A Heuristic Strategy for Performance Optimisation of Stream Programs (Extended Abstract)

Vu Thien Nga Nguyen, Raimund Kirner
University of Hertfordshire
\{v.t.nguyen, r.kirner\}@herts.ac.uk

I. INTRODUCTION

Programming models based on stream programming have become an active research topic, as stream programming has some nice benefits for parallel programming. For example, it makes some forms of parallelism explicit and the communication over streams facilitates implicit synchronisation. Because of this advantage, several research projects have introduced stream programming frameworks such as StreamIt [6], Brook [1], S-Net [3], and CnC [2] to name a few.

In this paper we present a design of a stream scheduler aiming at optimising throughput and latency of streaming programs with dynamic program structures. The scheduler uses heuristics based on the demand of data in communications streams. As we address dynamic structures of streaming programs, the particular challenge is that static scheduling based on formal constraints or probabilities is not applicable.

II. BACKGROUND

Stream programming is a paradigm that allows us to express parallelism by decoupling computations and communications [6], [1], [3]. The structure of stream programs is usually illustrated as a graph whose vertices are computation nodes and edges are communication channels called streams. This paper refers computation nodes as nodes and assumes that streams are unidirectional.

Data arrives to a stream program as a virtually infinite sequence of messages. Input messages coming from the external environment are called external input messages. The structure of stream programs is usually illustrated as a graph whose vertices are computation nodes and edges are communication channels called streams. This paper refers computation nodes as nodes and assumes that streams are unidirectional.

Data arrives to a stream program as a virtually infinite sequence of messages. Input messages coming from the external environment are called external input messages. Output messages sent to the external environment are called external output messages. An external input message is completed when all the associated external output messages are produced.

Conceptually the stream execution model includes two layers: a runtime system (RTS) and a scheduler. At the RTS layer, each stream is represented as a FIFO buffer and each node is transformed into one task. Tasks consuming external input messages are entry tasks, and tasks producing external output messages are exit tasks. Other tasks are called middle tasks. A task is ready to be executed if all required messages are available on its input streams. Lying under the RTS, the scheduler employs a policy to distribute ready tasks to physical resources. The scheduler’s policy decides: i) which ready task will be processed; and ii) which physical resource will process the ready task.

The throughput of stream programs is measured as the number of external input messages that are completed per time unit. The latency of an external input message is the time interval from when it is consumed by the program to when it is completed. The performance of stream programs is evaluated by the average latency as the arithmetic mean of the latency of all external messages. For convenience, we refer to the average latency simply as latency.

III. ANALYSIS OF THROUGHPUT AND LATENCY

As uni-directional streams can be considered as queues of messages, a stream program is constructed by queues of messages and therefore can be considered as a queuing system. We consider here only stable queuing systems where the average number of external input messages currently processed the stream program $M_{cp}$ is bounded [4].

A. Throughput Analysis

Consider a stream program deployed on a platform of $N$ homogeneous physical resources for a time period $P = [0, t]$. After the period $P$, $M$ external messages have been completed and $M_{cp}$ external messages are partially processed. Let the average computational time required to complete one external message be $C$. The total computational time required to complete these $M$ messages is $C_M \approx M \cdot C$. The total computational time for partly processing these messages is $C_{M_{cp}}$. Since $M_{cp}$ is bounded, $M_{cp}$ and $C_{M_{cp}}$ are bounded.

During the period $P$, the total processing time of the $N$ resources is $T = N \cdot t$. The total idling time of the $N$ resources is $W$. The relative idling time of the system is defined as $\bar{W} = \frac{W}{\bar{C}}$. During the period $P$, the $N$ resources contribute to the computations of $M$ completed messages; the computations of $M_{cp}$ partly processed messages; and idling time. We therefore have:

\[
T = N \cdot t = C_M + C_{M_{cp}} + W \\
= M \cdot C + C_{M_{cp}} + \bar{W} \cdot t
\]

Therefore,

\[
M = \frac{N \cdot t - C_{M_{cp}} - \bar{W} \cdot t}{\bar{C}}
\]

The throughput over the period $P$ is:

\[
TP = \frac{M}{t} = \frac{N \cdot t - C_{M_{cp}} - \bar{W} \cdot t}{\bar{C} \cdot t} \\
= \frac{1}{\bar{C}} \cdot \left( N - \frac{C_{M_{cp}}}{t} - \bar{W} \right)
\]
When the stream program processes infinite external input messages, the overall throughput is obtained when $t \rightarrow \infty$. Because $C_{M_{cp}}$ is bounded, $\lim_{t \rightarrow \infty} \frac{C_{M_{cp}}}{c_{M_{cp}}} = 0$. Therefore, the overall throughput is:

$$TP_{t \rightarrow \infty} = \frac{(N - \bar{W})}{C}$$

As $C$ varies with the implementation and the underlying hardware, it is not under the sphere of control of the scheduler. Therefore, to optimise the throughput the scheduler should: i) keep $M_{cp}$ bounded and ii) reduce $\bar{W}$.

