

# FP6 Project IST-034869

## Deliverable D3.2

Code generation for streaming applications  
based on an abstract machine description

IST-034869

Public

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

|                                  |   |                                                                                     |
|----------------------------------|---|-------------------------------------------------------------------------------------|
| <b>Deliverable Number</b>        | : | D3.2                                                                                |
| <b>Title of Deliverable</b>      | : | Code generation fro streaming applications based on an abstract machine description |
| <b>Nature of Deliverable</b>     | : | Public                                                                              |
| <b>Internal document version</b> | : | 0.1                                                                                 |
| <b>Contractual Delivery Date</b> | : | 31.05.2007                                                                          |
| <b>Actual Delivery Date</b>      | : | 06.09.2007                                                                          |
| <b>WP(s)</b>                     | : | WP 3                                                                                |
| <b>Author(s)/Affiliation</b>     | : | Paul Carpenter , David Rodenas, Xavier Martorell, Alex Ramirez, Eduard Ayguade      |

# Table of Contents

|                                                      |           |
|------------------------------------------------------|-----------|
| <b>Table of Contents.....</b>                        | <b>2</b>  |
| <b>0 Abstract.....</b>                               | <b>3</b>  |
| <b>1 Introduction.....</b>                           | <b>4</b>  |
| <b>2 Stream Programming Model .....</b>              | <b>5</b>  |
| 2.1 Directives.....                                  | 5         |
| 2.2 Unoptimized transformation algorithm .....       | 7         |
| 2.3 Graph Optimization.....                          | 9         |
| <b>3 Abstract Streaming Machine .....</b>            | <b>11</b> |
| 3.1 Program description .....                        | 11        |
| 3.2 Machine description .....                        | 12        |
| 3.3 System description .....                         | 15        |
| <b>4 Experiments.....</b>                            | <b>18</b> |
| 4.1 Runtime system .....                             | 18        |
| 4.2 Results .....                                    | 18        |
| <b>5 Limitations and potential improvements.....</b> | <b>22</b> |
| <b>6 Related work.....</b>                           | <b>23</b> |
| <b>7 Conclusions .....</b>                           | <b>24</b> |
| <b>8 Acknowledgements.....</b>                       | <b>25</b> |
| <b>9 References .....</b>                            | <b>26</b> |

## 0 Abstract

In this paper we present the initial development of a streaming environment based on a programming model and machine description. The stream programming model consists of an extension to the C language and its translation towards a streaming machine. The extensions will be a set of OpenMP-like directives. The machine description will be used to implement a simulator to decide the partitioning and scheduling of streaming tasks at compile time. We present an algorithm to convert a serial application into a real streaming parallel application using the proposed annotations. We also show how the simulator can be parametrized for different architectures to predict their performance, and how it allows to decide the best task partitioning and scheduling for each architecture. All results and studies are visualized using Paraver traces, which allow us to see what is happening at each timestamp with a minimum overhead.

# 1 Introduction

Multicore processors are becoming increasingly more common, as traditional monolithic CPUs fail to scale effectively to take advantage of Moore's law. As wire delays become more significant to performance, it is becoming increasingly expensive to implement shared memory in hardware, and many modern systems have distributed memory (Cell [ChT05], Merrimac [Dal03], GPUs [ChJ05,Gal05,Har03] and others). At the same time, intrinsically parallel applications such as digital video, software radio, multimedia and 3D graphics are becoming more important [Gor05,Blo04]. Compilers should face the new reality by offering developers the ability to write portable efficient code for parallel applications on parallel platforms. Our long term objective is to provide a set of powerful tools to bridge the gap between hardware and software; and in this paper we describe our current work targeting streaming machines. We consider a streaming machine as multiple independent processors communicating and synchronizing primarily via one-dimensional streams of data [Lee87].

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. We do not expect the programmer to learn a whole new language before any benefit can be seen, nor do we assume that the compiler can automatically extract tasks and streams from the original code without modifications. Our Stream Programming Model (SPM) 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 tasks within it, also specifying the tasks' inputs and outputs. We also present an algorithm that translates the annotated serial source code into a streaming program. Furthermore, we have developed a streaming run-time library, *acolib*, which allows us to run the resulting streaming applications, currently on shared memory, and experiment with program transformations to be performed by the compiler. In our initial experiments, we have applied the transformation manually to small benchmarks, and have obtained traces showing the internals of the execution.

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 have implemented a cost-model simulator, based on 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. The inputs to the cost-model simulator are the machine description, application description and the proposed mapping of the program onto the hardware. The simulator generates output which can be visualized using the Paraver tool [Par], dumped to standard output or processed internally using one of the available filters; e.g. to determine the steady state cost. It is also possible to describe the program using a Paraver trace; in which case, the simulator generates a new trace using the cost-model parameters of the system in a similar manner to the Dimemas tool [Dim].

The rest of the paper is structured as follows: Section 2 presents the Stream Programming Model, Section 3 presents the Abstract Streaming Machine, Section 4 describes our initial experiments, Section 5 describes the limitations of the work so far, Section 6 compares our approach to related work and Section 7 concludes the paper.

## 2 Stream Programming Model

The 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 of 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 during execution. The consumer task blocks when it pops from an empty input stream and the producer blocks when it pushes to a full output stream.

### 2.1 Directives

We have taken OpenMP [OMP] as a simple, well-designed and widely understood set of directives. The OpenMP directives 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 OpenMP and learn more about it only as further knowledge is required. As motivated in the introduction, we see this property as desirable for the SPM. Of course, there are many differences between the OpenMP and streaming models of parallelism; for example, OpenMP is designed for shared memory homogeneous machines, whereas the streaming directives target distributed memory and heterogeneous systems; for example the Cell Processor [ChT05].

