The Abstract Streaming Machine: Compile-time Performance Modelling of Stream Programs on Heterogeneous Multiprocessors *

Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade

Barcelona Supercomputing Center, C/Jordi Girona, 31, 08034 Barcelona, Spain
{paul.carpenter,alex.ramirez,eduard.ayguade}@bsc.es

Abstract. Stream programming is a promising step towards portable, efficient, correct use of parallelism. A stream program is built from kernels that communicate only through point-to-point streams. The stream compiler maps a portable stream program onto the target, automatically sizing communications buffers and applying optimizing transformations such as blocking, task fission and task fusion.
This paper presents the Abstract Streaming Machine (ASM), the machine description and performance model used by the ACOTES stream compiler. We explain how the parameters of the ASM and the ASM coarse-grain simulator are used by the partitioning and queue length assignment phases of the ACOTES compiler. Our experiments on the Cell Broadband Engine show that the predictions from the ASM have a maximum relative error of 15% across our benchmarks.

1 Introduction

As the industry moves towards multiprocessors, there is a growing need for software that exploits concurrency [2]. Many embedded systems provide multiple processors, often with multiple ISAs or microarchitectures to optimize for diverse applications and mitigate Amdahl's law [3, 4]; e.g. TI OMAP [5], ARM MPCore [6], Intel IXP2850 [7], Nexperia Home Platform [8], and ST Nomadik [9]. Many embedded systems [5, 10] and the Cell Broadband Engine (CBE) [11] provide distributed memory, which has lower power consumption and higher storage density than shared memory, but it is harder to program. Different types of applications contain different kinds of parallelism, which may need to be expressed in different ways [12].

One important class of applications comprises multimedia, graphics and signal processing. These applications can be represented as *stream programs*, in which independent blocks communicate and synchronize only via regular streams of data. Such a representation exposes task and data parallelism that would be hidden if the program were written in C or a similar sequential programming language. Fig. 1 is a graphical representation of our GNU radio FM demodulation program.

* This article is an extended version of a paper presented at SAMOS 2009 [1].

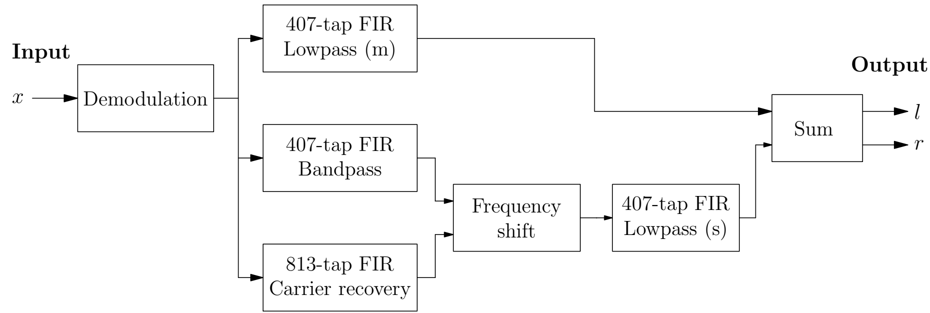


Fig. 1. Block diagram of an example stream program: GNU radio FM demodulation, showing seven kernels, connected by point-to-point streams

The key benefit of stream programming is that the stream compiler can perform high level transformations such as task fusion, fission and blocking. These transformations are necessary for good performance, and the right choice of transformations depends on the number and type of processors, the interconnect, and target topology.

The main contributions of this paper are:

- A novel simulation methodology for streaming applications,
- a flexible hardware description model,
- a flexible streaming application description model,
- and an explanation of how the ASM is used by two specific compiler optimizations in the ACOTES compiler.

2 The ACOTES Stream Compiler

This work is part of the ACOTES project[1] [13, 14], which has been developing a complete open-source stream compiler for embedded systems. The ACOTES compiler partitions a stream program to use task-level parallelism, aggregates communications through blocking, and statically allocates communications buffers. The stream program is written using the Stream Programming Model (SPM) [15, 16], an extension of C that uses pragma annotations to identify streaming tasks. If these pragmas are ignored, the result is a valid sequential program.

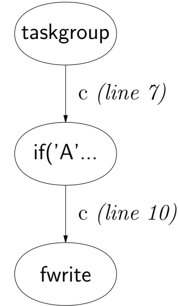
Fig. 2 shows an example program using the SPM. The streaming part of a program is known as a *taskgroup*, and it comprises the loop following the `acotes taskgroup` pragma. This taskgroup has two *subtasks*, each of which contains the statement or block following an `acotes task` pragma. The task's inputs and outputs are identified using the `input` and `output` clauses.

The SPM program begins running in a single thread. When execution reaches the taskgroup, the subtasks are created, and the program starts processing data

[1] Advanced Compiler Technologies for Embedded Streaming

```
1  int main()
2  {
3      char c;
4  #pragma acotes taskgroup
5      while (fread(&c, sizeof(c), 1, stdin))
6      {
7  #pragma acotes task input(c) output(c)
8          if ('A' <= c && c <= 'Z')
9              c = c - 'A' + 'a';
10
11 #pragma acotes task input(c)
12         fwrite(&c, sizeof(c), 1, stdout);
13     }
14     return 0;
15 }
```

(a) SPM source code for tolower



(b) Streaming graph (flattened)

Fig. 2. Example SPM program from [14]

in streams, passing inputs and outputs through the streams. More information can be found in the ACOTES documentation [15]. From the point of view of the mapping and queue sizing algorithms, the taskgroup and its subtasks have equal status, and are all known as *kernels*. This paper shows *flattened* stream graphs like Fig. 2(b), meaning that the hierarchical relationship between the taskgroup and its subtasks is ignored.

This paper describes the Abstract Streaming Machine (ASM), which represents the target system to the compiler. The ASM defines the target via coarse-grain simulation, since a closed-form analytical model is unlikely to capture the real behaviour of the program. This is because kernels can have variable execution times, and their relative firing rates can vary during the execution of the program—if they are inside conditional statements or loops.

Fig. 3 shows the compilation flow. The source program is converted from SPM to C, using the Mercurium [17] source-to-source converter. This step translates pragma annotations into calls to the *acolib* run-time system, and inserts calls to the trace collection functions. It fuses kernels that are mapped to the same processor and inserts statically sized communication buffers, as required, between kernels on different processors. The mapping is determined by the search algorithm [18, 19] (see Section 8). The resulting multi-threaded program is compiled using GCC, which has been extended within the ACOTES project to perform blocking, polyhedral transformations, and vectorization. Additional mapping information is provided to GCC using the Iterative Compilation Interface (ICI) [20]. The ACOTES compiler is iterative, meaning that the program may be compiled several times, as the search algorithm adjusts the mapping.

