

IST ACOTES Project
Deliverable D2.2

Report on Streaming
Programming Model and
Abstract Streaming
Machine Description Final
Version

IST-034869

Public

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

| | | |
|---|---|---|
| Deliverable Number | : | D2.2 |
| Title of Deliverable | : | Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version |
| Nature of Deliverable | : | Report |
| Internal Document Number/version | : | acotes-d2.2-final |
| Contractual Delivery Date | : | 30 May 2008 |
| Actual Delivery Date | : | 8.September 2008 |
| 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 final version of the Acotes 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..... | 3 |
| 1 Introduction..... | 5 |
| 2 Streaming Programming Model..... | 6 |
| 2.1 Overview | 6 |
| 2.2 Execution model | 6 |
| 2.2.1 Task parallelism..... | 7 |
| 2.3 Data parallelism | 8 |
| 2.4 Memory model..... | 9 |
| 2.5 Construction of the task graphs..... | 9 |
| 2.5.1 Connecting named ports | 10 |
| 2.5.2 Connecting unnamed ports | 10 |
| 2.5.3 Connecting control ports | 10 |
| 2.6 Pragmas | 10 |
| 2.6.1 Taskgroup and task pragmas | 11 |
| 2.6.2 Taskgroup and task clauses | 12 |
| 2.6.3 Taskgroup and task example | 13 |
| 2.6.4 Teamreplicate pragma | 14 |
| 2.6.5 Pragmas supporting optimizations | 14 |
| 2.6.6 Auxiliary pragmas | 17 |
| 2.6.6.1 Update state variables asynchronously | 17 |
| 2.6.6.2 Setting the requirements of tasks..... | 18 |
| 2.6.6.3 Pragmas supporting vectorization | 18 |
| 2.7 Runtime library definitions | 18 |
| 2.7.1 Intrinsic functions..... | 18 |
| 2.7.1.1 Task_team_me function | 19 |
| 2.7.1.2 Task_team_size function | 19 |
| 2.7.1.3 Task_leader function | 19 |
| 2.7.2 Internal ACOLib functions | 19 |
| 2.7.2.1 Task_init..... | 20 |
| 2.7.2.2 Task_start | 20 |
| 2.7.2.3 Task_wait | 20 |
| 2.7.2.4 Task_copyin | 20 |
| 2.7.2.5 Task_copyout | 21 |
| 2.7.2.6 Task_oport..... | 21 |
| 2.7.2.7 Task_iport..... | 21 |
| 2.7.2.8 Iport_replicate | 22 |
| 2.7.2.9 Port_connect..... | 22 |

| | | |
|----------|---|-----------|
| 2.7.2.10 | Task_shared..... | 22 |
| 2.7.2.11 | Shared_async..... | 23 |
| 2.7.2.12 | Shared_sync..... | 23 |
| 2.7.2.13 | Task_close..... | 23 |
| 2.7.2.14 | Task_allopen..... | 23 |
| 2.7.2.15 | Copyin_acquire..... | 24 |
| 2.7.2.16 | Copyout_acquire..... | 24 |
| 2.7.2.17 | Oport_acquire..... | 24 |
| 2.7.2.18 | Oport_tryacquire..... | 24 |
| 2.7.2.19 | Oport_peek..... | 24 |
| 2.7.2.20 | Oport_push..... | 25 |
| 2.7.2.21 | Iport_acquire..... | 25 |
| 2.7.2.22 | Iport_tryacquire..... | 25 |
| 2.7.2.23 | Iport_peek..... | 25 |
| 2.7.2.24 | Iport_pop..... | 26 |
| 2.7.2.25 | Shared_acquire..... | 26 |
| 2.7.2.26 | Shared_update..... | 26 |
| 2.7.2.27 | Shared_check..... | 26 |
| 3 | Abstract Streaming Machine..... | 27 |
| 3.1 | Introduction to the ASM..... | 27 |
| 3.2 | Modeling the platform..... | 28 |
| 3.2.1 | Overview..... | 28 |
| 3.2.2 | Processor definition..... | 28 |
| 3.2.3 | Memory definition..... | 29 |
| 3.2.4 | Interconnect definition..... | 29 |
| 3.2.5 | Static routing..... | 30 |
| 3.2.6 | Summary of ASM Parameters..... | 31 |
| 3.3 | Modeling the compiled stream program..... | 33 |
| 3.3.1 | Overview..... | 33 |
| 3.3.2 | Tasks and subtasks..... | 33 |
| 3.3.3 | Sequencing of irregular tasks..... | 34 |
| 3.3.4 | Definition of a task..... | 34 |
| 3.3.5 | Statistical model..... | 36 |
| 3.4 | The cost model simulator..... | 38 |
| 3.4.1 | Trace-driven simulation..... | 38 |
| 3.5 | Integration with the compiler..... | 39 |
| 4 | Conclusions..... | 41 |
| | References..... | 42 |

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 final versions of both the SPM and the ASM. They are currently mostly stable, although the runtime system interface can still evolve during the last year of the project, as we plan to work on the adaptation of the programming model to use the Multicore Streaming Framework developed by IBM Haifa to support streaming applications in generic multicore environments and the Cell BE processor.

The rest of the document is structured as follows: Section 2 presents the Streaming Programming Model, Section 3 presents the Abstract Streaming Machine description, and Section 4 concludes the document.

2 Streaming Programming Model

2.1 Overview

The Streaming Programming Model (SPM) is a C extension. It consists of a set of directives extending serial code semantics. The SPM has to satisfy some requirements set in the context of the Acotes Project: it has to be easy to learn, easy to use and reuse, and provide data and task parallelism.

SPM is easy to learn: it can be learnt step by step. Like OpenMP, the programmer does not need to know all the programming model to start using it. There are only two main directives, *taskgroup* and *task*. They allow to exploit task parallelism. The average programmer can learn, and apply, one by one these directives. The programmer will see initial results by adding such a few lines.

SPM is easy to use: it is possible to incrementally modify an existing application. Like OpenMP, the programmer does not need to transform the whole application at once. During the transformation process, the programmer can evaluate the results and decide to go ahead or try another solution. The programming model assumes that the original serial application is correct. So it is better to debug first the application in its serial form, and then annotated it incrementally. This will ensure that existing data flow is correct as well.

SPM is easy to reuse: as it is based on plain C any code made in C is subject to be reused. The SPM can be used also in the building of libraries. Libraries are written in annotated C, and they can be used as C (by ignoring directives) or as a streaming library (by interpreting the directives).

The Streaming Programming Model 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 (*ports* from now on) from the point of view of the task, so that the producer has an output port to generate data into the stream and the consumer has an input port to read data from the 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 Execution model

The SPM allows to express parallelism based on a streaming model. Multiple processes (or threads) execute task kernels defined explicitly by SPM directives. SPM is intended to support programs that will execute correctly both as parallel programs (multiple processes of execution and full SPM library support) and sequential programs (directives ignored and simple SPM stubs library). However, it is possible and permitted to develop a program that executes correctly as streaming program but not as sequential program. It may happen that a program produces a different result when executed in parallel, compared to when it is executed sequentially.

The SPM is designed to be able to exploit task and data parallelism available in a program. Task parallelism is supported through taskgroups and tasks. The taskgroup establishes a portion of the program where streaming is present. Tasks are used inside taskgroups to express task parallelism. Teams of tasks allow to exploit data parallelism, by replicating the task, and distributing the work among them.

2.2.1 Task parallelism

A SPM program begins as a single process, called the *initial process*. The initial process executes sequentially, as the sequential part of the program. It has no streams or any kind of parallel synchronization.

When a process encounters a taskgroup construct, the process creates all tasks, instances and streams defined inside the taskgroup itself. Tasks created are the taskgroup implicit task, and its descendant tasks. For each task defined by the programmer, at least one instance is created. Each instance may require to create a new underlying process/thread or to reuse an existing one. Ports are created at this point to later establish the connections among tasks. Each process will execute the code contained in its task (the *kernel* of the task). Figure 1 shows an application with three taskgroup regions, each one with its tasks and stream connections.

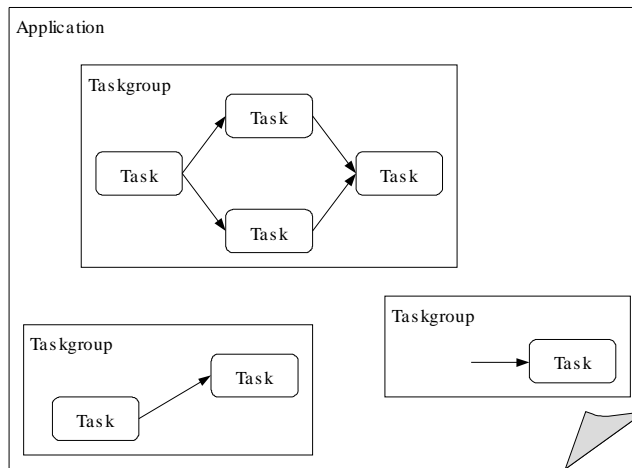


Figure 1: Application with three taskgroups

There is an implicit barrier at the end of the taskgroup construct. Beyond the end of the taskgroup construct, only the original process resumes the execution when all tasks have finished.

During the execution of a taskgroup, its implicit task and its descendant tasks are executed concurrently. Task constructs can be nested to facilitate stream connections, and they will also execute in parallel with the parent task. Task execution is sliced on task iterations. Task iterations are data driven. They are activated when input data set is available. Usually, the taskgroup implicit task produces the initial data. This initial data will activate all tasks and trigger all task iterations. When an instance of a task reads input data, and no data is available, the task instance blocks until new data arrives into the input stream. All tasks finalize when there is no more input data to consume.

A task is defined as a code block (the task kernel) with ports and state (see Figure 2). Port types are input or output. Input ports receive data to process, output ports are used to send results for further processing by other tasks. The task state consists of private memory, not shared, used during the execution of the task kernel. A task is activated for each data input set, it executes its kernel code, updates its state, and possibly generates an output.