Directives control how the application is converted into a stream program, by defining the tasks and the streams between them. All directives apply to the single statement following 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 and is named after the similar directive in the proposal for OpenMP 3.0 [Ayg07]. 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. The second directive is `#pragma task`, which must be inside the `#pragma taskgroup` to define a task; any statements not enclosed by a `#pragma task` belong to the control task. It is possible to nest task definitions. The final directives are `#pragma input (v1,v2,...,vn)` and `#pragma output (w1,w2,...,wm)`, which define the inputs and outputs of a task. `#pragma input(v1,v2,...,vn)` creates  $n$  streams that send the values of  $v_1, v_2, \dots, v_n$  from the outer task to the defining task. `#pragma output(w1,w2,...,wm)` is the opposite, meaning that it creates  $m$  streams that send the values of  $w_1, w_2, \dots, w_m$  produced by the defining task back to the outer task. The body of an inner task will be executed repeatedly in an implicit while loop until the end of stream on any input. When the body is executed it consumes exactly one element from each input stream, and produces exactly one element on each output stream. If a task has no inputs, then it is given a virtual control stream as an implicit input, which carries one data element each time that the task

```

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

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

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

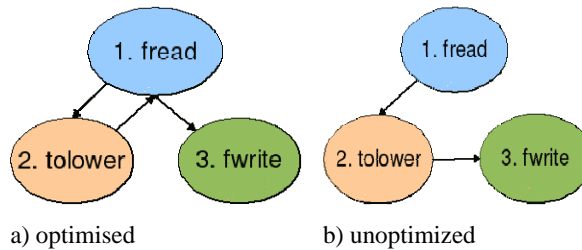
    fclose(in);
    fclose(out);
    return 0;
}

```

□ **Figure 1.** Annotated C code for tolower sample

should be executed.

Figure 1 defines a taskgroup region that contains the whole loop, and two explicit task definitions, each containing a single line of code. In total, there are therefore three tasks, the first is the #pragma 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*), is for the value of the variable *c*, produced at *fread* and sent towards *tolower*. The second stream, *fread* output(*x*), is for value of variable *x* produced by *tolower* and sent to *fread*. The third stream, *fwrite* input(*x*), is for 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 directive or an output; see Figure 2 a). Also note that we expect the compiler to connect *tolower* output(*x*) directly to *fwrite* input(*x*), but consider it to be an optimization. This optimization is possible because *x* is not consumed by *fread* in Figure 2 b).



**Figure 2.** Tolower graph

## 2.2 Unoptimized transformation algorithm

The transformation algorithm will process all taskgroups, their tasks and inputs and outputs. The algorithm is presented in Figure 3. For each task in each taskgroup (lines 01-03), the algorithm collects the information needed to build the interface of the task (lines 05-12). Such interface consists of the initial value of the variables used by the task, its input and output streams, and the streams created by tasks immediately defined inside the current one. The interface is used to initially invoke the task. Next, the algorithm constructs the body of the task, using the interface previously collected and the original code enclosed inside the task (lines 13-25). For each input stream, a pop operation is generated to get a value from it, to be consumed by the task (lines 15, 19). In the same way, for each output stream, a push operation is generated to send a value from the task to the stream (line 18). Also, a while control loop is generated (line 16). This loop controls the iterations of the task until any of its input streams is closed. The task body (line 17) is generated inside this loop. Upon completion of the loop, the output streams are closed (lines 21-23). Lines 26-30 are used to replace the original task directive and body with statements to push/pop data to and from the input/output streams. Lines 31 and 32 simply declare the variables representing the interface of the task, and add the outlined function representing the task, into the internal compiler representation.

In addition, the algorithm generates the initialization and finalization code for each taskgroup. First, it uses the information about the task interfaces to declare and initialize the streams created by the inner tasks (lines 35-40). Lines 41 and 42 connect the ends of the streams together. Lines 43 and 44 set the values of the parameters of the tasks, and start them. Line 45 generates the body of the taskgroup based on the original code enclosed in the taskgroup directive. Lines 46 and 47 then close the output streams used by the immediately inner tasks. Line 49 then makes the taskgroup wait until all inner tasks have finished. Finally, line 51 replaces the original taskgroup directive and body with the code resulting from this algorithm. This is not the most efficient algorithm, but is a first step. It can be considered as a reference algorithm that helps to explain the meaning of the directives.

```

00 source= Parse([input.c]);
01 For(taskgroup: source.taskgroups)
02 {
03   for(task: taskgroup.tasks)
04   {
05     task.states= Flatten(
06       task.vars,
07       task.inputs.istream,
08       task.outputs ostream,
09       { i task from task.tasks and NestedLevel(task,i task) == 1 |
10         i task.inputs ostream, i task.outputs.istream
11       }
12     );
13     task.outline.body= Flatten(
14       { state from task.states|Declare(state), Assign(state,task.outline.argument[state]) },
15       { stream from task.inputs|stream.var= Pop(stream.istream) },
16       While ( And({ stream from task.inputs | !Eos(stream.istream) })),
17         Flatten( task.body,
18           { stream from task.outputs | Push(stream ostream, stream.var) },
19           { stream from task.inputs | stream.var= Pop(stream.istream) }
20         ) ),
21       { stream from task.outputs | Close(stream ostream) },
22       { i task from task.tasks and NestedLevel(task,i task) == 1 |
23         { stream from i task.inputs | Close(stream ostream) }
24       }
25     );
26     task.stub= Flatten(
27       { stream from task.inputs | Push(stream ostream, stream.var) },
28       { stream from task.outputs | stream.var= Pop(stream.istream) }
29     );
30     taskgroup.source.Replace(task,task.stub);
31     source.Add(Declare(task.states));
32     source.Add(task.outline);
33   }
34   taskgroup.stub= Flatten(
35     { task from taskgroup.tasks and NestedLevel(taskgroup,task) == 1 |
36       { stream from task.inputs | Declare(stream ostream) },
37       { stream from task.outputs | Declare(stream.istream) }
38     },
39     { stream from taskgroup.tasks.inputs | Create(stream ostream), Create(stream.istream) },
40     { stream from taskgroup.tasks.outputs | Create(stream ostream), Create(stream.istream) },
41     { stream from taskgroup.tasks.inputs | Connect(stream ostream, stream.istream) },
42     { stream from taskgroup.tasks.outputs | Connect(stream ostream, stream.istream) },
43     { state from taskgroup.tasks.states | state= [state] },
44     { task from taskgroup.tasks | Start(task.outline) },
45     taskgroup.body,
46     { task from taskgroup.tasks and NestedLevel(taskgroup,task) == 1 |
47       { stream from task.inputs | Close(stream ostream) }
48     },
49     { task from taskgroup.tasks | Join(task.outline) }
50   );
51   source.Replace(taskgroup, taskgroup.outline);
52 }

