

IST ACOTES Project
Deliverable D2.1

Report on Streaming
Programming Model and
Abstract Streaming
Machine Description 1st
version

Project Number	:	IST-034869
Project Title	:	Advanced Compiler Technologies for Embedded Streaming
Deliverable Type	:	Report

Deliverable Number	:	D2.1
Title of Deliverable	:	Report on Streaming Programming Model and Abstract Streaming Machine Description 1 st version
Nature of Deliverable	:	Report
Internal Document Number/version	:	1.0
Contractual Delivery Date	:	28.02.2007
Actual Delivery Date	:	15.04.2007
WP(s)	:	WP 2 Streaming Programming / Abstract Machine
Author(s)/Affiliation	:	Paul Carpenter/UPC, Alex Ramirez/UPC, Xavier Martorell/UPC, David Rodenas/UPC, Roger Ferrer/UPC

Abstract

This deliverable presents the preliminary descriptions of the Streaming Programming Model (SPM) and the Abstract Streaming Machine (ASM).

Keyword list

ASM Abstract Streaming Machine
SPM Streaming Programming Model
WP Work Package

Table of Contents

Table of Contents	2
1 Introduction.....	4
2 Streaming Programming Model.....	5
2.1 Overview	5
2.2 Pragmas	5
2.2.1 Streaming pragmas	5
2.2.2 Pragmas supporting optimizations	7
2.2.3 Other auxiliary pragmas	9
Update asynchronous state variables	9
Setting the requirements of tasks.....	10
2.2.4 Work in progress	10
Task naming	10
Pragmas supporting parallelism.....	10
2.3 Extended C supporting the programming model.....	10
2.3.1 Data structures representing streams	10
2.3.2 Description of the streaming interface.....	11
Input stream interface.....	11
Output stream interface	11
Auxiliary functions.....	12
2.3.3 Merging directives and Extended C	12
3 Abstract Streaming Machine.....	14
3.1 Introduction to the ASM.....	14
3.2 Modeling the platform	15
3.2.1 Overview	15
3.2.2 Static routing	15
3.2.3 Processor and link definitions	17
3.3 Modelling the Streaming Program	18
3.3.1 Overview	18
3.3.2 Simple tasks.....	18
3.3.3 Modelling irregular tasks.....	18
3.3.4 Modelling variable computation time and pattern of communication.....	19
3.3.5 Data dependent scheduling of subtasks and control flow graph	25
3.4 Defining the mapping and scheduling	27
3.5 Integration with the compiler.....	28
4 Summary	30
References.....	31

1 Introduction

The Acotes project is defining a Streaming Programming Model (SPM) and an Abstract Streaming Machine (ASM) making possible to develop parallel streaming applications, and analyze their performance for several target architectures. The SPM will allow to express the specific parallel processing features of streaming data applications, and the ASM is to provide a target architecture description to be used by the compiler.

We aim to efficiently deal with intrinsically parallel applications such as digital video, software radio, multimedia and 3D graphics. Compilers should face the new reality by offering developers the ability to write portable efficient code for streaming applications on parallel platforms.

We consider a streaming machine as multiple independent processors communicating and synchronizing primarily via one-dimensional streams of data [Lee87]. The programming model is based on pragma extensions to bridge the gap between hardware and software in multicore processors oriented towards streaming data applications.

We believe that a stream programming model is most likely to be adopted by the mainstream if it is possible to incrementally modify an existing sequential application into a streaming one. Our stream programming model consists of an annotated version of the C programming language, which uses a set of new directives, implemented as pragmas. We have proposed two basic directives, which define the streaming environment and the parallel tasks within it, also specifying the tasks' inputs and outputs as streams. The compiler analyzing the pragmas will generate code onto the Extended C interface, to deal with building the streams environment, and managing them to transfer the data. We foresee that the programmer may want to use directly the Extended C environment, so that its interface is also part of the programming model, and it is described here.

The same annotated source code is intended to be retargetable to any streaming architecture, with the compiler estimating the costs of computation and communication on the target and merging and scheduling tasks for optimal performance. For this reason, we also define an Abstract Streaming Machine (ASM), which can be used directly to evaluate costs in a prototype compiler and potentially used as a test-bed for future work on estimating streaming performance analytically.

In this document, we describe the first versions of both the SPM and the ASM. They are both preliminary, and still evolving. The goal now is to work with them and finalize them for the next deliverable which is due for month 15.

2 Streaming Programming Model

2.1 Overview

The Streaming Programming Model (SPM) describes an application as multiple tasks connected via point-to-point streams. Each task may be viewed as an independent process, with all its data private. Communication and synchronization among tasks happens only via streams. A stream is directed, and we refer to its ends from the point of view of the task, so that the producer has an output stream and the consumer has an input stream; the two ends are permanently connected together, and cannot be moved to connect other tasks during execution. The consumer task blocks when it tries to read from an empty input stream and the producer blocks when it writes to a full output stream.

2.2 Pragmas

We have taken OpenMP [OMP] as a simple, well-designed and widely understood set of pragmas or directives. The OpenMP pragmas allow one to start with an existing serial application and parallelize it incrementally, checking at each step that the program still works. This allows non-expert users to obtain immediate benefit from the programming model and learn more about it only as further knowledge is required. As there are many differences between the OpenMP and streaming models of parallelism, we are adapting the OpenMP constructs to work with streaming, adding new features when necessary. For example, OpenMP is designed for shared memory homogeneous machines, whereas the streaming pragmas target distributed memory and heterogeneous systems, for example the Cell Processor [ChT05].

2.2.1 Streaming pragmas

Pragmas control how the application is translated into a streaming program, by defining the tasks and the streams between them. All pragmas apply to the single statement following the specification of the pragma, so if a larger block is required it should be enclosed in braces. The outermost directive is the *pragma taskgroup*, which defines a region in which the tasks exist. It is named after the similar directive that is being proposed for the task support in the proposal for OpenMP 3.0. It first initializes all its tasks, starts them, executes the body of the outer control task, finally waiting for all of the tasks to finish; i.e. it defines an implicit barrier at the end. The syntax is as follows:

```
#pragma acotes taskgroup [async (var1, ...)]  
code-block
```

A taskgroup defines an initial task that controls the execution of the inner tasks, and it may give support to some of the streaming communications between them. Depending on the level of optimization, this support will not be needed. This pragma has one optional clause, *async(v1, ...)* which defines the set of shared variables that will be explicitly updated from inside the inner tasks or the taskgroup itself.

The second directive is *pragma task*, which must be lexically inside the *pragma taskgroup* to define a task. Any statements inside a taskgroup not enclosed by a *pragma task* belong to the control task. The task directive also allows the specification of input and output streams connecting to other tasks in the same taskgroup. The syntax is as follows:

```
#pragma acotes task [input (var1, ...)] [output (var2, ...)] [private(var3, ...)]
[firstprivate(var4, ...)] [lastprivate(var5, ...)] [async(var6, ...)] [shared(var7, ...)]
[import(stream1, ...)] [export(stream2, ...)]
```

code-block

Taskgroups and tasks may be nested together to force barriers at different points in the task hierarchy. All tasks nested inside a task group execute in parallel when there are enough resources available. Otherwise, the system scheduler makes them share the processors.

The clauses *input* (*v1, ...*) and *output* (*v2, ...*) define the input and output streams of a task or taskgroup. *Input* (*v1,v2,...,vn*) establishes *n* streams that will be used to receive the values for the variables *v1,v2,...,vn*. The task will automatically issue the corresponding pop operations to get the values from the streams. *Output* (*v1,v2,...,vm*) establishes *m* streams used to automatically send the values of the variables *v1,v2,...,vm* to further tasks.

The clauses *async*(*v1, ...*) and *shared*(*v2, ...*) are defining shared variables. The clause *async*(*v1, v2,...,vn*) has the same meaning than when used in the taskgroup directive. *Shared*(*v1,v2,...,vn*) defines *n* shared variables, and it implies the need of shared memory between the tasks accessing those variables.

