46], providing yet another motivation for integrating streaming with task parallelism.

Chapter 8

Conclusion and future work

8.1 Conclusion

This work has established that integrating task and streaming parallelism by using the same high level modelling language, such as Concurrent Collections, is both possible and profitable in both time and space. The results show good speedup for the applications studied, as well as reductions in memory footprint.

We also show that converting task based parallelism to streaming need not be difficult (when a solution exists) and we give an algorithm that can help with this task, both at the graph level (CnC specification) and at the implementation level (get/put parameters).

We also propose using places for dynamic parallelism in slit-join nodes, as an additional way to obtain performance, while maintaining an abstraction level close to the Concurrent Collections model.

8.2 Future Work

In addition to the current SCnC model, it would be interesting to add support for dynamic pipeline parallelism. Offering both split-join and pipeline parallelism would complete our dynamic parallelism work for the streaming model. This possible extension is needed to better implement applications such as the Sieve of Eratosthenes which offers even better performance for pipeline implementations.

The integration of the task based and streaming runtimes inside a single common runtime that can adaptively decide which approach to use for which program component is an interesting direction of research: we can imagine algorithms that automatically analyse a CnC specification, detect if parts of it can be converted to streaming parallelism and if it is profitable to do such a conversion. This integration would lead to faster programs and less worry for the performance expert who tunes the application.

Extensions of the algorithms that identify if an application can be used with the streaming runtime would help streaming reach more applications. Right now some streaming patterns are not accepted (such as multiple consumers when all the consumers consume the same data). There are subtle conditions here that may cause deadlocks in the streaming case that would not appear in a task based implementation, thereby requiring that we avoid those conditions.

Implementing more applications and auto-generating the dynamic parallelism runtime code are other notable directions for future work.

Bibliography

[1] X. Teruel, C. Barton, A. Duran, X. Martorell, E. Ayguadé, P. Unnikrishnan, G. Zhang, and R. Silvera, "Openmp tasking analysis for programmers," in *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '09, (New York, NY, USA), pp. 32–42, ACM, 2009.

[2] T. Willhalm and N. Popovici, "Putting intel&#174; threading building blocks to work," in *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, (New York, NY, USA), pp. 3–4, ACM, 2008.

[3] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," in *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, (New York, NY, USA), pp. 227–242, ACM, 2009.

[4] B. C. Kuszmaul, "Cilk provides the "best overall productivity" for high performance computing: (and won the hpc challenge award to prove it)," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, (New York, NY, USA), pp. 299–300, ACM, 2007.

[5] R. e. a. Barik, "The habanero multicore software research project," in *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, (New York, NY, USA), pp. 735–736, ACM, 2009.

[6] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, (London, UK), pp. 179–196, Springer-Verlag, 2002.

[7] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, (Los Alamitos, CA, USA), pp. 3–13, IEEE Computer Society Press, 1998.

[8] Z. Budimlic, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. M. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[9] J. Jisheng, Zhao andShirako and V. Sarkar, "Habanero-java: the new adventures of old x10," 9th International Conference on the Principles and Practice of Programming in Java (PPPJ).

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, (New York, NY, USA), pp. 207–216, ACM, 1995.

[11] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1 –12, may 2009.

[12] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, “Phasers: a unified deadlock-free construct for collective and point-to-point synchronization,” in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS ’08, (New York, NY, USA), pp. 277–288, ACM, 2008.

[13] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, “Phaser accumulators: A new reduction construct for dynamic parallelism,” in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2009.

[14] J. Shirako, D. M. Peixotto, D. Sbirlea, and V. Sarkar, “Phaser beams: integrating task and streaming parallelism,” in *X10 workshop colocated with PLDI*, 2011.

[15] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, “A stream compiler for communication-exposed architectures,” in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, (New York, NY, USA), pp. 291–303, ACM, 2002.

[16] A. Meyerson, “Online facility location,” in *Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, FOCS ’01, (Washington, DC, USA), pp. 426–, IEEE Computer Society, 2001.

[17] J. Zhang, “Approximating the two-level facility location problem via a quasi-greedy approach,” in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA ’04, (Philadelphia, PA, USA), pp. 808–817, Society for Industrial and Applied Mathematics, 2004.

[18] G. Şahin and H. Süral, "A review of hierarchical facility location models," *Comput. Oper. Res.*, vol. 34, pp. 2310–2331, August 2007.

[19] D. Fotakis, "Online and incremental algorithms for facility location," *SIGACT News*, vol. 42, pp. 97–131, March 2011.

[20] M. Charikar, C. Chekuri, T. Feder, and R. Motwani, "Incremental clustering and dynamic information retrieval," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, (New York, NY, USA), pp. 626–635, ACM, 1997.