```

Figure 3. Transformation algorithm

## 2.3 Graph Optimization

In the above algorithm all output streams are routed back to the outer task, even if the values are not required by the outer task and will later be forwarded to the real destination. This means that the outer task will always wait for a response from an inner task that has one or more outputs, eliminating potential parallelism. We prefer to send the data directly to the sibling task when it is not being consumed by the outer task. In this case, the outer task will continue running and the serial semantics will still be valid. We have two potential approaches: the first uses additional `#pragma` directives to optimize flow control, and the second uses dependence analysis in the compiler.

Both methods make visible both ends of the streams. In the first approach, the programmer uses standalone `#pragma` output and `#pragma` input directives to indicate where values are produced and consumed. `#pragma output(v)` should connect to the following `#pragma task input(v)` and `#pragma task output(v,...)` should connect the following `#pragma input(v)`. This approach is inconvenient because it increases the number of directives needed. These directives allow to have full control on source transformation. An example is seen in Figure 4.

```
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 taskgroup /* fread */
    while (!feof(in))
    {
        #pragma output(c)

        #pragma task input(c) output(x) /*tolower*/
        if ('A' <= c && c <= 'Z') x = c - 'A' + 'a';
        else x = c;

        #pragma task input(x) /* fwrite */
    }
}
```

The second approach is to use automatic dependence analysis in the compiler which can take place either when building the graph of stream tasks, or in the transformation phase on the intermediate representation. The former is at the original source level. The compiler will obtain the positions where data is consumed ( $\dots=v \rightarrow \#pragma\ input(v)$ ) and where data is produced ( $v=\dots \rightarrow \#pragma\ output(v)$ ) and automatically add the necessary directives. The latter is at transformed intermediate representation level. Pushes have to be performed as soon as possible, and Pops as late as possible. When a Pop for one variable is immediately followed by one or more Pushes for the same variable new streams will be created directly between the source of the Pop and the destination of the Push. Old Pushes and its streams will be removed. If the popped variable is overwritten or not consumed, its Pop and stream will be also removed.

In the optimized program, outer tasks should not always have to wait for the results from nested tasks. We can apply blocking (task sends a block of elements instead of sending one

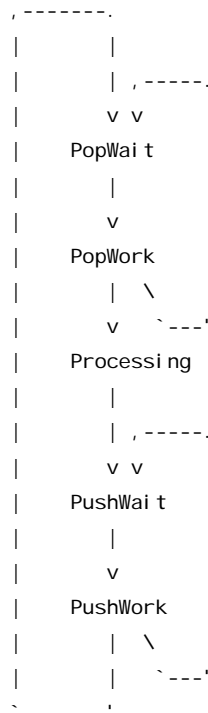
single element at a time) when there aren't such cycles. Blocking allows the task control loop to be unrolled.

## 3 Abstract Streaming Machine

The Abstract Streaming Machine description is defined in three parts: the program, machine and system. The program description defines the application, its tasks and its streams. The machine description defines the available processors, buses and communications properties. The system description is the glue that maps the program into the machine.

### 3.1 Program description

For the purposes of the cost-model simulator, a streaming program is a directed graph of tasks and streams. Each stream carries a sequence of homogeneous values between tasks. Each task has, as the elementary unit of work, a work function, which consists of three distinct stages: a pop stage, in which a fixed number of elements are popped from each input stream, 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 output stream—see Figure 5. Pops and pushes block when the stream becomes empty or full respectively. The work function is executed repeatedly in an implicit while loop, optionally preceded by a similar initialization function, allowing the behaviour at the start of the stream to be different from the steady state; for example a Finite Impulse Response (FIR) filter may fill it's input window at initialization without generating any output. This definition of a streaming program is thus similar to Synchronous Data Flow (SDF) [Lee87].



Task  **Figure 5.** Task as a finite state machine

Note the difference in perspective between the ASM and the SPM. In the SPM, a (source) task without any input streams is given an implicit input stream. In the ASM, the task simply has no input. Also, the transformation algorithm described above will not necessarily produce tasks that pop, compute and push in that order. The cost-model as it currently stands can be used to generate a conservative estimate, or *complex* tasks may be defined as a sequence of *simple* ones. In the future, we will extend the simulator to natively support *complex* tasks, and our method of implementation of the simulator makes this straightforward.

The program description specifies the graph topology, data rates, and processing costs; see the tolower example in Figure 6. The program description does not define the actual computation performed by any of the kernels, so it is not possible for the simulator to know the values of data elements on the streams. This has the advantage, however, that the simulator does not require knowledge of the input data set. The execution time of each kernel is currently constant, and estimated by the compiler, although it could come from a statistical model, with parameters estimated by the compiler. There is no concept of data arrivals from outside the streaming network.

```
# Define program
def setup_program():

    # Streams
    #           num  name  elemSize
    streams = [ Stream ( 1, 'c',   1 ),
               Stream ( 2, 'x',   1 ) ]

    # Tasks
    #           num  name      inputs                outputs                cpu_time
    #           :: [ (inEdge, num, cost)] :: [ (outEdge, num, cost)]
    tasks = [ Task ( 1, 'fread',  [],                    [ ('c', 1, 2)],        5 ),
             Task ( 2, 'tolower', [ ('c', 1, 2)],    [ ('x', 1, 2)],        5 ),
             Task ( 3, 'fwrite',  [ ('x', 1, 2)],    [],                    5 ) ]

    return Program ( tasks, streams )
```

Figure 6. Tollower program

It is possible that the producer and consumer tasks do not write and read the same number of elements per iteration. In this case, the size of the transmitted packets is controlled by the producer, and the consumer waits for a new packet when any of the data in it is required. The producer and consumer therefore execute at different rates. The stream constructor has an optional argument to pre-queue elements onto the stream, as required for feedback loops.