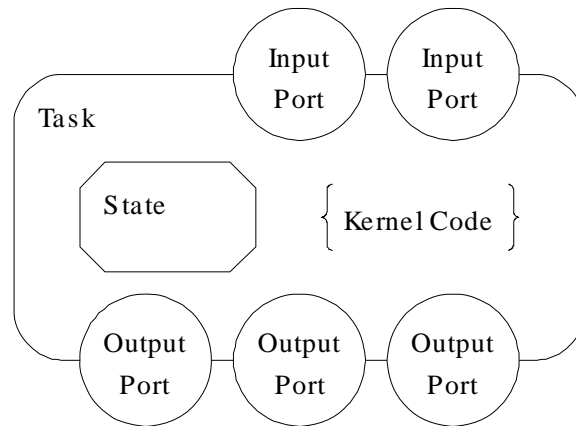


Figure 2: Task with two input ports and three output ports

2.3 Data parallelism

Data parallelism is defined generally as the distribution of data across multiple resources for parallel processing. Our SPM can exploit data parallelism, processing chunks of streams in parallel. To do so, a task is replicated in multiple instances, where each instance processes and generates a different set of data chunks. Tasks can have many inputs and many outputs. Inputs can be split (distribute data chunks among task instances) or replicated (all task instances receive the same data). The output must be merged, so each output chunk is generated by a single instance only.

We introduce the concept of team to deal with data parallelism in stream environments. We define a team as multiple instances of the same task, sharing the same ports, and replicating the task state. The state of each task instance is updated according to the input, but only one task instance (the leader instance) generates output. The leader itself changes with each chunk of data, so every task instance collaborates in generating the output across time. Figure 3 shows a team with three task instances.

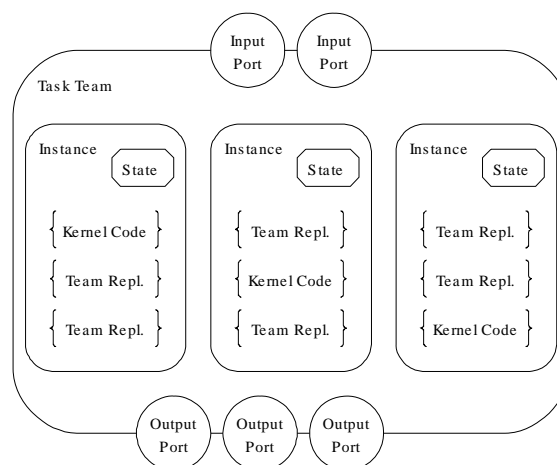


Figure 3: Team of three task instances

Input and output ports are shared among all instances. There are two types of input ports when working with teams: split and replicate. For each input port in split mode, data from this

port is sent only to the leader of the team at any time. For each input port in replicate mode, data from this port is sent to all instances (thus having the effect of replicating the data). Inputs in replicate mode are defined with the *inputreplicate* clause. Output data must be received from the team leader, so a merge operation and chunk reordering may be required.

When working with teams, each task instance is required to consume a certain number of consecutive elements, and produce a certain number of consecutive elements.

State is replicated for each task instance. State is not shared and each instance updates its own state. The *teamreplicate* pragma defines a subset of the kernel code as responsible to update the state. *Teamreplicate* code is executed for each iteration, regardless of the instance being the leader or not. The whole kernel code is executed only when the instance is the leader.

2.4 Memory model

The SPM assumes a distributed memory model. Multiple processes define independent and possibly distributed memory regions. The initial process, and the implicit task executed on it define the memory region of the sequential part of the application. Tasks created inside taskgroups have their own independent memory regions.

There are three kinds of variables, whose behaviour is defined by the SPM: local, state, and shared variables. Any variable not explicitly make state or shared is a local variable. Local variables are private to task instances, and they loose their values after each iteration of the task. Initialization and finalization of local variables default to the rules of the C language.

State variables are replicated for each task instance, and they keep their values across task iterations. All task instances define the same set of state variables, and they may keep different values on them. The initial value of state variables can be undefined, initialized or copied. The final value of state variables may be lost, finalized or copied. The *state* clause defines state variables with undefined initial values. The *initializestate* clause defines state variables with an associated initializer. The initializer code is copied to the beginning of the task region, and it is used to initialize each instance state variable. The *copyinstate* clause defines state variables whose values are copied from the implicit task executing the taskgroup. In this case, all instances are initialized with the same value. The final state value is lost by default. The *finalizestate* clause defines state variables with an associated finalizer. The finalizer code is copied at the end of the task region and finalizes each instance state variable. The *copyoutstate* clause defines state variables whose value is copied to the task executing the enclosing taskgroup. Only the leader instance value is copied.

Shared variables are state variables which values are synchronized by the SPM, from other task instances. There are two kinds of shared variables: synchronous, and asynchronous. The former are defined by the *sync* clause. Synchronously updated variables are updated from a source task using the *update* clause, and they are accessed from another task sharing the variable using the *check* clause. Asynchronous shared variables have the extra property that they may be updated automatically by the runtime system at any moment, so there is no need to use the check/update pair of clauses to get their values. More specifically, an implementation may choose to update such variables at the end of each iteration of the task.

2.5 Construction of the task graphs

The compiler builds the task graphs at compile time. During the construction of the graph, every port attached to a task must be connected with another port. Each connection will be established between one input port and one output port. Each output port can have multiple port connections to multiple input ports. In this case, the output data is automatically replicated

for each input port. Each input port must have one and only one connection, from an output port.

Graph construction is divided in three parts: graph construction for named ports, graph construction for unnamed ports, and graph construction for control ports. Named ports allow programmer to explicitly create portions of the program graph. Port labels (names) are used to identify which ports must be connected and which connections have to be created. Unnamed ports free the programmer of the tedious task of building the graph. In this case, the compiler uses variables specified in the pragmas to connect ports and create connections. Control ports are automatically created by the compiler for tasks without input ports. These ports ensure that those tasks are executed the correct number of times.

2.5.1 Connecting named ports

Connections of named ports are defined explicitly by the programmer using port labels. Each label is a name local to the taskgroup where the connection is made explicit. A label must be specified once for output, and it may be specified one or more times as input, so that multiple input ports can be connected to a single output port.

The port connection is done regardless of the variables affected and their types. Variables can be scalars or arrays. If the types of the variables are different, it is the programmer responsibility to ensure that the amount of data sent will match the amount of data expected in the receiver side. This allows interesting type transformations to happen during the communication of the data across tasks.

2.5.2 Connecting unnamed ports

Unnamed port connections are defined implicitly and automatically by the compiler, given a set of non-labeled (unnamed) pragmas specifying streams. The compiler has to create the connections of the input and output ports following the serial semantics expressed in the program regarding dependences of variables. This means that for each unnamed input port, a connection must be made with either the previous assignment to the variable, or the previous output of the variable.

Variables can be scalars or arrays. Their types are not checked by the compiler, and they have to agree in the amount of data sent and received.

2.5.3 Connecting control ports

A control port is automatically created by the compiler when a task has no other inputs. It is used to artificially send data, at a rate of one element per task activation, to ensure that the task is executed the appropriate number of times, following the serial semantics of the program.

2.6 Pragmas

We have taken OpenMP [OMP3.0] as a simple, well-designed and widely understood set of pragmas or directives. The OpenMP pragmas allow programmers 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].

2.6.1 Taskgroup and task 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 the region in which the tasks exist. It is named after the similar directive that was being proposed for the task support in the proposal for OpenMP 3.0 [OMP3.0]. It first initializes all its tasks, starts them, executes the body of the outer control task, and finally waits 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 (var, ...)]
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, in an asynchronous way.

When a process executing a task or a taskgroup encounters another taskgroup directive, it creates a new set of tasks. The enclosing task/taskgroup execution is suspended until the new taskgroup and its tasks are finished.

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 (var, ...)] [inputreplicate (var, ...)] [output (var, ...)]
[private(var, ...)] [shared (var, ...)] [async (var, ...)] [sync (var, ...)] [state(var, ...)]
[copyinstate(var, ...)] [copyoutstate(var, ...)] [initializestate (var, ...)] [finalizestate(var,
...)] [(team(scalar-integer-expr))]
structured-block
```

The task construct defines the structured-block as the task kernel. Any clause provided specifies a specific behaviour when translating the task kernel. The clauses are explained in the next section.

Code in structured-block that is surrounded by other task *pragmas* is excluded from this task kernel. The kernel code is executed for each input data set. When executed, the kernel computes the output data set, to be sent to the output streams.

Tasks can access variables in a number of ways defined by the different clauses. A task can have private and shared variables, synchronously and asynchronously updated variables, and state variables with different initialization and copy behaviours.

Tasks can be converted into teams using the *team* clause at the task construct. Task teams create multiple instances of the same task. The suggested number of instances is defined by

the *team* clause. The region of the code that is surrounded by the *taskreplicate* construct is replicated in all tasks of the team. Other computation is not replicated, and it is performed only in one of the tasks of the team. Task input data is split if the *input* clause is used, and replicated, when *inputreplicate* is specified on the input data. Task output data is produced in a round-robin fashion by the different members of the team (when the *task_team_leader()* primitive returns true). The *task_team_me()* primitive returns the identifier of the task inside the team, and *task_team_size()* returns the total number of task instances created in the team.

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 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 control stream as an implicit input. The format of such control stream is implementation dependent. It could for example carry one data element each time that the task should be executed.

2.6.2 Taskgroup and task clauses

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

When using teams, the *input* clause causes the data to be split and distributed across all instances of the task. To avoid splitting, the *inputreplicate* clause must be used. With the *inputreplicate* clause, an automatic replicator is set to provide each member of the team with exactly the same data.

Private (*v1, v2, ..., vn*) defines variables to be local to the task, with undefined initial values. The clauses *shared* (*v1, ...*), *async*(*v1, ...*), and *sync*(*v1, ...*) define shared variables with different update characteristics. *Shared*(*v1, v2, ..., vn*) defines *n* shared variables, and it implies the need of shared memory between the tasks accessing those variables. The clause *async*(*v1, v2, ..., vn*) defines the set of shared variables that will be explicitly updated from inside the inner tasks or the taskgroup itself, in an asynchronous way. The clause *sync*(*v1, ...*) defines the same kind of shared variables, but forcing a synchronous update, for instance, at the beginning of each task iteration.

State clauses define variables that will keep their values across task activations. Such variables will be private for each task in a team. The modifier clauses *copyinstate* and *copyoutstate* change the way the state variables are initialized and copied back. With *copyinstate*, a state variable is initialized in all task instances with the value available in the taskgroup. The clause *copyoutstate* causes the final value of the state variables in the team leader to be copied back to the original variable at the taskgroup level. The same variable can be annotated both as *copyinstate* and *copyoutstate*.

Sometimes the initializer/finalizer code for some state variables is in a different place in the serial application, with respect to the definition of the tasks where the variables are used. For such situation, we provide the *initializestate* and *finalizestate* clauses. They name the variables that have an associated *initializer* and/or *finalizer* codes, so that such codes are copied into the task, before and after the task kernel, to correctly initialize and finalize the variables.

The *team* clause allows the creation of task teams. The *team* clause enables task replication and suggests the number of instances that would be appropriate to generate. It is used to exploit data parallelism inside a task.

2.6.3 Taskgroup and task example

Figure 4 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 5a. Figure 5b 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. Such examples are provided in the current AcotesCC prototype [Rod08].

