Orchestration by Approximation

Mapping Stream Programs onto Multicore Architectures

S. M. Farhad[1,3] * Yousun Ko[2] § Bernd Burgstaller[2] § Bernhard Scholz[1] ‡

[1]The University of Sydney
Sydney, Australia
{smfarhad, scholz}@it.usyd.edu.au

[2]Yonsei University
Seoul, Korea
{yousun.ko, bburg}@cs.yonsei.ac.kr

[3]NICTA
Locked Bag 9013
Alexandria NSW 1435, Australia

Abstract

We present a novel 2-approximation algorithm for deploying stream graphs on multicore computers and a stream graph transformation that eliminates bottlenecks. The key technical insight is a data rate transfer model that enables the computation of a "closed form", i.e., the data rate transfer function of an actor depending on the arrival rate of the stream program. A combinatorial optimization problem uses the closed form to maximize the throughput of the stream program. Although the problem is inherently NP-hard, we present an efficient and effective 2-approximation algorithm that provides a lower bound on the quality of the solution. We introduce a transformation that uses the closed form to identify and eliminate bottlenecks.

We show experimentally that state-of-the art integer linear programming approaches for orchestrating stream graphs are (1) intractable or at least impractical for larger stream graphs and larger number of processors and (2) our 2-approximation algorithm is highly efficient and its results are close to the optimal solution for a standard set of StreamIt benchmark programs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors Compilers

General Terms Languages, Algorithms, Performance

Keywords StreamIt, multicore, stream programming

1. Introduction

Multicore processors have become the industry standard because the von Neumann computing model [2] of uniprocessor architectures has ceased to scale effectively. Example systems include the IBM Cell BE processor with 9 cores [13], the IBM Power7 with 8 cores [14], the Sun UltraSPARC T3 with 16 cores [26], NVIDIA's GeForce GTX 480 that provides 15 streaming processors each with 32 CUDA cores [23], and the Cisco CRS-1 and CRS-3 routers that utilize Tensilica's Metro and QuantumFlow processors with 188 and 40 4-threaded cores [8, 10]. Intel and AMD are already producing x86 systems with 8 and 12 cores respectively.

Sequential programming languages are ill-suited to multicore architectures, because they assume a single instruction stream and a single, uniform memory system. Identifying coarse-grained parallelism for efficient multicore execution is left to the programmer and the compiler. Multicore architectures vary in their communication primitives and in the number and capabilities of their CPU cores. Sequential programming languages provide insufficient parallel hardware abstractions, which greatly hampers performance and portability of software on multicore architectures.

The stream programming paradigm has turned out to be an effective approach for programming multicore architectures. Stream programming languages facilitate application domains characterized by regular sequences of data, such as digital signal processing, audio, video, graphics and networking. Examples of stream programming languages and language extensions include StreamIt [30], Brook [5], Baker [7], StreamFlex [27], Cg [21] and SPUR [36].

In our work we use StreamIt, which represents a program as a set of actors that interact through FIFO data channels (see Figure 1). During program execution, actors are invoked repeatedly in a periodic schedule [3]. Because each actor has a separate program counter and an independent address space, dependencies between actors are made explicit by the data channels. StreamIt employs the synchronous data-flow model (SDF, [19]), which requires the number of data items produced and consumed by each actor to be known a-priori (with StreamIt this information is already specified in the source code). Compilers can then leverage the static dependence information to orchestrate parallel execution.

The major optimization objective with stream programs is to maximize data throughput. Although stream programs contain an abundance of parallelism, obtaining an efficient mapping onto parallel architectures is nevertheless a challenging problem. The gains obtained from parallel execution are easily overshadowed by the costs of communication and synchronization. Resource limitations of the system must be incorporated into the mapping process to avoid stalls and load-imbalances. Resource limitations of shared-memory multiprocessors include processing capabilities and memory bandwidth. A compiler must take into account the structural properties of actors and actor dependencies to resolve *bottlenecks* that constrain the throughput of a stream program. Stream pro-

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

§ Research partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0005234), and the OKAWA Foundation Research Grant (2009).

‡ Research partially supported by the Australian Research Council through ARC DP grant DP1096445.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

grams contain task, data and pipeline parallelism, and it is critical for a compiler to leverage a synergistic combination while avoiding the hazards associated with each.

Recently, significant research effort has been spent for orchestrating stream graphs on multicore architectures [6, 7, 11, 12, 16, 18, 28, 31, 32, 34, 35]. It has been shown in [18] how the unfolding and partitioning of actors onto a multicore architecture can be formulated as an Integer Linear Programming (ILP) problem. This ILP formulation does not take into account communication costs between actors. The ILP solution-space for unfolding and partitioning of actors grows exponentially in the number of processors, actors and data channels. Our experiments with existing methods using CPLEX indicate that ILP formulations become intractable or at least impractical already for three or four cores (see also Table 2 where $P\#$ is the number of cores and t_{ILP} is the CPLEX solve time). An approach that approximates the optimal solution of the ILP formulation seems therefore desirable. In contrast to previous work, our work uses a data transfer model that sets in relation the input and output data rates of actors. In our model, we maximize the arrival rate whereas previous work optimizes makespan, i.e., the runtime of the longest running processor, which is an indirect measure of arrival rate and throughput.

This paper makes the following contributions:

- a novel data rate transfer model for stream programs that introduces data rate functions for actors that depend on the arrival rate of the stream program,
- a quantitative analysis to identify bottlenecks and "hot regions" in stream programs,
- a transformation for stream programs that is region based for eliminating bottlenecks and, hence, reduces communication costs, and
- a 2-approximation algorithm that maps actors onto a parallel system by maximizing the arrival rate of the stream program.

Experiments show that our approximation approach solves the orchestration problem instantly (i.e., within milliseconds), with a solution that achieves 95–100% of the throughput obtained by an ILP formulation.

The paper is organized as follows: in Section 2 we motivate problems and optimization opportunities with stream program orchestration on multicore architectures. Section 3 introduces our analytical performance model. In Section 4 we discuss the elimination of bottlenecks. Section 5 contains the actor allocation problem (AAP) and our approximation thereof. In Section 6 we discuss the experimental results. We survey related work in Section 7 and draw our conclusions in Section 8. We provide a glossary in Appendix B.

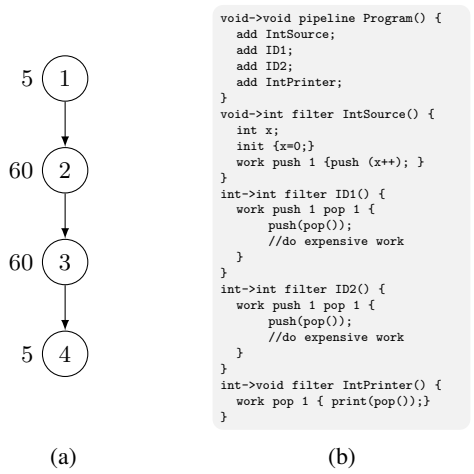

Figure 1. (a) Stream graph (b) Corresponding StreamIt source code.

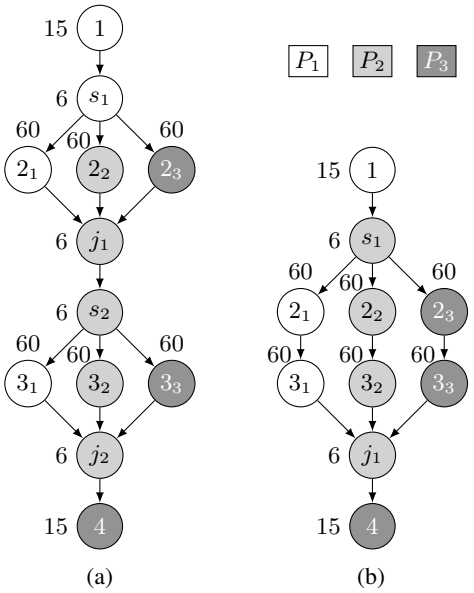

Figure 2. (a) Integrated bottleneck resolution and actor allocation approach of [18] (b) hot region bottleneck resolution.