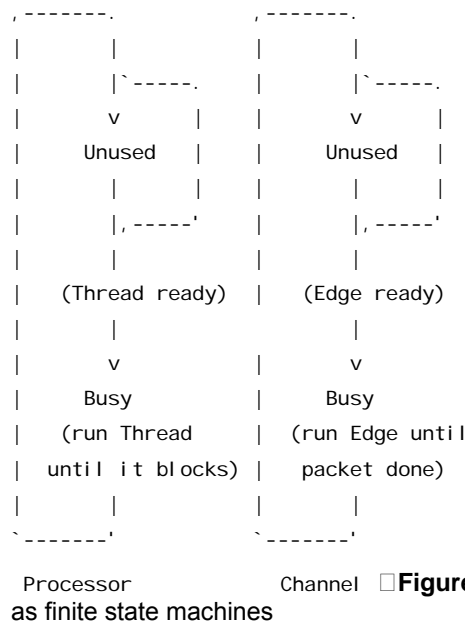
The program description defines the default length of queue for each stream, normally set to a value large enough to prevent deadlock. For example, setting the queue length to be smaller than the number of elements pushed by the producer at each iteration would cause it to immediately deadlock. The system mapping file can override these values as necessary to get good performance.

### 3.2 Machine description

The machine description defines the platform on which a streaming program may be executed, and is represented by a hypergraph of Processors connected via communication hardware

Links, each joining 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.

Figure 7 illustrates the behaviour of each processor and channel as a finite state machine. Communication links are not executed directly, instead corresponding to shared state and parameters for the channels. Threads and Edges are used to model the compiled Binary (see next section), but here it is useful to know that when a stream is statically mapped onto the available Links in the hardware, it is divided into Edges, each of which corresponds to one hop in the route.



As can be seen from Figure 7, edges cannot be interrupted once they begin transmission, fully occupying the channel until the packet is complete.

When a processor is connected to a link with more than one channel, we assume that it is not possible for the same processor to transmit onto more than one channel in the same link simultaneously; similarly for receive. The interface between the processor and link may be configured as either full-duplex or half-duplex, depending whether it is possible to transmit and receive on different channels at the same time.

The simulator supports both store-and-forward and virtual cut-through routing. Any Processor on more than one Link can potentially forward packets from one Link to another. If configured as store-and-forward, retransmission of the message begins once the entire message has been received and buffered. If configured as virtual cut-through, the behaviour depends on whether the outgoing link has higher bandwidth than the incoming one. If not, then retransmission begins after a fixed delay. If the outgoing bandwidth is higher, then the incoming message is divided into packets (of configurable maximum size) for retransmission. Each packet is transmitted, assuming an output channel is available, at the earliest point such that its last word arrives just before it has to be forwarded.

Figure 8 shows a simplified model of the Cell processor [ChT05]. Following Girona et al. [Gir00] we approximate the four rings of the EIB using a set of buses (in this example four buses). The interface of each processor to the EIB has bandwidth equal to a single channel of the EIB, so it can be represented using the model of the interface, as presented above (in full-

duplex mode). We are currently in the process of validating our approach and determining the optimal values of the parameters.

```
# Define platform
def setup_platform():

    # define processors
    processors = [ Processor ( 1, 'PPE' ),
                  Processor ( 2, 'SPE0' ), Processor ( 3, 'SPE1' ),
                  Processor ( 4, 'SPE2' ), Processor ( 5, 'SPE3' ),
                  Processor ( 6, 'SPE4' ), Processor ( 7, 'SPE5' ),
                  Processor ( 8, 'SPE6' ), Processor ( 9, 'SPE7' ) ]

    # Model EIB as four buses; although it's actually four rings

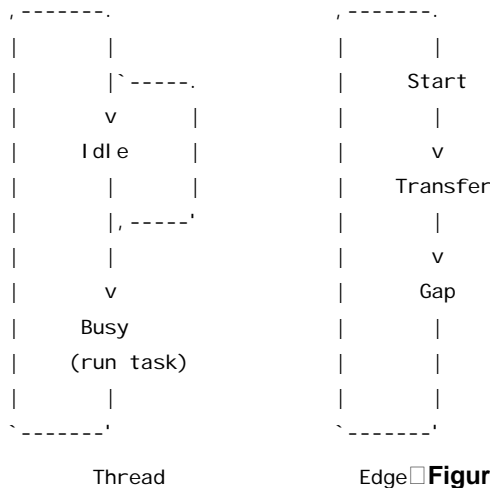
    # All processors are on the bus
    processorsOnBus = [ proc.name for proc in processors ]

    # Define bus
    #           num name   processors   start bandwidth gap
    links = [ Link ( 1, 'EIB', processorsOnBus, 159, 8, 2,
                    numChannels=4) ]

    return Platform ( processors, links )
```

**Figure 8.** Example platform

description



Edge **Figure 9.** Thread and Edge as finite state machines

### 3.3 System description

The system description references the program and machine descriptions, and maps the former onto the latter to provide the Binary. Together, the Program, Platform and Binary form the executable system to be simulated. The Binary is comprised of Threads, each of which executes a list of Tasks on a particular Processor, and Edges, each of which is part of a Stream statically routed onto a fixed Link.