```
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 4: Annotated C code for the *tolower* example

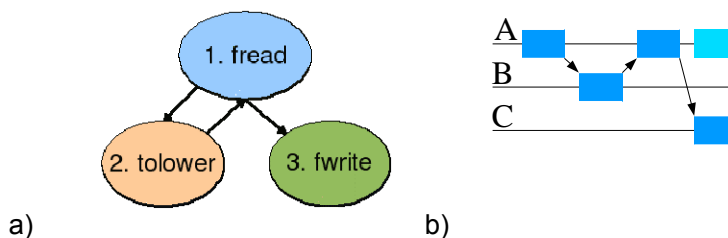


Figure 5: *Tollower* unoptimized (a) graph, and b) pipeline structure

2.6.4 Teamreplicate pragma

The *teamreplicate* construct defines a code region to be replicated in all instances of a team of tasks. The code region is computed in all task instances for all task activations. For instance, the code outside any *teamreplicate* construct is executed only by the leader instance. *Task_team_leader()* is always true outside a *teamreplicate* construct. An example of use is provided in the AcotesCC prototype [Rod08].

```
#pragma acotes teamreplicate
structured-block
```

2.6.5 Pragmas supporting optimizations

The fact that all stream communication should go through the control task (the one associated with the taskgroup) 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 providing this second option at the directive level. There is an easy solution, adding a *bypass* clause, that indicates, in addition to the variables involved in the communication, the name of the variable that can be bypassed from the source task to the destination one. The *bypass* syntax is as follows:

```
#pragma ... bypass (var, ...)
code-block
```

Figure 6 shows the *tolower* example again, now with the use of this clause. It allows the compiler to automatically connect the output stream of task *tolower* to the input of task *fwrite*. Figure 7a 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 7b. This optimized version corresponds to what the programmer expects from the optimized 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 bypass (x) /* fread */
    while (!feof(in))
    {
        #pragma acotes task input(c) output(x:WRITE) /* tolower */
        if ('A' <= c && c <= 'Z') x = c - 'A' + 'a';
        else x = c;

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

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

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

```

Figure 6: Optimized code for the tolower example

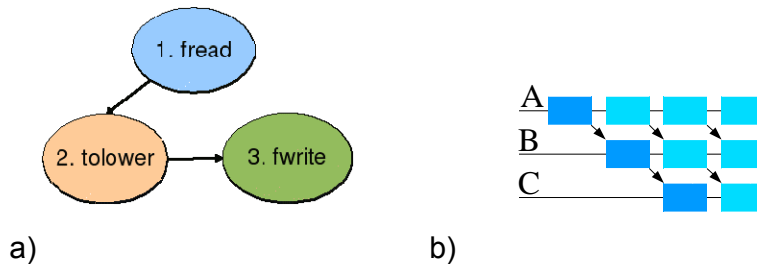


Figure 7: Tolower optimized (a) graph, and (b) pipeline structure

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 8 presents an example of such a construct, and Figure 9 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 8: Code showing the use of the for-replicate directive

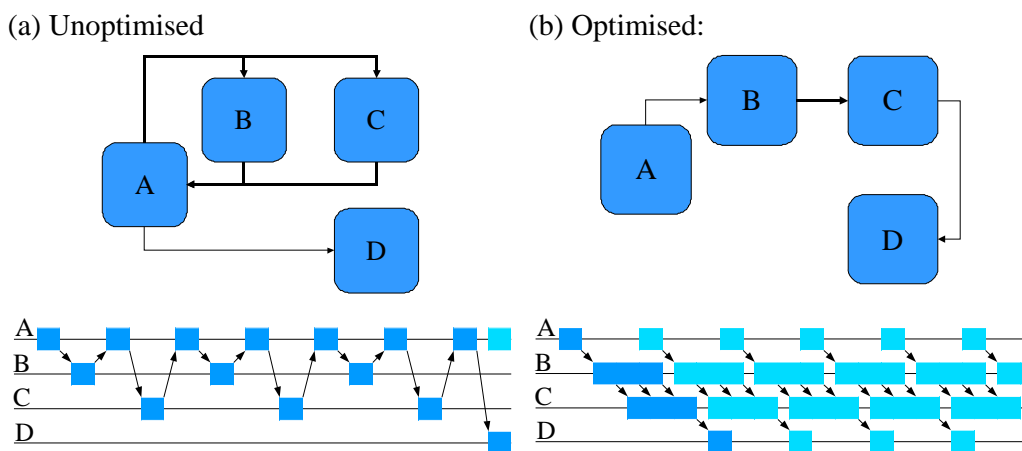


Figure 9: Graph and pipeline for the code of Figure 8

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.

Port constructs are used to improve task interfaces. They allow to express explicit synchronization, port optimizations and improve graph construction.

```

#pragma acotes port [input (v, ...) [inputreplicate (v, ...) [output (v, ...) ]

```

The port pragma adds ports to the current task interface, and performs explicit synchronizations. It is used to set the data movement and synchronization points to the place where the port pragma is used. The execution of this pragma consists of receiving any input data from the input ports, and sending any output data to the output ports. Synchronization is performed by blocking the current task, when input data cannot be received or output data cannot be sent.

The ports defined with this pragma, have a name corresponding to labels associated to the named variables. Port labels are mandatory in this case. An example of use is provided in the AcotesCC prototype [Rod08].

Streaming applications usually express the peek optimization to access directly to the input streams of data, checking for some future data to arrive. In Acotes, this is supported through the *peek* pragma.

```
#pragma acotes peek [inputvalue;historyarray[windowsize][;index])
    structured-block
```

We have implemented a kind of idiom recognition, to transform the associated structured-block, from a regular array access to the peek operation on the input stream. The peek construct allows the compiler to translate history access array into port peek operations. It requires an structured-block, an *inputvalue*, a *historyarray*, a *windowsize*, and optionally an *index*. The *structured-block* is the serial program code region that stores the value of the variable *inputvalue* inside the *historyarray* and updates the *index* (if the index is specified). The *inputvalue* is the symbol of one input port. The *historyarray* is the symbol of a state array variable. The *windowsize* is the size of the maximum number of elements accessed in the history. The *index* is an optional pointer to the eldest element of *historyarray*, if not specified the eldest is inside position 0.

The *structured-block* is removed and replaced by the consumption of the *inputvalue* element. The *historyarray* variable accesses are replaced by peek operations on *inputvalue* port, and the *historyarray* state variable is removed. If *historyarray* state variable is a copyinstate variable, the *windowsize* initial values are used as initial *inputvalue* elements. If *historyarray* is not a copyinstate the first task iteration is delayed until *windowsize* elements have been received. The *windowsize* is the minimum size of the peek window. If the *index* variable is specified, all accesses to the index variable are replaced by 0, and the *index* variable is removed. An example of use of the *peek* pragma is provided in the AcotesCC prototype [Rod08].

2.6.6 Auxiliary pragmas

2.6.6.1 Update state variables asynchronously

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.

2.6.6.2 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.

2.6.6.3 Pragmas supporting vectorization

Vector directives are hints used to assist the compiler to automatically generate vectorized code. The directives presented here provide hints about lack of loop dependences, expected values and variable alignments. Currently, the acotes compiler leaves such pragmas in the output code, so that they can be processed as GCC pragmas by the backend compiler.

```
#pragma acotes ivdep  
for-statement  
  
#pragma acotes dividesby (expression, value)  
  
#pragma acotes aligned (var, alignment)
```

The *pragma ivdep* is the usual one, that is already recognized by some vectorizing compilers, and it indicates that the following loop can be vectorized, even if the compiler cannot analyze properly its data dependences.

The *pragma dividesby* indicates that the given expression divides perfectly by the value provided. If the compiler is told that the limits of a loop are divisible by 4, for instance, then it will be able to apply vectorization without considering the need of generating prologues or epilogs.

The *pragma aligned* indicates that the given variable (*var*) is aligned in memory to at least the amount of memory indicated by *alignment*. This also will help the vectorizer to decide which is the appropriate approach for vectorization on such a variable.

2.7 Runtime library definitions

This section presents the runtime API available to the Acotes programmers, and the internal primitives used to support the programming model, currently implemented as part of the ACOLib library.

2.7.1 Intrinsic functions

These functions can be used by the programmer to get information about the current execution environment, and identify tasks and teams.

2.7.1.1 Task_team_me function

```
int task_team_me(void);
```

The `task_team_me` routine returns the number of instance inside a task team. All task instances of the same team have an identifier ranging from 0 to the team size minus one. It is expected that the instance 0 is responsible (`task_team_leader` returns true) for the first input data set chunk, the 1 for the second, and so on.

The binding region for the `task_team_me` function is a team task region. Any call outside this region returns 0.

2.7.1.2 Task_team_size function

```
int task_team_size(void);
```

The `task_team_size` routine returns the number of task team instances in the current team.

The binding region for the `task_team_size` primitive is a team task region. Any call outside this region returns always 1.

2.7.1.3 Task_leader function

```
int task_leader(void);
```

The `task_leader` routine returns a non-zero positive integer (true) if the current task instance is responsible to process the task input data set and generate an output data set. Returns zero (false) otherwise.

The binding region for a `task_leader` function call is a *teamreplicate* construct region. Any call outside this region returns always true.

2.7.2 Internal ACOLib functions

This subsection presents a short description of all runtime library calls generated by the compiler to deal with application level routines. These routines are used from the *taskgroup* construct to create and define all tasks and the stream graph.

Task are represented by the opaque `task_t` type defined as follows:

```
typedef struct task* task_t;
```

2.7.2.1 Task_init

```
int task_init(task_t* task, void (*start)(task_t), int team_count);
```

The `task_init` routine initializes the `task_t` type, and creates a new task. The parameters are the following:

- `task`: a valid `task_t` pointer to an uninitialized `task_t` variable.
- `start`: a pointer to the task routine. This routine will process all stream elements.
- `team_count`: a suggested number of instances, 1 for a single instance. A larger number, indicates the number of suggested members of the team.

2.7.2.2 Task_start

```
void task_start(task_t task);
```

The `task_start` routine starts the task provided. After this function is executed, task ports, state variables, shared variables, and connections cannot be changed. The parameters are the following:

- `task`: an initialized task to start.

2.7.2.3 Task_wait

