Influences on Throughput and Latency in Stream Programs

Vu Thien Nga Nguyen
University of Hertfordshire
v.t.nguyen@herts.ac.uk

Raimund Kirner
University of Hertfordshire
r.kirner@herts.ac.uk

Abstract

Stream programming is a promising approach to execute programs on parallel hardware such as multi-core systems. It allows to reuse sequential code at component level and to extend such code with concurrency-handling at the communication level. In this paper we investigate in the performance of stream programs in terms of throughput and latency. We identify factors that affect these performance metrics and propose an efficient scheduling approach to obtain the maximal performance.

1. Introduction

With the current trend towards increasing number of execution units running in parallel, stream programming has been emerged by its elegant way of exposing useful types of paralleling. This programming paradigm separates communication from computation. This separation relieves the programmer of managing concurrent communication and synchronisation at the same time.

Because of this advantage, several research projects have introduced stream programming frameworks such as StreamIt [14], Brook [2] and S-Net [7]. However, it is intricate to understand the performance of stream programming. Possessing properties similar to communication networks, stream programs are usually evaluated in terms of latency and throughput. These two performance metrics depends not only on internal factors such as scheduling policies but also on external factors such as the arrival rate of data.

In this paper, we study the influences on the performance of in stream programs in terms of throughput and latency. The paper i) shows that different ranges of the data arrival rates give different influence on the performance; ii) analyses in which way the scheduling policy affects the performance; and iii) proposes a scheduler aiming to achieve the optimal performance of stream programs in multicore systems. The proposed scheduler consists of a static strategy targeting at the maximum throughput; and a heuristic dynamic strategy which observes the runtime behaviours and automatically tunes the policy to obtain the optimal latency while keeping the maximal throughput.

The paper is organised as follows. Section 2 provides basic backgrounds including stream programming paradigm and the stream execution model. Section 3 discusses latency and throughput in stream programming. Section 4 presents different ranges of input rate with different affects on latency and throughput. Section 5 investigates on the scheduler's influences on the latency and throughput. Based on these analysis, Section 6 proposes a scheduling strategy to achieve the optimal performance. Section 7 describes related work and Section 8 presents conclusions and planned future work.

FD-COMA 2013 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures, part of the 8th International Conference on High-Performance and Embedded Architectures and Compilers, Berlin, Germany, January 21-23, 2013
ACM SIGPLAN conference style.

2. Backgrounds

2.1 Stream Programming Paradigm

Stream programming is a paradigm that allows to express the parallelism by decoupling computations and communications [2, 7, 14]. In this model, a program is represented as a directed graph whose vertices are computation nodes and edges are communication channels called streams. For short, we from now use nodes instead of computation nodes.

Data is expressed as a flow of messages moving between nodes via streams. Streams connect nodes in different ways such as pipeline, parallel, feedback, etc.

A node receives messages from a set of streams, called input streams. The node then processes these messages to produce new messages and sends to a set of streams, called output streams. For one execution, a node may consume *n* messages and process to produce *m* messages. The value of *n* and *m* may change at the runtime, for example depending on the input data. The node can progress only when *n* required messages from its input streams are available.

Nodes that receives messages from outside of the stream programs are entry nodes. Similarly, nodes sending messages to outside of the stream programs are exit nodes. A stream program can have multiple entry nodes and multiple exit nodes. Messages coming from outside are called input messages and messages sending to outside are called output messages. Other messages (messages inside the stream program) are called intermediate message.

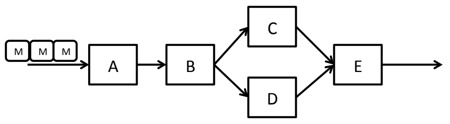

Figure 1. An example stream program

Figure 1 shows an example of a stream program. In this example, the computations are done on 5 nodes: *A, B, C, D, E. A* is an entry node and connected *B* in pipeline. *B* is followed by the parallel combination of *C* and *D*. This combination is then pipelined with an exit node *E*. A message produced by *B* can take either the route via *C* or *D*, i.e. can be processed either by *C* or *D*.

When a node consumes a message *X* (and possibly other messages) to produce a message *Y* (and possibly other messages), it is said that *X* derives to *Y*, or Y is derived from *X*. In this case, *X* is a predecessor of *Y* and *Y* is a successor of *X*. The predecessor - successor relation is inherited, i.e. if *X* is predecessor of *Y* and *Y* is predecessor of *Z* then *X* is also predecessor of *Z*.