The simulator will automatically generate a route for each stream, assuming one exists, using the minimum number of hops. It does not however attempt to balance the total communication load across the network. Local communication within a processor is not normally modelled by the simulator, so is effectively zero latency and infinite bandwidth. In all cases it is possible to override the routing decisions in the system description file.

Each message is transferred using a single (uni-directional) transfer on the bus, which carries both the data and the necessary control.

Figure 10 and Figure 11 show the mapping of the tolower program on a CPU plus accelerator. It is assumed that the accelerator does not have the ability to perform I/O, so the fread and fwrite tasks have been mapped to the CPU, in different threads to allow concurrency. Note that the system description only deals with issues of partitioning and scheduling. Compiler optimizations such as loop transformations or blocking may be done at the program level.

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

    # Threads
    #
    # num name proc tasks
    threads = [ Thread ( 1, 'A', 'CPU', [ 'fread1', 'fread2' ] ),
                Thread ( 2, 'B', 'Accelerator', [ 'tolower' ] ),
                Thread ( 3, 'C', 'CPU', [ 'fwrite' ] ) ]
```

return Binary( threads ) **Figure 10.** Unoptimized version of tolower on CPU plus accelerator □

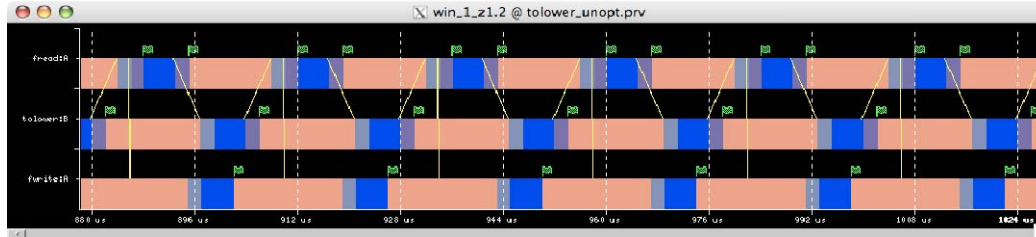
```
# Define binary
def setup_binary( program ):

    # Threads
    #
    # num name proc tasks
    threads = [ Thread ( 1, 'A', 'CPU', [ 'fread' ] ),
                Thread ( 2, 'B', 'Accelerator', [ 'tolower' ] ),
                Thread ( 3, 'C', 'CPU', [ 'fwrite' ] ) ]
```

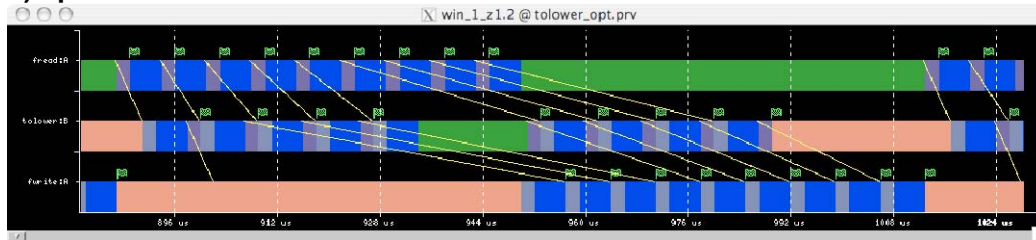
return Binary( threads ) **Figure 11.** Optimized version of tolower on CPU plus accelerator

Figure 12 shows Paraver traces for the simulation of the unoptimized and optimized versions of the tolower program, as defined by the programs in Figure 10 and Figure 11 respectively. Each row corresponds to a Thread, but it is possible, using the Visualizer module of Paraver, to plot Processors or Tasks instead from the same trace.

**a) unoptimized version**



**b) optimized version**



**Legend:**

- Processing
- Push Work
- Pop Work
- Push Wait
- Pop Wait

**Figure 12.** Paraver traces for cost-model simulation of tolower program

## 4 Experiments

We have applied the algorithm by hand to two examples: tolower, and wordhash. We have three translated versions of tolower: standard, optimized and blocking, and two translated versions for wordhash: optimized and blocking. The resultant graph for tolower standard is presented at Figure 2 a), the resultant optimized graph for tolower is presented at Figure 2 b). The optimized graph for word hash is at Figure 13.

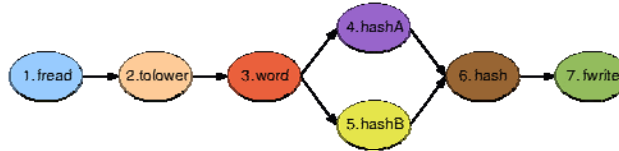


Figure 13. Wordhash optimized graph.

### 4.1 Runtime system

The runtime system consists of two libraries: pthreads and acolib. Application source code is translated to use explicit calls to pthreads and acolib. Pthreads are used to create parallel tasks and wait for them to complete. Acolib is our library to support streaming. This library is used to create the streams and send and receive data through them. Calls used are:

