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 offers a portable way for regular applications such as digital video, software radio, multimedia and 3D graphics to exploit a multiprocessor machine. The compiler maps a portable stream program onto the target, automatically sizing communications buffers and applying optimizing transformations such as task fission or fusion, unrolling loops and aggregating communication. We present a machine description and performance model for an iterative stream compilation flow, which represents the stream program running on a heterogeneous multiprocessor system with distributed or shared memory. The model is a key component of the ACOTES open-source stream compiler currently under development. Our experiments on the Cell Broadband Engine show that the predicted throughput has a maximum relative error of 15% across our benchmarks.

1 Introduction

Many people [1] have recognized the need to change the way software is written to take advantage of multi-core systems [2] and distributed memory [3–5]. This paper is concerned with applications such as digital video, software radio, signal processing and 3D graphics, all of which may be represented as block diagrams, in which independent blocks communicate and synchronize only via regular streams of data. Such applications have high task and data parallelism, which is hidden when the program is written in C or a similar sequential programming language, requiring the programmer to apply high level optimizations such as task fusion, fission and blocking transformations by hand. Recent work on stream programming languages, most notably StreamIt [6] and Synchronous Data Flow (SDF) [7], has demonstrated how a compiler may potentially match the performance of hand-tuned sequential or multi-threaded code [8].

This work is part of the ACOTES project [9], which is developing a complete open-source stream compiler for embedded systems. This compiler will automatically partition a stream program to use task-level parallelism, size communications buffers and aggregate communications through blocking. This paper describes the Abstract Streaming Machine (ASM), which represents the target system to this compiler. Figure 1 shows the iterative compilation flow, with a

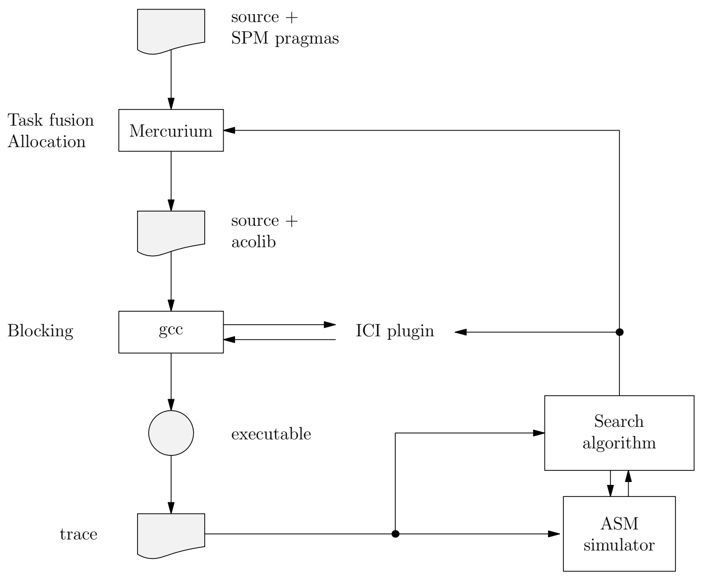


Fig. 1. The ACOTES iterative stream compiler

search algorithm determining the candidate mapping, which is compiled using Mercurium [10] and GCC. The Mercurium source-to-source convertor translates from the SPM source language [11, 12], and performs task fusion and allocation. The resulting multi-threaded program is compiled using GCC, which we are extending within the project to perform blocking to aggregate computation and communication. Running the executable program generates a trace, which is analysed by the search algorithm to resolve bottlenecks. An alternative feedback path generates a trace using the ASM simulator, which is a coarse-grain model of the ASM. This path does not require recompilation, and is used when resizing buffers or to approximate the effect of fission or blocking.

2 Stream programming

There are several definitions of stream programming, differing mostly in the handling of control flow and restrictions on the program graph topology [13]. All stream programming models, however, represent the program as a set of *kernels*, communicating only via unidirectional *streams*. The producer has a blocking *push* primitive and the consumer has a blocking *pop* primitive. 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 particular time [14].

When the stream program is compiled, one or more kernels are mapped to each task, which is executed in its own thread. The communications primitives are provided by the ACOTES run-time system, *acolib*, which also creates and initializes threads at the start of the computation, and waits for their completion at the end. The run-time system supports two-phase communication, and can be implemented for shared memory, distributed memory with DMA, and hardware FIFOs. On the producer side, `pushAcquire` returns a pointer to an empty array of n_p elements; the n_p parameter is equal to the producer's blocking factor, and is supplied during stream initialization. When the task has filled this buffer with new data, it calls `pushSend` to request that acolib delivers the data to the consumer. On the consumer side, `popAcquire` returns a pointer to the next full block of n_c elements. When the consumer has finished with the data in this block, it calls `popDiscard` to mark the block as empty.