The *ASM simulator* executes a mapped stream program at a coarse granularity, and generates statistics which are used to improve the mapping. The inputs to the simulator are the *ASM machine description*, which describes the

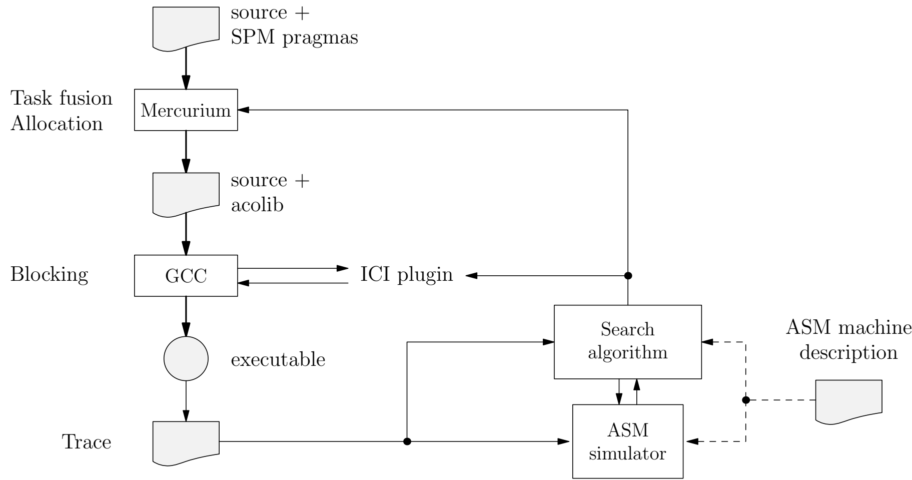


Fig. 3. The ACOTES iterative stream compiler

target, and the *ASM program model*, which describes the program. The ASM simulator is driven by a trace, which allows it to follow conditions and model varying computation times.

The ASM simulator is used inside the small feedback loop in the bottom right of Fig. 3; for example in determining the size of the stream buffers [18]. The trace format has been designed to allow a single trace to be reused for several different mappings. In Section 8 we describe in more detail the search algorithm and its interaction with the ASM.

3 Stream programming and *acolib*

There are several definitions of stream programming, going back to the 1960s, differing mostly in the handling of control flow and restrictions on the program graph topology [21]. All stream programming models, however, represent the program as a set of *kernels*, communicating only via unidirectional *streams*. The producer on a stream has a blocking *push* primitive, which pushes an element on the stream, and the consumer has a blocking *pop* primitive. In the SPM, push and pop primitives are inserted at the boundaries between kernels, as indicated by the pragmas. This programming model is deterministic provided that the kernels themselves are deterministic, there is no other means of communication between kernels, each stream has one producer and one consumer, and the kernels cannot check whether a push or pop would block at a given time [22].

When the stream program is compiled, one or more kernels are mapped to each *task*, which is executed in its own thread. Tasks are managed by the ACOTES run-time system, *acolib*, which uses POSIX pthreads or some other threading library. It creates and initializes tasks, manages communication, and waits for their completion.

Tasks communicate using four *acolib* communications primitives, which use a push model similar to the DBI (Direct Blocking In-order) variant of TTL [23]. These primitives push or pop *buffers*, which contain a fixed number of elements chosen by the compiler. The buffer sizes can be different at the producer and consumer ends, but the following description assumes they are the same, to avoid extraneous detail. A *block* is the contents of one buffer, and i and j count blocks, starting at zero. The first argument, s, is the stream. Each end of the stream has a fixed number of buffers, chosen by the compiler, and denoted $n_p(s)$ and $n_c(s)$.

ProducerAcquire(s, k) Wait for the producer buffer for block $i + k$ to be available, meaning that the DMA transfer of block $i + k - n_p(s)$ has completed
ProducerSend(s) Wait for the consumer buffer for block i to be available, meaning that the producer has received acknowledgement that block $i - n_c(s)$ has been discarded. Then send the block and increment i
ConsumerAcquire(s, k) Wait for block $j + k$ to arrive in the consumer buffer
ConsumerDiscard(s) Discard block j, send acknowledgement, and increment j

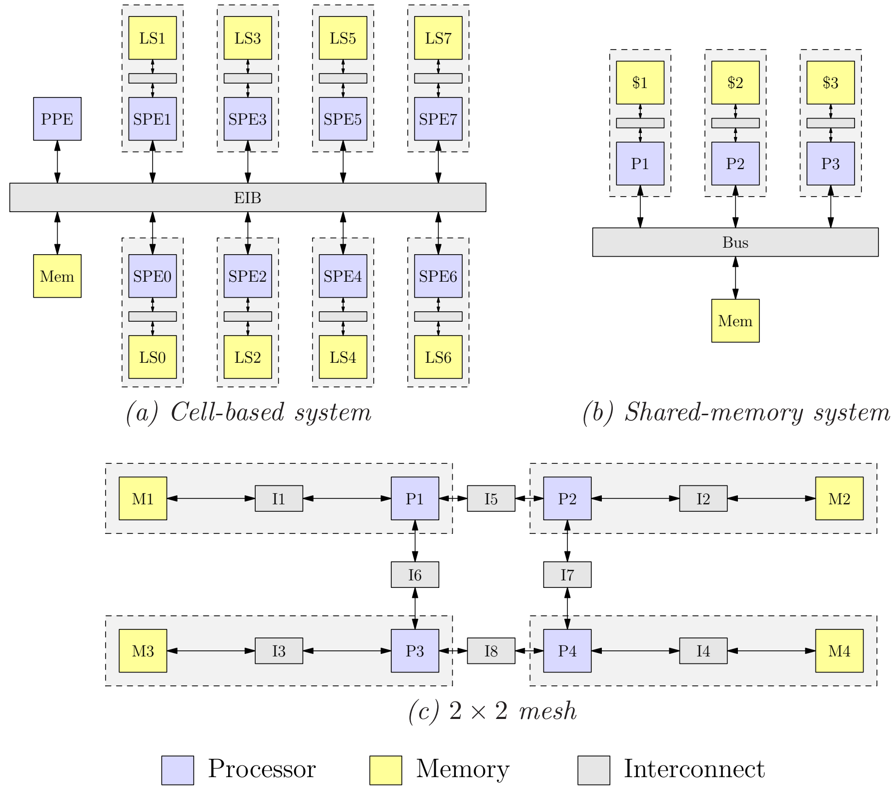
4 ASM Machine Description

The target is represented as a undirected bipartite graph $H = (V, E)$ with *processors* and *memories* in one partition, and *interconnects* in the other. Fig. 4 shows the topology of three example targets. The machine description defines the machine visible to software, provided by the OS and *acolib*, which may be different from the physical hardware. For example, the OS in a Playstation 3 makes six of the eight SPEs available to software, and does not make available the mapping from virtual to physical processor. We assume that the processors used by the stream program are not time-shared with other applications while the program is running.