The clauses *private*(*v1, ...*), *firstprivate*(*v2, ...*), and *lastprivate*(*v3, ...*) define private variables of a task. *Private*(*v1,v2,...,vn*) defines *n* state variables for the task, whose initial values are undefined. *Firstprivate*(*v1,v2,...,vn*) defines *n* state variables initialized with the value of the enclosing taskgroup. *Lastprivate*(*v1,v2,...,vn*) defines *n* state variables which values will be copied to the taskgroup at the end of the task. The same variable can be firstprivate and lastprivate.

The clauses *import*(*stream1, ...*) and *export*(*stream2, ...*) allow the use of explicit user input and output Extended C streams. *Import*(*stream1,stream2,...,streamn*) defines *n* user streams that will be popped from within the task-body. *Export*(*stream1,stream2,...,streamn*) defines *n* user streams that will be pushed to within the task-body.

The compiler will generate the body of each task enclosed on an implicitly generated loop that will execute until any of the input streams reaches the end-of-stream condition. If a task has no inputs, then it is given a virtual control stream as an implicit input. The format of the virtual control stream is implementation dependent, but it could for example carry one data element each time that the task should be executed.

Figure 1 presents an example of use of these directives. It defines a taskgroup region that contains the whole loop, and two explicit task definitions, each containing a single sentence of code. Therefore, in total there are three tasks. The first is the *taskgroup* control task (referred to as the *fread* task), and the other two are the first *#pragma task* (referred to as *tolower*) and the second *#pragma task* (referred to as *fwrite*). Three streams are also created. The first stream, *fread input(c)*, carries the value of the variable *c*, produced at *fread* and sent towards *tolower*. The second stream, *tolower output(x)*, carries the value of variable *x* produced by *tolower* and sent to *fread*. The third stream, *fwrite input(x)*, carries the value of variable *x* produced by *fread* and consumed by *fwrite*. Note that exactly one end of each stream is visible in the program, either as an input or an output clause; see Figure 2a. Figure 2b presents the resulting pipeline of tasks, which shows that the program has a little degree of parallelism. Note that we expect the compiler to connect *tolower output(x)* directly to *fwrite input(x)*, but consider it to be an optimization.

```

int main(int argc, char **argv)
{
    FILE *in, *out;
    char c,x,y;

    in = fopen(argv[1], "r");
    out = fopen(argv[2], "w");

    fread(&c, sizeof(char), 1, in);
    #pragma acotes taskgroup /* fread */
    while (!feof(in))
    {
        #pragma acotes task input(c) output(x) /* tolower */
        if ('A' <= c && c <= 'Z') x = c - 'A' + 'a';
        else x = c;

        #pragma acotes task input(x) /* fwrite */
        fwrite(&x, sizeof(char), 1, out);

        fread(&c, sizeof(char), 1, in);
    }

    fclose(in);
    fclose(out);
    return 0;
}

```

Figure 1: Annotated C code for the tolower example

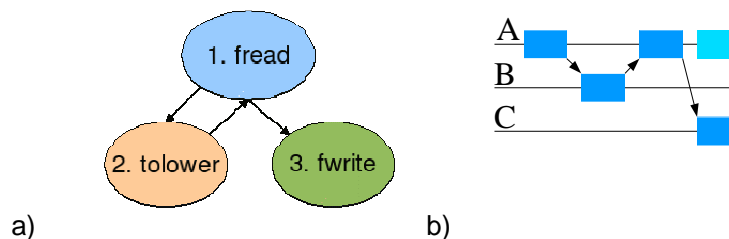


Figure 2: Tolower unoptimized (a) graph, and b) pipeline structure

2.2.2 Pragas supporting optimizations

The fact that all stream communication should go through the control task (the one associated with the task group) reduces the parallelism that can be achieved with this proposal. There are two solutions to this problem: either the compiler detects the shortcuts that can be implemented, and applies the optimization, or the programmer by hand indicates such possibility. For experimentation purposes, we are considering to provide this second option at the directive level. There is an easy solution, adding two new clauses that indicate, in addition to the variables involved in the communication, the name of the origin or destination targets of the communication. Their syntax is as follows:

```

#pragma ... target-input (var1, ...: target-name)
code-block

#pragma ... target-output (var1, ...: target-name)
code-block

```

Figure 3 shows the tolower example again, now with the use of these clauses. They allow the

compiler to automatically connect the output stream of task `tolower` to the input of task `fwrite`. Figure 4a shows the resulting graph after applying the optimization. The fact that we have removed the communication between `tolower` and `fread` allows to execute the program using the pipeline presented in Figure 4b. This optimized version corresponds to what the programmer expects from the streaming programming model.

```
int main(int argc, char **argv)
{
    FILE *in, *out;
    char c,x,y;

    in = fopen(argv[1], "r");
    out = fopen(argv[2], "w");

    fread(&c, sizeof(char), 1, in);
    #pragma acotes taskgroup /* fread */
    while (!feof(in))
    {
        #pragma acotes task input(c) target-output(x:WRITE) /* tolower */
        if ('A' <= c && c <= 'Z') x = c - 'A' + 'a';
        else x = c;

        #pragma acotes task target-input(x:WRITE) /* fwrite */
        fwrite(&x, sizeof(char), 1, out);

        fread(&c, sizeof(char), 1, in);
    }

    fclose(in);
    fclose(out);
    return 0;
}
```

Figure 3: Optimized code for the `tolower` example

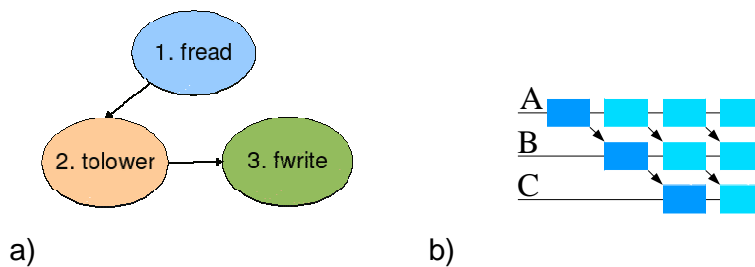


Figure 4: `ToLower` optimized (a) graph, and (b) pipeline structure

There are situations where a piece of code is used to compute a single element, given a number of elements in the input stream. This decimation is usually implemented using a control loop structure with a defined runtime limit. In these cases, it is useful to replicate the loop structure across the enclosed tasks, so that they are executed the same amount of times, before generating a value. The proposed syntax to implement this situation is as follows:

```
#pragma acotes for_replicate
for-block
```

Figure 5 presents an example of such a construct, and Figure 6 shows the task graphs and pipelined executions for the unoptimized and optimized versions:

```
#pragma acotes taskgroup /* taskgroup: A */
while (0 < fread(&values, sizeof(int), N, if)
{
  min= values[0];

  #pragma for_replicate
  for (i= 1; i < N; i++)
  {
    #pragma task input(values) output(value) /* task: B */
    value= decode(values[i]);

    #pragma task input(value,min) output(min) /* task: C */
    min= value < min ? value : min;
  }

  #pragma acotes task input(min) /* task: D */
  printf("%d\n", min);
}
```

Figure 5: Code showing the use of the for-replicate directive

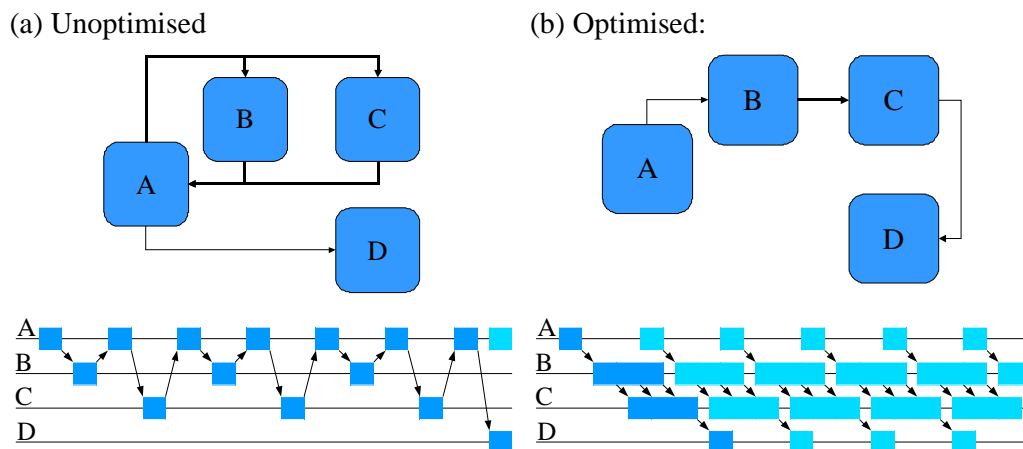


Figure 6: Graph and pipeline for the code of Figure 5

This directive avoids passing the loop index variable of the replicated loop to each of the tasks as input, which would have been implemented using an input stream. We think that even if the directive is not present, the compiler can detect simple cases, and introduce itself the transformation.