3 ASM Machine Description

The target is represented as a bipartite graph of *processors* and *memories* in one partition, and *interconnects* in the other. Figure 2 shows the topology of two example targets. Each processor and interconnect is defined using the parameters summarized in Figures 3 and 4, and described below. The machine description defines the machine visible to software, which may not exactly match the physical hardware. For example, the OS in a Playstation 3 makes six of the eight SPEs available to software. We assume that the processors used by the stream program are not time-shared with other applications while the program is running.

Each processor is defined using the parameters shown in Figure 3(a). The details of the processor's ISA and micro-architecture are described internally to the back-end compiler, so are not duplicated in the ASM. The processor description includes the costs of the acolib library calls. The costs of the `pushSend` and `popAcquire` primitives 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 `pushSend` to program multiple DMA transfers. The discontinuity at 16K in Figure 5 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. 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.

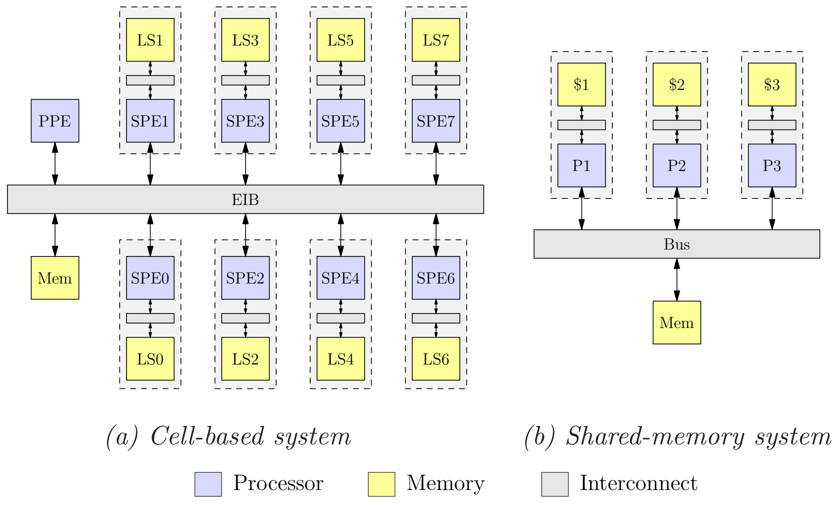


Fig. 2. Topology of two example targets

Each interconnect is defined using the parameters shown in Figure 3(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 [15]. Each bus has a single unbounded queue to hold the messages ready to be transmitted, and one or more channels on which to transmit them. 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.

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*.

Each memory is defined using the parameters shown in Figure 4. 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.

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

(a) *Definition of a processor*

Parameter	Description	Value
`name`	Unique name in platform namespace	`‘EIB’`
`clockRate`	Clock rate, in GHz	`1.6`
`elements`	List of the names of the elements (processors and memories) on the bus	`[‘PPE’,‘SPE0’, ···, ‘SPE7’]`
`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]`
`interfaceRouting`	Define for each processor the type of routing from this bus: *storeAndForward*, *cutThrough*, or *None*	`[None, ···, None]`
`startLatency`	Start latency, L, in cycles	`80`
`startCost`	Start cost on the channel, S, in cycles	`0`
`bandwidthPerCh`	Bandwidth per channel, B in bytes per cycle	`16`
`finishCost`	Finish cost, F, in cycles	`0`
`numChannels`	Number of channels on the bus	`3`
`multiplexable`	False for a hardware FIFO that can only support one stream	`True`

(b) *Definition of an interconnect*

Fig. 3. Processor and interconnect parameters of the Abstract Streaming Machine and values for the Cell Broadband Engine

Parameter	Description	Value
`name`	Unique name in platform namespace	`'LS`*n*`'`
`size`	Size, in bytes	`262144`
`clockRate`	Clock rate, in GHz	`3.2`
`latency`	Access latency, in cycles	`2`
`bandwidth`	Bandwidth, in bytes per cycle	`128`

Fig. 4. Memory parameters of the Abstract Streaming Machine and values for the Cell Broadband Engine

4 ASM Program Description

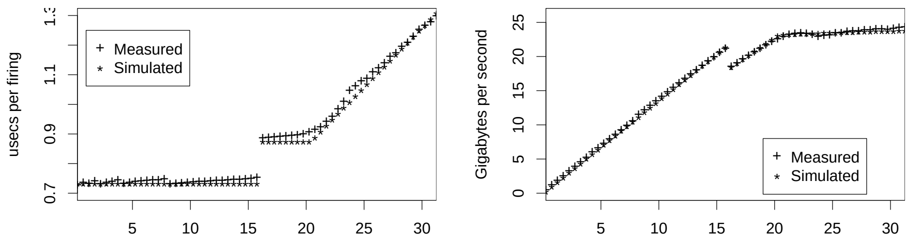
The compiled stream program is a connected directed graph of tasks and point-to-point streams, as described in Section 2. All synchronization between tasks happens in the blocking acolib communications primitives described above.