2. Orchestration

The motivation for our approximation algorithm for actor unfolding and partitioning are scalability problems with the existing, optimal ILP formulation. We observed impractical solve times already for a program consisting of 26 actors on 4 processors (see Section 6). However, a recent survey ([29]) on the characteristics of stream programs reports increasing program sizes of up to 2868 actors per program. In conjunction with the semiconductor industry's projected increases in core counts an approach that approximates the optimal solution of the ILP formulation is important.

We motivate our approach using a simple StreamIt program consisting of four actors depicted in Figure 1. Let us assume that actors 1 and 4 are *stateful* and actors 2 and 3 are *stateless*. Only stateless actors may be duplicated to exploit data-parallelism, because a stateful actor propagates local state information between actor invocations. As a result, every invocation of a stateful actor depends on a previous invocation, and hence must not be duplicated.

In Figure 1(a) actor execution times are depicted next to stream graph nodes. Actors 2 and 3 consume more processing time (60 time units each) than actors 1 and 4. The stream program takes 130 time units when executed on a uniprocessor with a single invocation for all actors. Note that actors 2 and 3 will cause a bottleneck if the program is executed on a multicore system due to the high execution costs. We refer to these actors as *hot actors*. To improve the throughput of stream programs on multi-core systems, hot actors are duplicated to spread the load evenly among processors. In this example the hot actors form a *hot region* since they are adjacent to each other.

An optimal allocation is presented in Figure 2(a) following the method described in [18]. Actors 2 and 3 are duplicated two times,

totaling in three versions each. Splitters s_1, s_2 and joiners j_1, j_2 are added to distribute data among the three versions of actors 2 and 3. Nodes 2_1, 2_2, 2_3 constitute the three versions of actor 2 and nodes 3_1, 3_2, 3_3 constitute the three versions of actor 3. Node colors indicate the processor allocation. In this example, the workloads assigned to processors P_1, P_2 and P_3 are 141, 138 and 135 time units, respectively, if we account a workload of 6 time units for splitters and joiners. We calculate speedups by dividing the weighted uniprocessor execution-time by the load on the maximally loaded processor. The weight of the uniprocessor execution-time is 3, since the transformed program represents three invocations of the original program. The maximally loaded processor is processor P_1 with 141 time units. Hence, the obtained speedup for the stream graph in Figure 2(a) is $\frac{3\times130}{141} = 2.77$.

In our approach, the solution in Figure 2(a) is further improved by combining actors 2 and 3 into a hot region. The result of duplicating the hot region as a whole is depicted in Figure 2(b). Because the overhead of joiner j_1 and splitter s_2 is avoided, the speedup increases to $\frac{3\times130}{135} = 2.89$. Our approach separates hot actor identification from actor allocation, which enables the duplication of hot regions. In this way our method can achieve improved bottleneck elimination over existing techniques.

Essential for the hot region transformation is a data transfer model for actors that is able to identify hot actors. The data transfer model permits the computation of closed forms that express the output data rate of an actor as a function that depends solely on the arrival rate of the whole program, i.e., the closed form incorporates all immediate and intermediate data transfer dependencies of an actor. With the closed form the allocation problem simplifies as well since the topology of the graph does not need to be considered. An overview of the proposed approach to statically optimize the throughput of a stream program is illustrated in Figure 3.

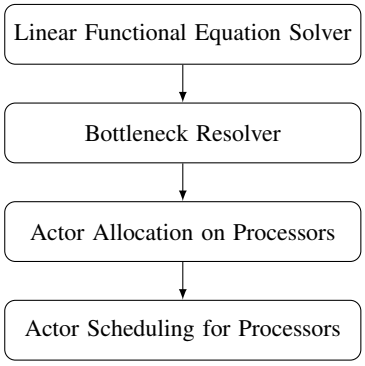

Figure 3. Framework overview.

Our static optimization consists of four components: a linear functional equation solver, a bottleneck resolver, actor allocation and actor scheduling. An analytical performance model is created for a stream program that employs a linear functional equation solver to compute the closed form of data transfers rates in the stream program. After computing closed forms, we perform a quantitative analysis to detect bottlenecks and duplicate hot regions. The remaining steps are the mapping of actors to processors via an approximation algorithm and computing the schedule using standard techniques.

3. Data Transfer Model

The dataflow model [9] represents a program as a *stream graph* $G(V, E)$ whose vertices $V = \{1, \ldots, n\}$ are called *actors* and whose edges $E \subseteq V \times V$ are called *channels*. A channel $(i, j) \in E$ queues data elements called *tokens* which are passed from the output of computation i to the input of computation j. Synchronous dataflow (SDF, [3, 19]) restricts the dataflow model by fixing the number of consumed tokens denoted by c_i and the number of produced tokens denoted by p_i of an actor i at compile time. StreamIt [30] employs the SDF model except for the following differences: (1) StreamIt has a non-consuming read (`peek`) operation from the input channel similar to the computation model introduced in [17], (2) the number of tokens consumed and produced are specified for actors rather than along data channels, and (3) stream graphs in StreamIt are "structured", i.e., they are composites of filters, pipelines, split/joins, and feedback loops.

There is a cornucopia of scheduling work for synchronous dataflow [6, 11, 16, 18, 19, 24, 31, 32] ensuring that a schedule requires finite resources for storing tokens along channels, and that actors do not dead-lock whilst executing the schedule. A *periodic static* schedule [19] consists of a finite sequence of actor invocations; the periodic schedule is computed at compile time, invokes each actor of the stream graph at least once, and produces no net change in the system state, i.e., the number of tokens on each edge is the same before and after executing the schedule. Hence, a periodic schedule can be executed ad-infinitum without exhausting memory, and we refer to the state before and after the execution of a periodic schedule as the *steady-state*.

A periodic schedule has a positive integer vector $\mathbf{q} \in \mathbb{N}^n$ called *repetition vector* [19] whose elements correspond to actors in the stream graph. Element q_i for all $i \in \{1, \ldots, n\}$ is equal to the number of occurrences of actor i in the periodic schedule. Because every actor needs to be invoked at least once in the schedule, q_i is greater than or equal to 1.

A parallel periodic schedule distributes the invocation of actors on a parallel system and has a finite sequence of actor invocations for each processor. Synchronization between processors is necessary to wait for the longest running processor before executing the next iteration of the periodic schedule. We refer to the time duration of the longest running processor as *make span* Π, which affects the performance (i.e., throughput) of the stream graph.

3.1 Performance of a Periodic Static Schedule

The performance of a periodic schedule may be determined by the number of input tokens it can process in time duration Π. Without loss of generality we assume there is a unique actor that solely reads the input of the program and receives no input from other actors in the stream graph. We refer to this actor as the *start node* and assign it the ordinal number 1.

The *arrival rate* z of a periodic schedule measures the number of input tokens processed in time duration Π and is obtained by

$$z = \frac{\iota_1 q_1}{\Pi}, \tag{1}$$

where ι_1 is the number of bytes[1] read from the program input for a single invocation of the start node. Note that ι_i for all $i \in \{1, \ldots, n\}$, is the product of the size of an input token in bytes times the number of tokens c_i consumed by an invocation of actor i. The *output data rate* of actor i is defined by

$$\xi_i = \frac{\omega_i q_i}{\Pi}, \quad \text{for all } i \in \{1, \ldots, n\}, \tag{2}$$

where ω_i represents the number of bytes produced by a single invocation of actor i. Parameter ω_i is the product of the size of an output token in bytes and the number of tokens p_i produced by actor i for a single invocation.