```

int pthread_create(pthread_t*pth, void**nul, void *(*task_outline)(void*), void**state);
int pthread_join(pthread_t pth, void**nul);
void ostream_create(ostream_t*, int element_size);
void istream_create(istream_t*, int element_size);
void ostream_connect(ostream_t*, istream_t*);
void ostream_push(ostream_t*, void**element);
void istream_pop(ostream_t*, void**element);
bool istream_eos(istream_t*);
void ostream_close(ostream_t*);
  
```

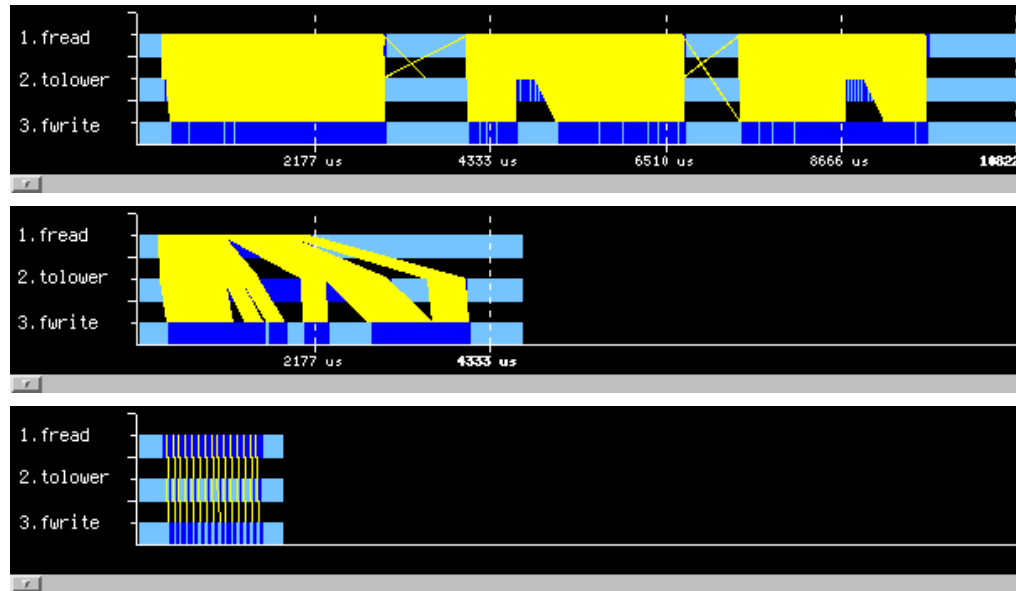
The current runtime implementation works on shared memory processors using efficient lock free structures. It uses an active wait whenever blocking is required. Acolib has been tested on Intel and PowerPC architectures without changes. It has two main data structures: ostream\_t and istream\_t. The ostream\_t structure contains next (next push element number), size (element size), first (next pop element number), and istream (a pointer to its istream). istream\_t structure contains next, size, first, ostream (a pointer to its ostream), and buffer (input element stream buffer, where elements are stored from push to pop). Push stores the next element at istream buffer (remote), and next updates its ostream next (local) and istream next (remote). Pop reads the element from its istream buffer (local), and next updates its istream first (local) and ostream first (remote). Contention are done using next and first, always will be satisfied  $0 \leq \text{next} - \text{first} \leq \text{BUFFER\_SIZE}$ .

### 4.2 Results

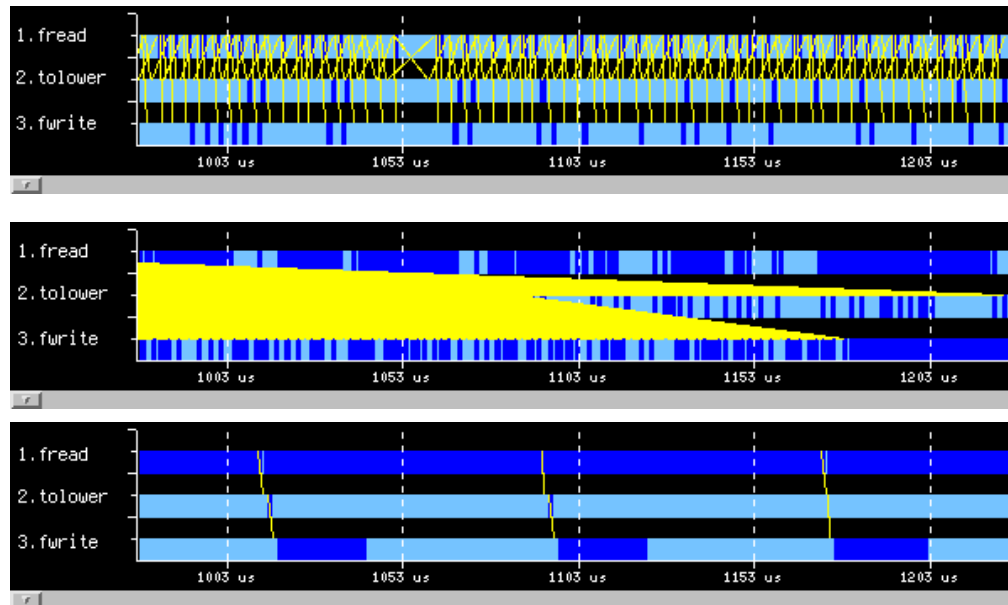
The translated source code uses the previously presented runtime, and has been tested on Intel, PowerPC and OpenPower platforms. Each task has its own dedicated processor. We present traces for the tolower example on a two-processor dual core PowerPC970 (total 4 cores), and the wordhash example on a two-processor dual core dual thread Power5 (total 8 threads).

Tolower resultant traces are presented in Figure 14 a) for unoptimized version, Figure 14 b) for optimized, and Figure 14 c) for 128 elements blocking. Each row is a different **task** (1. fread,

2. tolower, 3. fwrite). Cyan represents idle state, blue represents computing state, and yellow are the communications lines. These figures are in the same time scale. We can see that optimized is about 3 times faster than not optimized, and blocking is about 2 times faster. Figure 15 a), Figure 15 b) and Figure 15 c) are a zoom from previous Figure 14. Those figures have the same time scale. Unoptimized version shows communications in both directions, optimized communications shows lots of communications per work. Blocking version shows a good ratio between compute and communication, but it also shows an imbalance between tasks.



**Figure 14.** Tolower application Paraver traces. a) unoptimized, b) optimized, c) blocking



**Figure 15.** Zoom at tolower application Paraver traces. a) unoptimized, b) optimized, c) blocking

The wordhash example (not listed in this paper) has two main characteristics: it sends complex data structures through streams (an array of five elements), and it has one task inside a conditional statement. Wordhash has seven tasks in total. Resultant traces are in Figure 16 a) for optimized, and Figure 16 b) for 128 elements blocking. These figures are in the same timescale. Blocking version is about six times faster than optimized. Figure 17 a), Figure 17 b) are a zoom from previous Figure 16. Those figures are in the same timescale. We can see that we have the same problems that we had in tolower. Blocking version shows that tasks 4 through 7 are executed once every five iterations of the outer task. This is because their tasks are defined inside the conditional statement that only is satisfied every five iterations. Note that because the source transformation places the push to task 4 inside the conditional statement, the data flow rates are correct. It will satisfy the serial semantics.

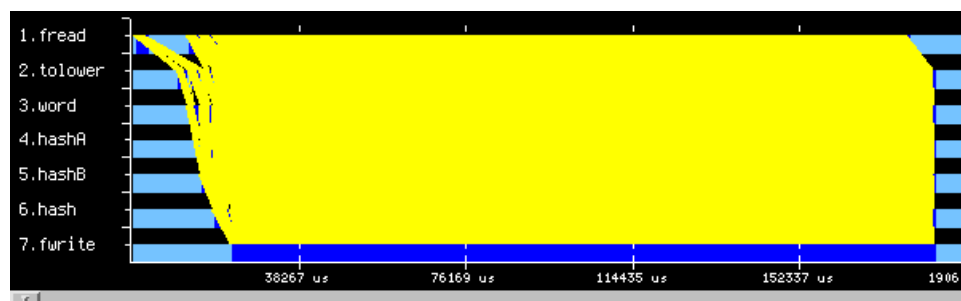




Figure 16. Wordhash application Paraver traces. a) optimized, b) blocking

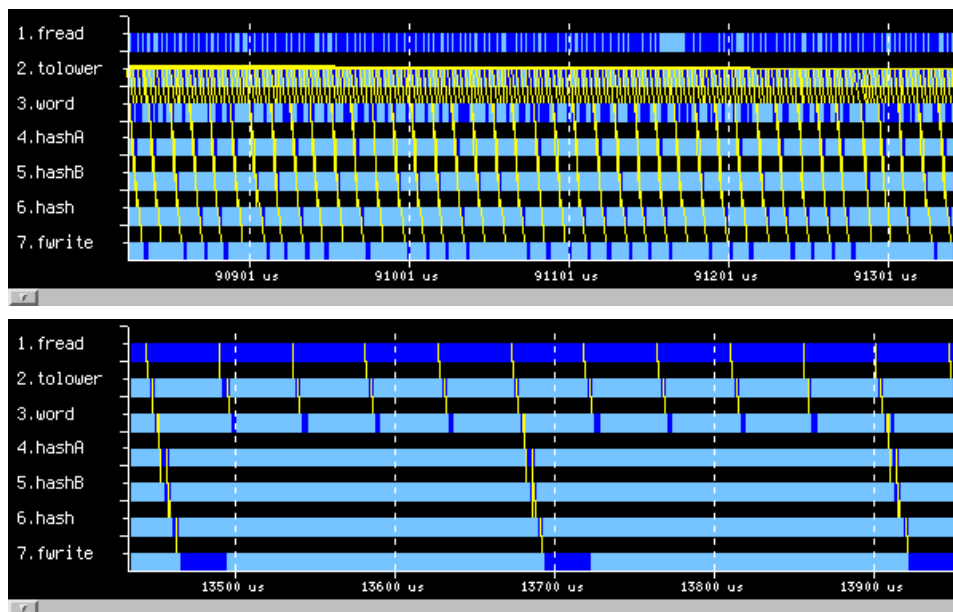


Figure 16. Wordhash application Paraver traces. a) optimized, b) blocking