Fig. 5 shows the parameters used to characterize each resource in the system, together with their values for the Cell B.E. with the current *acolib*, and estimated values for an SMP. Each core has a separate definition, allowing the ASM to support both heterogeneous and homogeneous systems.

Each processor is defined using the parameters in Fig. 5(a). The details of the processor's ISA and micro-architecture are already described in the compiler's back-end, so are not duplicated in the ASM. The ASM processor description lists the costs of the *acolib* calls. The costs of ProducerSend and ConsumerAcquire are given by a staircase function; i.e. a fixed cost, a block size, and an incremental cost for each complete or partial block after the first. This variable cost is necessary both for FIFOs and for distributed memory with DMA. For distributed memory, the size of a single DMA transfer is often limited by hardware, so that larger transfers require additional processor time in ProducerSend to program multiple DMA transfers. The discontinuity at 16K in Fig. 11 is due to this effect.

The addressSpace and hasIO parameters provide constraints on the compiler mapping, but are not required to evaluate the performance of a valid mapping.



(a) Cell-based system *(b) Shared-memory system*

(c) 2×2 *mesh*

Fig. 4. Topology of three example targets

The former defines the local address space of the processor; i.e. which memories are directly accessible and where they appear in local virtual memory, and is used to place stream buffers. The model assumes that the dominant bus traffic is communication via streams, so either the listed memories are private local stores, or they are shared memories accessed via a private L1 cache. In the latter case, the cache should be sufficiently effective that the cache miss traffic on the interconnect is insignificant. 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.

Each interconnect is defined using the parameters shown in Fig. 5(b). The system topology is given by the elements parameter, which for a given interconnect lists the adjacent processors and memories. Each interconnect is modelled as a bus with multiple channels, which has been shown to be a good approximation to the performance observed in practice when all processors and memories on a single link are equidistant [24]. If there are more messages than channels, then messages have to wait, and are arbitrated using a first-come-first-served policy. There is a single unbounded queue per bus to hold the messages ready

Parameter	Description	Cell	SMP
name	Unique name in platform namespace	'SPE*n*'	'CPU*n*'
clockRate	Clock rate, in GHz	3.2	2.4
hasIO	True if the processor can perform IO	False	True
addressSpace	List of the physical memories addressable by this processor and their virtual address	[(LS*n*,0)]	[('Mem',0)]
pushAcqCost	Cost, in cycles, to acquire a producer buffer (before waiting)	448	20
pushSendFixedCost	Fixed cost, in cycles, to push a block (before waiting)	1104	50
pushSendUnit	Number of bytes per push transfer unit	16384	0
pushSendUnitCost	Incremental cost, in cycles, to push pushUnit bytes	352	0
popAcqFixedCost	Fixed cost, in cycles, to pop a block (before waiting)	317	50
popAcqUnit	Number of bytes per pop transfer unit	16384	0
popAcqUnitCost	Incremental cost, in cycles, to pop popUnit bytes	0	0
popDiscCost	Cost, in cycles, to discard a consumer buffer (before waiting)	189	20

(a) *Definition of a processor*

Parameter	Description	Cell	SMP
name	Unique name in platform namespace	'EIB'	'FSB'
clockRate	Clock rate, in GHz	1.6	0.4
elements	List of the names of the elements (processors and memories) on the bus	['PPE', 'SPE0',···, 'SPE7']	['CPU0', ···, 'CPU3']
interfaceDuplex	If the bus has more than one channel, then define for each processor whether it can transmit and receive simultaneously on different channels	[True, ···, True]	[False, ···, False]
interfaceRouting	Define for each processor the type of routing from this bus: storeAndForward, cutThrough, or None	[None,···, None]	[None,···, None]
startLatency	Start latency, L, in cycles	80	0
startCost	Start cost on the channel, S, in cycles	0	0
bandwidthPerCh	Bandwidth per channel, B in bytes per cycle	16	16
finishCost	Finish cost, F, in cycles	0	0
numChannels	Number of channels on the bus	3	1
multiplexable	False for a hardware FIFO that can only support one stream	True	True

(b) *Definition of an interconnect*

Fig. 5. Processor and interconnect parameters of the ASM and values for two example targets (measured on Cell and estimated for a four-core SMP)

Parameter	Description	Cell	SMP
name	Unique name in platform namespace	'LS*n*'	'Mem'
size	Size, in bytes	262144	2147483648
clockRate	Clock rate, in GHz	3.2	0.4
latency	Access latency, in cycles	2	4
bandwidth	Bandwidth, in bytes per cycle	128	8

Fig. 6. Memory parameters of the ASM and values for two example targets

to be transmitted. The compiler statically allocates streams onto buses, but the choice of channel is made at runtime. The interfaceDuplex parameter defines for each resource; i.e. processor or memory, whether it can simultaneously read and write on different channels.

The bandwidth and latency of each channel is controlled using four parameters: the start latency (L), start cost (S), bandwidth (B), and finish cost (F). In transferring a message of size n bytes, the latency of the link is given by $L + S + \lfloor \frac{n}{B} \rfloor$ and the cost incurred on the link by $S + \lfloor \frac{n}{B} \rfloor + F$. This model is natural for distributed memory machines, and amounts to the assumption of cache-to-cache transfers on shared memory machines. Fig. 7 shows the temporal behaviour of a single message transfer on a bus.

Hardware routing is controlled using the interfaceRouting parameter, which defines for each processor whether it can route messages from this interconnect: each entry can take the value storeAndForward, cutThrough or None. Memory controllers and routers are modelled as a degenerate type of processor.

Each memory is defined using the parameters shown in Fig. 6. The latency and bandwidth figures are currently unused in the model, but may be used by the compiler to refine the estimate of the run time of each task. The memory definitions are used to determine where to place communications buffers, and provide constraints on blocking factors.

5 ASM Program Description

The compiled stream program is a connected directed graph of tasks and point-to-point streams, as described in Section 3. All synchronization between tasks happens in the *acolib* communications primitives also described in that section.

The program model uses a trace, and the same trace can be reused for several different mappings of the program onto the target—as illustrated by the small feedback loop in the bottom right of Fig. 3. This reuse avoids recompiling the whole program via Mercurium and GCC, just to obtain a new trace. Because tasks may have complex irregular behaviour, the trace contains contral flow information inside the tasks.