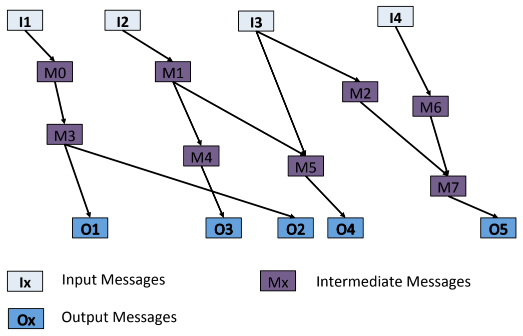

Figure 2. An example of message derivation

An input message I_i when processed by a stream program derives to multiple intermediate messages M_j before deriving to output messages O_k. Figure 2 shows an example of input messages deriving to output messages. In this example, input message I_2 derives to M_1, M_1 derives to M_4 and M_4 derives to output message O_3. M_1 (together with input message I_3) also derives to M_5 and then M_5 derives to O_4. An input is said to be **completed** when all of its successors no longer exist in the stream program. That is usually when all its output messages are produced.

2.2 Stream Execution Model

A typical framework for executing a stream program is shown in Figure 3. A stream program first is compiled into runtime objects managed by the Runtime System (*RTS*). These objects include streams and tasks. Each stream is represented as a FIFO buffer for storing messages. Each computation node is associated with a task. Typically each task receives messages from a set of *n* streams, called input streams. The task then processes messages before sending them to a set of *m* streams, called output streams.

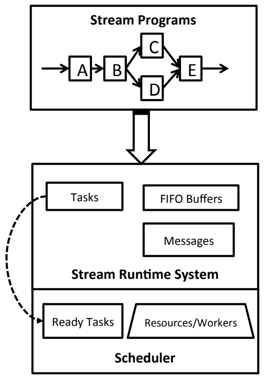

Figure 3. An Execution Model for Stream Programs

The RTS's duty is to assure the semantics of the stream program. In particular, an input message's successors have to be generated in order. In the example of Figure 2, input message I_2 is completed when it is first processed to produce message M_1. M_1 is then processed to produce M_4 and M_5. O_3 is produced by processing M_4 and O_4 is produced by processing M_5. There is no order between M_4 and M_5. To preserve this semantics, the RTS ensures that each task receives messages from the correct input streams and sends messages to the correct output streams.

Additionally, the RTS ensures that all obligatory messages have arrived before a task can make any progress. In the example in Figure 2, to produce M_5 a node has to wait until M_1 and I_3 have arrived. Another semantics is reflected in the order of messages within a stream. In most of the stream programming languages, messages are transferred within streams in the FIFO manner. This is guaranteed as the RTS uses FIFO buffers to represent streams.

The first and second semantic constraints decide whether a task is **ready** or not. A task is ready when it has all the required messages to make progress. In case of finite stream buffer sizes, a task when trying to write to a full stream can not proceed and therefore becomes not-ready.

Besides the RTS, a scheduler is required to distribute tasks to physical resources. In the context of the scheduler, each physical resource is addressed as a worker. A task is executable only if it is ready. The scheduler employs a policy to decide:

- a ready task to be executed
- a worker to execute the task
- how long the worker will execute the task

3. Throughput and Latency in Stream Programs

Stream programs are similar to communication networks in the sense that they transfer messages from one end to another via interconnected nodes. For these reasons, the performance of stream programs should be evaluated with similar metrics of communication networks, i.e. throughput and latency. However, unlike in communication networks, nodes in stream programs contain computations. These computations need to be executed on a mutual platform of physical resources (e.g. CPUs). This section discusses the influences on throughput and latency in stream programs.

3.1 Throughput

Similar to communication networks, **Throughput (Tp)** of stream programs is measured as the number of input messages are completed per time unit. Each node in a communication network is a physical resource with its own throughput. Queuing theory therefore can be used to analyse the throughput with some assumptions [12].

However, queuing theory is not applicable for stream programs as nodes perform their computation on a shared platform of physical resources. The throughput of stream programs highly depends on the scheduling policy.

3.2 Latency

Generally, latency is the delay experienced in the system, i.e. the delay to transfer one message from one entry point to an exit point. In stream programming, latency of an input message is the time interval from when the input message arrives to when it is completed.