2.2.3 Other auxiliary pragmas

Update asynchronous state variables

We have defined a special case of shared variable that is updated explicitly by the programmer under the control of the clause `async`. The update occurs when the following directive is used from inside a task:

```
#pragma acotes update(var1, ...)  
code-block
```

The directive `update(var1,var2,...,varn)` modifies the original copies of all variables listed, at the end of the code-block. It cannot be assumed that the update is executed atomically on all tasks.

Setting the requirements of tasks

Sometimes a task needs a special resource to run on, like a hardware accelerator or simply to use a primitive that it is only available on a full-fledge processor, like file I/O. The proposal is to use a clause of the following type:

```
#pragma acotes task requires [keyboard, libc, spu, fmadd, ...]  
code-block
```

Where the specific features that can be specified are left open to each implementation.

2.2.4 Work in progress

Task naming

We foresee that it will be useful to assign names to the tasks to achieve further expressiveness in the programming model. For instance, in the original sequential program, the code belonging to a single task may be intermixed with code from other tasks. Setting names at the task level will allow the compiler to fuse all the code with the same name into a single task.

Also, special initialization sequences may have to be done inside the hardware accelerators to define the value of internal variables. Task naming will allow the compiler to merge this initialization code with the code of the task using such variables.

Pragmas supporting parallelism

In the specific places where the application exhibits shared memory parallelism, the usual OpenMP directives/pragmas can be used.

2.3 Extended C supporting the programming model

This section outlines the interface provided by the runtime system to support the directives presented. In addition, this interface is available to programmers, in case they want to use user-managed streams.

2.3.1 Data structures representing streams

Streams are represented in extended C source as special types. These types represent one of both streams ends. In these types, the semantics of input and output are defined from the task view: input allows the task to consume elements, and output allows the task to produce data. The proposal for these types is:

```
istream_t // input stream type
ostream_t // output stream type
```

Ostream_t defines an output stream, which allows to push elements. Istream_t defines an input stream, which allows to pop elements. The internals of the streams types is implementation dependent.

2.3.2 Description of the streaming interface

The streaming interface is defined in two parts: the interface for input streams, and the interface for output streams. The operations that can be performed on streams are to create and destroy them, push and pop elements. The following are the Extended C primitives supporting streams:

Input stream interface

```
void istream_create(istream_t*, int element_size);
```

istream_create creates a new input stream, defining the element size as element_size bytes. The behavior of the rest of the interface on an uninitialized stream is undefined before this call.

```
void istream_destroy(istream_t);
```

istream_destroy destroys and frees resources associated with a previously created stream. The behavior of the rest of the interface on the specified stream is undefined since this call is effective.

```
const void * istream_peek(const istream_t, int n_element);
```

istream_peek returns a pointer to the element numbered *n_element* in the input stream. The element number 0 is the oldest element introduced in the stream, and not popped. The number 1 is the next element pushed, and so on and so forth. The current task blocks, until the requested element is available in the stream (it is pushed by the source task). The returned pointer will be valid till the first pop operation is issued.

```
void istream_pop(istream_t, int count);
```

istream_pop removes count elements from the input stream.

```
int istream_eos(const istream_t);
```

istream_eos queries for the end of the data in the stream, and it returns true if there will be no more elements available, and the other side of the stream (ostream) is closed.

Output stream interface

```
void ostream_create(ostream_t*, int element_size);
```

ostream_create creates a new output stream. It defines the element size as element_size bytes. The behavior of the rest of the interface on an uninitialized stream is undefined before this call is issued.

```
void ostream_destroy(ostream_t);
```

ostream_destroy destroys and frees resources of a previously created stream. The behavior of the rest of the interface on the specified stream is undefined since this call is effective.

```
void ostream_push(ostream_t, const void* element_ref);
```

ostream_push sends an element, pointed to by element_ref, through the ostream. The size of the element was previously defined on the ostream_create call. Element_ref cannot be null. The implementation of push may block the current task, when there are not enough resources available.

```
void ostream_close(ostream_t);
```

ostream_close notifies that there will be no more elements pushed at the specified output stream. The behavior of ostream_push is undefined after this call is effective.

```
void ostream_connect(ostream_t, istream_t);
```

`ostream_connect` connects one output stream with one input stream. None of the streams are already connected. The size of the elements of both streams must be the same or one multiple of the other.

Auxiliary functions

We have observed that when the sequential execution of the Extended C is required, additional functions are needed to avoid deadlock. The primitives that we are currently using to overcome this problem are the following:

```
int istream_count(const istream_t);
```

`istream_count` asks for the counter of elements present in the stream. The value returned is the number of elements that can be popped from the stream.

```
int istream_wait(const istream_t, int count);
```

`istream_wait` has two different behaviours depending on whether the execution is serial or streaming. In a serial execution, this primitive returns 0 when the input stream has at least *count* elements available. And it returns non-zero when any of the *count* peek operations would block. In a streaming execution, this function always waits until *count* elements are available, and it always returns 0.

```
int ostream_room(const ostream_t);
```

`ostream_room` asks for the minimum number of elements that can be pushed onto the stream, before the runtime eventually blocks the task.

```
int ostream_wait(const ostream_t, int room);
```

`ostream_wait` has two different behaviours depending on whether the execution is serial or streaming. In a serial execution, this primitive returns 0 when the output stream has at least *room* elements available. And it returns non-zero when any of the *room* push operations would block. In a streaming execution, this function always waits until *room* elements would be pushed without blocking, and it always returns 0.

2.3.3 Merging directives and Extended C

There are situations where it may be useful to reuse code already written in plain C. Current applications are usually developed using functions that act as filters. We can take advantage of this legacy code by providing compatibility support to allow the use of user-managed streams in functions implementing such filters.

For instance, the following code implements such a feature:

```
#pragma acotes taskgroup
{
  for (it=0; it < N; it++) {
    #pragma acotes task firstprivate (taps, decimation, coeff) import (history)
    output (sum)
      sum = low_pass_filter (history, taps, decimation, coeff);
    #pragma acotes task input (sum)
      printf ("%f\n, sum);
  }
}

float low_pass_filter (istream_t history, int taps, int decimation, double *
coeff)
{
  float sum = 0.0;
  int i;
  for (i=0; i < taps; i++) {
    float * v = (float *) istream_peek (history, i);
    sum += *v * coeff[i];
  }
  istream_pop (history, decimation);
  return sum;
}
```

This code creates two tasks, being the first one a low pass filter, and the second one simply the printing of the resulting data. The stream *history* is user-managed, containing the input data that has to be filtered. The management of the stream is done in the `low_pass_filter` function, where the user implements a sliding window on the input data. The function uses *taps* values from the input stream to compute the new output value, and afterwards it pops *decimation* elements from the stream.

After the output value sum is computed by the filter function, it is sent to the next task using a regular output stream, which will be managed by the runtime system, thus providing the connection between the directives model, and the Extended C model.

3 Abstract Streaming Machine

3.1 Introduction to the ASM

The *Abstract Streaming Machine* (ASM) is a model of the target multiprocessor architecture in a form that allows the compiler to automatically partition and schedule a streaming program to run on it. The ASM is parameterized by the *Machine Description*, which provides the description of a particular target. It is not necessary or desirable for the machine description to duplicate information already present in the compiler's own machine description, which gives local information for each one of the processors such as the details of its ISA and micro-architecture. Instead, it should identify which processors are available and how they are connected, referencing if possible the compiler's relevant machine descriptions.