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 detail, the work function for a subtask is

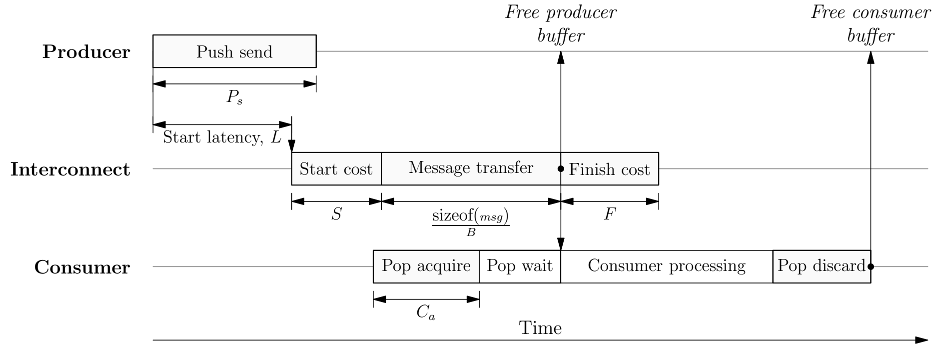
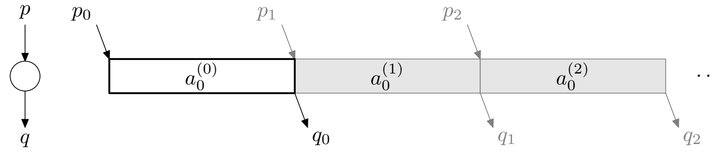
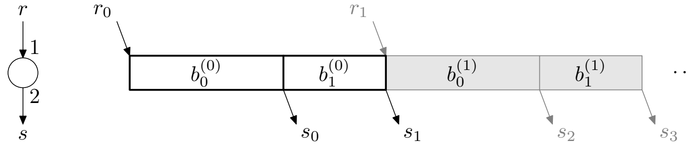


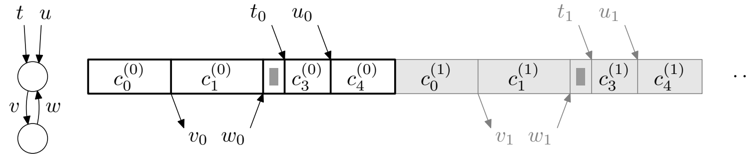
Fig. 7. Cost and latency of communication between tasks



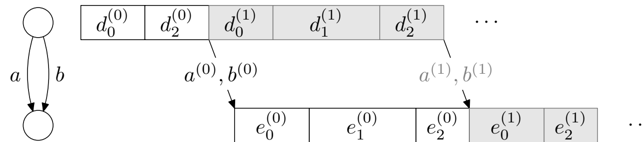
(a) Task containing a single subtask a_0; the superscript is the iteration number



(b) Interpolation task containing subtasks b_0 and b_1



(c) Irregular task containing subtasks c_0, c_1, c_2, c_3 and c_4



(d) Execution and communication of the program of Fig. 9 and Fig. 10

Fig. 8. Building tasks from subtasks

divided into three consecutive phases. First, the *acquire phase* obtains the next set of full input buffers and empty output buffers, using ProducerAcquire and ConsumerAcquire. Second, the *processing phase* works locally on these buffers, and is modelled using a fixed processing time, determined from a Paraver [25] trace. Finally, the *release phase* discards the input buffers using ConsumerDiscard, and sends the output buffers using ProducerSend, releasing the buffers in the same order they were acquired. This three-stage model is not a deep requirement of the ASM, and was introduced as a convenience in the implementation of the simulator, since our compiler will naturally generate subtasks of this form.

A task is the concatenation of one or more subtasks. Fig. 8(a), (b) and (c) show how to represent some tasks that perform an arbitrary fixed sequence of communication and computation. The superscript is the iteration number of the task. Although a stream has exactly one producer and one consumer task, it may be accessed from more than one subtask. For example, Fig. 8(b) has two subtasks, b_0 and b_1, and they both push elements on stream s. In order to support control flow, all subtasks of all tasks are placed into a common *control-flow hierarchy*. Subtasks are executed conditionally or repeatedly based on a Paraver trace attached to this control-flow hierarchy, with this common trace ensuring that communicating tasks behave consistently.

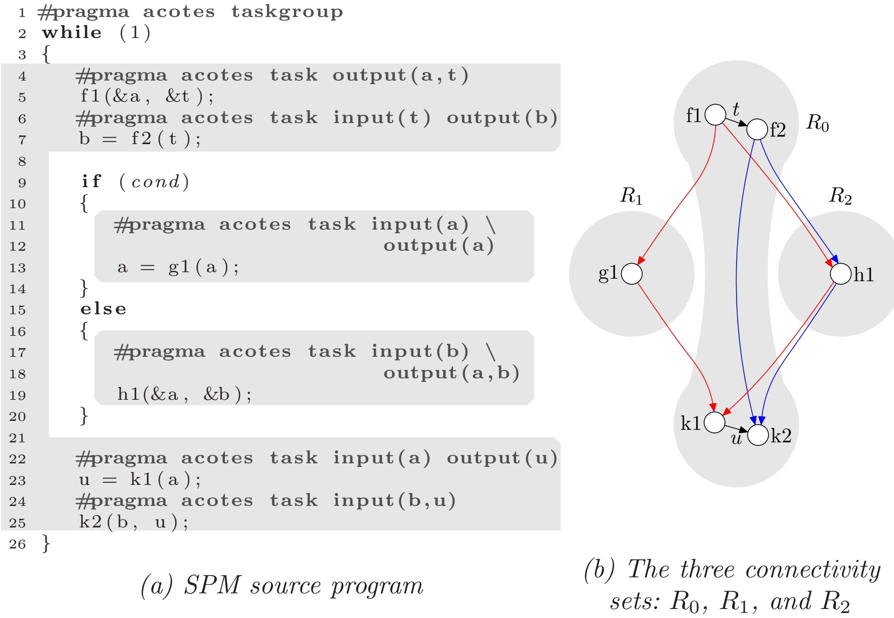
Each *if* or *while* statement has an associated *control variable*, which gives its sequence of arguments. As part of the conversion from SPM to C, the Mercurium tool inserts calls to the trace collection functions, which record the control variables in the Paraver trace. The control variables are represented using *event records* in the trace; the *event type* identifies the control variable, and the *event value* gives its value.

Fig. 9(a) is the source code for an example stream program containing six kernels and an *if* statement. Fig. 9(b) is the stream graph. Because the program contains an *if* statement, the multiplicities of the kernels depend on the data. However, within each shaded region, R_0, R_1, and R_2, the program is homogeneous Synchronous Dataflow (SDF).

The ASM sees the program after it has been partitioned. Imagine that the partition is as given in Fig. 10(a), so that task D contains kernels f1, f2, and g1, and task E contains kernels h1, k1, and k2. The tasks execute at the same frequency, but they both contain kernels from inside and outside the *if* statement. Conditional execution of g1 and h1, including modelling of computation times and their pushes and pops, is driven using a control variable in the trace.

Fig. 10(c) is one way for the compiler to implement the given partition. The tasks are decomposed into subtasks, d0, d1, and d2, and e0, e1, and e2. Fig. 10(b) shows the control flow hierarchy that controls the execution. The subtasks at the root are always executed, d1 is executed if the control variable is True, and e1 is executed if it is False. Fig. 8(d) shows an execution trace where the decision values for this node are False, True, $\cdots$. The control variable attached to a *while* node is similar, but it counts the number of iterations of the loop.