## 5 Limitations and potential improvements

Our current SPM definition has many limitations and unspecified cases. We are assuming that all the code is in a single source file. If tasks are defined across multiple source files, the files need to be combined before we can merge tasks and optimize graphs. We cannot currently handle the case where a function contains a task directive without the enclosing taskgroup directive, but in the future we will. All defined tasks are created statically at compile time, so we will not support dynamic task creation and recursive functions, as supported by OpenMP. There are also some limitations to simplify the control flow: we do not allow return statements within the body of the directives or break, switch, goto or continue statements if they cross the boundary. We have to provide extensions for the SPM to be able to handle pure synchronization without data, sliding windows (as required for example by a Finite Impulse Response filter), and other cases as we discover them. Sliding windows are in fact possible but are not efficient: either the whole window has to be transferred each iteration as the element on the stream, which increases communication, or the window is managed internally by the consumer, which involves extra copy operations and increases memory use.

The applications in this paper have been translated manually, as proof of concept. We have begun work on a simple source-to-source translator that will implement the SPM by automatically translating the programs using a simple heuristic: a variable is considered to be consumed and produced by the outer task if it appears anywhere within its body. We believe that this approach is reasonable because the programmer can rename or duplicate the variable or isolate its uses by the outer task inside new tasks, but further investigation is needed. Despite its limitations, this simple source-to-source translator will enable us to evaluate a larger number and more complicated benchmarks than manual conversion. Later, we intend to develop a more sophisticated translator that works at the GIMPLE level.

We have tested the runtime library on shared memory machines, but it is designed for distributed memory, and only requires a few changes. We are currently working to port it to Cell, which requires two output source files for each input: one for Power PPU processor, and the other for the SPU accelerator.

The major limitation of the work on the ASM so far is that the cost-model simulator has not yet been validated against real hardware. Once we have ported the acolib run-time system to the Cell processor, we will be able to obtain realistic parameters to characterize the platform.

There are also some minor limitations in the cost-model simulator that we intend to remove or improve upon; for example, we will replace the push-execution- pop three-stage model of tasks with a more flexible model that allows tasks to perform a sequence of pushes, pops and computation in any order. We also intend to experiment with different thread scheduling algorithms, including pre-emptive multitasking.

Also, when the ASM becomes more mature, we will define a domain-specific language for the system description, bearing in mind the needs of our target compiler, which is GCC. Currently it is convenient to simply use python modules to define the system.

## 6 Related work

There are some other models similar to the SPM; for example StreamIt[Gor06], and GNU Radio[Blo04]. StreamIt is a whole infrastructure with its own language: a mix between C and Java oriented to filters, and a compiler able to deal with the filters and interconnections. There are some limitations: each filter has exactly one input and output stream, and streams can only carry values from a small set of primitive types. GNU Radio is a framework developed in C++ and Python. The graph of filters and connections is described using Python, and the filters are constructed as C++ classes. GNU Radio has its own scheduler and the system can be deployed on multiple architectures, even including FPGA. GNU Radio has more than 100 blocks already encoded. Both StreamIt and GNU Radio are designed for signal processing applications, and require the program to be written specifically in terms of streaming processes.

The proposal for OpenMP 3.0[Ayg07] supports task parallelism using the new taskgroup and task directives. The task directive specifies that the serial code within it should be executed by another thread inside the taskgroup's scope. The similarly named directives in the SPM intentionally have similar behaviour, although we have input and output streams. In OpenMP every time the task directive is reached a new task is created to execute its body. In the SPM, all the inner tasks are created once when the taskgroup directive is reached, and a value is sent on each input stream each time the task directive is reached. This is a form of synchronization that does not exist in the OpenMP 3.0 proposal. However, there are other proposals for OpenMP that add synchronization between threads, Gonzalez et al.[Gon01,Gon03] propose three new directives, which they call PRED, SUCC and NAME. The NAME directive labels a worksharing, and this label can be used by PRED and SUCC directives, which specify synchronization. Another approach using annotated C is Cell superscalar[Bel06], which uses a task directive to mark the inputs and outputs of a function. Each time the function is called, a new thread and dependencies between them are tracked using the information about the inputs and outputs.