The machine description must be sufficient to allow the compiler, or an external tool invoked by the compiler, to determine the performance of a particular program, given a proposed mapping. This is the function of the *cost model* associated with the ASM, initially defined and implemented as a coarse grain simulator. In future, it may be feasible to define a closed form statistical model that does not require simulation, and would potentially be faster, but we believe that such work should be outside the scope of the ACOTES project.

The cost model requires from the compiler a specification of both the streaming program and the proposed mapping of the program onto the hardware. The space of potential mappings of a fixed program onto a fixed platform is of course usually huge, and it is the compiler that should manage the search for the optimum. The compiler queries the cost model, as necessary, to assign a cost value to each candidate mapping. Each time it is called, the cost model runs a short simulation to find out.

In the above, we have made a distinction between the definition of the streaming program and its mapping; although in practice there will be some overlap between the two because the description of the program is influenced by how it is mapped. For example, if several tasks have been fused into one by the compiler, they will cease to have independent identities and it would be hard and pointless to try to separate them. Furthermore, the program description describes the program after optimization, so is influenced by certain transformations such as loop blocking, or loop stripmining.

Many programs have streaming behaviour that is difficult or impossible to predict statically. For this reason, the cost model should be able to take as input a dynamic trace of the streaming program, identifying the precise pattern of communications. In a similar manner to the Dimemas [Dim] tool, the cost model will ignore any existing timing information in the trace, and use the cost model parameters to estimate the performance obtained. Like Dimemas, we intend to take as input a trace in the Paraver format, as we are familiar with these tools at the UPC. The pattern of communications between tasks will be exactly as given by the trace, so it will not be possible to reuse a trace with a different partitioning or a different blocking factor, if doing so would affect the sequence of messages passed between the tasks.

We have implemented a prototype cost model simulator, written in the Python language, taking its inputs as Python source code. The prototype simulator runs for a given number of time units, and generates a Paraver trace [Par], a cycle-by-cycle dump or statistics such as the number of cycles per iteration.

3.2 Modeling the platform

3.2.1 Overview

The Machine Description defines the platform on which the streaming program will be executed, and is represented by a set of processors connected via communication links, each of which joins two or more processors. Each communications link has a single unbounded queue to hold the messages ready to be transmitted, and one or more channels on which to transmit them. The reason for having two levels: channels and links, rather than just one, is that it allows the choice of channel within a link to be deferred until run time. Otherwise streams would not just be statically mapped onto a particular link; e.g. the Cell EIB for example, but onto a particular link; e.g. one of the rings on the EIB.

The bandwidth and latency of a channel can be specified using four parameters, the *start latency* (L), *start cost* (S), *bandwidth* (B), and *finish cost* (F), as shown in Figure 7. The start cost and bandwidth combine to give a function for the transmission time of the body of a message as a function of its size in bytes, $T = S + \text{ceil}(\text{message size}/B)$. The latency of the link is then given by $L + T$ and the cost incurred on the link by $T + F$. It is clear that there is some redundancy in this model, as it controls two values using three independent variables, but it has the advantage of being easy to understand and implement.

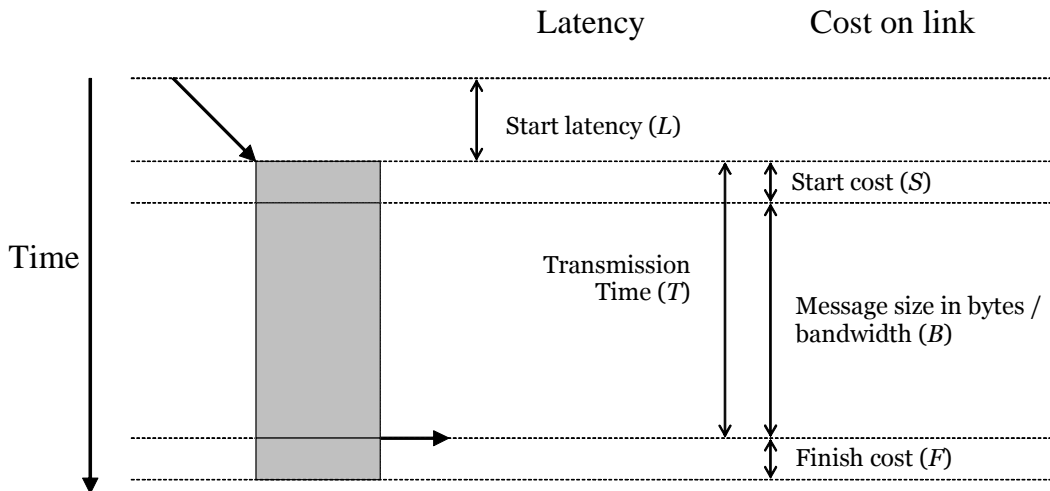


Figure 7: Cost and latency of a message on a given channel

In the ideal case most of the non-local traffic on the real platform will be due to communication via streams, and therefore be explicit in our model. We may also model the reloading of instructions across the bus at context switch. Even so, there may remain a significant number of other transfers due to data cache refills or explicit spilling to external memory, all of which we will ignore for now.

3.2.2 Static routing

When a processor is connected to a link with more than one channel on it, we would like to assume that it is not possible for the same processor to transmit onto more than one channel in the same link simultaneously; and similarly for receive. The interface may be either full duplex or half duplex, depending whether it is possible to transmit and receive on different channels at the same time. On the Cell processor, the bus interfaces of the SPEs and PPEs are full duplex, but this will depend on the platform.

If there is a stream between tasks executing on two processors that are not on the same link, then, so long as the hardware supports it, it is given a static route and the intermediate nodes between the source and destination will forward the messages automatically. The *interfaceRouting* item in the link definition is used to indicate such feature.

The behaviour of multi-hop communications is still open to discussion. Figure 8 shows an example of *store and forward* and *virtual cut-through* routing, which are both supported in the prototype model. In this example, a single message is routed via the three links *P*, *Q* and *R*, the middle of which has a much lower bandwidth. Store and forward is the easiest to implement, but it is unlikely that the hardware would buffer the entire message at each node before attempting to retransmit it. Rather, the outgoing message will begin transmission as soon as possible, after a fixed latency if the bandwidth of the outgoing link is the same or smaller than the incoming one. If the outgoing bandwidth is higher, then the message will be divided into packets of fixed maximum length for forwarding. Presumably the buffer size at the intermediate node is limited, so it should theoretically apply back pressure; but maybe we can ignore this problem at run-time for now. If we do model the buffer sizes and back pressure, then we should also support wormhole routing.

If a message needs forwarding, then as soon as it appears on a link, the simulator should be able to determine precisely how it will be propagated to the next; e.g. that the message will be divided into three, of sizes 1024 bytes, 1024 bytes and 452 bytes respectively, and put in the next link's input buffer at times 4362, 4924 and 5486. Doing so avoids having to make decisions on a cycle-by-cycle basis. This means that if the two links have multiple channels, the set of channels on each link must all have the same cost parameters, in particular bandwidth.

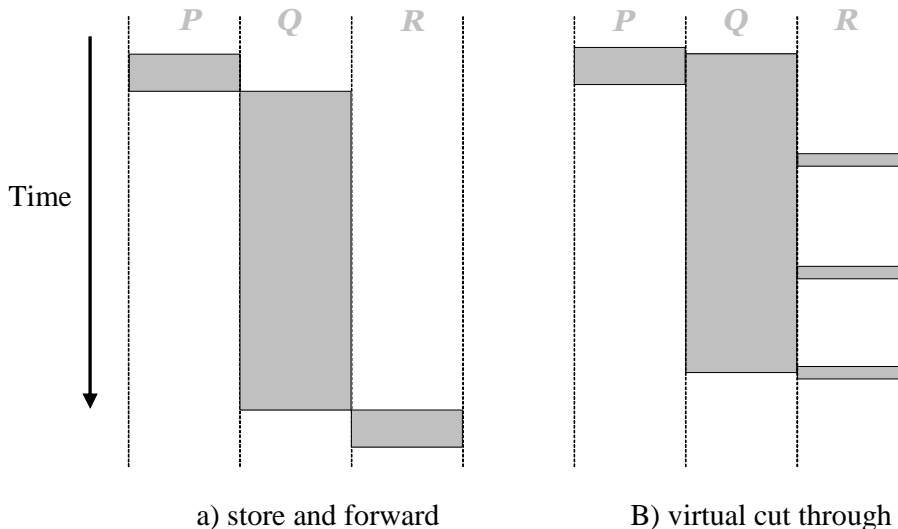


Figure 8: Store and forward and cut-through routing

3.2.3 Processor and link definitions

In summary, the machine description should define the set of processors and the set of links, specifying the information given in Table A and Table B.