There are no explicit streams carrying the control variables of *if* or *while* statements between tasks. The compiler ensures that such tasks are consistent



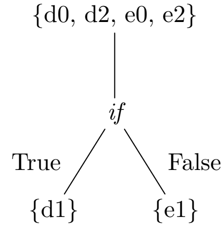
(a) SPM source program

(b) The three connectivity sets: R_0, R_1, and R_2

Fig. 9. Example stream program with data-dependent flow

Task	**Kernels**
D	f1, f2, g1
E	h1, k1, k2

(a) Partition



(b) Control flow hierarchy

```
1  void D(void)
2  {
3    while (1)
4    {
5      f1(&a, &t);       // d0
6      b = f2(t);        // d0
7      if (cond)         // d1
8         a = g1(a);     // d1
9      push(s, a);       // d2
10     push(t, b);       // d2
11   }
12 }
13
14 void E(void)
15 {
16   while (1)
17   {
18     a = pop(s);       // e0
19     b = pop(t);       // e0
20     if (!cond)        // e1
21        h1(&a, &b);    // e1
22     u = k1(a);        // e2
23     k2(b, u);         // e2
24   }
25 }
```

(c) Extended C for partition

Fig. 10. Representation of data-dependent flow between tasks and subtasks

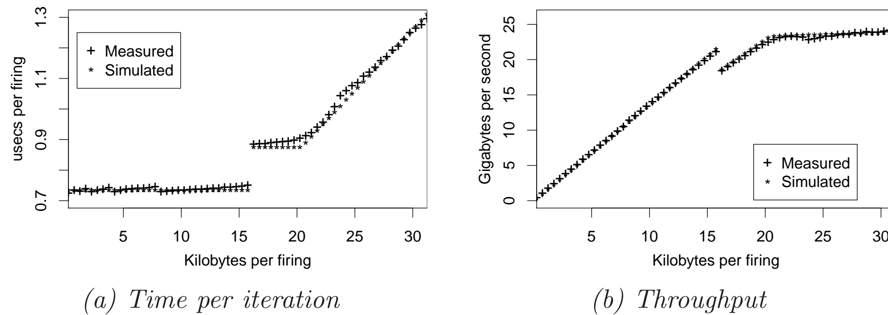
with each other, and may in the general case have to add such streams to do so. There are, however, examples where it would be unnecessary. It is assumed that the compiler produces correct code, and the ASM uses the control-flow hierarchy to ensure that its own model is consistent.

A stream is defined by the size of each element, and the location and length of either the separate producer and consumer buffers (distributed memory) or the single shared buffer (shared memory). These buffers do not have to be of the same length. If the producer or consumer task uses the peek primitive, then the buffer length should be reduced to model the effective size of the buffer, excluding the elements of history. The Finite Impulse Response (FIR) filters in the GNU radio benchmark of Section 7 are described in this way. It is possible to specify a number of elements to prequeue on the stream before execution begins.

6 Platform characterisation

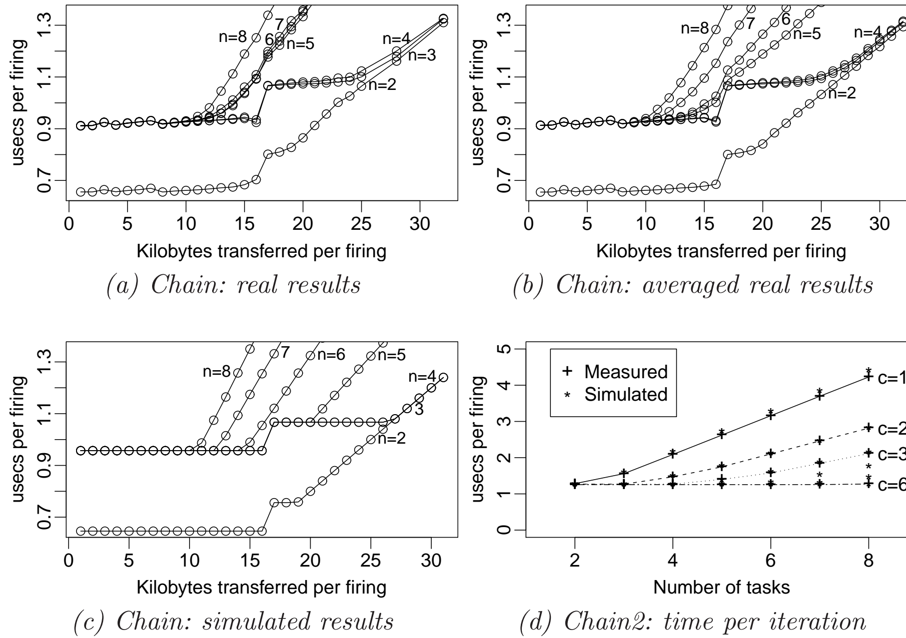
We use a small suite of benchmarks and target platforms, which have been translated by hand into the description files. The benchmarks were evaluated on an IBM QS20 blade, which has two Cell processors. The *producer-consumer* benchmark is used to determine basic parameters, and has two actors: a producer, and consumer, with two buffers at each end. The *chain* benchmark, is a linear pipeline of n tasks, and is used to characterize bus contention. The *chain2* benchmark is used to model latency and queue contention, and is a linear pipeline, similar to chain, but with an extra *cut* stream between the first and last tasks. The number of blocks in the consumer-side buffer on the cut stream is a parameter, c. For all benchmarks, the number of bytes per iteration is denoted b.

Fig. 11 shows the time per iteration for *producer-consumer*, as a function of b. The discontinuity at $b = 16\text{K}$ is due to the overhead of programming two DMA transfers. For $b < 20.5\text{K}$, the bottleneck is the computation time of the producer task, as can be seen in Fig. 13(a) and (b), which compares real and simulated traces for $b = 8\text{K}$. For $b > 20.5\text{K}$, the bottleneck is the interconnect,



(a) Time per iteration *(b) Throughput*

Fig. 11. Results for the producer-consumer benchmark



(a) Chain: real results *(b) Chain: averaged real results*

(c) Chain: simulated results *(d) Chain2: time per iteration*

Fig. 12. Time per iteration for the chain and chain2 benchmarks

and the slope of the line is the reciprocal of the bandwidth: 25.6GB/s. Fig. 13(c) and (d) compares real and simulated traces for $b = 24\text{K}$. The maximum relative error for $0 < b < 32\text{K}$ is 3.1%.