Latency in stream programming may vary for different input messages for two reasons. First, each node may take different amounts of time to process different messages. Second, a message can take different route inside the stream programs. In the example shown in Figure 1, a message produced by *B* can pass either *C* or *D* which may take different time to process the message.

On a platform with limited physical resources, the latency of an input message also depends on the work load when the input message arrives and the scheduling policy which decides when the stream program starts to process the input message. If the scheduler decides to process the input message immediately, it may increase the latency of previous input messages as processing the new input will bind some physical resources. Otherwise, the input message will wait in the queue before getting into the stream program. Since the queuing time of an input message is affected by resource limitation and the previous input messages' successors still to be processed, it is should be separated from the processing time of the input message. We therefore propose different kinds of latency as follows.

- **Queuing Latency (QL)** of an input message is the time it waits before getting processed by the stream program. The queuing latency depends on when the scheduler decides to start processing a new input message; and how frequently new input messages arrive at the stream program.
- **Processing Latency (PL)** of an input message is the time interval from when it is taken into the program until it is completed . Processing the input message's successors may not be continuous for two reasons. One reason is that one of its successors when reaching a node needs to wait for other messages so that the node has sufficient messages to make progress. Another reason is that the scheduler decides to halt processing the input message's successors due to the lack of physical resources.
- **Overall Latency (L)** is the sum of queuing latency and processing latency. For the simplicity, from now on we use latency to refer to overall latency.

For communication networks in which each node has its own physical resource, the latency over the whole network can be reasoned from the latency over individual nodes. This is non-trivial for stream programs in which nodes share a mutual number of physical resources within the scheduler's control.

4. Input Rate affects on Throughput and Latency

For most of stream programs, input messages arrive with a rate, called **Input Rate (IR)**. The IR along with the scheduling policy are key factors controlling on the program's throughput and latency. In this section, we focus on the effects of the input rate alone on the throughput and latency.

With a specific platform of physical resources and a specific scheduling policy, Figure 4 shows a typical change in latency and throughput according to the input rate. There are two marks that divide the IR value into three ranges. Within each of these ranges, the IR affects the throughput and latency in different ways. For the simplicity, we assume that the stream program has exclusive usage of the physical resources, i.e., no other application is running at the same time on the platform.

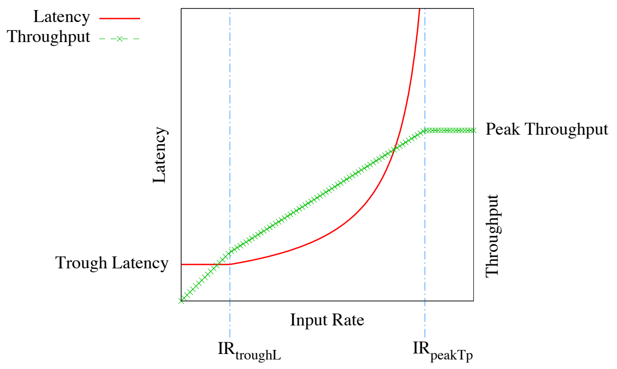

Figure 4. Latency and Throughput are affected by Input Rate

4.1 Idling Range

As explained in the previous section, the latency of an input message depends on the system load, i.e. how busy all the resources are. At the beginning, when the first input message arrives all the resources are free and immediately invoked to process the input message and its successors. The smallest latency is therefore achieved for the input message.

When the IR value is small enough that all the physical resources are always free whenever an input message appears, all input messages have the smallest latency value. Thus the average latency is also smallest and this value is called **Trough Latency**. The throughput in this case equal to the IR.

There might be a case where the average latency is decreased when the IR increased. That is when a node requires successors from two input messages to make progress. That means the processing the former input message can not continue until the later input message arrives. In this case, the latency is decreased when the IR gets higher but this is a temporary improvement. When the IR reaches a value that the faster arrival of input messages do not help nodes make progress earlier, the latency stops decreasing and reaches the stable value. The trough latency in this case is this stable latency value. Throughput in this case is smaller than but still proportional to the IR.

The highest IR value at which the system still keep the trough latency is called $IR_{troughL}$. Whenever the actual IR is smaller than this value, there are time intervals where the physical resources are idling and waiting for input messages. For this reason, the period from 0 to $IR_{troughL}$ is called **Idling Range**.