An alternative measure of performance for stream programs is the notion of *throughput*. Throughput of a stream program may be

[1] SDF semantics implies static push and pop rates of actors.

defined as the sum of output data rates of its actors, i.e., $\sum_{i=1}^{n} \xi_i$. In Section 5, Eq. (14), we show that with an SDF model although arrival rate and throughput are different optimization objectives, the solutions are identical.

In this work, we seek a parallel periodic schedule that maximizes the arrival rate by placing actors on processors and transforming the stream graph. Our approach relies on a performance model of stream graphs that is able to compute the output data rate of an actor as a linear function of the arrival rate, independent of a concrete periodic schedule and the make span Π.

3.2 Model

Actors in the stream graph consume and produce a fixed number of tokens for a single invocation in the schedule. Under the assumption that we know the input data rate of an actor, we can compute the output data rate using *data rate transfer functions* $\Phi_i : \mathbb{R}^+ \rightarrow \mathbb{R}^+$, which are defined as $x \mapsto \alpha_i x$, where α_i is the ratio between the number of bytes produced and consumed for a single invocation of actor i, i.e., $\alpha_i = \frac{\omega_i}{\iota_i}$.

For example, all channels in Figure 1 are of type integer. The push and pop rates of the second actor are 1. If we assume a 4-byte integer representation, we get $\iota_2 = 4{\cdot}1$ and $\omega_2 = 4{\cdot}1$, i.e., invoking the second actor once consumes 4 bytes from the input channel and produces 4 bytes on the output channel. Hence, the parameter of the data transfer function is 1, i.e., $\alpha_2 = \frac{\omega_2}{\iota_2} = 1$, which implies that the input data rate of the actor is equal to the output data rate.

An actor may have more than one outgoing channel, and SDF semantics allows a fixed ratio for splitting the output rate of an actor i among its outgoing channels. In our model, the data rate of channel $(i, j) \in E$ is given by term $w_{ij}\Phi_i(x)$, where weight w_{ij} denotes the fraction of the output data rate of actor i that is diverted from actor i to channel (i, j). If w_{ij} is zero, it denotes that there is no dataflow between actor i and actor j. To conserve dataflow in a stream program, it must hold that $\sum_{j=1}^{n} w_{ij} = 1$, for all $i \in \{1, \ldots, n\}$.

For example, the weights of the channels in Figure 2(b) are all one except for edges $(s_1, 2_1)$, $(s_1, 2_2)$, and $(s_1, 2_3)$, whose weights are $\frac{1}{3}$.

The input data rate of an actor i is the sum over all data rates of the incoming channels of i, i.e., $\sum_{j=1}^{n} w_{ji}x_j$, where x_j is the output data rate of actor j. We obtain *data rate equations* by setting the data rate transfer functions of an actor in relation to its input and output data rates.

DEFINITION 1. *The data rate equations of a stream graph are defined as*

$$x_1 = \Phi_1(z) \quad \textit{and}$$
$$x_i = \Phi_i\left(\sum_{j=1}^{n} w_{ji}x_j\right), \qquad \textit{for all } i \in \{2, \ldots, n\}, \tag{3}$$

where the output data rate for actor i is $x_i \in \mathbb{R}^+$, and z is the arrival rate.

LEMMA 1. *For a given periodic schedule, the output data rate ξ is a solution of the data rate equations.*

PROOF 1. *To preserve the steady-state of a periodic schedule, the number of consumed tokens $c_i q_i$ of an actor i must be equal to the number of received tokens $\sum_{j=1}^{n} w_{ji}p_j q_j$ for all $i \in \{2, \ldots, n\}$. Hence, the following equations hold for a periodic schedule:*

$$c_i q_i = \sum_{j=1}^{n} w_{ji}p_j q_j, \quad \textit{for all } i \in \{2, \ldots, n\}. \tag{4}$$

The relationship between q_i and ξ_i is established in Eq. (2) and is transformed to $q_i = \frac{\Pi}{\omega_i}\xi_i$. We rewrite the data rate equations of Eq. (4) to Eq. (3) for all $i \in \{2, \ldots, n\}$ as follows,

$$\tfrac{c_i\Pi}{\omega_i}\xi_i = \textstyle\sum_{j=1}^{n} \tfrac{w_{ji}p_j\Pi}{\omega_j}\xi_j \Rightarrow \xi_i = \tfrac{\omega_i}{c_i}\textstyle\sum_{j=1}^{n} \tfrac{w_{ji}p_j}{\omega_j}\xi_j \Rightarrow$$
$$\xi_i = \alpha_i \textstyle\sum_{j=1}^{n} w_{ji}\xi_j \Rightarrow \xi_i = \Phi_i\left(\textstyle\sum_{j=1}^{n} w_{ji}\xi_j\right)$$

since the ratio $\frac{\omega_j}{p_j}$ is the byte size of an output token of actor j, and it is equal among all predecessors j of actor i, the inverse $\frac{p_j}{\omega_j}$ can be moved out of the sum reducing the term to α_i. Substituting Eq. (1) in Eq. (2) for the start node by eliminating q_1 gives the equation $\xi_1 = \Phi_1(z)$ and the lemma follows.

3.3 Closed Form

A *closed-form* $\Phi_i^* : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ of actor i is a data rate transfer function that solely depends on arrival rate z of the dataflow program, i.e., $z \mapsto \alpha_i' z$. With the existence of a closed form, the output rate of an actor can be determined by knowing the arrival rate z. In this work, the closed form is fundamental for placing actors on processors and eliminating bottlenecks.

The solution of the simultaneous equation system in Eq. (3) is a set of linear functions that depend on z which can be solved by employing standard linear algebra. We introduce a new matrix $\mathcal{A}$ with elements $a_{ik} = \alpha_i w_{ki}$ and vector $\mathbf{b}$ with elements $b_1 = \alpha_1$, and $b_i = 0$ for all $i \in \{2, \ldots, n\}$.

LEMMA 2. *The closed form of stream graph G is*

$$\alpha' = (\mathcal{I} - \mathcal{A})^{-1}\boldsymbol{b} \tag{5}$$

where $\mathcal{I}$ is the identity matrix.

PROOF 2. *We rewrite the data rate equations of Eq. (3) as*

$$\boldsymbol{x} = \mathcal{A}\boldsymbol{x} + z\boldsymbol{b} \tag{6}$$

which is reduced to $\mathcal{I}\boldsymbol{x} = \mathcal{A}\boldsymbol{x} + z\boldsymbol{b} \Rightarrow (\mathcal{I} - \mathcal{A})\boldsymbol{x} = z\boldsymbol{b} \Rightarrow \boldsymbol{x} = z(\mathcal{I} - \mathcal{A})^{-1}\boldsymbol{b}$, and the lemma follows.

For large numbers of actors, the computation of the inverse of matrix $\mathcal{A}$ is expensive. An alternate method that computes the closed form more efficiently is required for practical purposes.

LEMMA 3. *For a program that has a periodic schedule, matrix $(\mathcal{I} - \mathcal{A})$ is nonsingular.*

Due to space limitations we only sketch the proof here. The proof in Lemma 1 establishes a connection between the data rate equations and the equilibrium of consumed and produced tokens of actors. In [19] the equilibrium was expressed in form of a topological matrix mapping the repetition vector to a null vector, and it was shown that if the stream graph is connected, the rank of the topology matrix is $n - 1$. This result can be used to argue that the matrix $(\mathcal{I} - \mathcal{A})$ is nonsingular.