Fig. 12 shows the time per iteration for *chain*, as a function of n, the number of tasks, and b, the block size. Fig. 12(a) shows the measured performance on the IBM QS20 blade, when tasks are allocated to SPEs in increasing numerical order. The EIB on the Cell processor consists of two clockwise and two anticlockwise rings, each supporting up to three simultaneous transfers provided that they do not overlap. The drop in real, measured, performance from $n = 4$ to $n = 5$ and from $n = 7$ to $n = 8$ is due to contention on certain hops of the EIB, which the ASM does not attempt to model. As described in Section 4, the ASM models an interconnect as a set of parallel buses. Fig. 12(b) shows the average of the measured performance of three random permutations of the SPEs. The simulated results in Fig. 12(c) are hence close to the expected results, in a probabilistic sense, when the physical ordering of the SPEs is not known.

Fig. 12(d) shows the time per iteration for *chain2*, as a function of the number of tasks, n, and the size of the consumer-side buffer of the *shortcut* stream between the first and last tasks, denoted c. The bottleneck is either the computation time of the first task (1.27us per iteration) or is due to the latency of the chain being exposed due to the finite length of the queue on the shortcut stream. Fig. 13(e) and (f) shows real and simulated traces for the latter case, with $n = 7$ and $c = 2$.

Kernel	Multiplicity	History buffer	Time per firing (us)	% of total load
Demodulation	8	n/a	398	1.7%
Lowpass (middle)	1	1.6K	7,220	3.8%
Bandpass	8	1.6K	7,246	30.4%
Carrier	8	3.2K	14,351	60.2%
Frequency shift	8	n/a	12	0.1%
Lowpass (side)	1	1.6K	7,361	3.9%
Sum	1	n/a	13	0.0%

(a) Kernels

Task	Kernel	Blocking factor
1	Demodulation	512
2	Lowpass (middle)	128
3	Bandpass	1024
4	Carrier	1024
5	Frequency shift	1024
6	Lowpass (side)	128
7	Sum	128

(b) Naive mapping

Task	Kernel	Blocking factor
1	Demodulation	1024
	Bandpass	1024
2	Carrier (even)	1024
3	Carrier (odd)	1024
4	Lowpass (middle)	128
	Frequency shift	1024
	Lowpass (side)	128
	Sum	128

(c) Optimized mapping

Table 1. Kernels and mappings of the GNU radio benchmark

7 Validation

This section describes the validation work using our GNU radio benchmark, which is based on the FM stereo demodulator in GNU Radio [26]. Table 1(a) shows the computation time and multiplicity per kernel, the latter being the number of times it is executed per pair of l and r output elements. Four of the kernels, being FIR filters, peek backwards in the input stream, requiring history as indicated in the table. Other than this, all kernels are stateless.

Table 1 shows two mappings of the GNU radio benchmark onto the Cell B.E. The first allocates one task per kernel, using seven of the eight SPEs. Based on the resource utilization, the *Carrier* kernel was split into two worker tasks and the remaining kernels were partitioned onto two other SPEs. This gives 79% utilization of four processors, and approximately twice the throughput of the unoptimized mapping, at 7.71ms per iteration, rather than 14.73ms per iteration. The throughput and latency from the simulator are within 0.5% and 2% respectively.

8 Using the ASM

This section explains how the ACOTES stream compiler uses the ASM machine description and simulator.

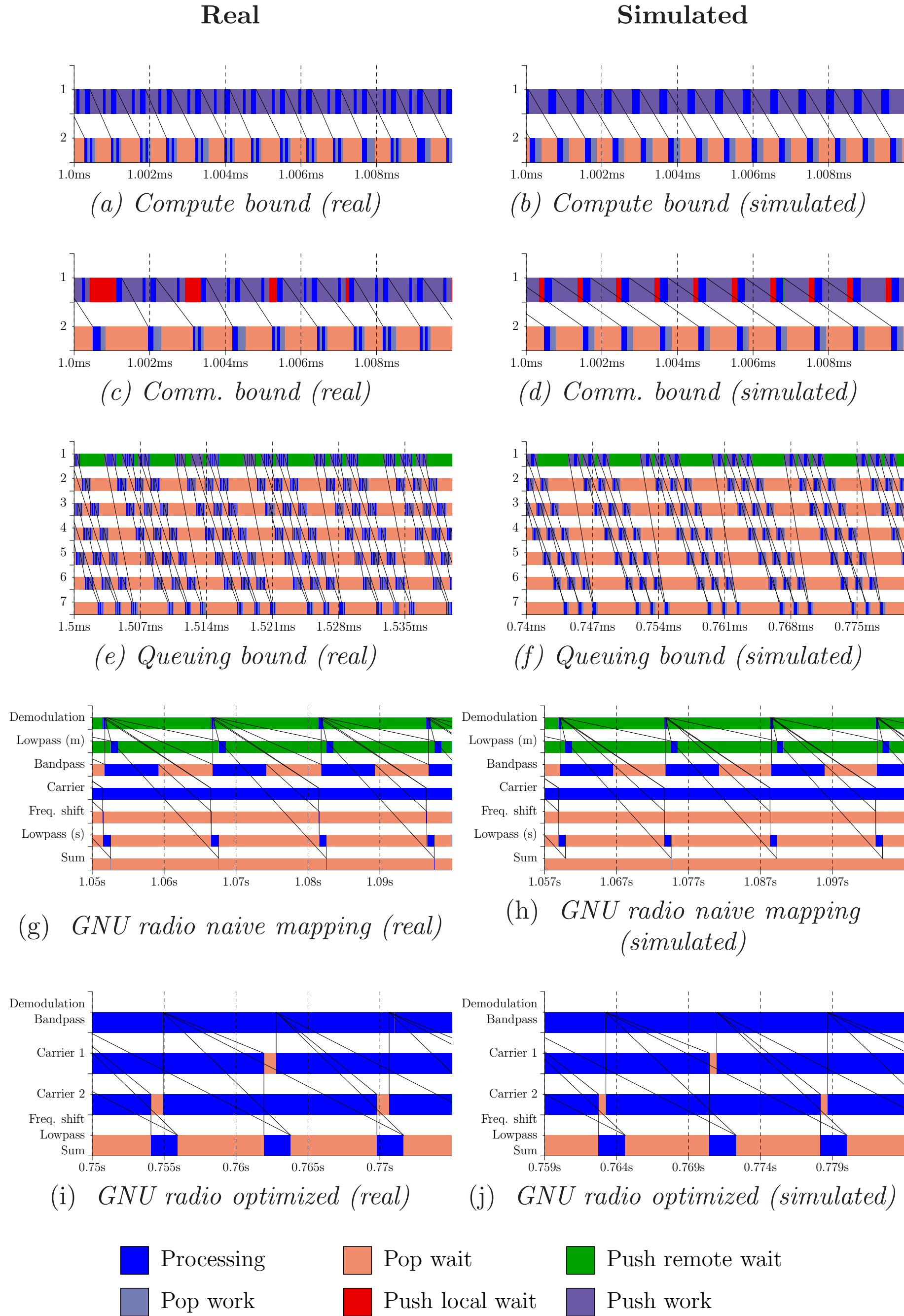
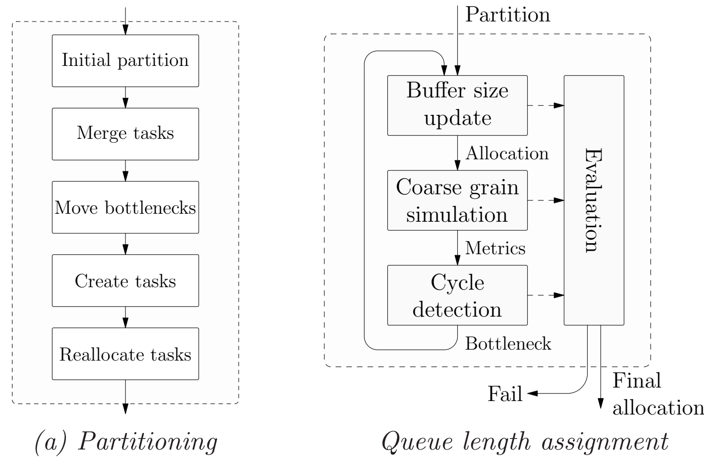