B. Latency Analysis

According to Little’s law [4], we have:

$$L = \frac{M_{cp}}{\lambda_{consume}}$$  (1)

Where $L$ is the latency and $\lambda_{consume}$ is the consumption rate, i.e., the rate at which the stream program consumes external messages. To reduce the latency, the scheduler needs to increase $\lambda_{consume}$ and at the same time keep $M_{cp}$ low. Within stable systems, $\lambda_{consume}$ is equivalent to the throughput, therefore to maximise throughput is also to contribute to minimising the latency.

IV. A STREAM SCHEDULER DESIGNED FOR PERFORMANCE OPTIMISATION

According to the previous analysis, the scheduler has to minimise the resources’ relative waiting time $\bar{W}$ and the number of external messages inside the program $M_{cp}$. In the following, we introduce the design of a space scheduler that decides on which resource a task is executed; and a time scheduler which decides when a task is executed.

The space scheduler does not map tasks permanently to a particular resource. Instead, ready tasks are stored in a central queue (CTQ). When a resource becomes free, a ready task is picked and assign to it. Dynamic stream program structures are well supported by using the CTQ with its dynamic scheduling of tasks to available resources. The CTQ approach helps to reduce the $\bar{W}$ but does not guarantee to minimise it. This depends on the time scheduler which controls the availability of ready tasks. This design of the space scheduler allows flexibility for the time scheduler to control the ready task availability as well as $M_{cp}$.

The time scheduler on one side has to activate enough ready tasks and on the other side controls $M_{cp}$. Note that the availability of ready tasks is equivalent to the availability of messages inside the stream program. When a resource is free, the time scheduler chooses from the CTQ the highest priority task to send to the free resource. The task priority is defined by a demand-based function as follows.

**Demand-based Task Priority.** This heuristics is based on the positive demand $S_I$ and the negative demand $S_O$, where $S_I$ is the total number of messages in the input streams and $S_O$ is the total number of messages in the output streams. The proposed heuristics works as follows:

- **The priority of an entry task should have a negative correlation with its $S_O$.** Entry tasks are ready as soon as there are external messages. Their execution makes their successor tasks ready. This heuristic helps entry tasks be executed when the potential of ready tasks is low. Once executed, their priority is reduced and after a certain time they have to release the resources for other tasks keeping the $M_{cp}$ bounded.

- **The priority of exit tasks should be higher than other types of tasks.** This is because exit tasks send messages to the external environment, they should be executed as soon as possible to keep $M_{cp}$ as low as possible.

- **The priority of a middle task should have a positive correlation with $S_I$ and a negative correlation with $S_O$.** Exit tasks should be executed as soon as possible, however they become ready only when messages are transferred over the stream program passing other middle tasks. A middle task $T_0$, while performing the associated node’s computations, consumes $n$ messages from its input streams and produces $m$ messages to its output streams which are read by successor tasks $T_i \mid 1 \leq i \leq n$. The task’s $S_I$ is reduced by $n$ and its $S_O$ is increased by $m$. With this heuristics, $T_0$’s priority is reduced and its chance to hold physical resources is reduced. Meanwhile the $S_I$ of tasks $T_i \mid 1 \leq i \leq n$ is increased. That means that the tasks $T_i \mid 1 \leq i \leq n$ will have a higher chance to be scheduled and the newly created messages are likely to move forward to the output.

V. DISCUSSION

In this paper we have presented the design of a heuristics-based scheduler to optimise in terms of throughput and latency the performance of stream programs with dynamic network structures. The scheduler deploys a centralised approach with demand-based heuristics for task selection, which is geared towards optimising throughput and latency. The scheduler has been implemented as a new scheduler for the execution layer LPEL [5] to support S-Net stream programs [3]. In contrast to the default scheduler of the reference system, the new scheduler does consider knowledge of the structure and state of the streaming network. Experiments show that the new scheduler shows significant improvement in throughput and latency compared to the default scheduler.

REFERENCES