LEMMA 4. *Given the repetition vector $\boldsymbol{q}$ of a stream program, the parameter α_i' of the closed form is given by*

$$\alpha_i' = \frac{\omega_i q_i}{\iota_1 q_1} \tag{7}$$

PROOF 3. *We set in relationship the data output rate of Eq. (1) and the closed form. We substitute the arrival rate z by the right-hand side of Eq. (2), i.e., α_i' as $\frac{\xi_i}{z}$ that further reduces to*

$$\xi_i = \alpha_i' z \Rightarrow \frac{\omega_i q_i}{\Pi} = \alpha_i' \frac{\iota_i q_1}{\Pi} \Rightarrow \alpha_i' = \frac{\omega_i q_i}{\iota_1 q_1} \tag{8}$$

By Lemma 4, the parameter α' of the closed form may be computed by knowing $\mathbf{q}$, which is a property of the stream graph [19].

Fast solvers for computing the repetition vector of a stream graph are known to compute solutions in linear time complexity for unstructured graphs [3] and structured graphs [15]. Computing the parameter α' by employing linear equation solvers exhibits a higher computational complexity though fast equation solvers as used for data flow analysis [25] might deliver sufficient performance in practice.

For example, the second and third actor in Fig. 1 have data transfer functions whose α-parameters are 1, i.e., the input data rate is equal to the output data rate. As stated in the previous sub-section, the output rate of an actor is expressed as a linear functional equation system. Consider the third actor. The equation of the third actor is $x_3 = \Phi_3(x_2)$, which simplifies to $x_3 = x_2$ since $\alpha_3 = 1$. By expanding x_2 and so forth, we obtain a closed form $x_3 = z$ which implies that the output data rate of the third actor is the arrival rate. Instead of performing Gaussian elimination which exhibits a cubic runtime complexity to find the closed form of the actor, we employ Eq. (8) that requires a repetition vector. In our example, a most trivial periodic schedule can be found that contains a single invocation for each actor. Hence, the repetition vector of this periodic schedule is $\mathbf{q} = (1\,1\,1\,1)$. Using the formula from Eq. (8), we obtain $\alpha'_3 = \frac{\omega_3 \cdot q_3}{\iota_3 \cdot q_1} = 1$ and $x_3 = \alpha'_3 z = z$.

3.4 Processor Utilization

For the bottleneck elimination we require the notion of processor load incurred by the execution of actors depending on the input data rate. We have an underlying assumption that the multi-core system has the same *execution time* t_i of an actor invocation i on all parallel processors, and we obtain the execution time by profiling (see Section 6).

In our model the measurement of utilization is in percentage of the processor load. Assuming that a processor of the multi-core system exclusively executes actor i, the load of this processor would become 100% and the input data rate of actor i is $\frac{\iota_i}{t_i}$. Hence, $\frac{\iota_i}{t_i}$ bytes per second constitutes an upper bound for the input data rate of actor i, i.e., $\sum_{j=1}^{n} w_{ji} x_j \leq \frac{\iota_i}{t_i}$. This upper bound is used in the following section, which discusses a quantitative model for bottleneck elimination.

For the actor allocation and the bottleneck elimination we introduce a *processor utilization function* $\Gamma_i{:}\mathbb{R}^+ \to \mathbb{R}^+$, that maps the input data rate of actor i to a processor load which is defined as $x \mapsto \gamma_i x$, where

$$\gamma_i = \frac{t_i}{\iota_i}, \quad \text{for all } i \in \{1, \ldots, n\}.$$

We observe that $\Gamma_i(\frac{\iota_i}{t_i})$ represents a load of 100%, because $\frac{\iota_i}{t_i}$ is the upper bound of the input data rate.

For example, the parameter of the processor utilization function of the second actor in Figure 1 is $\frac{\iota_3}{t_3} = \frac{4}{60} = 0.066$ and therefore the maximal input data rate of actor 3 is 15 bytes per time unit.

For sake of readability, we define the processor load of actor i depending on the arrival rate:

$$U_1(z) = \Gamma_1(z)$$
$$U_i(z) = \Gamma_i\left(\sum_{j=1}^{n} w_{ji} \Phi_j^*(z)\right), \quad \text{for all } i \in \{2, \ldots, n\}.$$

Due to the linearity of the processor utilization function, we obtain

$$U_i(z) = \gamma_i \left(\sum_{j=1}^{n} w_{ji} \alpha'_j\right) z.$$

4. Elimination of Bottlenecks

The input data rate z of a stream program is limited by the processor capacity of the cores and the memory bandwidth used by storing tokens along channels. Bottlenecks limit the arrival rate of a stream program although free processing capacity in the parallel system is available. Hence, it is of key importance to eliminate bottlenecks in order to maximize the arrival rate of a stream program.

We perform the elimination of bottlenecks in a stream graph in two steps. In the first step, a quantitative analysis that is based on the performance model described in Section 3 identifies bottlenecks. In the second step, a stream graph transformation duplicates hot-regions, i.e., sub-graphs of the stream graph that cause a bottleneck. After transforming the stream graph the arrival rate is bound by the system resources.

4.1 Identification of Hot Actors

A quantitative analysis determines two types of upper bounds for the arrival rate: (1) an upper bound that is imposed by an actor in the stream graph, and (2) an upper bound that is imposed by the processing capacity and memory bandwidth of the parallel system. If the upper bound imposed by an actor is smaller than the upper bound of the parallel system, then the actor is *hot* and causes a bottleneck because free processing capacity is available but cannot be utilized.

The upper bound ub_i of the arrival rate for an actor i is discussed in the following. Assume that R is the maximum data rate in bytes at which an actor can receive tokens. Data rate R is limited by the memory bandwidth of the underlying hardware; we determine R through profiling. We can state the following inequalities for an actor:

$$\sum_{j=1}^{n} w_{ji} \Phi_j^*(z) \leq R, \quad \text{for all } i \in \{2, \ldots, n\}$$
$$U_i(z) \leq 1, \quad \text{for all } i \in \{1, \ldots, n\}$$

The first constraint limits the data rate of an input stream to R and the second constraint restricts the arrival rate z by the processor capacity of a single core whilst executing a single actor solely. Since the left-hand sides of both constraints are linear functions in z, we obtain:

$$z \leq \frac{R}{\sum_{j=1}^{n} w_{ji} \alpha'_j}, \quad \text{for all } i \in \{2, \ldots, n\}$$
$$z \leq \frac{1}{\sum_{j=1}^{n} w_{ji} \alpha'_j \gamma_i}, \quad \text{for all } i \in \{1, \ldots, n\}$$

Hence, the upper bound ub_i for actor $i \in \{2, \ldots, n\}$ is

$$z \leq \min\left(\frac{R}{\sum_{j=1}^{n} w_{ji} \alpha'_j}, \frac{1}{\sum_{j=1}^{n} w_{ji} \alpha'_j \gamma_i}\right) = \mathrm{ub}_i,$$

whereas for the start node the upper bound is $\mathrm{ub}_1 = \min(R, \frac{1}{\gamma_1})$.

The upper bound ub_s of the arrival rate imposed by the parallel system is given by

$$\sum_{i=1}^{n} \Phi_i^*(z) \leq pR$$
$$\sum_{i=1}^{n} U_i(z) \leq p,$$

where the first inequality states that the sum of the output data rates of actors cannot exceed pR assuming that each processor has the same memory bandwidth and the second constraint limits the overall load of the system to p times 100%. Since the left-hand sides of the above inequalities are linear, z can be factored out and

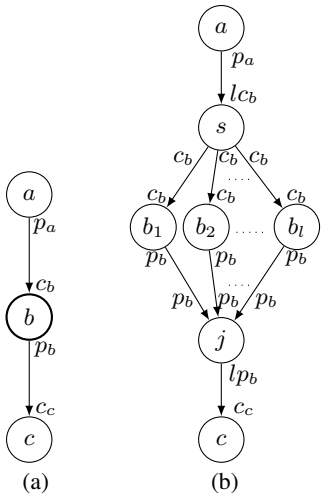

Figure 4. Hot actor duplication: (a) A pipeline consists of one hot actor b with duplicity l. (b) Duplication of actor b.

we obtain

$$z \leq \frac{pR}{\sum_{i=1}^{n} \alpha'_i}$$
$$z \leq \frac{p}{\gamma_1 + \sum_{i=2}^{n} \gamma_i \sum_{j=1}^{n} w_{ji} \alpha'_j}.$$

Hence, the upper bound ub_s is obtained by

$$z \leq \min\left(\frac{pR}{\sum_{i=1}^{n} \alpha'_i}, \frac{p}{\gamma_1 + \sum_{i=2}^{n} \gamma_i \sum_{j=1}^{n} w_{ji} \alpha'_j}\right) = \mathrm{ub}_s\,.$$

The upper bound ub_i of an actor i and the upper bound of the parallel system ub_s are conservative since a concrete placement can further lower the upper bound.