Fig. 13. Comparison of real and simulated traces



(a) Partitioning *Queue length assignment*

Fig. 14. Detail on the main phases in the search algorithm

Partitioning The partitioning phase decides how to fuse kernels into tasks, and allocates these tasks to processors [19]. It represents the target as an undirected bipartite graph, $H = (V_H, E_H)$, taken directly from the ASM. The weight of processor p, denoted w^p is its clock rate in GHz, and the weight of interconnect u, denoted w^u is its bandwidth in GB/s. The static routing table is determined using minimum distance routing, respecting the interfaceRouting parameters. We didn't find it necessary to store the routing table explicitly in the ASM.

Fig. 14(a) shows the main stages in the partitioning phase. An initial partition is constructed by recursively subdividing the target and program. The partition is then improved using several optimization passes.

The partitioning phase uses *connectivity sets* [19] to constrain the mapping to make sure the compiler can support it. In particular, the ACOTES compiler can only fuse kernels that are lexicographically adjacent in the same basic block. Each connectivity set is therefore a pair of adjacent kernels in the same basic block. In a more advanced compiler, we would expect the connectivity sets to be as illustrated in Fig. 9(b).

Buffer sizing The queue length assignment phase allocates memory for stream buffers, subject to memory constraints, and taking account of variable computation times and task multiplicities [18]. The objectives are to maximise throughput and minimize latency.

This phase is an iterative algorithm, which uses the ASM simulator to find the throughput, utilization, and latency, given the candidate buffer sizes. As mentioned in Section 2, simulation is used because a mathematical model is unlikely to capture the real behaviour.

Fig. 14(b) shows the main stages in the queue length assignment phase. A *cycle detection algorithm* uses statistics from the ASM simulator to find the

bottleneck. There are two cycle detection algorithms: the *baseline* algorithm uses only the total wait time on each primitive on each stream, and the *token* algorithm tracks dependencies through tasks. The *buffer size update algorithm* chooses the initial buffer sizes, and adjusts them to resolve the bottleneck. The *evaluation algorithm* monitors progress and decides when to stop, choosing the buffer sizes that achieved the best performance-latency tradeoff.

The inputs to the queue length assignment phase are the stream program, minimum buffer sizes, and the *memory constraint graph*. The minimum buffer sizes can be one block, because an SPM stream program is acyclic. The memory constraint graph is a bipartite graph, $\mathcal{H} = (R_{\mathcal{H}}, E_{\mathcal{H}})$, where the vertices are the processors and memories, and the edges connect processors to their local memories. Fig. 15 shows a memory constraint graph for the Cell B.E.

The memory constraint graph is generated from the addressSpace parameter for each processor. The remaining capacities are taken from the size parameters of the memories, *minus* the sizes of any code and data already in them.

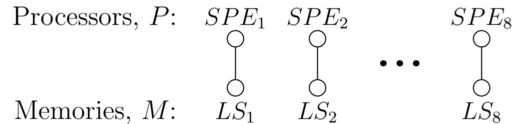


Fig. 15. Memory constraint graph for the Cell Broadband Engine

9 Related work

Recent work on stream programming languages, most notably StreamIt [27] and Synchronous Data Flow (SDF) [28], has demonstrated how a compiler may potentially match the performance of hand-tuned sequential or multi-threaded code [29].

Most work on machine description languages for retargetable compilers has focused on describing the ISA and micro-architecture of a single processor. Among others, the languages ISP, LISA, and ADL may be used for simulation, and CODEGEN, BEG, BURG, nML [30], EXPRESSION [31], Maril and GCC's .md machine description are intended for code generation (see; e.g. [32]). The ASM describes the behaviour of the system in terms of that of its parts, and is designed to co-exist with these lower-level models.

The Stream Virtual Machine (SVM) is an intermediate representation of a stream program, which forms a common language between a high-level and low-level compiler [33, 34]. Each kernel is given a linear computation cost function, comprised of a fixed overhead and a cost per stream element consumed. There is no model of irregular dataflow. The SVM architecture model is specific to graphics processors (GPUs), and characterizes the platform using a few parameters such as the bandwidth between local and global memory. The PCA Machine

Model [35], by the Morphware Forum, is an XML definition of a reconfigurable computing device, in terms of *resources*, which may be processors, DMA engines, memories and network links. The reconfigurable behaviour of a target is described using *ingredients* and *morphs*. Unlike the ASM, the PCA Machine Model describes the entire target, including low-level information about each processor's functional units and number of registers.

ORAS is a retargetable simulator for design-space exploration of stream-based dataflow architectures [36]. The target is defined by the *architecture instance*, which defines the hardware as a graph of architecture elements, similar to the resources of the ASM. The purpose is performance analysis rather than compilation, and the system is specified to a greater level of detail than the ASM.

Gordon et al. present a compiler for the StreamIt language targeting the Raw Architecture Workstation, and applying similar transformations to those discussed in this paper [37]. As the target is Raw, there is no general machine model similar to the ASM. The compiler uses simulated annealing to minimize the length, in cycles, of the critical path. Our approach has higher computational complexity in the compiler's cost model, but provides retargetability and greater flexibility in the program model.

Gedae is a proprietary stream-based graphical programming environment for signal processing applications in the defence industry. The developer specifies the mapping of the stream program onto the target, and the compiler generates the executable implementation [38]. There is no compiler search algorithm or cost model. A version of Gedae has been released for the Cell processor.

Kupriyanov et al. describe an architecture description language (ADL) for *weakly-programmable arrays* [39] of simple processors in a regular two-dimensional mesh.