4.2 Working Range

When the IR exceeds the $IR_{troughL}$ value it can no longer be guaranteed that the system exclusively processes one message at a time. Instead the processing of messages tends to get overlapped. When an input message arrives, the physical resources are partly or completely busy for processing previous input messages' successors. The latency of an input message gets higher either because the input message has to wait in the queue or because the resources are not fully used to process the input message and its successors but to contribute to process others. For this reason, the average latency is therefore higher than the trough latency.

While the latency gets worse, the throughput becomes better. The high input arrival rate gives the stream program the chance to consume input messages faster. More input messages are processed and therefore the throughput gets higher. However when the IR is high enough to saturate the system, the throughput stops increasing. The highest throughput value that the system can achieve is called **Peak Throughput**. The IR value at which this happens is called IR_{peakTP}.

We call the IR range from $IR_{troughL}$ to IR_{peakTP} as **Working Range** as with the IR in this range the platform does not have any moment idling but working constantly.

4.3 Overload Range

With an IR higher than IR_{peakTP}, the throughput does not exceed the peak value. The system cannot consume input messages as fast as input rate. The platform gets saturated after an initial period. After this initial period input messages have to wait before getting processed. The later the input message comes, the more input messages are already waiting in the queue before it. The queuing latency therefore rises up and eventually becomes infinity. As the result, the average latency is infinity.

In this circumstance, the platform is saturated and cannot keep up with the requested input rate. Hence the range from IR_{peakTP} up is called **Overload Range**.

5. Scheduler affects on Throughput and Latency

As presented in Section 2.2, the scheduler manages a set of workers and a set of tasks. One worker is associated to a physical resource of the platform and one task is associated to a computation node of the stream program. The scheduler decides when and how long a worker performs a task. This decision has a strong influence in throughput and latency of the stream program.

5.1 Throughput

Consider a stream program deployed on a platform of homogeneous physical resources. Let **N** be the number of workers. To process **M** input messages, it takes the stream program the time period **P**. The total time that N workers have spent over that period is:

$$T = N \times P \tag{1}$$

Within the period P, N workers spend time **C** on computations and time **W** on idling. The sum of computation time and the idling time must is equal to the total time that N workers have:

$$T = C + W \tag{2}$$

From Equation 1 and Equation 2, we have:

$$T = N \times P = C + W \tag{3}$$

We have the throughput of the stream program is equal to the number of input messages over the time period:

$$Tp = \frac{M}{P} \tag{4}$$

From Equation 3, we have:

$$Tp = \frac{M \times N}{C + W} \tag{5}$$

As C is the required computation time for processing M input messages, it varies on the implementation and the underlying hardware. These factors are not under the sphere of control of the scheduler. We thus can assume C is constant in the scheduler's context. Throughput then is inverse-proportional to the idling time.

There are two situations that a worker gets idling. One situation is that there is no ready task because the input rate is too low. As explained in Section 4, when input rate is in idling range the throughput is as low as the input rate. The other situation happens when the input rate falls in two other ranges. There are ready tasks and also free workers at the same time. This happens usually due to the mapping strategy of the scheduler.

The scheduling policy determines the idling time and therefore controls the throughput value. If the scheduling policy helps to achieve higher throughput during the working range, the stream program can consume more input messages and therefore can deal with higher input rate. As a result, it expands the working range by increasing IR_{peakTP} value, i.e. IR needs a higher value to saturate the system. The peak throughput also is increased.

5.2 Processing Latency

As described in Section 2.2, latency is the sum of queuing latency and processing latency. The queuing latency of an input message reflects the time the input message has to wait before getting processed. Therefore the queuing latency depends on the speed at which the stream program consumes input messages. That means the queuing latency depends on the throughput. In this section, we focus on the processing latency which is directly controlled by the scheduler.

The processing latency of an input message is the time it takes to make the input message completed, i.e. the period from when it is consumed until all its output messages are produced. Within this period, the input message itself and all its successors are processed. If these messages are processed continuously, the latency is minimal. This is decided by the RTS and the scheduler.

As explained in Section 2.2, to preserve the stream program's semantics the RTS decides when a task is ready. The scheduler affects processing an input messages and its successors by deciding the order of executing ready tasks. In particular, the scheduler determines when a ready task is executed and for how long.

The scheduling policy affects the latency in two first ranges of the input rate as the latency rises up to infinity when the input rate falls in the overload range. Within these two ranges, while deciding the executing order of ready tasks, the scheduler effectively has an influence on the latency and also the trough latency. If the scheduler helps to reduce the trough latency, the stream programs can process input messages faster. That means the workers get idling more often in the idling range. Therefore the input rate needs a higher value to get in the working range, i.e., the $IR_{troughL}$ is reduced.