```
void task_wait(task_t task);
```

The `task_wait` routine waits for the completion of the provided task. After this call, none of the task ports can be used. The parameters are the following:

- `task`: an started valid task.

2.7.2.4 Task_copyin

```
void task_copyin(task_t task, int state, void * src, int size);
```

The `task_copyin` routine defines a state variable with an initial value. The parameters are the following:

- `task`: an initialized valid task.
- `state`: an id number for the state variable. If it is the first time it is supplied, the state variable is created and initialized.
- `src`: a pointer to a local variable containing the initial value for the state variable. The value of the local variable will be copied before starting the target task.
- `size`: size of the state variable.

2.7.2.5 Task_copyout

```
void task_copyout(task_t task, int state, void * dst, int size);
```

The task_copyout routine defines a state variable whose final value must be saved. The parameters are the following:

- *task*: an initialized valid task.
- *state*: an id number for the state variable. If it is the first time it is supplied, the state variable is created and initialized.
- *dst*: a pointer to a local variable where to save the final value of the state variable. The value will be copied upon task termination.
- *size*: size of the state variable.

2.7.2.6 Task_oport

```
void task_oport(task_t task, int port, int esize, int gcount, int  
               bsize);
```

The task_oport routine defines an output port. The parameters are the following:

- *task*: an initialized valid task.
- *port*: an id number for the port buffer. If it was not already defined, a new port is defined.
- *esize*: element size in bytes.
- *gcount*: a group element count. This is a group element count that must be processed consecutively to achieve a correct result. This is used to compute elements distribution when the destination of the port is a team with several instances.
- *bsize*: the minimum number of elements required to be placed into the port buffer.

2.7.2.7 Task_iport

```
void task_iport(task_t task, int port, int esize, int gcount, int  
               bsize, void *ivalues, int isize);
```

The task_iport routine defines an input port. The parameters are the following:

- *task*: an initialized valid task.
- *port*: an id number for the port buffer. If it was not already defined, a new port is defined.
- *esize*: element size in bytes.
- *gcount*: a group element count. This is a group element count that must be processed consecutively to achieve a correct result. This is used to compute elements distribution when the destination of the port is a team with several instances.

- *bsize*: the minimum number of elements required to be placed into the port buffer.
- *ivalues*: a pointer to a buffer with the initial values to be pushed back to the input port. This affects only to the stream end defined by *task* and *port*.
- *isize*: the number of elements to be pushed back from *ivalues* buffer.

2.7.2.8 Iport_replicate

```
void iport_replicate(task_t task, int port);
```

The `iport_replicate` routine defines an input port as a team replicated input port. This port will replicate the values towards all task instances instead of splitting them. The parameters are the following:

- *task*: an initialized valid task.
- *port*: a code number for the port buffer previously defined with a task `iport` operation.

2.7.2.9 Port_connect

```
void port_connect(task_t ot, int oport, task_t it, int iport);
```

The `port_connect` routine defines a connection between one output port and one input port. An output port can be connected to multiple input ports, but input ports are only connected to one input port. This routine can create more than one stream, when the tasks involved have more than one instance. The parameters are the following:

- *ot*: an initialized valid task where the output port is defined.
- *oport*: a valid port number previously defined with task `oport` operation.
- *it*: an initialized valid task where the input port is defined.
- *iport*: a valid port number previously defined with task `iport` operation.

2.7.2.10 Task_shared

```
void task_shared(task_t task, int state, void* ivalue, int size);
```

The `task_shared` routine defines a state variable as a shared variable. The parameters are the following:

- *task*: an initialized valid task.
- *state*: an id number for the state variable.
- *ivalue*: a pointer to a local variable containing the initial value for the state variable. On shared memory implementations, shared state may reside on the *ivalue* buffer.
- *size*: size of the shared state variable created.

2.7.2.11 Shared_async

```
void shared_async(task_t utask, int ustate, task_t atask, int
                   astate);
```

The `task_async` routine defines an asynchronous update relationship between two shared state variables. Modifications on the task updating the value are propagated asynchronously towards the shared state variable. The parameters are the following:

- *utask*: an initialized valid task where the state variable is defined.
- *ustate*: a valid shared variable id previously defined with a task shared operation.
- *atask*: an initialized valid task where the asynchronous state is defined.
- *astate*: a valid shared variable id previously defined with a task shared operation.

2.7.2.12 Shared_sync

```
void shared_sync(task_t utask, int ustate, task_t stask, int
                   sstate);
```

The `task_sync` routine defines a synchronous update relationship between two shared state variables. Modifications on the task updating the value are propagated synchronously towards the shared state variable. The parameters are the following:

- *utask*: an initialized valid task where the state variable is defined.
- *ustate*: a valid shared variable id previously defined with a task shared operation.
- *stask*: an initialized valid task where the synchronous state is defined.
- *sstate*: a valid shared variable id previously defined with a task shared operation.

2.7.2.13 Task_close

```
void task_close();
```

The `task_close` routine closes all output streams, and signals to the following tasks in the graph that there are no more stream elements to process. After this call no output port operations are permitted.

2.7.2.14 Task_allopen

```
int task_allopen();
```

The `task_allopen` routine checks for the state of all input ports. This is a non-blocking call that requires a previous call to an acquire operation to check that there are no more elements. It returns 0 (false) if there is at least one acquire operation on any task input port unable to peek

all requested elements because its output port was closed previously. It returns non-zero (true) otherwise.

2.7.2.15 Copyin_acquire

```
void* copyin_acquire(int state);
```

The copyin_acquire routine obtains a pointer to the state buffer of a task state variable. The result is a pointer to the state buffer to work with. The parameters are the following:

- *state*: the id number for a state variable matching a task copyin operation.

2.7.2.16 Copyout_acquire

```
void* copyout_acquire(int state);
```

The copyout_acquire routine obtains a pointer to the state buffer of a task state variable. The result is a pointer to the state buffer to work with. The parameters are the following:

- *state*: the id number for a state variable matching a task copyout operation.

2.7.2.17 Oport_acquire

```
void oport_acquire(int port, int ecount);
```

The oport_acquire routine prepares the next elements from the port to peek. This function is a blocking function. The parameters are the following:

- *port*: a valid task output port number.
- *ecount*: number of requested elements to peek.

2.7.2.18 Oport_tryacquire

```
int oport_tryacquire(int port, int ecount);
```

The oport_tryacquire routine prepares the next elements from the port to peek. This function is a non-blocking function. It returns the maximum number of elements acquired, or zero(0) if it would block. The parameters are the following:

- *port*: a valid task output port number.
- *ecount*: number of requested elements to peek.

2.7.2.19 Oport_peek

```
void* oport_peek(int port, int enumber);
```

The oport_peek routine requests the buffer location of a selected element of the stream. This is a non-blocking call. An advanced runtime system should provide a pointer with all valid

elements consecutively starting from *enumber* 0 to *gcount*, if they are properly acquired. The result is a pointer to the buffer where to store the result. The parameters are the following:

- *port*: a valid task output port number.
- *enumber*: the element number to be peeked. 0 is the first to be pushed, 1 the second, and so on. All peeked elements must be acquired.

2.7.2.20 **Oport_push**

```
void oport_push(int port, int ecoun);
```

The `oport_push` routine pushes a set of elements to all connected input ports. This can be a blocking call, but some platforms may ensure that push operations are always possible by blocking at acquire operation. The parameters are the following:

- *port*: a valid task output port number.
- *ecoun*: number of requested elements to push. All elements must have been previously acquired.

2.7.2.21 **Iport_acquire**

```
void iport_acquire(int port, int ecoun);
```

The `iport_acquire` routine prepares the next elements from the port to peek. This function is a blocking function. The parameters are the following:

- *port*: a valid task input port number.
- *ecoun*: number of requested elements to peek.

2.7.2.22 **Iport_tryacquire**

```
int iport_tryacquire(int port, int ecoun);
```

The `iport_tryacquire` routine prepares the next elements from the port to peek. This function is a non-blocking function. It returns the maximum number of elements acquired. The parameters are the following:

- *port*: a valid task output port number.
- *ecoun*: number of requested elements to peek.

2.7.2.23 **Iport_peek**

```
void* iport_peek(int port, int enumber);
```

The `iport_peek` routine requests the buffer location of a selected element of the stream. This is a non-blocking call. An advanced runtime system should provide a pointer with all valid elements consecutively starting from *enumber* 0 to *gcount*, if they are properly acquired. The result is a pointer to the buffer where to store the result. The parameters are the following:

- *port*: a valid task output port number.
- *enumber*: the element number to be peeked. 0 is the first to be popped, 1 the second, and so on. All peeked elements must have been acquired.

2.7.2.24 **lport_pop**

```
void lport_pop(int port, int ecount);
```

The `lport_pop` routine pops a set of elements from the given port. This is a non-blocking call. The parameters are the following:

- *port*: a valid task output port number.
- *ecount*: number of requested elements to pop. All elements must have been previously acquired.

2.7.2.25 **Shared_acquire**

```
void* shared_acquire(int state);
```

The `shared_acquire` routine obtains a pointer to the shared state buffer of a task shared state variable. The result is a pointer to the state buffer to work with. The parameters are the following:

- *state*: a code number for a state variable matching a task shared operation.

2.7.2.26 **Shared_update**

```
void shared_update(int state);
```

The `shared_update` routine requests an explicit update of the remote shared buffers with the value of the buffer obtained by the shared acquire operation. Some architectures may implement shared buffers on shared memory, so explicit update will have no effect.

2.7.2.27 **Shared_check**