A task may have complex data-dependent or irregular behaviour. The basic unit of sequencing inside a task is the *subtask*, which pops a fixed number of elements from each input stream and pushes a fixed number of elements on each output stream. In detail, the work function for a subtask is divided into three consecutive phases. First, the *acquire phase* obtains the next set of full input buffers and empty output buffers. Second, the *processing phase* works locally on these buffers, and is modelled using a fixed processing time, determined from a trace. Finally, the *release phase* discards the input buffers, and sends the output buffers, 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 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 that share the buffer. The Finite Impulse Response (FIR) filters in the GNU radio benchmark of Section 6 are described in this way. It is possible to specify a number of elements to prequeue on the stream before execution begins.

5 Implementation and methodology

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



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

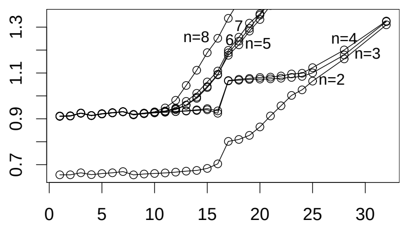
Fig. 5. Results for producer-consumer benchmark

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.

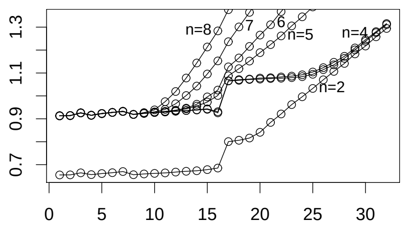
Figure 5 shows the time per iteration for *producer-consumer*, as a function of b. The discontinuity at $b = 16$K is due to the overhead of programming two DMA transfers. For $b < 20.5$K, the bottleneck is the computation time of the producer task, as can be seen in Figure 7(a) and (b), which compares real and simulated traces for $b = 8$K. For $b > 20.5$K, the bottleneck is the interconnect, and the slope of the line is the reciprocal of the bandwidth: 25.6GB/s. Figure 7(c) and (d) compares real and simulated traces for $b = 24$K. The maximum relative error for $0 < b < 32$K is 3.1%.

Figure 6 shows the time per iteration for *chain*, as a function of n, the number of tasks, and b, the block size. Figure 6(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 particular hops of the EIB, which the ASM does not attempt to model. As described in Section 3, the ASM models an interconnect as a set of parallel buses connecting an (unordered) set of processors. Figure 6(b) shows the average of the measured performance of three random permutations of the SPEs. The simulated results in Figure 6(c) are hence close to the expected results, in a probabilistic sense, when the physical ordering of the SPEs is not known.

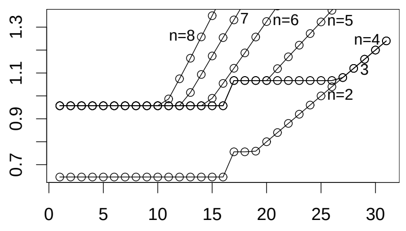
Figure 6(d) shows the time per iteration for *chain2*, as a function of the number of tasks, n, and the size of the consumer-side buffer of the *shortcut* stream between the first and last tasks, denoted c. The bottleneck is either the computation time of the first task (1.27us per iteration) or is due to the latency of the chain being exposed due to the finite length of the queue on the shortcut stream. Figure 7(e) and (f) shows real and simulated traces for the latter case, with $n = 7$ and $c = 2$.