10 Conclusions

This paper presents the Abstract Streaming Machine (ASM), which is the machine description used by the ACOTES stream compiler. The ACOTES stream compiler automatically partitions, blocks, and schedules a machine-independent stream program for best performance on a heterogeneous multiprocessor system.

The ASM is implemented by a coarse grain model, driven by a reusable trace. We explain how the ASM is used by the partitioning and queue length assignment stages in the ACOTES compiler. We also give the machine descriptions for two targets: the Cell Broadband Engine and an SMP.

Acknowledgements The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contract no. TIN2007-60625), the European Commission in the context of the ACOTES project (contract no. IST-34869) and the HiPEAC Network of Excellence (contract no. IST-004408). We would also like to acknowledge our partners in the ACOTES project for the insightful discussions on the topics presented in this paper.

References

1. Carpenter, P.M., Ramirez, A., Ayguade, E.: The abstract streaming machine: Compile-time performance modelling of stream programs on heterogeneous multiprocessors. In Bertels, K., Dimopoulos, N.J., Silvano, C., Wong, S., eds.: SAMOS. Volume 5657 of Lecture Notes in Computer Science., Springer (2009) 12–23
2. Olukotun, K., Hammond, L.: The future of microprocessors. Queue **3**(7) (2005) 26–29
3. Amdahl, G.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference, ACM New York, NY, USA (1967) 483–485
4. Kumar, R., Tullsen, D., Jouppi, N., Ranganathan, P.: Heterogeneous chip multiprocessors. Computer **38**(11) (2005) 32–38
5. Chaoui, J., Cyr, K., Giacalone, J., Gregorio, S., Masse, Y., Muthusamy, Y., Spits, T., Budagavi, M., Webb, J.: OMAP: Enabling Multimedia Applications in Third Generation (3G) Wireless Terminals. SWPA001 (2000)
6. Hirata, K., Goodacre, J.: ARM MPCore; The streamlined and scalable ARM11 processor core. In: ASP-DAC '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference, Washington, DC, USA, IEEE Computer Society (2007) 747–748
7. Intel: IXP2850 Network Processor: Hardware Reference Manual. (2004)
8. Dutta, S., Jensen, R., Rieckmann, A.: Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems. IEEE Design & Test of Computers (2001) 21–31
9. Artieri, A., Alto, V., Chesson, R., Hopkins, M., Rossi, M.: Nomadik Open Multimedia Platform for Next-generation Mobile Devices. STMicroelectronics Technical Article TA305 (2003)
10. ClearSpeed: http://www.clearspeed.com/docs/resources/ClearSpeed_Architecture_Whitepaper_Feb07v2.pdf (2005) CSX Processor Architecture.
11. Chen, T., Raghavan, R., Dale, J., Iwata, E.: Cell Broadband Engine Architecture and its first implementation. IBM developerWorks (2005)
12. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley (2006)
13. ACOTES IST-034869: http://www.hitech-projects.com/euprojects/ACOTES/ Advanced Compiler Technologies for Embedded Streaming.
14. Munk, H., et al.: ACOTES Project: Advanced Compiler Technologies for Embedded Streaming. International Journal of Parallel Programming (2010) 1–54 10.1007/s10766-010-0132-7.
15. Carpenter, P.M., Ródenas, D., Martorell, X., Ramírez, A., Ayguadé, E.: A streaming machine description and programming model. In Vassiliadis, S., Berekovic, M., Hämäläinen, T.D., eds.: SAMOS. Volume 4599 of Lecture Notes in Computer Science., Springer (2007) 107–116
16. ACOTES: IST ACOTES Project Deliverable D2.2 Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version. (2008)
17. Balart, J., Duran, A., Gonzalez, M., Martorell, X., Ayguade, E., Labarta, J.: Nanos Mercurium: a Research Compiler for OpenMP. In: Proceedings of the European Workshop on OpenMP. Volume 2004. (2004)

18. Carpenter, P.M., Ramirez, A., Ayguade, E.: Buffer sizing for self-timed stream programs on heterogeneous distributed memory multiprocessors. In: High Performance Embedded Architectures and Compilers, 5th International Conference, HiPEAC 2010, Springer (2010) 96–110
19. Carpenter, P.M., Ramirez, A., Ayguade, E.: Mapping stream programs onto heterogeneous multiprocessor systems. In: CASES '09. (2009) 57–66
20. Fursin, G., Cohen, A.: Building a Practical Iterative Interactive Compiler. In: 1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07). (2007)
21. Stephens, R.: A survey of stream processing. Acta Informatica **34**(7) (1997) 491–541
22. Kahn, G.: The semantics of a simple language for parallel processing. Information Processing **74** (1974) 471–475
23. van der Wolf, P., de Kock, E., Henriksson, T., Kruijtzer, W., Essink, G.: Design and programming of embedded multiprocessors: an interface-centric approach. Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis (2004) 206–217
24. Girona, S., Labarta, J., Badia, R.: Validation of Dimemas communication model for MPI collective operations. Proc. EuroPVM/MPI (2000)
25. CEPBA: http://www.cepba.upc.edu/paraver/ Paraver performance visualization and analysis tool.
26. GNU Radio: http://www.gnu.org/software/gnuradio/
27. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications. ICCC **4** (2002)
28. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9) (1987) 1235–1245
29. Gummaraju, J., Rosenblum, M.: Stream Programming on General-Purpose Processors. In: Proc. MICRO 38, Barcelona, Spain (November 2005)
30. Fauth, A., Van Praet, J., Freericks, M.: Describing instruction set processors using nML. In: Proceedings of the 1995 European conference on Design and Test. 503
31. Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., Nicolau, A.: EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In: Proceedings of the conference on Design, automation and test in Europe. (1999)
32. Ramsey, N., Davidson, J., Fernandez, M.: Design principles for machine-description languages. ACM Trans. Programming Languages and Systems (1998)
33. Labonte, F., Mattson, P., Thies, W., Buck, I., Kozyrakis, C., Horowitz, M.: The stream virtual machine. Proc. PACT 2004 267–277
34. Mattson, P., Thies, W., Hammond, L., Vahey, M.: Streaming virtual machine specification 1.0. Technical report, http://www.morphware.org (2004)
35. Mattson, P.: PCA Machine Model, 1.0. Technical report (2004)
36. Kienhuis, B.: Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools. Delft University of Technology, The Netherlands (1999)
37. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. Proc. ASPLOS 2006 (2006) 151–162
38. Lundgren, W., Barnes, K., Steed, J.: Gedae: Auto Coding to a Virtual Machine. Proc. HPEC (2004)
39. Kupriyanov, A., Hannig, F., Kissler, D., Teich, J., Schaffer, R., Merker, R.: An architecture description language for massively parallel processor architectures. In: Proceedings 9th ITG/GMM/GI Workshop, Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen. (2006)