## 7 Conclusions

As far as we know this is the first attempt to define a streaming language that allows an existing serial application to be converted into a streaming program using only pragma-style directives. This approach allows a non-expert programmer to modify the application step by step. We have proposed a small set of basic directives, which have easy to understand and predictable behavior; in future we may extend our set of directives as necessary.

We have presented some very simple benchmarks, which we have used to test the algorithm and experiment with the cost-model. We can see how simple applications can be converted automatically into stream programs with minimal programmer effort. We expect to extend our set of directives and the algorithm to handle new cases; for example tasks defined outside the lexical scope of the taskgroup. We have modelled these benchmarks using a simple cost-model simulator and visualized their internal behaviour using Paraver, and have seen behaviour similar to that seen in the real world. We expect to integrate the work into a real compiler that takes a serial application with minimal markup, automatically partitioning and scheduling the tasks for optimal performance on a given architecture.

## 8 Acknowledgements

We would like to acknowledge our partners in the Acotes project for the insightful discussions on the topics presented in this paper. This research is supported by the Spanish Government under contract CICYT TIN200407739C0201, and the IST program of the European Community under contract IST034869 (Acotes Project).

## 9 References

- [Dim] Dimemas: <http://www.cepba.upc.es/dimemas/>
- [Par] Paraver: <http://www.cepba.upc.es/paraver/>
- [OMP] OpenMP: OpenMP Application Program Interface. [www.openmp.org](http://www.openmp.org)
- [Lee87] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. Proceedings of the IEEE, September, 1987.
- [Gir00] Girona, S., Labarta, J., Badia, R.M.: Validation of Dimemas Communication Model for MPI Collective Operations. Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advance in Parallel Virtual Machine and Message Passing Interface (2000)
- [Gon01] M. Gonzalez, E. Ayguadé, X. Martorell and J. Labarta: Complex Pipelined Executions in OpenMP Parallel Applications. International Conference on Parallel Processing (ICPP'01). September 2001.
- [Dal03] William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, François Labonté, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, Ian Buck, "Merrimac: Supercomputing with Streams", SC2003, November 2003, Phoenix, Arizona.
- [Gon03] Marc Gonzalez, Eduard Ayguadé, Xavier Martorell and Jesús Labarta. Exploiting Pipelined Executions in OpenMP. 32nd Annual International Conference on Parallel Processing 2003, pg153-160. Kaohsiung, Taiwan.
- [Har03] Harris, M. J., III, W. V. B., Scheuermann, T., and Lastra, A. 2003: Simulation of cloud dynamics on graphics hardware. In Proceedings of the ACM SIG-GRAPH/EUROGRAPHICS conference on Graphics hardware, 92–101.
- [Blo04] Eric Blossom: GNU Radio: Tools for Exploring the RF Spectrum. Linux Journal, Issue 122, June 2004. <http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html>
- [ChJ05] Jiawen Chen, Michael I. Gordon, William Thies, Matthias Zwicker, Kari Pulli and Frdo Durand: A reconfigurable architecture for load-balanced rendering. HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. Pages 71-80. ISBN1-59593-086-8
- [ChT05] Thomas Chen, Ram Raghavan, Jason Dale, Eiji Iwata. Cell Broadband Engine Architecture and its first implementation. <http://www-128.ibm.com/developerworks/power/library/pa-cellperf/>
- [Gal05] Nico Galoppo, Naga K. Govindaraju, Michael Henson, Dinesh Manocha: LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In Proceedings of the ACM/IEEE SC'05 Conference. November 12-18, 2005.
- [Bel06] Pieter Bellens, Josep M. Perez, Rosa M. Badia and Jesus Labarta. CellSs: A Programming Model for the Cell BE Architecture . in proceedings of the ACM/IEEE SC 2006 Conference, , November 2006.
- [Gor06] Michael I. Gordon, William Thies, and Saman Amarasinghe: Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. Proceedings of the 2006 ASPLOS Conference Volume 34 , Issue 5 (December 2006)
- [Ayg07] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, Guansong Zhang: A proposal for task parallelism in OpenMP, Submitted to IWOMP2007