```
void shared_check(int state);
```

The `shared_check` routine requests an explicit update of its own buffer obtained by a shared acquire operation. Some architectures may update automatically shared buffers and this operation may have no effect.

3 Abstract Streaming Machine

3.1 Introduction to the ASM

The *Abstract Streaming Machine* (ASM) is a model of the target multiprocessor system that allows the compiler to automatically partition and schedule a streaming program to run on it. The ASM supplements the compiler's back-end machine description, which is still required for code generation. It defines the system topology, memories, interconnects, and the behaviour of processors beyond the ISA and micro-architecture.

As illustrated in Figure 10, the compiler controls the search for the optimal mapping, and repeatedly invokes a *cost model* based on the ASM. The cost model estimates the performance of a candidate static mapping, given a description of the program, the target architecture and the mapping. The compiler uses the throughput and resource utilization metrics calculated by the cost model to guide the search for a near optimum mapping.

We have implemented the cost model as a simulator, which evaluates the performance of a static mapping of the stream program onto the target, given the target's ASM machine description. It generates a trace in the Paraver format [Par08], as well as resource utilization and throughput figures. In the future, it may be feasible to define a closed form statistical model that does not require simulation, but this work is beyond the scope of the ACOTES project.

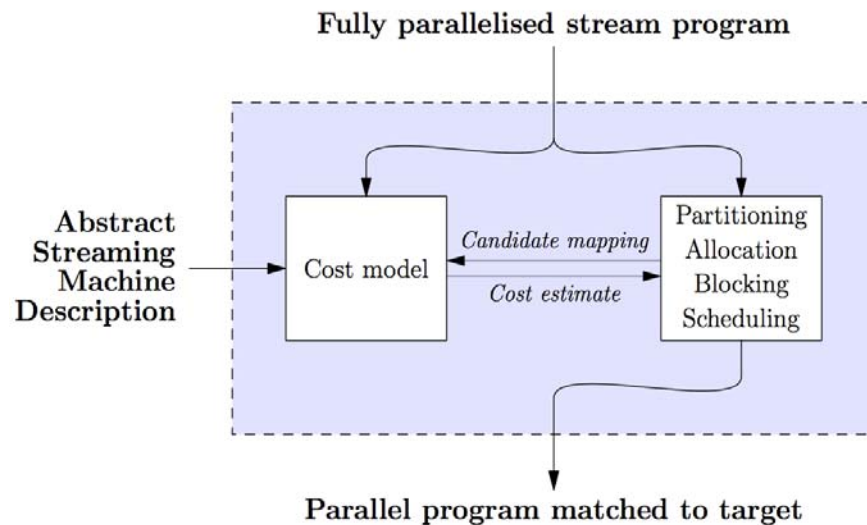


Figure 10: Cost model and search algorithm in the compiler

The cost model has two ways to control the precise behaviour of the stream program. In the *trace-driven* mode, the program simulation is controlled by a trace obtained from a real execution. This trace remains valid even after changing the partitioning or allocation of tasks. In the *static* mode, the program simulation is controlled by a statistical model.

3.2 Modeling the platform

3.2.1 Overview

The target system is represented as a bipartite graph of processors and memories in one partition, and interconnects in the other. Figure 11 shows the topology of a Cell processor, which includes an interconnect between each SPE and its local store.

The ASM describes the target visible to the stream program, and is therefore affected by the Operating System, virtual machine, and the implementation of *acolib* [Rod07]. The OS may hide certain aspects of the physical hardware, such as the mapping of logical to physical processor cores. We assume that while the stream program is running, it has exclusive access to the cores that it uses, so that it is not preempted by the OS in favour of other applications.

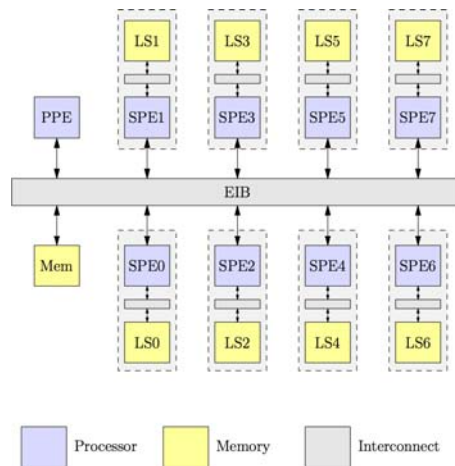


Figure 11: Topology of a Cell-based system

3.2.2 Processor definition

Each processor is defined using the parameters given in Figure 14(a). As noted above, the details of the processor's ISA and micro-architecture are described internally to the back-end compiler, so are not duplicated in the ASM. The compiler estimates the cost of the computational part of each task, excluding communications primitives, on its assigned processor. The cost model adds the processor cost of the communications primitives, given in this table. The cost of the `pushSend` and `popAcquire` primitives is defined by a staircase function; i.e. a fixed minimum cost, a block size, and an incremental cost for each complete or partial block after the first. This is required to accurately model the cost of programming multiple DMA transfers on, for example the Cell processor, which has a maximum DMA transfer size of 16 kilobytes. In our implementation of *acolib* for Cell SPEs, the DMA transfer is programmed in the `pushSend` primitive, and its execution time is clearly a staircase function. The ASM supports a single implementation of *acolib* per processor, so it is not possible for a given processor to support more than one means of communication; e.g. DMA and FIFO, with significantly different costs.

The `addressSpace` parameter defines the local address space of the processor; i.e. which memories are accessible by ordinary load-store instructions, and where they appear in local virtual memory. We assume that the local address space is known at compile time (but we do

not make the same assumption for the global address space used to program DMA). This information is primarily required to decide where to place the producer's and consumer's buffers.

The `hasIO` parameter defines which processors can perform system IO, and is a simple way to ensure that tasks that need system IO are mapped to a capable processor.

3.2.3 Memory definition

Each memory is defined using the parameters shown in Figure 14(b). The latency and bandwidth figures are currently unused in the cost model, but may be used by the compiler to refine the estimate of the computational part of each task.

3.2.4 Interconnect definition

Each interconnect is defined using the parameters shown in Figure 14(c). The `elements` parameter connects each interconnect to a set of processors and memories, thus defining the system topology. Each interconnect is modelled as a bus with multiple channels, which has been shown to be a good approximation to the real performance when processors and memories on the link are all equidistant [Gir00].

Each bus has a single unbounded queue to hold messages ready to be transmitted, and one or more channels on which to transmit them. Streams are statically allocated onto buses, but the choice of channel is made at runtime. It is not possible for the same processor to transmit on more than one channel of the same interconnect simultaneously. Similarly for receive, so the transmitter must gain exclusive access to the receiver for this interconnect before it begins transmission. The `interfaceDuplex` parameter defines whether a processor can simultaneously transmit and receive on different channels.

The bandwidth and latency of each channel is specified using four parameters: the *start latency* (L), *start cost* (S), *bandwidth* (B), and *finish cost* (F). Figure 12 shows the timeline of a single transfer. The start latency is measured from the beginning of the `pushSend` primitive on the producer until the earliest time at which the message can begin transmission. At this time, the message is added to the back of the queue for the interconnect. When there is an available channel, and both the transmitter and receiver are not already busy as described above, the message nearest, but not necessarily at, the head of this queue begins transmission. It is possible for the message to begin transmission in the next cycle. It is also possible for more than one message to be enqueued and dequeued in any particular cycle; in the former case the order in the queue is undefined.

The transmission is comprised of a start phase, of length S cycles, immediately followed by a transfer phase, at B bytes per cycle, immediately followed by a finish phase, of length F cycles. The message is available to the receiver immediately at the end of the message transfer phase. In summary, when transferring a message of size n bytes, the latency is given by $L + S + \text{ceil}(n/B)$ and the cost incurred on the interconnect is given by $S + \text{ceil}(n/B) + F$. Recall that the cost incurred by the processor is measured separately. There is some redundancy in this model, as it controls two values using three independent variables, but it is easy to understand and implement.

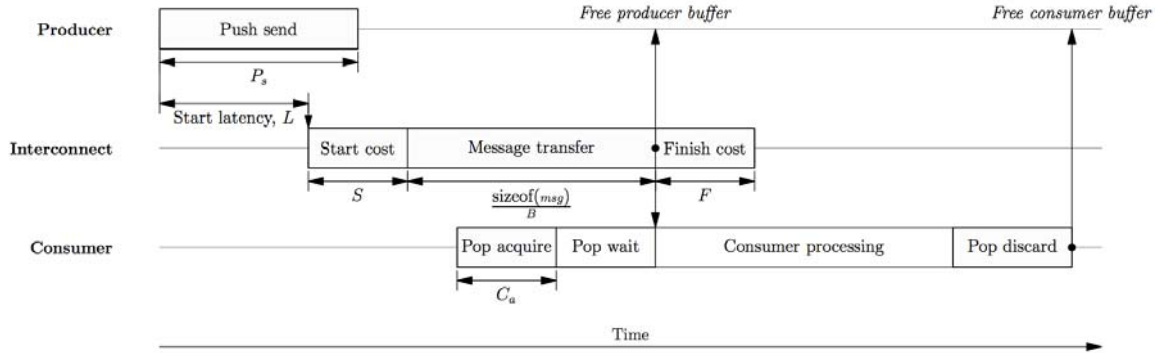


Figure 12: Timeline showing cost and latency of communication

3.2.5 Static routing

Hardware routing is controlled using the `interfaceRouting` parameter, which defines for each processor whether it can route messages from a particular interconnect. Each stream is assigned a static route, with messages being forwarded automatically if required by the mapping and supported by the hardware. The `interfaceRouting` parameter may be *None*, in which case no routing is supported at the node, or it may be one of the supported forwarding schemes: *storeAndForward*, or *cutThrough*. For both routing schemes, the intermediate buffering is unbounded, so there is no back pressure.

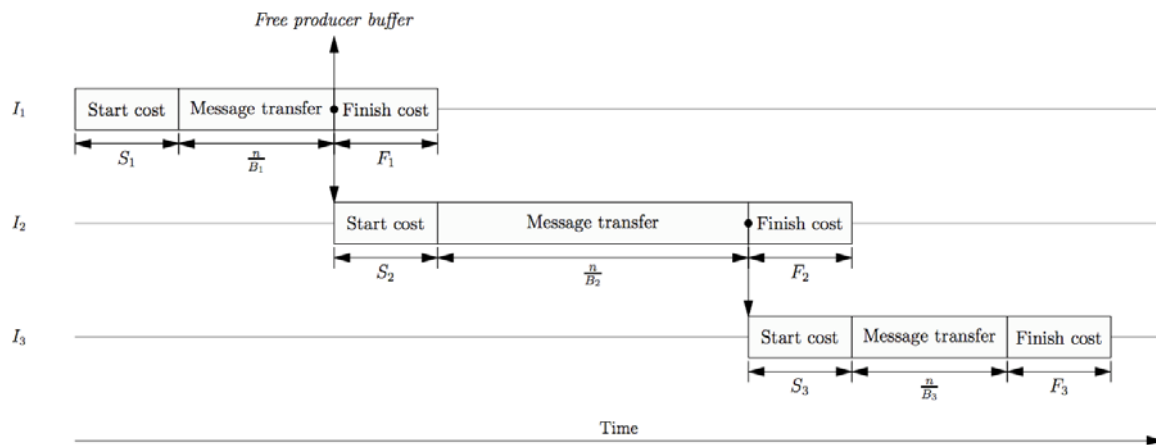
Figure 13 compares store and forward with cut through routing for an example message routed on three links, l_1 , l_2 , and l_3 , where l_1 and l_3 both have twice the bandwidth of l_2 . In the general case, a given stream follows the static route l_1, l_2, \dots, l_k , where l_j has parameters L_j, S_j, B_j , and F_j . Consider the progress of a particular message on the stream using the following notation. On interconnect l_j , the message is added to the back of the queue at time t_j , begins the start phase at some later time u_j and begins the finish phase at time $v_j \geq u_j + S_j + \text{ceil}(n/B_j)$.

For store and forward routing, illustrated in Figure 13(a), the whole message must be received completely on the incoming interconnect before it can begin retransmission. It is added to the back of the queue just after it finishes the transfer phase on the previous interconnect, and when it begins retransmission, it does so at full outgoing bandwidth. For cut through routing, illustrated in Figure 13(b), the message is added to the back of the queue after the start phase on the previous interconnect, but it may not use the full outgoing bandwidth if it is greater than the incoming bandwidth, since doing so may cause data to have to be retransmitted before it is received.

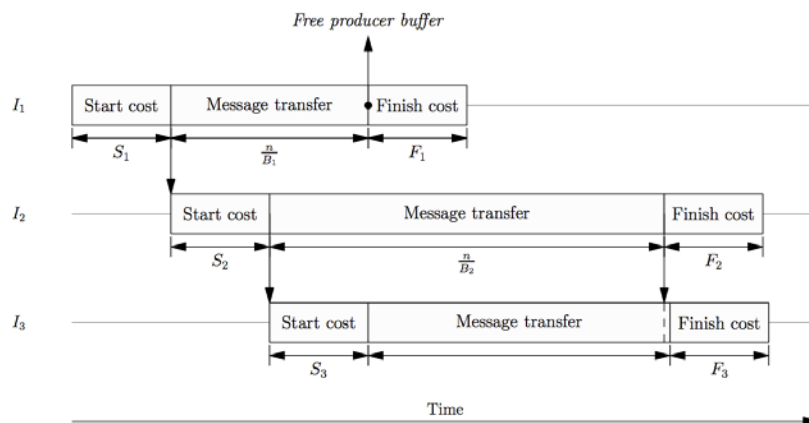
Hence,

$$t_j = \begin{cases} v_{j-1} & \text{store and forward} \\ u_{j-1} & \text{cut through} \end{cases}$$

$$v_j = \max[1+v_{j-1}, u_{j-1} + S_j + \text{ceil}(n/B_j)].$$



(a) Store and forward routing



(b) Cut through routing

Figure 13: Comparison of store and forward vs cut through routing

3.2.6 Summary of ASM Parameters

In summary, the machine description defines the processors, memories and interconnects using the parameters listed in Figure 14.

| Parameter | Type | Description |
|-------------------|-------------|---|
| name | String | Unique name in platform namespace |
| clockRate | Fixed point | Clock rate, in GHz |
| hasIO | Bool | True if the processor can perform IO |
| addressSpace | List | List of the physical memories in the system that are addressable by this processor with ordinary load-store instructions, and their virtual addresses |
| pushAcqCost | Int | Cost, in cycles, of acquiring an empty producer buffer (before waiting) |
| pushSendFixedCost | Int | Fixed cost, in cycles, of pushing a block (before waiting) |
| pushSendUnit | Int | Number of bytes per push transfer unit |
| pushSendUnitCost | Int | Incremental cost, in cycles, to push <code>pushSendUnit</code> bytes |
| popAcqFixedCost | Int | Fixed cost, in cycles, of popping a block (before waiting) |
| popAcqUnit | Int | Number of bytes per pop transfer unit |
| popAcqUnitCost | Int | Incremental cost, in cycles, to pop <code>popAcqSendUnit</code> bytes |
| popDiscCost | Int | Cost, in cycles, of discarding a consumer buffer (before waiting) |

(a) Definition of a processor

| Parameter | Type | Description |
|-----------|-------------|-----------------------------------|
| name | String | Unique name in platform namespace |
| clockRate | Fixed point | Clock rate, in GHz |
| size | Int | Size, in bytes |
| latency | Int | Access latency, in cycles |
| bandwidth | Int | Bandwidth, in bytes per cycle |

(b) Definition of a memory

| Parameter | Type | Description |
|------------------|-------------|---|
| name | String | Unique name in platform namespace |
| clockRate | Fixed point | Clock rate, in GHz |
| elements | List | List of the names of the elements (processors and memories) on the bus. |
| interfaceDuplex | List | If the bus 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 bus |
| multiplexable | Bool | False for a hardware FIFO that can only support one stream |

(c) Definition of an interconnect

Figure 14: Parameters of the Abstract Streaming Machine

Figure 15 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 15: Platform description for a 2x2 mesh

3.3 Modeling the compiled stream program

3.3.1 Overview

The compiled stream program is a directed graph of *tasks* and point-to-point *streams*. Each task repeats the work function inside an implicit loop, and may have any number of input and output streams. All synchronization between tasks is due to blocking on empty or full stream buffers. It is possible for a task to have no input streams and/or no output streams. The program must be a connected graph, but it may be collapsed into a single task with no inputs *and* no outputs. The parameters used in the program description are presented in Figure 20.

3.3.2 Tasks and subtasks

A task may have complex data-dependent or irregular behaviour. The basic unit of sequencing inside a task is the *subtask*, which pops a fixed number of elements from each input stream and pushes a fixed number of elements on each output stream. In more detail, a subtask is divided into three phases. First, the *acquire phase* obtains the next set of full input buffers and empty output buffers. Next, the *processing phase* works locally on these buffers, performing a fixed amount of computation. Finally, the *release phase* discards the used input buffers and sends the contents of the output buffers, releasing the buffers in the same order that they were acquired.

This three-stage model is a restriction on the collection of programs that can be represented. In particular, Figure 16 shows a task that cannot be split into subtasks of the above form, since the streams are not released in the same order in which they are acquired. It is not a deep requirement of the ASM, and was introduced as a convenience in the implementation of the simulator. We will ensure that the ACOTES compiler will naturally generate subtasks of the correct form.