DEFINITION 2. *An actor i is called hot if $\mathrm{ub}_i < \mathrm{ub}_s$, and cold otherwise.*

If a hot actor is stateless, the stream graph transformation will create new instances of the actor to reduce the data input rate. With the reduced data input rates of the new instances, the bottleneck will be eliminated. Note that actors with a peek-rate that is greater than the pop-rate carry implicit state and hence are treated as stateful actors. For stateful hot actors, there is no simple transformation to overcome the bottleneck.

For the stream graph transformation, we introduce the notion of *duplicity* of an actor i that gives the number of instances required to avoid a bottleneck in the stream graph.

DEFINITION 3. *The duplicity of an actor is defined by*

$$d_i = \lceil U_i(\mathrm{ub}_s) \rceil\,.$$

A duplicity of one means that the actor is cold; a duplicity greater than one makes an actor hot.

4.2 Stream Graph Transformation

If the total processing capacity and the memory bandwidth of the parallel system are the most restricting constraints for the arrival rate z, the stream program is called *bottleneck free*, i.e., the processing capacity and/or the memory bandwidth is fully utilized. If a stateless actor b is hot, the stream program has a *bottleneck* and we create new instances of the actor so that the transformed stream program becomes bottleneck free.

The duplication of a hot actor b with duplicity l is depicted in Figure 4. Additional $l-1$ instances of actor b, a new roundrobin splitter s and a joiner j are added to the vertex set V and incoming and outgoing streams for the instances of actor b are duplicated for the new vertices, i.e.,

$$E' = E \cup \{(a,s)|(a,b) \in E\} \cup \{(j,c)|(b,c) \in E\}$$
$$\cup \{(s,b_i)\} \cup \{(b_i,j)\}, \text{for all } i \in \{1,\ldots,l\} \quad (9)$$

The consumed and produced tokens for the newly introduced nodes s and j are adjusted according to the data rates of hot actor b and duplicity l.

To minimize the number of data channels introduced by the duplication, we extend the notion of hot actors to *hot regions*. A hot region is a maximal connected subgraph $HR = (V', E')$ in G whose actors are hot and stateless. We assume that splitters and joiners are special nodes which cannot be regarded as hot actors. Hence, a hot region has the shape of a pipeline consisting one or more hot filters. The heuristic approach for region duplication is outlined in Algorithm 1.

Algorithm 1 Heuristic for Bottlenecks

1: Determine hot regions (*HRs*) in G.
2: **for** each (V', E') in *HRs* **do**
3: $\quad d_{HR} = \max_{i \in V'} d_i$
4: $\quad$ Duplicate (V', E') in G
5: **end for**

Hot regions are determined by a DFS search (line 1). The duplicity of a hot region *HR* is calculated by taking the maximum duplicity of the actors in the *HR* (line 3). In duplicating *HR* (line 4), we apply the same approach as explained in Figure 4 except that all the internal edges of the *HR* will also be duplicated and the consumed and produced tokens for the newly added nodes s and j are adjusted according to the duplicity of the *HR* and the data rates of the first and last hot actors of the *HR*, respectively.

5. Actor Allocation Problem

The *Actor Allocation Problem* (AAP) seeks an actor placement onto processors such that the arrival rate z of the stream program becomes maximal. The actor allocation problem may be expressed as a mathematical program as stated below:

$$max.\ z \quad (10)$$

$$s.t. \sum_{j=1}^{p} y_{ij} = 1 \qquad \text{for all } i \in \{1,\ldots,n\} \quad (11)$$

$$\sum_{i=1}^{n} U_i(z) y_{ij} \leq 1 \quad \text{for all } j \in \{1,\ldots,p\} \quad (12)$$

$$y_{ij} \in \{0,1\} \qquad \text{for all } i \in \{1,\ldots,n\}, \text{ for all } j \in \{1,\ldots,p\} \quad (13)$$

The binary variable y_{ij} is one if actor i is placed on processor j; zero otherwise. The constraint in Eq. (11) ensures that exactly one processor executes actor i. The right-hand side of the constraint in Eq. (12) represents the processor load of processor j for an allocation and it is bounded by 100%, i.e., the maximal processor load.

The mathematical program optimizes for the arrival rate rather than the throughput. However, the throughput is a linear function of the arrival rate, i.e.,

$$\sum_{i=1}^{n} \xi_i = \sum_{i=1}^{n} \Phi_i^*(z) = \left(\sum_{i=1}^{n} \alpha'_i\right) z, \quad (14)$$

whose slope $\sum_{i=1}^{n} \alpha_i'$ is positive. Although arrival rate and throughput are different optimization objectives, the solutions are identical.

THEOREM 1. *The actor allocation problem is NP-hard.*

PROOF 4. *The proof is shown in Appendix A.*

In the following we devise an algorithm for AAP that uses an oracle. The oracle tests whether for a given arrival rate z, there exists an actor allocation. With the existence of an oracle, binary search can be employed to seek for the largest z for which there exists an actor allocation. The binary search needs an upper and lower bound to commence the search. The lower bound is zero and an upper bound can be derived directly from the AAP problem itself as shown later in this section. Since the problem is NP hard, we cannot hope for an oracle that delivers a precise answer in polynomial time (unless P=NP). Hence, we use an approximate oracle that has no false positives, i.e., if the approximate oracle answers positively, there always exists an allocation, however, if the answer is "no", there might or might not exist an allocation.

As an oracle for AAP we use the bin-packing problem. For a fixed arrival rate z, the mathematical program of AAP reduces to the standard bin-packing problem [33].

DEFINITION 4 (**Bin-packing**). *Given a set of items A with sizes $s_i \in (0, 1]$ for all $i \in \{1, \ldots, n\}$, find a k-partitioning $B_1, \ldots, B_k$ of k disjoint sets $B_j \subseteq A$ such that $\sum_{s_i \in B_j} s_i \leq 1$, for all $j \in \{1, \ldots, k\}$.*

Bin-packing is an NP-complete problem for which approximation algorithms with polynomial runtime complexity and bounded solution quality exist.

OBSERVATION 1. *Bin-packing is an oracle in the arrival rate z.*

PROOF 5. *Assume actors in AAP become items and processors in AAP bins, and set the item size to $s_i = U_i(z)$. The bin-packing problem either delivers a packing for AAP limiting the number of bins to p, or it fails.*

The binary search scheme is illustrated in Algorithm 2. Let ub_z denote an upper bound of the arrival rate. The feasibility test of allocating actors to processors is achieved by using a bin-packing oracle (cf. line 8) for a given arrival rate m, whether an allocation for p bins (i.e. processors) and n items (i.e. actors) can be found. In the algorithm we have the invariant that the lower bound l of the arrival rate represents a feasible solution and u represents an infeasible solution. If the initial upper bound of the algorithm represents a feasible solution, we do not enter the loop and terminate with a feasible and optimal solution. If the actor allocation is feasible at the mid point of u and l, the lower bound is assigned the midpoint; otherwise the upper bound is assigned the midpoint (see line 7–11 of Algorithm 2). We terminate the binary search when the gap between lower and upper bound is less than or equal to a small value ϵ. In each step i the initial gap of ub_z between feasible and infeasible solution halves and we gain one bit in precision. Hence, the inequality $\frac{\mathrm{ub}_z}{2^i} \leq \epsilon$ holds after the termination of the algorithm, where i is the number of executed steps. The inequality implies that after $\lceil \lg_2(\frac{\mathrm{ub}_z}{\epsilon}) \rceil$ steps the binary search scheme will terminate.