6. A Proposed Scheduler for Maximising Performance

In this section, we propose a scheduler aiming to maximise the throughput and minimise the processing latency. We focus only on the processing latency because the queuing latency depends on both the input rate and the throughput. When the throughput is maximised, the queuing latency is also minimised as messages are taken to by the program with the highest rate. Hence while maximising the throughput, the proposed scheduler effectively reducing the queuing latency.

Generally, all ready tasks are stored in a central task queue as in Figure 5. Each task in the task queue has a priority which is calculated dynamically during the runtime. A worker requests for a task from the task queue when it gets idling. The task with the highest priority at that time will be chosen and assigned to the requesting worker. The worker executes the task until when either the task becomes not ready or it is asked to yield.

The following of this section provides more detail about the scheduling policy and explains how it steers the performance.

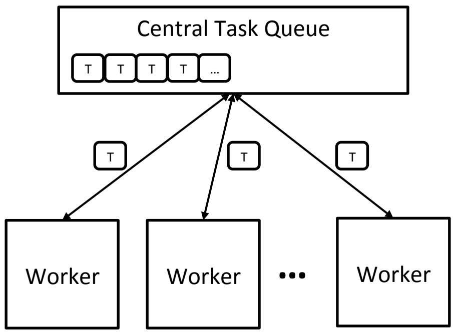

Figure 5. Latency of an individual processing component

6.1 Maximising Throughput

As shown in Section 5.1, the throughput is maximal when the idling time is minimal assuming that the individual processing times of messages are independent from the scheduling decisions. In the proposed scheduler, each worker requests for a task whenever it is idling. If there is a ready task in the central task queue, it will be assigned to the worker immediately. By doing this, each worker is kept busy as much as possible. A worker has to wait only when there is no ready task in the central task queue because input messages have not arrived. The input rate is not under the sphere of control of the scheduler.

Therefore by using a central task queue, the scheduler minimises the idling time. However this also raises a risk of overhead.

In particular, the central task queue can become a bottleneck when too many workers request at the same time.

For the implementation, an efficient data structure should be used to implement the central task queue. Also it is suggested to dedicate a worker as a conductor which controls the central task queue. In this case, a sufficient communication protocol between the conductor and the workers is necessary in order to reduce the delay from when a worker sends a task request till receiving one.

6.2 Minimising the processing latency

As presented in Section 5.2, how an input message and its successors are processed has a strong affect on the processing latency. As the RTS applies some constraints to preserve the stream program's semantics, the scheduler has control only on ready tasks. When a worker becomes free, the scheduler can make two decisions: which ready task is executed and for how long. In general, a message can be routed to streams dynamically during the runtime as explained in Section 2.1. Thus it is difficult to make these decisions at the compile time.

Our proposed scheduler uses a heuristic to compute the task priority based on the state of input and output streams of the task during the runtime. Let **I** be the number of messages in the task's input streams and **O** be the number of messages in the task's output streams. The **I** value of entry tasks are always zero and the **O** value of exit tasks are always zero. The task's priority is calculated as follows.

$$P = \frac{I+1}{(O+1)(I+O+1)} \tag{6}$$

Equation 6 implies that a task's priority has a positive correlation with its **I** value and a negative correlation with its **O** value. When a task consumes *n* messages and produces *m* messages, it lowers its **I** value by *n*, raises its **O** value by *m* and also raises the succeeding task's **I** value by *m*. That means, the task's priority is reduced while the following task's priority is increased. This increases the chance of scheduling the succeeding task and therefore the message's successor will be processed earlier. This therefore lowers the latency of the input messages that are the message's predecessors.

After a task has processed one message, the priorities are reevaluated. The scheduler then pick the highest-priority task to assign to a worker. By doing this, the priority is finely updated and therefore the heuristic is more accurate. However, this also raises the overhead for switching tasks and calculating the priority. For the implementation, a task when scheduled should process a limited but large number of messages in order to compensate the cost of loading the task.

7. Related Work

Stream programming has seen a revival over the last years, due to the main-stream multi-core computing. Gordon et al. have identified three forms of parallelism to be exploited in stream programming: task parallelism, data parallelism, and pipeline parallelism [5]. The performance of a stream program can be optimised by task allocation, resource mapping, and scheduling policy.