Id	Type	Description
name	String	Unique name in platform namespace
executable	Bool	True if the processor can execute code (see section 1.2.2)
localMachineDesc	Reference	Link to the compiler's machine description
keywords	List	List of keywords; e.g. the <code>hasIO</code> keyword would identify processors that can perform IO

Table A: Processor definition

Id	Type	Description
name	String	Unique name in platform namespace
processorsOnLink	List	List of the names of the processors on the link.
interfaceDuplex	List	If the link has more than one channel, then define for each processor whether it can transmit and receive simultaneously on different channels
interfaceRouting	List	Define for each processor the type of routing from this link: <i>storeAndForward</i> , <i>cutThrough</i> , or <i>None</i> .
startLatency	Int	Start latency, L , in cycles
startCost	Int	Start cost, S , in cycles
bandwidthPerCh	Int	Bandwidth per channel, B , in bytes per cycle
finishCost	Int	Finish cost, F , in cycles
numChannels	Int	Number of channels on the link
multiplexable	Bool	False if the link can only support one stream, which may be the case if it is a hardware FIFO

Table B: Link definition

Figure 9 shows the definition of a simple 2x2 mesh using our current Python syntax, which does not yet support the full specification above.

```
# Define platform
def setup_platform():

    # Define processors      name
    processors = [ Processor ('A'),
                  Processor ('B'),
                  Processor ('C'),
                  Processor ('D') ]

    # Define links          name      processors  latency  bandwidth  numCh
    #                       |         |           |         |         |
    links = [ Link ( 'A <-> B', [ 'A', 'B' ], 0, 2, 4, 2, 1 ),
             Link ( 'A <-> C', [ 'A', 'C' ], 0, 2, 4, 2, 1 ),
             Link ( 'B <-> D', [ 'B', 'D' ], 0, 2, 4, 2, 1 ),
             Link ( 'C <-> D', [ 'C', 'D' ], 0, 2, 4, 2, 1 ) ]

    # Define platform
    return Platform ( processors, links )
```

Figure 9: Example platform description for a 2x2 mesh

3.3 Modelling the Streaming Program

3.3.1 Overview

The streaming program is, essentially, a directed graph of tasks and streams, with each stream carrying a sequence of values from the *producer* to the *consumer*. Each task repeats a fixed basic unit of work, the work function, inside an implicit loop, and may have any number of input and output streams. All synchronization between tasks will be managed via streams. It is possible for a task to have no input streams and/or no output streams. The only reason to support no inputs *and* no outputs, assuming that programs are usually connected, is to allow the compiler to collapse the whole program into a single task.

3.3.2 Simple tasks

The simplest kind of task worth mentioning has a work function consisting of three distinct stages: a *pop stage*, in which a fixed number of elements are popped from each of its input streams, a *processing stage*, in which a fixed amount of work is performed, and a *push stage*, in which a fixed number of elements are pushed onto each of its output streams. We call such tasks *simple tasks*, but it is clear that such tasks are not sufficient, in two orthogonal ways. Firstly, the computation time and/or communication may depend on the data, or more accurately simply fail to be constant. Secondly, the task may not be divided neatly into three distinct stages; instead being an arbitrary sequence of computation, pushes and pops.

3.3.3 Modelling irregular tasks

We first deal with how a work function could fail to be divided into these three stages, and later talk about the data dependent behaviour. Figure 10 shows a *simple* task (a), together with some reasonable non-simple ones. Plates (b) and (c) show how the compiler may wish to aggregate communication to get a good trade-off between speed on one hand and latency and memory use on the other, when implementing interpolation (b), and decimation (c). In these examples, there are two approaches short of implementing the general case directly. The first method is to generate a conservative cost estimate by moving all the pops to the beginning of the work function, and all the pushes to the end, and increasing the queue sizes accordingly. This increases latency, but more seriously, may cause deadlock. As an example of this, plate (d) shows how the unoptimized SPM translation algorithm implements the communication

between two tasks A, and B. It is clear that moving the pop of the response from B to the beginning of the task A and the corresponding push to the end will cause deadlock.

The second method is to split such a task into a sequence of simple ones as shown in Figure 11, obtaining a sequence of *subtasks*. It may be that more than one of the subtasks obtained in this way produce or consume elements on a particular stream. In the interpolation example of Figure 11(a), subtasks A and B both push onto the same stream marked p . The program model should therefore distinguish tasks from subtasks, allowing only a single producer and consumer *task* for each stream, but potentially more than one subtask. All the subtasks of course execute on the same processor.

3.3.4 Modelling variable computation time and pattern of communication

The other limitation of simple tasks, mentioned above, is that both the cost of computation and pattern of communication are fixed. Note that even when the cost model is implemented by coarse-grain simulation, the values of the data on the streams are clearly not going to be known. The compiler may supply a static distribution function for the computation cost of the work function, so that each iteration has a cost given by an independent random variable following that distribution. As noted above, a single iteration may have invoked several subtasks, with different mutually independent distributions.

As remarked in the Section 3.1 (Introduction to the ASM), it is possible to handle data-dependent patterns of communication by reading a trace. We would also like, if possible, to define a statistical model; but doing so is harder for communication than computation because it is necessary to ensure consistency between the two ends of the streams.

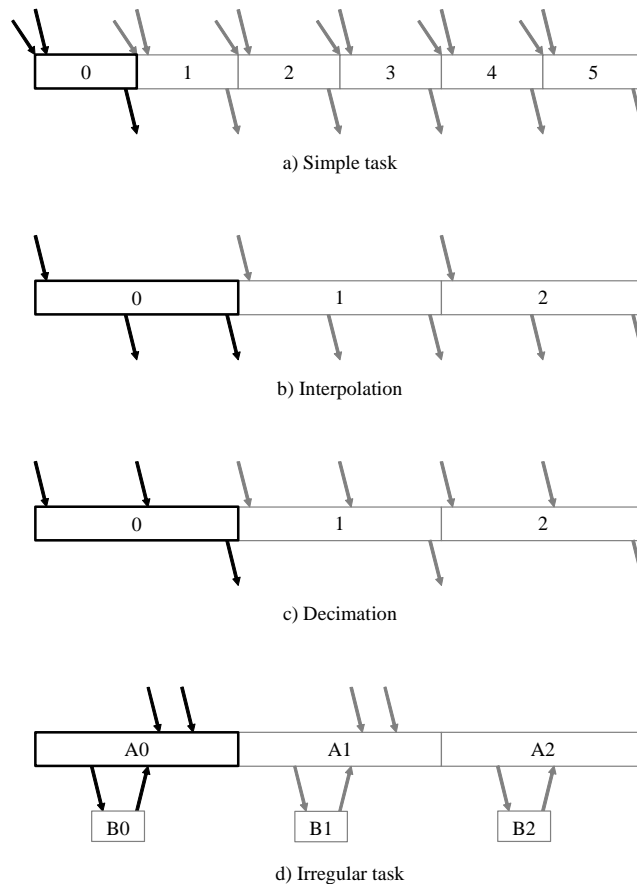


Figure 10: Fixed patterns of task communication

Figure 12 shows source code for a simple example, where depending on some condition on x , either g or h is performed; and for any of a number of reasons, g and h are to be put on different processing engines¹. Figure 13 represents the same program as a graph, where Task f may push the value of x to either task g or task h , in fact depending on the data but simulated using a sequence of boolean random variables, denoted X . Task f therefore has two outgoing streams for x , predicated by X and $\neg X$ respectively in the diagram, corresponding to the cases where the random variable is True and False. Similarly task k has two incoming streams for y , with the selection between them based on the same random variable. In this example, the compiler will probably have to introduce an extra stream carrying the successive values of $\text{cond}(x)$, otherwise in the actual binary, task k will not know what to do. However, the cost model shouldn't assume that there is always such a stream.