```

while (1)
{
    instream_acquire(x);
    ostream_acquire(y);
    ostream_send(y);
    instream_discard(x);
}

```

Figure 16: A task that cannot be split into subtasks

Streams are point-to-point, meaning that each stream has exactly one producer task and exactly one consumer task. Those tasks may, however, access the same stream from more than one subtask, assuming that the subtasks either all pop or all push.

3.3.3 Sequencing of irregular tasks

The ASM uses the sequential semantics of the original SPM program to control the sequencing of subtasks in the compiled stream program. In particular, there must be a mapping from the subtasks of the stream program onto a single common sequential program, built up in the normal way from basic blocks, *if* statements and *while* statements. Gotos and *break* and *continue* statements are not supported by the ASM; *switch* statements and ternary operators can be translated into *if-elses*, and *for* and *do* statements are handled in the same way as *whiles*.

In order to run the cost model in the trace-driven mode, the program is instrumented to record the outcome each time a *control statement* is executed. A control statement is an *if* or *while* statement in the sequential program made visible to the cost model by the ACOTES compiler. An *if* or *while* should be visible if its dynamic behaviour has a significant effect on the simulated program; i.e. if it controls either any communication or a large amount of computation. The resulting list of outcomes for the control statement is known as its *control variable*, each element of which takes the values 1 (true) or 0 (false) for an *if* statement, or the non-negative iteration count for a *while* statement. When the simulator is used in the trace-driven mode, the program model is driven by the set of control variables obtained from a real execution.

The set of control variables can be reused with a different allocation of the same tasks onto different processors. It can also be reused if one or more tasks are fused together. However, it cannot usually be reused with a different blocking factor or if any compiler transformation modifies the structure of the program in certain ways described below.

3.3.4 Definition of a task

The sequencing of subtasks in a task is driven by its *subtask tree*, built up from subtasks, *If* nodes and *While* nodes, with each *If* or *While* node associated with a similar statement in the sequential program. In order to motivate the definition of a task, Figure 17(a) shows an example SPM program with an example partition into two tasks. Figure 17(b) shows how these tasks may be described using the above program model. In this example, stream *d* crosses levels of the program hierarchy, and has been compiled by moving the push operation immediately before the pop. Hence, the push on stream *d* is moved inside the innermost loop in task 1, and each value of the variable *d* may be pushed on the stream multiple times.

Figure 18(a) shows the same example after a compiler transformation that would require the generation of a new trace. In this case, task 2 pushes the value onto stream *d* exactly once only if the innermost loop is due to execute at least once—it is implicit in this program that *cond3* corresponds to the value of *cond4* on the first iteration of the loop. The new trace contains four control variables and the definition of the tasks is given in Figure 18(b).

The simulator reads the program description as Python source code, and uses the following grammar to define the subtask tree for a task. Condition variables and subtasks are identified by their numeric Ids. The condition variable for an *if* node may be inverted. In fact, if the *if* statement contains both an *if* part and an *else* part in the same task, then they are represented as two *If* nodes, sharing the same condition variable.