One of the prominent stream programming approaches is StreamIt [13], which is conceptually based on synchronous data flow (SDF). In the initial version of the language, all flow rates have to be static, even though this can be avoided in later versions of the language.

There are many approaches of performance-optimised scheduling of StreamIt programs. All encountered approaches assume a static structure of the streaming graph, as StreamIt currently requires at compilation time fixed message routes, i.e., the connection of network nodes is determined at compile time. Karczmarek et al. introduced the concept of *phase scheduling* for StreamIt programs, exploiting the static nature of the streaming graph [10]. The goal of phase scheduling is to address the trade-off between code size and data size [1] without considering the actual performance of the program. The concept of phase scheduling is to construct the stream processing as a repetition of phases. Based on this construction, one strategy is to deploy each phase on one resource [15]. The phases are executed in parallel to take advantage of data parallelism. Another scheduling strategy is to execute on each resource a separate phase [6]. In this case relatively long waiting times can happen if synchronisation between different phases happens.

Farhad et al. recently used efficient approximative algorithms with bounded inaccuracy to schedule StreamIt programs [3]. In a different approach they built a model of communication costs to improve performance-guided optimisations [4].

Stream programming has been also introduced for hard real-time computing. Giotto is a coordination language where nodes are scheduled in a strict static time-triggered way [9]. Message processing is synchronous, i.e., a constant number of messages is consumed and produced per invocation by a node. With such a static language as Giotto is, one actually has to solve the inverse problem to that ours: with Giotto one starts with the performance values as specification, and has to produce a schedule in order to fulfill the required performance constraints.

With our discussion of performance of stream programming we include also more dynamic network structures of stream programs, as this, for example, is the case with the coordination language S-Net [8]. Streaming programs in S-Net can have a dynamic network structure, which requires the use of a dynamic scheduling algorithm. In S-Net there are no constraints over the flow rate of input. Thus, actual throughput and latency depend on the concrete message values and timings of the input stream. CAL (constraint aggregation language) has been developed to provide contracts of the behaviour of network nodes [11].

8. Conclusion & Future Work

In this paper we have discussed the performance measures throughput and latency for stream programming. We have contrasted them to the established calculation of throughput and latency in the domain of communication networks. Even though there are many similarities, the subtle but important difference is that communication networks typically have a separate resource for each network node, while in stream programming typically multiple nodes have to share each processing resource. Thus the behaviour of stream programs get an additional dependency, the resource allocation policy, often described by the allocation, mapping, and scheduling of tasks on resources.

Most discussions of performance of stream programs have been based on the assumption of the special case of static network structures, synchronous communication, and fixed data rates. Important insights are the formation of schedule-dependent trough-latency and peak throughput. Within this paper we discuss more generally the influence of scheduling on throughput and latency of stream programs. In addition, we outline a scheduling approach for such dynamic streaming programs.

References

[1] S. S. Bhattacharyya. Optimization trade-offs in the synthesis of software for embedded dsp. In *IN CASES*, 1999.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, Aug. 2004.

[3] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Orchestration by approximation: mapping stream programs onto multicore architec-

tures. In *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, pages 357–368, New Port Beach, CA, USA, Mar. 2011.

[4] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Profile-guided deployment of stream programs on multicores. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'12)*, pages 79–88, Beijing, China, June 2012.

[5] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, 2006.

[6] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. *SIGOPS Oper. Syst. Rev.*, 36(5):291–303, Oct. 2002.

[7] C. Grelck, S.-B. Scholz, and A. Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[8] C. Grelck, S.-B. Scholz, and A. Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[9] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed Giotto. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM Press, 2005.

[10] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. *SIGPLAN Not.*, 38(7):103–112, June 2003.

[11] F. Penczek, R. Kirner, R. Poss, C. Grelck, and A. Shafarenko. An infrastructure for multi-level optimisation through property annotation and aggregation. In *Proc. 4th Int. Workshop on Non-functional System Properties in Domain Specific Modeling Languages (NF-PinDSML'12)*, Innsbruck, Austria, Oct. 2012.

[12] T. G. Robertazzi. *Computer Networks and Systems: Queueing Theory and Performance Evaluation*. Springer-Verlag, 1994.

[13] B. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. 11th International Conference on Compiler Construction (CC'02)*, pages 179–196, London, UK, 2002. Springer Verlag.

[14] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[15] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36(2):18–27, May 2008.