¹ Perhaps for load balancing, or because only one of them is large enough to warrant the overhead of putting in a separate task.

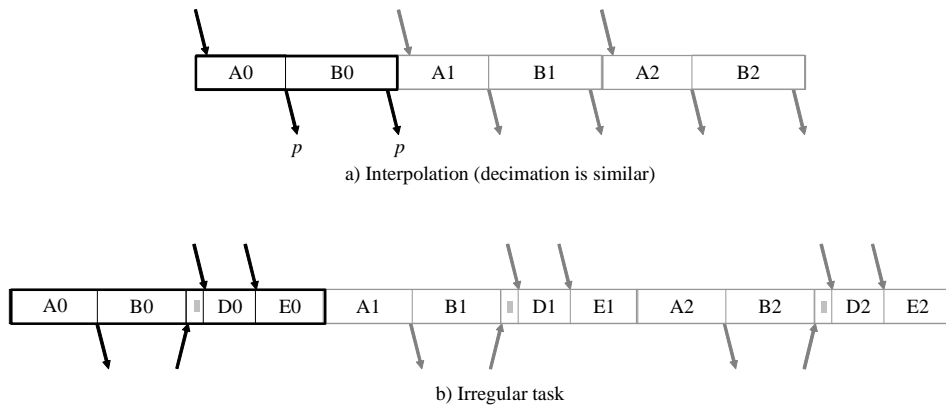


Figure 11: Splitting a task into simple subtasks

```
#pragma acotes task input(a) output (x) // task f
x = f(a);

if (cond(x))
{
#pragma acotes task input(x) output(y) // task g
y = g(x);
else
{
#pragma acotes task input(x) output(y) // task h
y = h(x);
}

#pragma acotes task input(y) output (z) // task k
z = k(y);
```

Figure 12: Data dependent flow in the source code

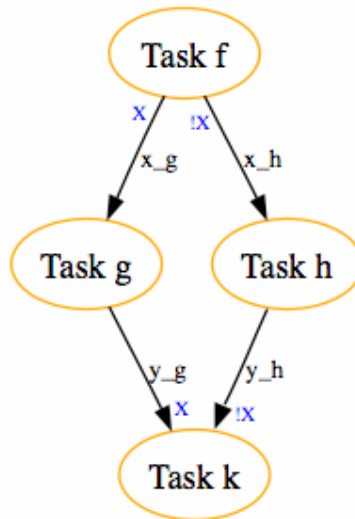


Figure 13: Data dependent flow at the graph level

In the context of the ACOTES project, the starting point for streaming programs is valid sequential C code. Assuming that the programmer is not given unrestricted access to the underlying stream data type, this may constrain the class of graphs encountered by the ASM.

A particular basic block in the original program may be divided up into tasks in any manner, yielding a flat Synchronous Data Flow graph, but given that the repetition counts are all the same, it does not internally present the problem being discussed here.

In the serial program, larger blocks are built up hierarchically from smaller ones using *if* statements, suggesting that if each basic block identified above is collapsed into a single node, the resulting graph is essentially a two-terminal series-parallel (TTSP) graph, such as that of StreamIt. We may allow any non-negative number of streams to pass through a single edge of the TTSP graph, as doing so adds no extra power over a language like StreamIt that allows record types. Furthermore, we shall allow streams to take short-cuts directly from their source to their destination, so long as there is also a valid indirect route.

We do, however, have to be careful with this scheme: Figure 14 shows an example where there is ostensibly communication, of the variable *t*, between the two branches of an *if* statement, albeit in different iterations of the loop. However, it is likely that when compiled, there would be no direct communication between *g* and *h*, and these tasks would communicate via their common neighbours; i.e. task *f* and/or task *k*. Figure 15 shows a) how the program could be implemented using direct communication, and b) one several possible ways in which direct communication could be avoided.

```
#pragma acotes task input(a) output (x) // task f
x = f(a);

if (cond(x))
{
#pragma acotes task input(x) output(y, t) // task g
  y = g0(x);
  t = g1(x);
}
else
{
#pragma acotes task input(x, t) output(y) // task h
  y = h(x, t);
}

#pragma acotes task input(y) output (z) // task k
z = k(y);
```

Figure 14: Potentially non-hierarchical data-dependent communication in the source code

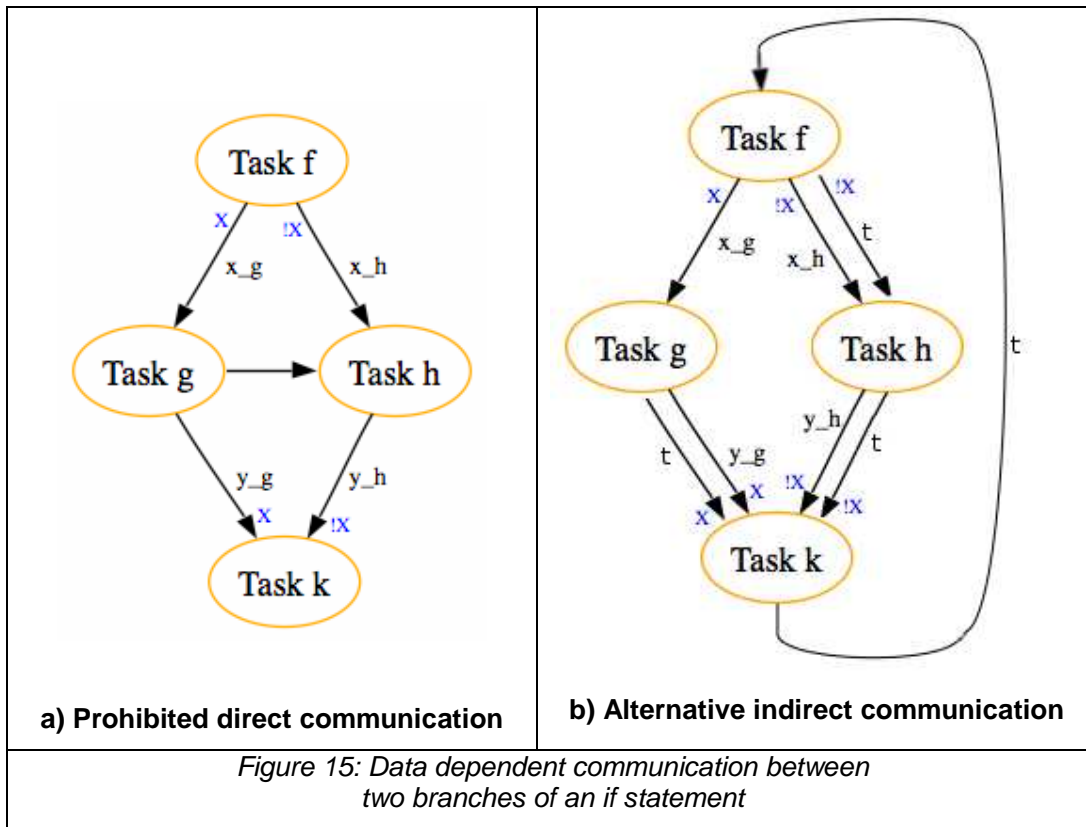


Figure 15a is the standard prohibited *W graph*, which is not possible in a TTSP graph [Val79]. It is difficult to model in the cost model simulator, and can only arise from aggressive optimizations. It is preferable if we simply disallow it in the ASM, and support only an alternative such as Figure 15b, which requires a cycle involving more than one task. The program description should therefore inherit the hierarchical structure of the original program. Each *if* statement has an associated sequence of random variables, generated lazily as required and shared by all those tasks that require it.

As mentioned above, it is possible for a task to not have any inputs, begging the question of how it knows when to stop. The simple answer in the *ASM* is that it doesn't, so it may iterate more times than are strictly necessary. In fact, it is likely not to be necessary, at least initially, to model the end of the streaming part of the program. Streams are of effectively infinite length, as far as the cost model is concerned.