[21] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies, "Language and compiler design for streaming applications," *Int. J. Parallel Program.*, vol. 33, pp. 261–278, June 2005.

[22] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 365–376, ACM, 2010.

[23] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, "Teleport messaging for distributed stream programs," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, (New York, NY, USA), pp. 224–235, ACM, 2005.

[24] M. Karczmarek, W. Thies, and S. Amarasinghe, "Phased scheduling of stream programs," in *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, (New York, NY, USA), pp. 103–112, ACM, 2003.

[25] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, (New York, NY, USA), pp. 114–124, ACM, 2008.

[26] P. M. Carpenter, A. Ramirez, and E. Ayguade, "Mapping stream programs onto heterogeneous multiprocessor systems," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '09, (New York, NY, USA), pp. 57–66, ACM, 2009.

[27] A. Hagiescu, W.-F. Wong, D. F. Bacon, and R. Rabbah, "A computing origami: folding streams in fpgas," in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, (New York, NY, USA), pp. 282–287, ACM, 2009.

[28] W. Che, A. Panda, and K. S. Chatha, "Compilation of stream programs for multicore processors that incorporate scratchpad memories," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, (3001 Leuven, Belgium, Belgium), pp. 1118–1123, European Design and Automation Association, 2010.

[29] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe, "A lightweight streaming layer for multicore execution," *SIGARCH Comput. Archit. News*, vol. 36, pp. 18–27, May 2008.

[30] A. Douillet and G. R. Gao, "Software-pipelining on multi-core architectures," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, (Washington, DC, USA), pp. 39–48, IEEE Computer Society, 2007.

[31] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, (New York, NY, USA), pp. 151–162, ACM, 2006.

[32] S.-w. Liao, Z. Du, G. Wu, and G.-Y. Lueh, "Data and computation transformations for brook streaming applications on multiprocessors," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, (Washington, DC, USA), pp. 196–207, IEEE Computer Society, 2006.

[33] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," in *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, (New York, NY, USA), pp. 777–786, ACM, 2004.

[34] N. Bellas, S. M. Chai, M. Dwyer, and D. Linzmeier, "Mapping streaming architectures on reconfigurable platforms," *SIGARCH Comput. Archit. News*, vol. 35, pp. 2–8, June 2007.

[35] Y. A. M. U. H. Mori, "A case study on predictive method of task allocation in stream-based computing," in *Proceedings of the 13th International Conference on Information Networking*, ICOIN '98, (Washington, DC, USA), pp. 0316–, IEEE Computer Society, 1998.

[36] R. L. Collins and L. P. Carloni, "Flexible filters: load balancing through backpressure for stream programs," in *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, (New York, NY, USA), pp. 205–

214, ACM, 2009.

[37] F. Aleen, M. Sharif, and S. Pande, “Input-driven dynamic execution prediction of streaming applications,” in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP ’10, (New York, NY, USA), pp. 315–324, ACM, 2010.

[38] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz, “Orchestration by approximation: mapping stream programs onto multicore architectures,” in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS ’11, (New York, NY, USA), pp. 357–368, ACM, 2011.

[39] Z. Wang and M. F. O’Boyle, “Partitioning streaming parallelism for multi-cores: a machine learning based approach,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT ’10, (New York, NY, USA), pp. 307–318, ACM, 2010.

[40] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain, “Deadlock avoidance for streaming computations with filtering,” in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA ’10, (New York, NY, USA), pp. 243–252, ACM, 2010.

[41] C. Miranda, A. Pop, P. Dumont, A. Cohen, and M. Duranton, “Erbium: a deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes,” in *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, CASES ’10, (New York, NY, USA), pp. 11–20, ACM, 2010.

[42] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, “Performance evaluation of concurrent collections on high-performance multicore computing systems,” in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, april 2010.

[43] X. Yang, J. Du, X. Yan, and Y. Deng, “Matrix-based streamization approach for improving locality and parallelism on ft64 stream processor,” *J. Supercomput.*, vol. 47, pp. 171–197, February 2009.

[44] M. P. Antoniu Pop, C. D. R. E. Informatique, and M. E. Systmes, “Automatic streamization in gcc.”

[45] V. Pankratius, A. Jannesari, and W. F. Tichy, “Parallelizing bzip2: A case study in multicore software engineering,” *IEEE Softw.*, vol. 26, pp. 70–77, November 2009.

[46] A. Pop and A. Cohen, “A stream-computing extension to openmp,” in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC ’11, (New York, NY, USA), pp. 5–14, ACM, 2011.