For achieving an acceptable runtime complexity, an approximate oracle is employed. The approximate oracle is implemented as a greedy approximation algorithm for bin-packing [33] that achieves a packing of $B_{apx} \leq 2B_{opt}$, where B_{opt} is the optimal number of bins required to pack the items, and B_{apx} is the worst-case result of the approximation algorithm for all instances of bin-packing.

Algorithm 2 Binary Search

Require: n: the number of actors, p: the number of processors

```
1: l ← 0
2: u ← ub_z
3: if ∃ bin-packing allocation for arrival rate u then
4:     return allocation for arrival rate u
5: end if
6: while u − l > ε do
7:     m ← (l + u)/2
8:     if ∃ bin-packing alloc. for p bins and z = m then
9:         l ← m
10:    else
11:        u ← m
12:    end if
13: end while
14: return allocation for arrival rate l
```

The greedy bin-packing algorithm is implemented as follows: At any intermediate step i, it has a list of partially packed bins, $B_1, \ldots, B_k$. The approximation algorithm picks an item of size s_i and attempts to pack it in one of the partially packed bins of any assumed order. If the item does not fit into any of the partially filled bins, the algorithm opens a new bin B_{k+1} and packs the item in it. If the algorithm requires B_{apx} bins then at least $B_{apx} - 1$ of the bins are more than half filled. Hence, $\frac{B_{apx}-1}{2} < \sum_{i=1}^{n} s_i$ implies $B_{apx} - 1 < 2B_{opt}$ since $\sum_{i=1}^{n} s_i$ is a lower bound and $B_{apx} \leq 2B_{opt}$ follows.

In the following we analyze the quality of the solution obtained by binary search using the approximate oracle.

LEMMA 5. *Given a packing $B_1, \ldots, B_k$ of an instance of bin-packing. There exists a packing with $\lceil \frac{k}{2} \rceil$ bins by scaling items by $\frac{1}{2}$, i.e., $s_i' = \frac{s_i}{2}$.*

PROOF 6. *In the packing $B_1, \ldots, B_k$ of k bins, $\sum_{s_i \in B_j} s_i \leq 1$, for all $j \in \{1, \ldots, k\}$. If all items are scaled by a factor $\frac{1}{2}$, the inequality*

$$\sum_{s_i' \in B_j} s_i' \leq \frac{1}{2} \quad \textit{for all } j \in \{1, \ldots, k\}$$

holds. Hence, two consecutive bins can be packed in a single bin, i.e., $B_j' = B_{2j-1} \cup B_{2j}$ such that $\sum_{s_i' \in B_j} s_i' \leq 1$ for all $j \in \{1, \ldots, \lfloor \frac{k}{2} \rfloor\}$. If k is an odd number, then $B'_{\lceil \frac{k}{2} \rceil} = B_k$ of size less than or equal to $\frac{1}{2}$; otherwise this bin is not required.

COROLLARY 1. *If there exists an optimal packing with k bins, a 2-approximation bin-packing algorithm obtains a packing of k bins with item sizes $s_i' = \frac{s_i}{2}$.*

THEOREM 2. *A 2-approximation for the bin-packing oracle gives a $2 + \epsilon$-approximation for AAP.*

PROOF 7. *W.l.o.g. we assume that p bins are required to pack items of item sizes $s_i = U_i(z_{opt} - \epsilon)$ with a worst-case arrival rate of $z_{opt} - \epsilon$ using a precise oracle. Note that the arrival rate may be reduced by ϵ due to the halting condition of the binary search and we assume that $U_i(z) \geq \epsilon$.*

By Corollary 1 we achieve an approximate packing with p bins by scaling the item sizes by $\frac{1}{2}$:

$$U_i(z_{apx}) \geq \frac{U_i(z_{opt} - \epsilon)}{2} \Rightarrow$$
$$\left(\sum_{j=1}^{n} \alpha'_j w_{ji}\right) z_{apx} \geq \frac{\left(\sum_{j=1}^{n} \alpha'_j w_{ji}\right)(z_{opt} - \epsilon)}{2} \Rightarrow$$
$$\frac{z_{opt}}{z_{apx}} \leq 2 + \frac{\epsilon}{z_{apx}} \leq 2 + \epsilon.$$

For each step in the binary search scheme, we have an invocation of the bin-packing approximation that exhibits a worst-case runtime complexity of $\mathcal{O}(np)$. There are at most $\lg_2 \frac{\mathrm{ub}_z}{\epsilon}$ invocations of the bin-packing heuristic, and, hence, the overall complexity of finding an approximate solution is $\mathcal{O}(np \lg_2 \frac{\mathrm{ub}_z}{\epsilon})$.

An *instance bound* gives an upper bound on the quality of an approximate solution without knowing the optimal solution. An instance bound can be found if an upper bound of the arrival rate is known, i.e.,

$$1 \leq \frac{z_{opt}}{z_{apx}} \leq \frac{\mathrm{ub}_z}{z_{apx}}. \tag{15}$$

We can deduce an upper bound by summing up the left- and right-hand side of Eq. (11) over $j \in 1, \ldots, p$:

$$\sum_{j=1}^{p}\sum_{i=1}^{n} U_i(z) y_{ij} = \sum_{i=1}^{n} U_i(z) \sum_{j=1}^{p} y_{ij} = \sum_{i=1}^{n} U_i(z) \leq p \Rightarrow$$
$$\sum_{i=1}^{n} \left(\gamma_i \sum_{j=1}^{n} \alpha'_j w_{ji}\right) z \leq p \Rightarrow$$
$$z \leq \frac{p}{\sum_{i=1}^{n} \left(\gamma_i \sum_{j=1}^{n} \alpha'_j w_{ji}\right)} = \mathrm{ub}_z \tag{16}$$

The upper bound ub_z is also employed for initializing the binary search scheme.

6. Experiments

We have implemented our proposed mapping technique as an extension to the StreamIt compiler framework [1]. Our backend generates a uniprocessor schedule for profiling actor execution times of a stream program. Profiling uses the x86-64's hardware cycle counters exported by the clock library from [4]. Actor profiles and the stream graph topology information are then used to compute the closed form on the input data rate z. Quantitative analysis determines bottleneck actors and their duplicity. After bottleneck removal we compute the periodic schedule for the transformed stream graph and allocate actors to processors. Bottleneck removal and actor allocation are parameterized by the user-supplied number of processors (p), and by the memory bandwidth (R) that we determine through profiling. Based on the processor allocation our StreamIt compiler backend generates C-code for the parallelized version of the stream program. With the parallelized version the main program will spawn a scheduler thread on each of the p processors. Each scheduler invokes the actors that have been allocated to the corresponding processor. Together, the schedulers execute a steady-state iteration of the parallel periodic schedule. In between iterations, schedulers synchronize at a barrier. Data-channels between actors are implemented as shared-memory FIFO-buffers. Buffer sizes are derived from the periodic schedule and buffers are statically allocated. We employ multi-buffering similar to [11] to remove filter dependencies within a steady-state iteration of the stream graph.

Table 1. Benchmark characteristics.