Tables C, D and E show, respectively, the information required to define streams, subtasks and tasks in the program description. It is also necessary for the program to define the hierarchical control flow graph—see Section 1.3.5. Figure 16 shows a possible way to define the example of Figure 12 using Python syntax.

Id	Type	Description
name	String	Unique name in program namespace
elemSize	Int	Size of each element on the stream, in bytes
queueLength	Int	Number of elements in the producer's send queue and the consumer's receive queue
route	List	Static route for the stream, given by an alternating list of processors and links. If left undefined, a minimum length route will be found, but it may not be optimal.

Table C: Stream definition

Id	Type	Description
name	String	Unique name in program namespace
cpuTime	Int	Fixed execution time—may be extended to a statistical distribution
inputs	List	List of input streams, each specifying the name of the stream, number of elements to pop and the cost incurred by the processor in popping them
output	List	List of output streams, in the same format as the input streams

Table D: Subtask definition

Id	Type	Description
name	String	Unique name in program namespace
subtasks	List	Ordered list of the subtasks
codeSize	Int	Code size, in bytes, used to estimate the cost of reloading code at context switch
codeLocation	String	Name of the communications link on which the memory containing the code to reload at context switch is located

Table E: Task definition

```

# Define program
def setup_program():

    # Control flow graph of subtasks
    cfg = [ Fixed(1),
            'f0',
            [ Uniform(0,1), ['g0'], ['h0'] ]
            'k0'
          ]

    # Physical streams
    #          name elemSize  min queueLength
    streams = [ Stream ( 'x_g',  4,      1 ),
                Stream ( 'x_h',  4,      1 ),
                Stream ( 'y_g',  4,      1 ),
                Stream ( 'y_h',  4,      1 ) ]

    # Subtasks
    #          name      inputs          outputs          cpu_time
    #          ::[ (inEdge,num,cost)    ::[ (outEdge,num,cost)]
    subtasks = [ Subtask ( 'f0', [],
                          [ ('x_g;x_h',1,2)], 10 ),
                Subtask ( 'g0', [ ('x_g',1,2)],
                          [ ('y_g',1,2)], 5 ),
                Subtask ( 'h0', [ ('x_h',1,2)],
                          [ ('y_h',1,2)], 25 ),
                Subtask ( 'k0', [ ('y_g;y_h',1,2)],
                          [], 10 ) ]

    # Tasks
    #          name list of subtasks
    tasks = [ Task ( 'f', ['f0']),
              Task ( 'g', ['g0']),
              Task ( 'h', ['h0']),
              Task ( 'k', ['k0']) ]

    return Program ( cfg, subtasks, tasks, streams )

```

Figure 16: Proposed program definition for the example of Figure 12

3.3.5 Data dependent scheduling of subtasks and control flow graph

Figure 17 shows a more complicated example that contains data-dependent data flow. Assume that tasks f , g , h and k are merged to execute on one processor, and tasks j and l are merged to execute on another. This does not necessarily mean that the ASM is given two tasks, containing the subtasks $[f, g, h, k]$ and $[j, l]$ respectively. The compiler schedules a merged task as a single unit and should only split it into subtasks for the reasons given in Sections 3.3.3 and 3.3.4.

```

while (!feof(fp1))
{
    x = fgetc(fp1);                // task f
    a = f(x);

    if ( cond1(x) )
    {
        #pragma task input(a) output(b)    // task g
        b = g(a);
        if ( cond2(x) )
        {
            #pragma task input(a,b) output(u) // task h
            u = h(a,b);

            #pragma task input(u) output(v) // task j
            v = j(u);
        }
    }
    #pragma task input(a,b) output(v)    // task k
    c = k(a,b);

    #pragma task input(v, c)             // task l
    fputc ( l(v,c), fp2);
}

```

Figure 17: Example program to schedule

Figure 18 shows possible extended C code for the example, together with an indication of how it could be split into simple subtasks, $[p, q, r]$ and $[x, y, z]$. This is not the only way of partitioning the program, and in fact the partitioning depends on how the compiler schedules the code. It also depends on how accurately the computation costs should be modelled; a finer grain partitioning could capture the fact that, in this example, function g is only called if $\text{cond1}(x)$ is true.

<pre> while (!feof(fp1)) { x = fgetc(fp1); push(x, ostream_x); a = f(x); if (cond1(x)) { b = g(a); if (cond2(x)) { u = h(a,b); push(u, ostream_u); } } c = k(a,b); push(c, ostream_c); } </pre>	$\left. \vphantom{\begin{array}{l} p \\ q \\ r \end{array}} \right\}$	p q r	<pre> while (...) { x = pop(istream_x); if (cond1(x)) { if (cond2(x)) { u = pop(istream_u); v = j(u); } } c = pop(istream_c); fputc(l(v,c), fp2); } </pre>	$\left. \vphantom{\begin{array}{l} x \\ y \\ z \end{array}} \right\}$	x y z
--	---	---------------------------	--	---	---------------------------

a) Task containing f, g, h and k

b) Task containing j and l

Figure 18: Extended C code for the example partitioning of the program in Figure 17

Figure 19 shows a possible execution trace for this example; subtasks q and y are shown with a vertical offset to show their depth in the control flow graph. Figure 20 shows how we could define the control flow graph for this program. Each node represents a basic block, and specifies the unordered set of subtasks that it contains. Each node can also contain one or more pairs of children, with each pair corresponding to the true and false cases of a particular *if* statement and labelled with the statistical distribution of the predicate. Initially we should support Bernoulli random variables with a single parameter giving the probability of executing the true case.

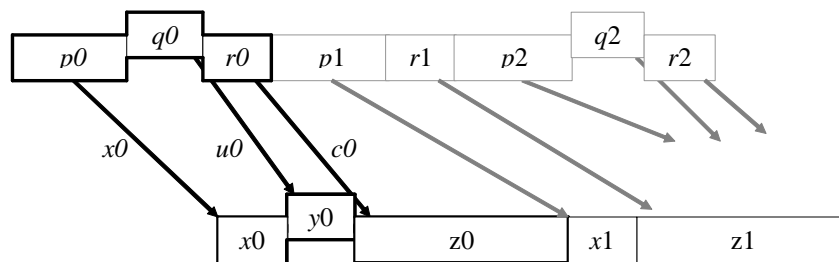


Figure 19: Example execution of the extended C code in Figure 18

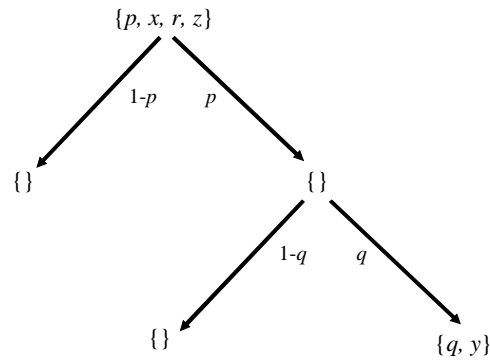


Figure 20: Control flow graph for the extended C code in Figure 18

3.4 Defining the mapping and scheduling

The final piece of information to be supplied by the compiler is the proposed mapping of tasks onto processors and how they will be scheduled. As mentioned above, each task is assigned to a fixed processor and each stream is given a fixed route; specifying this information is not so hard.

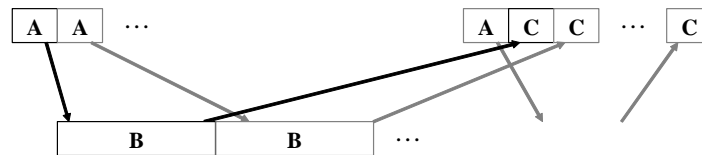


Figure 21: FIFO-driven scheduling

There may be several tasks allocated to the same processor. For example, Figure 21 shows a pipeline of three tasks: task A reads from a file, task B does some work and task C writes to the output file. Tasks A and C both need file IO, so they may have to run on the same processor; e.g. the PPE on a Cell processor, but they should not have a fixed static schedule. We define a list of schedules, one per processor, and each specifying the information in Table F. Figure 22 shows an example of a simple mapping file.