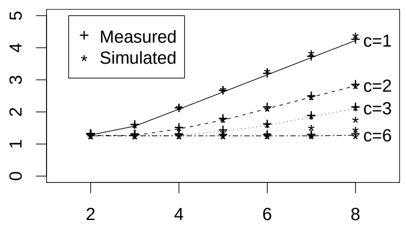
(a) Chain: real results



(b) Chain: averaged real results



(c) Chain: simulated results



(d) Chain2: time per iteration

Fig. 6. Time per iteration for the chain and chain2 benchmarks

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

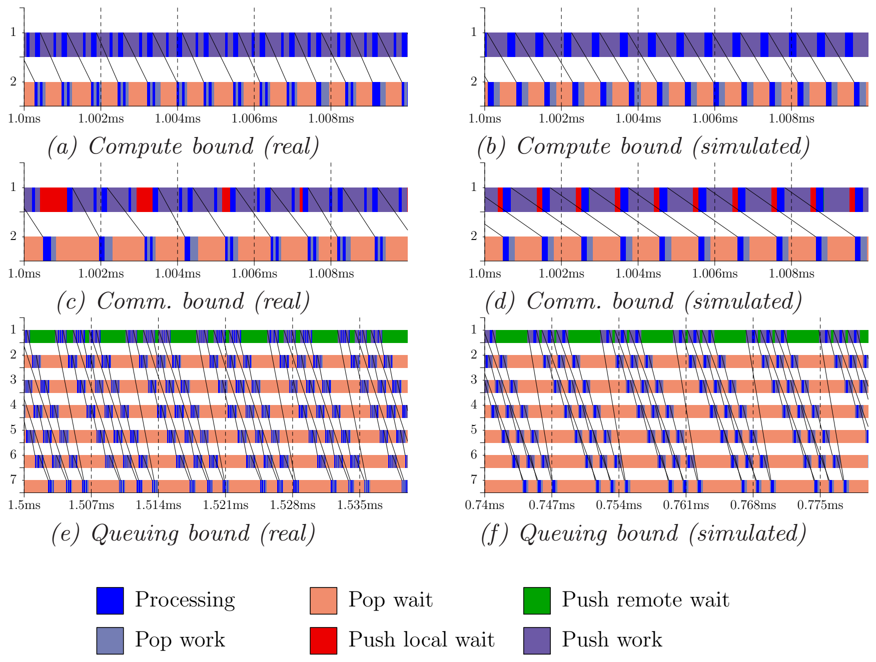


Fig. 7. Comparison of real and simulated traces

6 Validation

This section describes the validation work using our GNU radio benchmark, which is based on the FM stereo demodulator in GNU Radio [16]. 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 possible mappings of the GNU radio benchmark onto the Cell processor, being the mapping of kernel to task and blocking factors. The first allocates one task per kernel, using a total of seven of the eight available 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.

7 Related work

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, EXPRESSION, Maril and GCC's .md machine description are intended for code generation (see; e.g. [17]). 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 [18, 19]. 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 [20], 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 [21]. The target is specified by the *architecture instance*, which defines the hardware as a graph of architecture elements, similar to the resources of the ASM. Since the purpose is performance analysis rather than compilation, 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 [22]. 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 defense industry. The developer specifies the mapping of the stream program onto the target, and the compiler generates the executable implementation [23]. There is no compiler search algorithm or cost model. A version of Gedae has been released for the Cell processor.

8 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. Sutter, H., Larus, J.: Software and the concurrency revolution. Queue **3**(7) (2005) 54–62
2. Parkhurst, J., Darringer, J., Grundmann, B.: From single core to multi-core: preparing for a new exponential. In: Proc. ICCAD 2006, ACM Press (2006) 67–72
3. 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)
4. Chen, T., Raghavan, R., Dale, J., Iwata, E.: Cell Broadband Engine Architecture and its first implementation. IBM developerWorks (2005)
5. ClearSpeed: http://www.clearspeed.com/docs/resources/ClearSpeed_-Architecture_Whitepaper_Feb07v2.pdf (2005) CSX Processor Architecture.
6. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications. ICCC **4** (2002)
7. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9) (1987) 1235–1245
8. Gummaraju, J., Rosenblum, M.: Stream Programming on General-Purpose Processors. In: Proc. MICRO 38, Barcelona, Spain (November 2005)
9. ACOTES IST-034869: http://www.hitech-projects.com/euprojects/ACOTES/ Advanced Compiler Technologies for Embedded Streaming.
10. 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)
11. Carpenter, P., Rodenas, D., Martorell, X., Ramirez, A., Ayguadé, E.: A streaming machine description and programming model. Proc. SAMOS 2007
12. ACOTES: IST ACOTES Project Deliverable D2.2 Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version. (2008)
13. Stephens, R.: A survey of stream processing. Acta Informatica **34**(7) (1997) 491–541
14. Kahn, G.: The semantics of a simple language for parallel processing. Information Processing **74** (1974) 471–475
15. Girona, S., Labarta, J., Badia, R.: Validation of Dimemas communication model for MPI collective operations. Proc. EuroPVM/MPI (2000)
16. GNU Radio: http://www.gnu.org/software/gnuradio/
17. Ramsey, N., Davidson, J., Fernandez, M.: Design principles for machine-description languages. ACM Trans. Programming Languages and Systems (1998)
18. Labonte, F., Mattson, P., Thies, W., Buck, I., Kozyrakis, C., Horowitz, M.: The stream virtual machine. Proc. PACT 2004 267–277
19. Mattson, P., Thies, W., Hammond, L., Vahey, M.: Streaming virtual machine specification 1.0. Technical report, http://www.morphware.org (2004)
20. Mattson, P.: PCA Machine Model, 1.0. Technical report (2004)
21. Kienhuis, B.: Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools. Delft University of Technology, Amsterdam, The Netherlands (1999)
22. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. Proc. ASPLOS 2006 (2006) 151–162
23. Lundgren, W., Barnes, K., Steed, J.: Gedae: Auto Coding to a Virtual Machine. Proc. HPEC (2004)