Benchmark	Actors	Stateful	Peeking
`DCT`	22	18	16
`FMRadio`	67	23	22
`TDE`	55	27	2
`FFT`	26	14	0
`MergeSort`	31	2	0
`FilterBank`	53	34	16
`RadixSort`	13	2	0
`Equalizer`	65	22	20
`BitonicSort`	452	2	0
`DES`	375	180	1
`MPEG`	39	7	0
`MatrixMult`	52	2	0

To have a comparison for the quality of our achieved solutions we implemented the ILP formulation from [18]. The resulting ILP problems were solved using CPLEX. We evaluated our technique using 12 StreamIt applications depicted in Table 1.

We considered programs which have both stateful and stateless actors and actors with peeking operations (the ILP formulation from [18] was extended to work with stateful actors: we added additional constraints that prevent duplication of stateful actors, which reduces the solution space as well). Note again that actors where the peek-rate is greater than the pop-rate were identified as stateful actors and thus no actor duplication was performed. We encountered only one benchmark, FMRadio, that showed this characteristic.

The details of each benchmark are available online [1]. We have disabled the println statements because they were used in the sink nodes imposing high overheads. Sink nodes are stateful and hence cannot be duplicated. However, having a stateful node with dominating costs makes actor allocation trivial and does not allow a meaningful comparison of the approaches. Each benchmark was compiled and executed on a 2.33GHz dual quad-core Intel Xeon computer with 16GB main memory and Linux kernel version 2.6.23.

Table 2 shows the experimental results on 2–4 processors. The time to solve our actor allocation problem was in the range of milliseconds and we omitted it because it does not provide any useful insight. On the other hand, the ILP solve times vary greatly, with a tendency to grow substantially for larger numbers of processors (column t_{ILP} in Table 2). E.g., with `FFT` the ILP solve time increases from 0.46 seconds to 12 hours when the processor number is increased to three. The solver takes about 70 hours to solve the mapping for four processors.

We compared the input data rates achieved by our method with the data rates from the ILP formulation (the optimal solution). Column $\frac{z_{apx}}{z_{opt}}$ in Table 2 shows that the quality of our solutions stays within 5% of the optimum. The data rates achieved by our method are given in megabytes/second (column z_{apx} of Table 2). The lowest data rate was achieved for the `FFT` benchmark for 4 processors, and it is still 95% of the optimal data rate. The `DCT` and `MergeSort` benchmarks achieve 96% and 97% of the optimal rates respectively. Other than these, our approximation is mostly identical to the optimum. Data rates increase with the number of processors.

Figure 5 presents the speedups obtained by the approximation method for 2, 4, 6 and 8 processors over uniprocessor execution. Our approximation obtains near-linear average speedups of 6.95x for 8 processors. `MPEG`, `RadixSort` and `MatrixMult` achieve lower speedups due to stateful filters that cannot be paral-

Table 2. Experimental results for 2–4 processors.

Benchmark	*P#*	t_{ILP} (s)	$\frac{z_{apx}}{z_{opt}}$	z_{apx} (MB/s)
DCT	2	0.27	0.99	42.56
	3	1585.69	0.97	62.46
	4	2285.01	0.96	82.57
FMRadio	2	0.08	1.00	2.89
	3	3.22	1.00	4.00
	4	1.29	0.99	5.34
TDE	2	0.09	1.00	6.08
	3	0.17	1.00	19.80
	4	274.69	1.00	28.81
FFT	2	0.46	0.98	43.91
	3	44694.25	0.98	46.85
	4	249240.09	0.95	95.50
MergeSort	2	0.07	1.00	26.35
	3	0.09	0.99	56.24
	4	1.62	0.97	70.12
FilterBank	2	0.06	1.00	1.68
	3	0.62	0.99	2.51
	4	3.56	1.00	5.29
RadixSort	2	0.64	0.99	32.91
	3	0.09	1.00	49.98
	4	0.47	0.99	66.83
Equalizer	2	0.06	1.00	0.56
	3	5.29	1.00	0.83
	4	57553.83	0.99	1.59
BitonicSort	2	0.3	1.00	3.16
	3	3.06	1.00	4.73
	4	16371.99	1.00	10.14
DES	2	0.51	1.00	0.12
	3	2.73	1.00	0.18
	4	11.24	1.00	0.24
MPEG	2	0.09	1.00	36.59
	3	1.37	0.99	54.68
	4	0.44	1.00	73.22
MatrixMult	2	0.14	1.00	18.74
	3	4.85	1.00	28.16
	4	103.00	0.99	37.37

lelized as explained in Section 2. All other benchmarks scale well with increasing numbers of processors.

7. Related Work

Unlike classical data flow [9], SDF [20] fixes the number of tokens produced and consumed by an actor already at compile time. The static nature of SDF facilitates static program optimizations wrt. streamgraph transformations, partitioning and scheduling. A wide range of static scheduling algorithms for the SDF model exist [11, 16, 18, 19, 24, 31, 32]. SDF has been fundamental for contemporary stream languages including StreamIt [28, 30].

The StreamIt compiler [11] targets the Raw Microprocessor [22], shared-memory multicore architectures and clusters of workstations. To increase the computation to communication ratio, adjacent actors are fused as long as the result is stateless. A heuristic for actor fission is then applied to increase data-parallelism to the extent that a communication-efficient balance between task and data parallelism is maintained. Coarse-grained software-pipelining of actors eliminates actor dependencies within the same steady-state iteration, which increases flexibility of the program partitioning and scheduling phases. A greedy partitioning heuristic that

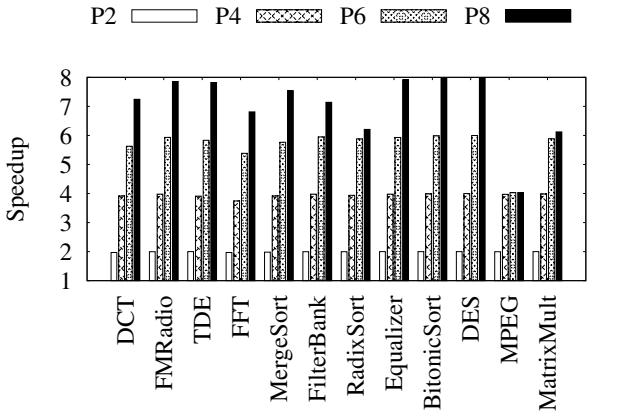

Figure 5. Speedups obtained for 2, 4, 6 and 8 processors.

minimizes the makespan is applied to load-balance actors among processors.

Kudlur and Mahlke's stream graph modulo scheduling [18] employs an ILP formulation to evenly distribute StreamIt actors among the synergistic processing elements of the Cell processor [13]. The ILP formulation consists of an integrated unfolding and partitioning technique that spreads data-parallel actors and maximally packs actors onto cores. Coarse-grained software pipelining is then applied such that computations of the current steady-state iteration are overlapped with data transfers for future iterations. In contrast to our model, (1) no approximation scheme was provided, making the approach intractable for larger problem instances, (2) the mathematical model optimizes makespan rather than arrival rate, (3) the proposed stream graph transformation duplicates single actors rather than regions, which may lead to a larger number of communication channels, and (4) we separate the bottleneck elimination from the actor allocation.

Udupa et al. devised an ILP formulation to partition and software-pipeline StreamIt programs on GPUs. The optimal execution configuration of a stream program in terms of the number of registers per thread and the number of data-parallel actor instances is determined through profiling. A buffer layout technique for GPUs that coalesces accesses to device memory is presented. The proposed approach minimizes makespan.

Orchestrating the execution of stream programs on multicore platforms with accelerators like GPGPUs is examined in [32]. Udupa et al. formulated a communication-aware ILP problem for partitioning computations between CPU cores and GPGPU streaming multiprocessors (SMs). They proposed a heuristic algorithm for the ILP problem, with solutions on average within 9.05% of the optimal solution across the benchmark suite. Partitioned tasks are then software-pipelined to execute on CPU cores and on the SMs of the GPU.