Id	Type	Description
processor	String	Name of the processor to run it on
task	List	List of tasks to run on this processor (the order of this list specifies the initial ordering of the queue)

Table F: Schedule definition

```
# Define binary
def setup_binary( program, platform ):

    # Threads
    #
    # proc tasks
    schedules = [ Schedule ( 'PPE', [ 'A', 'C' ] ),
                  Schedule ( 'SPE', [ 'B' ] ) ]

    # Return the binary
    return Binary( schedules )
```

Figure 22: A simple mapping file

There should be a cost associated with context switching, both the constant overhead and the cost of loading the code. For now, we assume that the whole body of code for a task needs to be reloaded if a different task executed previously.

3.5 Integration with the compiler

As described above, the cost model will be an external tool, invoked by the compiler to determine the performance of each candidate mapping. The previous sections have discussed what the cost model needs from the compiler in order to be able to characterize the system. The cost model also requires a limit on the number of iterations or cycles to run the simulation for.

Assuming that the descriptions are valid and consistent and the simulation doesn't deadlock, the cost model returns, by dumping to stdout, the throughput and latency either for the steady state or averaged over the whole simulation, and the percentage utilization for each processor and link on the system. The definition of the steady state is open to discussion, as it may not be straightforward to detect. Figure 23 shows an example of the statistics generated by the prototype simulator for a simple example with a defined steady state.

```

Time      f (AA)      f->g(AA<->AB)  f->h(AA<->BA)  g (AB)      g->k(AB<->BB)  h (BA)      h->k(BA<->BB)  k (BB)
-----
t= 182    Processing 16  Transfer 15    Start 15     Processing 11  -             Processing 11  Gap 10         Processing 6
t= 183    Processing 16  Transfer 15    Transfer 15   Processing 11  -             Processing 11  Gap 10         Processing 6
t= 184    Processing 16  Transfer 15    Transfer 15-> Processing 11  -             Processing 11  -             Processing 6
t= 185    Processing 16  Transfer 15->  Gap 15        Processing 11  -             Processing 11  -             Processing 6
t= 186    Processing 16  Gap 15         Gap 15        Processing 11  -             Processing 11  -             Processing 6
t= 187    (PushWait 16) Gap 15         -             Processing 11  -             Processing 11  -             Processing 6
t= 188    (PushWait 16) -             -             Processing 11  -             Processing 11  -             Processing 6
t= 189    (PushWait 16) -             -             (PushWait 11) -             Processing 11  -             Processing 6
t= 190    (PushWait 16) -             -             (PushWait 11) -             Processing 11  -             Processing 6
t= 191    (PushWait 16) -             -             (PushWait 11) -             (PushWait 11) -             Processing 6
t= 192    (PushWait 16) -             -             (PushWait 11) -             (PushWait 11) -             Processing 6
t= 193    (PushWait 16) -             -             (PushWait 11) -             (PushWait 11) -             Processing 6
t= 194    (PushWait 16) -             -             (PushWait 11) -             (PushWait 11) -             Processing 6
t= 195    (PushWait 16) -             -             (PushWait 11) -             (PushWait 11) -             Processing 6
t= 196    (PushWait 16) -             -             (PushWait 11) -             (PushWait 11) -             Processing 6
t= 197    (PushWait 16) -             -             (PushWait 11) -             (PushWait 11) -             Processing 6
t= 198    (PushWait 16) -             -             (PushWait 11) -             (PushWait 11) -             ->PopWork 7
t= 199    (PushWait 16) -             -             (PushWait 11) -             (PushWait 11) -             PopWork 7
t= 200    (PushWait 16) -             -             PushWork 11 -> -             (PushWait 11) -             ->PopWork 7
t= 201    (PushWait 16) -             -             ->PopWork 12 Start 11 PushWork 11-> -             PopWork 7
t= 202    PushWork 16-> -             -             PopWork 12 Transfer 11 PushWork 11 ->Start 11 Processing 7
t= 203    PushWork 16 ->Start 16 -             -             Processing 12 Transfer 11-> ->PopWork 12 Start 11 Processing 7
t= 204    PushWork 16-> Start 16 -             -             Processing 12 Gap 11 PopWork 12 Transfer 11 Processing 7
t= 205    PushWork 16 Transfer 16 ->Start 16 Processing 12 Gap 11 Processing 12 Transfer 11-> Processing 7

Iterations until steady state: 15
Iterations per steady state period: 1
Cycles until steady state: 158
Cycles per steady state period: 24

BeforeSteadyState
-----
NotRunning      0      0      0      0
(0%)            (0%)      (0%)      (0%)
WaitInput       0      0      0      0
(0%)            (0%)      (0%)      (0%)
WaitFull        38     10     13     0
(20%)           (5%)      (7%)      (0%)
Work            144    172    169    182
(79%)           (94%)     (92%)     (100%)

AAf      ABg      BAh      BBk
---      ---      ---      ---
NotRunning 0      0      0      0
(0%)      (0%)      (0%)      (0%)
WaitInput  0      0      0      0
(0%)      (0%)      (0%)      (0%)
WaitFull   38     10     13     0
(20%)     (5%)      (7%)      (0%)
Work       144    172    169    182
(79%)     (94%)     (92%)     (100%)

AA<->AB_0  BA<->BB_0  AA<->BA_0  AB<->BB_0
-----
NotRunning 0      0      0      0
(0%)      (0%)      (0%)      (0%)
WaitInput  44     118    91     116
(24%)     (64%)     (50%)     (63%)
WaitFull   0      0      0      0
(0%)      (0%)      (0%)      (0%)
Work       138    64     91     66
(75%)     (35%)     (50%)     (36%)

DuringSteadyState
-----
NotRunning      0      0      0      0
(0%)            (0%)      (0%)      (0%)
WaitInput       0      0      0      0
(0%)            (0%)      (0%)      (0%)
WaitFull        15     10     10     0
(62%)           (41%)     (41%)     (0%)
Work            9      14     14     24
(37%)           (58%)     (58%)     (100%)

AAf      ABg      BAh      BBk
---      ---      ---      ---
NotRunning 0      0      0      0
(0%)      (0%)      (0%)      (0%)
WaitInput  0      0      0      0
(0%)      (0%)      (0%)      (0%)
WaitFull   15     10     10     0
(62%)     (41%)     (41%)     (0%)
Work       9      14     14     24
(37%)     (58%)     (58%)     (100%)

AA<->AB_0  BA<->BB_0  AA<->BA_0  AB<->BB_0
-----
NotRunning 0      0      0      0
(0%)      (0%)      (0%)      (0%)
WaitInput  15     18     18     18
(62%)     (75%)     (75%)     (75%)
WaitFull   0      0      0      0
(0%)      (0%)      (0%)      (0%)
Work       9      6      6      6
(37%)     (25%)     (25%)     (25%)

```

Figure 23: Statistics, generated using the command line `sim.py --statistics examples/square_on_mesh2x2.py`

4 Summary

In this deliverable, we have presented the current status of the definition of the Streaming Programming Model (SPM) and the Abstract Streaming Machine (ASM). There are still some issues under discussion in the context of the WP 2, and WP4, with the goal of establishing the definitive versions of both topics in the deliverable D2.2. As a result of the interaction with WP4, we will add the support for vectorization onto the programming model in the final version of the deliverable in D2.2

References

[ChT05] Chen, T., Raghavan, R., Dale, J., Iwata, E. Cell Broadband Engine Architecture and its first implementation. IBM Developerworks, 2005.

[Dim] Dimemas: <http://www.cepba.upc.es/dimemas/>

[Lee87] Lee, E., Messerschmitt, D. Synchronous data flow. Proceedings of the IEEE, 75 (9), pp. 1235-1245, 1987.

[OMP] OpenMP Specification, <http://www.openmp.org/>

[Par] Paraver: <http://www.cepba.upc.es/paraver/>

[Val79] Jacobo Valdes, Robert E. Tarjan and Eugene L. Lawler. *The recognition of Series Parallel digraphs*. Proceedings of the eleventh annual ACM symposium on Theory of computing, 1979.