```

Tree      :: (Subtask | IfNode | WhileNode)*
IfNode    :: If (CondVar, Invert, Tree)
WhileNode :: While (CondVar, Tree)
CondVar   :: <Integer>
Subtask   :: <Integer>
Invert    :: True | False

```

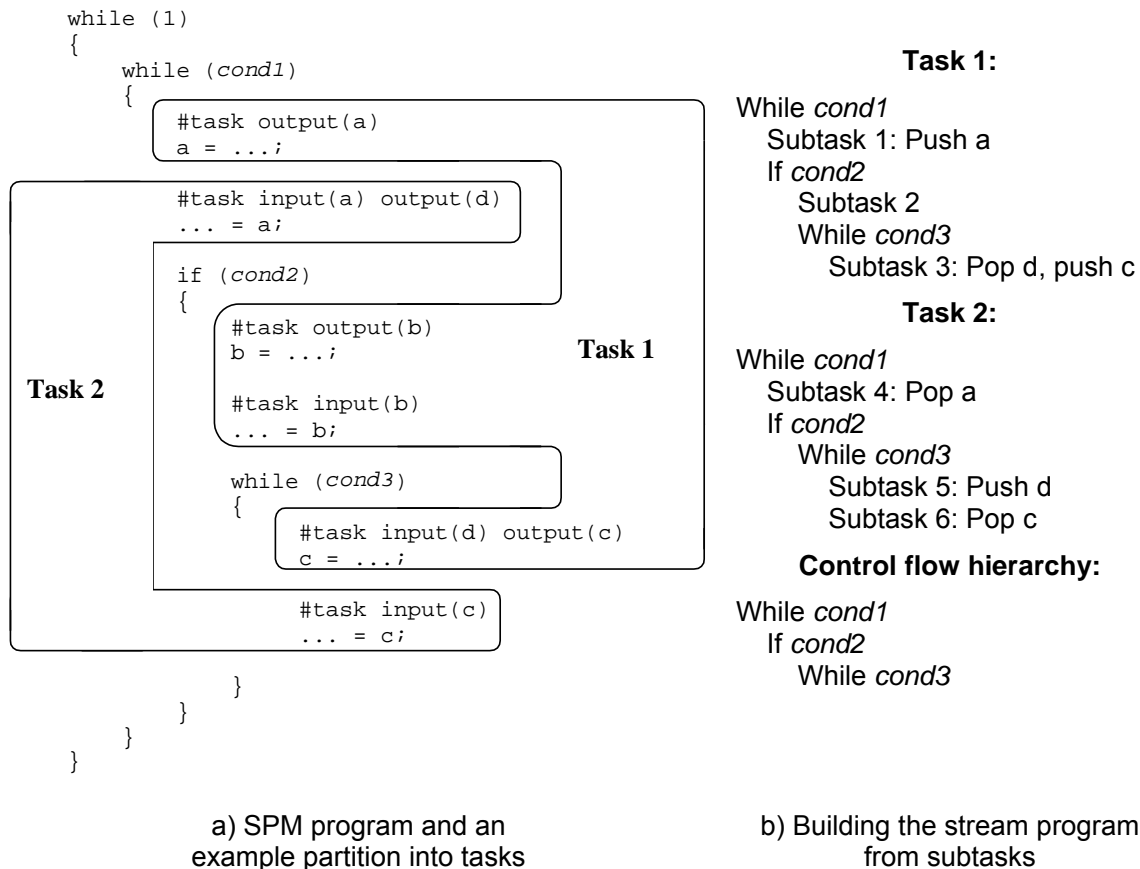


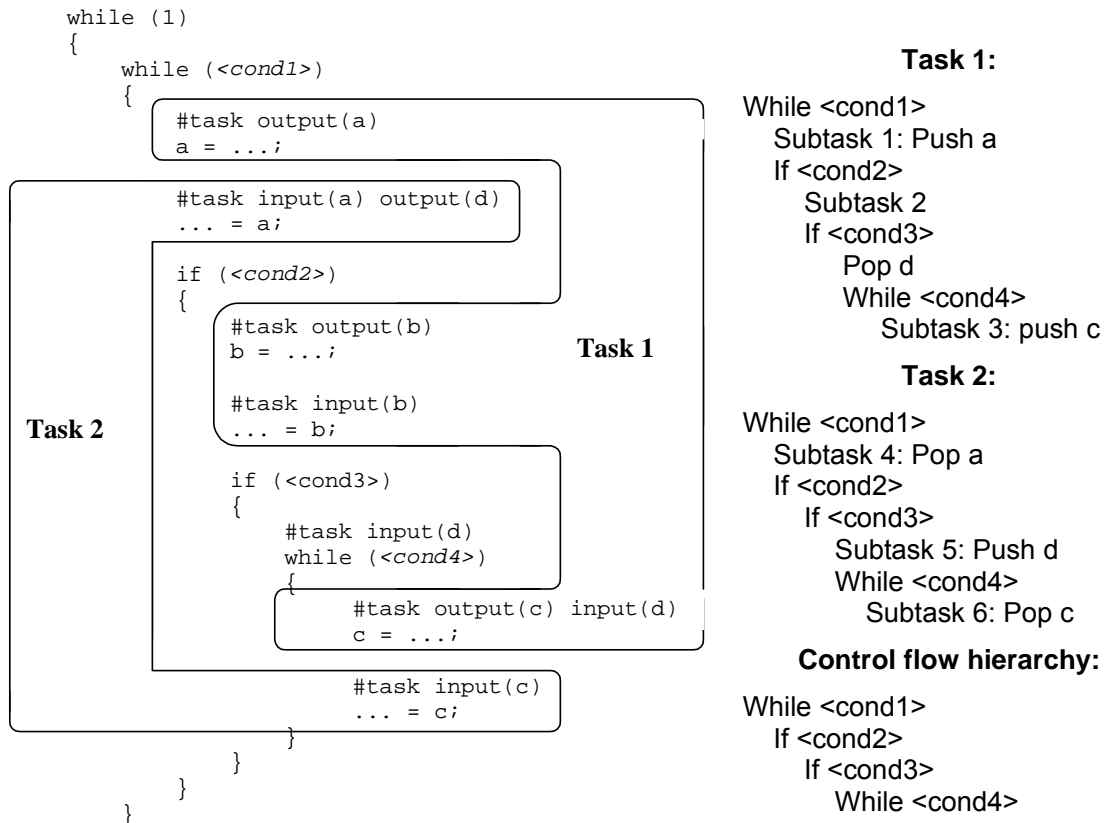
Figure 17: Example program with hierarchical structure

The simulator also requires the control flow hierarchy, shown in the above-referenced figures, which defines how the condition variables are arranged into a tree in the original program. Each condition variable appears exactly once in the control flow hierarchy. The control flow hierarchy has the same grammar as a subtask tree, except that it does not itself contain any subtasks and no condition variables are inverted.

The tree of subtasks in a task must be consistent with the control flow hierarchy. In particular, if node A is an immediate child of node B in any task, then the control variable for node A must be an immediate child of node B in the control flow hierarchy. It is not necessary for a subtask tree to contain the root of the tree; i.e. it may omit one or more outermost control variables. It

is also possible for multiple nodes based on the same condition variable, inverted or not, to appear at the same level of a subtask tree.

The simulator executes a task by traversing the subtask tree. Each control node in the subtask tree is given its own index in the relevant control variable list. When execution reaches a control node, the simulator reads the next element in the control variable list, and uses this value to control the direction of an *If* node or the number of iterations of a *While* node.



a) A transformation of the above SPM program with the same partition into tasks

b) Building the stream program from subtasks

Figure 18: The example program of Figure 17 after a compiler transformation

3.3.5 Statistical model

If the program satisfies the conditions in this section, the simulator can generate the control variables using a statistical model. In this case, each control variable takes the sequence of values of a random variable, with a distribution either taken from a profile or estimated statically. Currently, the simulator supports Bernoulli random variables for *if* statements, but does not contain a statistical model for *while* loops.

In order to avoid deadlock, the statistical model requires that whatever values the condition variables take, on each stream the sequence of pushes in the producer task is the same as the sequence of pops in the consumer task. Hence, the control variables can be generated from independent statistical models, and the program cannot deadlock. This means that the

tree of push counts for the producer task is isomorphic to the tree of pop counts for the consumer task. Concretely, the tree of push (resp. pop) counts is generated from the subtask tree by replacing a subtask that pushes (resp. pops) on the stream with the integer count giving the number of elements, ignoring any other subtasks. Additionally, any outermost control nodes containing exactly one node should be removed; otherwise one tree may strictly contain the other, rather than being isomorphic to it.

This constraint is automatically satisfied by SPM programs, unless the programmer is directly making use of push and pop primitives. Figure 19(a) shows a program excluded by the above constraint. The program would deadlock unless *cond1* takes the values 0, 1, 0, 1, ... a constraint that the statistical model cannot enforce. The second program shown in Figure 19(b) is supported by the statistical model, since it does not deadlock for any sequence of values for *cond1*, and the tree of push counts on stream a is the same as the tree of pop counts.

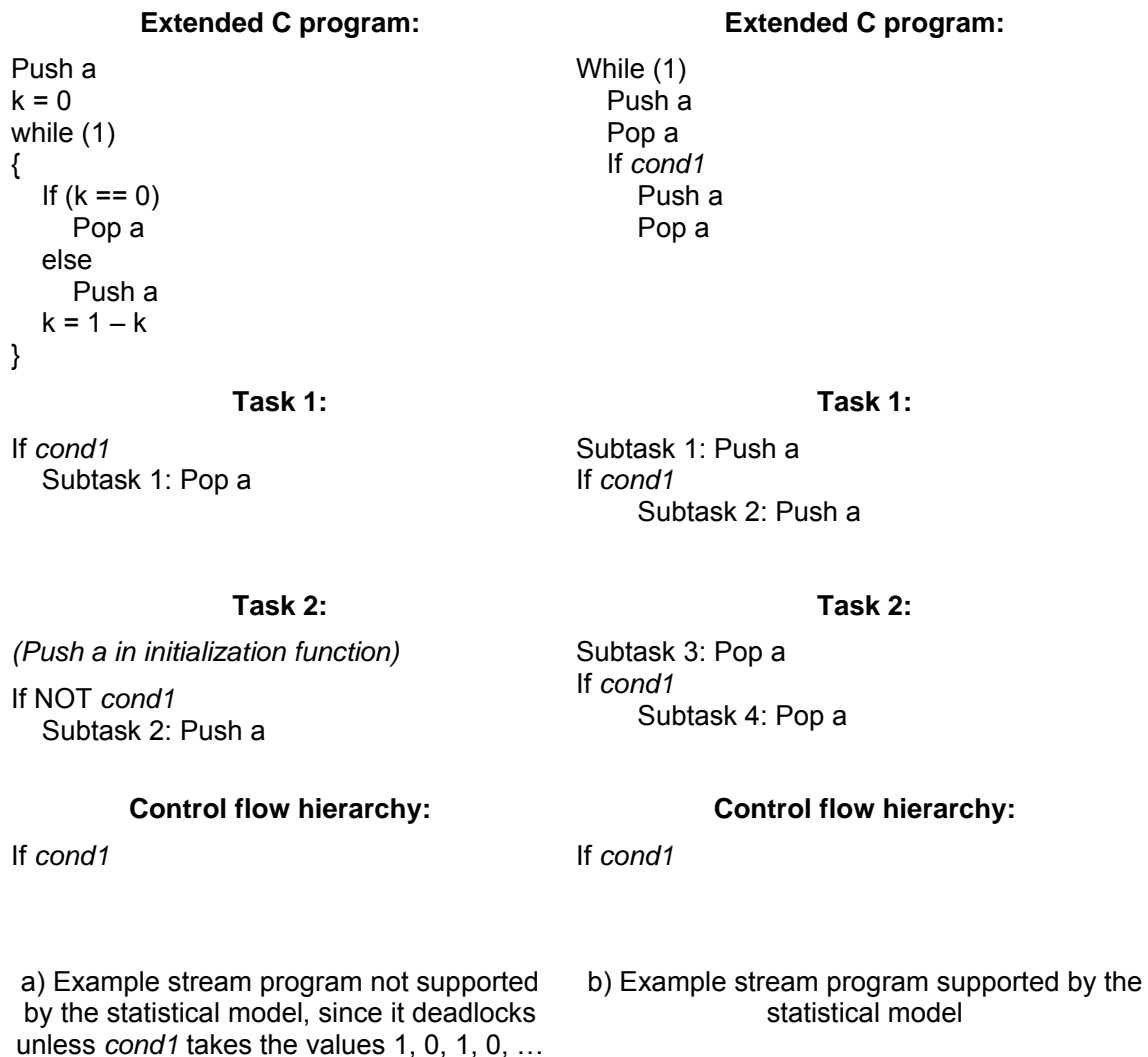


Figure 19: Example programs for the statistical model

| Parameter | Type | Description |
|---------------------|--------|--|
| name | String | Unique name in program namespace |
| elemSize | Int | Size of each element on the stream, in bytes |
| producerQueueLength | Int | Number of elements in the producer's buffer |
| consumerQueueLength | Int | Number of elements in the consumer's buffer |
| preQueue | Int | Number of elements to prequeue before execution begins |

(a) Definition of a stream

| Parameter | Type | Description |
|-----------|--------|--|
| name | String | Unique name in program namespace |
| cpuTime | Int | Fixed execution time, in cycles, on the target processor |
| inputs | List | List of inputs, each specifying the name of the stream and the number of elements to pop |
| outputs | List | List of outputs, each specifying the name of the stream and the number of elements to push |

(b) Definition of a subtask

| Parameter | Type | Description |
|--------------|--------|--|
| name | String | Unique name in program namespace |
| subtaskTree | List | Subtask tree |
| codeSize | Int | Code size, in bytes |
| codeLocation | String | Name of the memory containing the code |

(c) Definition of a task

Figure 20: Parameters of the Program Description

3.4 The cost model simulator

We have implemented a cost model simulator, written in the Python language. It takes its inputs as Python source code, and runs for a given number of time units or iterations. It generates a trace file compatible with Paraver [Par08] or displays statistics such as the number of cycles per iteration. This simulator is available from the ACOTES project. For more information refer to the README file in the package.

3.4.1 Trace-driven simulation

As described above, the program model in the simulator is driven by control variables related to the semantics of the sequential SPM program, and these control variables are obtained either from a real execution, or from a statistical model. In the former case, the simulator still needs to be given the program description, because the program description defines the computational cost of each unit of work.