Carpenter et al. [6] present an iterative heuristic partitioning and allocation algorithm that maps Kahn process networks with optional SDF-parts onto heterogeneous multiprocessors. Partitions with short software pipelines are favored to reduce memory, latency and startup overheads that increase with the number of pipeline stages. To shorten software pipelines, generated partitions are convex, i.e., they are connected and without circular dependencies. Partitioning is parameterizable through the provision of basic connected sets, which constitute collections of actors that the compiler is allowed to pairwise merge. Their assumption that partitions need to be convex is not proven to be code optimal.

8. Conclusion

In this paper, we presented an approximation algorithm for solving the actor allocation problem, and a data rate transfer model that resolves bottlenecks in stream graphs. In our approach, we separate the bottleneck elimination process from the actor allocation which enables efficient and effective bottleneck elimination and actor allocation for multicore architecture.

Current state-of-the-art approaches rely on integer linear programming that provides optimal solutions for actor allocation (though only sub-optimal ILP models for rewriting the stream graph are currently available). The runtime of commercial ILP solvers makes an ILP approach impractical even for small to moderate benchmark sizes and low numbers of processors. In contrast, our approximation framework performs the task within milliseconds. The quality of our solution is 5% off the optimal solution, and for up to 8 processors achieves a geometric mean speedup of 6.95x over single processor execution across the StreamIt benchmark suite.

Acknowledgements

We thank Professor Saman Amarasinghe for shepherding this paper. Much gratitude also goes to the anonymous referees who provided excellent feedback on this work.

References

[1] StreamIt Website. `http://groups.csail.mit.edu/cag/streamit`, retrieved 2010.

[2] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *ACM Turing Award Lectures*, 2007.

[3] S. S. Battacharyya, E. A. Lee, and P. K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[4] R. E. Bryant and D. R. O'Halloran. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2003.

[5] 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, 2004.

[6] P. M. Carpenter, A. Ramirez, and E. Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES '09: Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 57–66. ACM, 2009.

[7] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2005.

[8] Cisco. The Cisco QuantumFlow processor: Cisco's next generation network processor. White paper, 2008.

[9] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376. Springer-Verlag, 1974.

[10] W. Eatherton. The push of network processing to the top of the pyramid, 2005.

[11] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS '06: Proceedings of the 2006 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[12] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354. IEEE Computer Society, 2005.

[13] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA '05: Proceedings of the 2005 International Symposium on High-Performance Computer Architecture*, pages 258–262. IEEE Computer Society, 2005.

[14] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's next-generation server processor. *IEEE Micro*, 30(2):7–15, 2010.

[15] M. Karczmarek. Constrained and phased scheduling of synchronous data flow graphs for the StreamIt language. Master's thesis, Massachusetts Institute of Technology, 2002.

[16] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. *LCTES '03: Proceedings of the 2003 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 38(7):1235–1245, 2003.

[17] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[18] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008.

[19] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36:24–35, 1987.

[20] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[21] W. R. Mark, R. Steven G., K. Akeley, and M. J. Kilgard. Cg: a system for programming hardware in a C-like language. In *SIGGRAPH '03*. ACM, 2003.

[22] E. W. Michael, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: The Raw machine. *IEEE Computer*, 30:86–93, 1997.

[23] NVIDIA Corporation. CUDA C Programming Guide 3.1, 2010.

[24] S. Robert. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[25] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Comput. Surv.*, 18(3):277–316, 1986.

[26] J. L. Shin, K. Tam, D. Huang, B. Petrick, and H. Pham. A 40nm 16-core 128-thread CMT SPARC SoC processor. In *ISSCC '10, Solid-State Circuits Conference Digest of Technical Papers*. IEEE International, 2010.

[27] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput stream programming in Java. *OOPSLA '07: Proceedings of the 2007 ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, 42(10), 2007.

[28] W. Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, USA, 2009.

[29] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT '10 Proceedings of the 2010 Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010.

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

[31] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *CGO '09: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2009.

[32] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Synergistic execution of stream programs on multicores with accelerators. *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 44(7), 2009.

[33] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[34] H. Wei, J. Yu, H. Yu, and G. R. Gao. Minimizing communication in rate-optimal software pipelining for stream programs. In *CGO '10: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 210–217. ACM, 2010.

[35] 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, 2008.

[36] D. Zhang, Z. Li, H. Song, and L. Liu. A programming model for an embedded media processing architecture. In *SAMOS '05: Proceedings of the 2005 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. Springer LNCS, 2005.

A. NP Hardness

PROOF 8. *(of Theorem 1). We show the NP hardness by reducing the partitioning problem [33] to the actor allocation problem. The partitioning problem has a finite set of integer numbers* $A = \{a_1, \ldots, a_n\}$ *as input and determines whether there exists two disjoint subset* $X_1 \subseteq A$ *and* $X_2 \subseteq A$ *with* $A = X_1 \cup X_2$ *whose sums* $\sum_{a \in X_1} a$ *and* $\sum_{a \in X_2} a$ *are equal.*

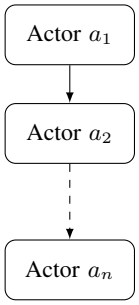

Figure 6. Reduction of partition problem to actor allocation problem.

Let's assume that the parallel system has two processors p_1 *and* p_2*. For each element in set* A *we introduce a vertex* $u \in V$*. Element* $a \in A$ *of associated vertex* u *is denoted by* a_u*. The principle decision whether a number* a *is either in the subset* X_1 *or* X_2*, is modeled by placing* u *on* p_1 *otherwise* u *is placed on* p_2*.*

We chain the filters in a pipeline as depicted in Figure 6 and define following constants for the bandwidth and processor utilization function

$$\begin{aligned} \alpha_u &= 1 \\ \gamma_u &= \frac{2a_u}{\sum_{a \in A} a} \end{aligned}$$

Hence, $\alpha'_u = 1$*, and* $U_u(z) = \gamma_u$*. The arrival rate can be at most one, if the AAP problem can pack the load of the actors equally on the two processors constituting a solution to the partitioning problem and the theorem follows.*

B. Glossary

α_i	The ratio between the number of bytes produced and consumed for a single invocation of actor i
Γ_i	The processor utilization function of actor i
Γ_i^{-1}	The inverse function the processor utilization function Γ_i
γ_i	$\frac{t_i}{\iota_i}$
ι_i	The number of bytes read from the program input for a single invocation of actor i
ξ_i	The output data rate of actor i in the periodic schedule
Π	Make span: the time duration of the longest running processor
Φ_i	Data rate transfer function of actor i
ω_i	The number of bytes produced by a single invocation of actor i
(i, j)	A channel $(i, j) \in E$ queues data elements which are passed from the output of computation i to the input of computation j
c_i	Number of consumed tokens on each execution of actor i
d_i	Duplicity of an actor i, i.e., how many instances in the stream graph are required to avoid a bottleneck.
E	A set of edges
$G(V, E)$	A stream graph consists of V vertices and E edges
$\mathcal{I}$	The identity matrix
n	Number of actors in the stream program
p	Number of processors in the parallel system
p_i	Number of produced tokens on each execution of actor i
$\mathbf{q}$	Repetition vector
q_i	The number of occurrences of actor i in the periodic schedule
R	Maximum data rate limited by memory bandwidth
t_i	The execution time of actor i
U_i	Utilization, $U_i = \gamma_i \sum_{j=1}^{n} \left(w_{ji} \alpha'_j \right) z$
ub_i	The upper bound of the arrival rate for an actor i
ub_s	The upper bound of the arrival rate imposed by the parallel system
ub_z	The upper bound of the arrival rate obtained by quantitative analysis
V	A set of vertices
w_{ij}	The fraction of the output data rate of actor i that is diverted from actor i to channel (i, j)
x_i	The output data rate of actor i
z	Arrival rate