The input trace is given in the Paraver format, with event records providing the values of the control variables. In particular, each task generates an event record each time it encounters a control statement; the event type is the control variable number, and the event value is the direction of the *if* statement or the number of iterations of a *while* statement, bearing in mind that the number of iterations is in general not known until the loop terminates.

3.5 Integration with the compiler

The ACOTES compiler will be an iterative compiler, in which a heuristic-based search algorithm controls a compiler built using Mercurium and GCC. Mercurium is a source-to-source converter, which translates from SPM to threads and acolib, and applies task fusion and allocation. The output from Mercurium is compiled using GCC, which applies blocking, autovectorization and loop interchange, alongside traditional compiler optimizations.

The parameters determined by the search algorithm are known as the mapping; i.e. the partitioning, allocation, blocking factors, and queue lengths. The mapping is provided to both Mercurium, possibly through additional pragmas in the source program, and to GCC, through a plugin using the Interactive Compilation Interface (ICI) [Fur07]. Decisions made internally to either compiler such as those controlling autovectorization and loop interchange are not included in the mapping. The structure of the ACOTES compiler is shown in Figure 21.

The search algorithm finds the optimal mapping by evaluating the performance of a sequence of candidate mappings, using the resource utilization and performance metrics of each candidate to choose the next candidate. Each candidate is compiled using Mercurium and GCC and executed on the real platform or a simulator to generate a trace and determine the average computation cost for each subtask. The resource utilization for buses is determined by running the trace through the ASM simulator, using the trace and the average computation costs. Within the ACOTES compiler, all changes in the mapping, with the exception of changing the allocation require recompilation. This is because it is difficult to predict how fusion and blocking will change the computational costs.

The compiler optimisations affected by the mapping are task fusion, allocation, and blocking. Task fusion merges several tasks in the SPM program into a single thread, and requires from the search algorithm the list of threads, each being a list of tasks to merge. Allocation maps threads to processors, and requires from the search algorithm this mapping function. Blocking unrolls loops and aggregates communication, and requires from the search algorithm the blocking factor at each end of each stream. As mentioned earlier, task fusion and allocation will be performed by Mercurium, and blocking will be performed by GCC.

The search algorithm requires the list of tasks from the SPM program, as this determines the search space for the partitioning search. In order for the ASM to be able to model the program, it also requires the program description and a trace, as defined in the previous section.

The interaction between the search algorithm and Mercurium will be defined at UPC. As mentioned above, the interaction between the search algorithm and GCC will use ICI. The ICI plugin will read a text file to control blocking, containing one line per stream, each with four space-separated fields: 1) the taskgroup ID assigned by Mercurium, 2) the stream ID assigned by Mercurium, 3) the producer blocking factor and, 4) the consumer blocking factor.

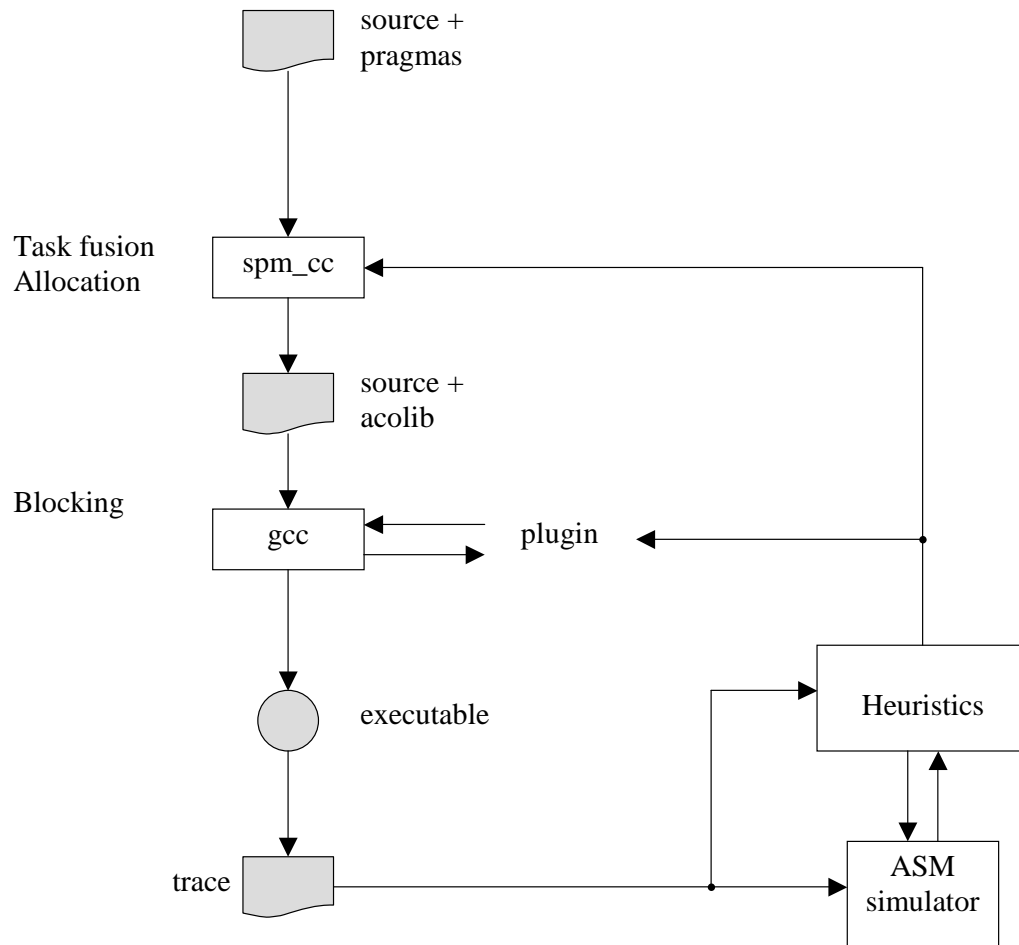


Figure 21: Proposed structure of the ACOTES compiler

4 Conclusions

In this deliverable, we have presented the final versions of the definition of the Streaming Programming Model (SPM) and the Abstract Streaming Machine (ASM) developed in the context of the Acotes Project.

The Streaming Programming Model has been completed. A second prototype has been implemented and released, consisting of a compilation pass of the Mercurium compiler and the ACOLib library. This document presents the details of the pragmas recognized by the compiler, intrinsic functions and internals of ACOLib. During the last year of the project, the ACOLib functionalities will be migrated to the Multicore Streaming Framework (MSF) being developed, in such a way that the programming model can benefit of the porting of MSF to the Cell BE processor and possibly other architectures.

The Abstract Streaming Machine has also been completed. It provides the way architectural cost models are defined, and a cost model simulator that has been seen accurate enough to model the communications in the Cell BE processor. During the last year of the project we will integrate the analysis done with the cost model simulation into the compiler to improve the quality of the code generated for the target architecture.

References

- [ChT05] Chen, T., Raghavan, R., Dale, J., Iwata, E. Cell Broadband Engine Architecture and its first implementation. IBM Developerworks, 2005.
- [Fur07] G. Fursin, A. Cohen. Building a Practical Iterative Compiler. 1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), 2007.
- [Gir00] S. Girona, J. Labarta, and R. Badia. Validation of Dimemas communication model for MPI collective operations, Proc. EuroPVM/MPI, 2000.
- [Lee87] Lee, E., Messerschmitt, D. Synchronous data flow. Proceedings of the IEEE, 75 (9), pp. 1235-1245, 1987.
- [OMP3.0] OpenMP Architectural Review Board, OpenMP Specification, <http://www.openmp.org/>
- [Par08] CEPBA. <http://www.cepba.upc.edu/paraver/>. Paraver Parallel Program Visualisation and Analysis Tool Tracefile Description.
- [Rod07] Paul Carpenter, David Rodenas, Xavier Martorell, Alex Ramirez and Eduard Ayguade, "A Streaming Machine Description and Programming Model, Lecture Notes in Computer Science, Embedded Computer Systems: Architectures, Modeling and Simulation, Springer Berlin/Heidelberg, 0302-9743, pp 107-116, Aug 30, 2007.
- [Rod08] David Rodenas, AcotesCC, second prototype, Feb. 2008, <https://wiki.ehv.campus.philips.com/twiki/bin/view/Acotes/WP2>