Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Risto Vuorio

Stream Processing on a Multi-core DSP with Open Event Machine

Master's Thesis
Espoo, May 20, 2016

Supervisor: Professor Heikki Saikkonen
Instructor: Vesa Hirvisalo D.Sc. (Tech.)

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Risto Vuorio

Title:
Stream Processing on a Multi-core DSP with Open Event Machine

Date: May 20, 2016 **Pages:** 87

Professorship: Software Systems **Code:** T-106

Supervisor: Professor Heikki Saikkonen

Instructor: Vesa Hirvisalo D.Sc. (Tech.)

Streaming data is responsible for a large fraction of the growth in creation and copying of data over the Internet. Video streams make up a sizable portion of this data. Stream processing is a computing paradigm that seeks to provide a useful abstraction of computing over streaming data. The stream processing and signal processing models are similar on a high level.

Digital Signal Processors (DSP) are computing units optimized for low energy consumption in signal processing tasks. The trend toward more cores in CPUs and GPUs has been adopted by the DSP vendors as well. Multi-core DSPs provide an interesting platform for stream processing as they combine an architecture optimized for signal processing tasks with the parallel computing power of the multiple cores.

Scheduling parallel computing is a non-trivial task. On DSPs it is often complicated by the lack of an OS as an abstraction layer for parallelism. Open Event Machine (OpenEM) provides a task-parallel computing model for Texas Instruments (TI) multi-core DSPs with a dynamic, hardware accelerated scheduler. In this thesis the performance of OpenEM in stream processing is investigated. The TI implementation of OpenEM for TMS320C6678 multi-core DSP is used in this thesis.

The concrete contributions of this thesis are the construction of an OpenEM based video stream processing measurement system, the construction of similar, statically scheduled measurement system, the comparison of the systems and the analysis of the stream processing performance of OpenEM using the measurement systems. The analysis uses the results of three experiments conducted in this thesis.

The results of the experiments show that the OpenEM runtime can be used for implementing high performance stream processing applications. Further research is needed to compare the performance to the competing computing platforms.

Keywords: stream processing, digital signal processor, parallelism, hardware accelerated scheduling

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Risto Vuorio

Työn nimi:
Virtalaskenta Moniydin DSP:llä Open Event Machine:a käyttäen

Päiväys:	20. Toukokuuta 2016	**Sivumäärä:**	87
Professuuri:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Heikki Saikkonen		
Ohjaaja:	TkT Vesa Hirvisalo		

Kasvavat datavirrat aiheuttavat suuren osan uuden datan luomisesta ja sen kopioimisesta internetin yli. Huomattava osa datavirtojen sisällöstä internetissä on videoita. Virtalaskenta on laskentaparadigma, joka pyrkii tarjoamaan käyttökelpoisen abstraktion virtaavan datan laskennasta. Virtalaskenta ja digitaalinen signaalinkäsittely ovat korkealla abstraktiotasolla samankaltaisia laskentamalleja.

Digitaaliset signaaliprosessorit (DSP) ovat laskentayksiköitä, jotka on suunniteltu tarjoamaan matalaa energian kulutusta signaalinkäsittelytehtävissä. DSP:iden valmistajat ovat omaksuneet tietokoneen suorittimissa ja grafiikkakiihdyttimissä yleisen suuntauksen kasvattaa yksiköiden ydinmäärää. Moniydin DSP:t tarjoavat mielenkiintoisen virtalaskenta-alustan sillä ne yhdistävät signaalinkäsittelyyn optimoidun arkkitehtuurin monen ytimen tarjoamaan rinnakkaislaskentatehoon.

Rinnakkaislaskennan ajoittaminen ei ole vähäpätöinen tehtävä. DSP:illä tätä tehtävää monimutkaistaa käyttöjärjestelmän tarjoaman rinnakkaislaskenta-abstraktion puuttuminen. Open Event Machine (OpenEM) tarjoaa tehtävärinnakkaisen laskentamallin Texas Instruments:in (TI) moniydin DSP:eille laitteistokiihdytetyllä dynaamisella tehtäväajoituksella. Tässä diplomityössä tutkitaan OpenEM:n suorituskykyä virtalaskennassa. Työssä käytetään TI:n TMS320C6678 moniydin-DSP:lle toteuttamaa versiota OpenEM:sta.

Diplomityön tuloksena on toteutettu OpenEM:neen pohjautuva virtalaskennan mittausjärjestelmä, vastaava, staattisesti ajoitettu mittausjärjestelmä, järjestelmien vertailu ja OpenEM:nen virtalaskentasuorituskyvyn analyysi järjestelmien avulla. Analyysissä tutkitaan kolmen työtä varten tehdyn kokeen tuloksia.

Kokeiden tulokset näyttävät, että OpenEM:a voidaan käyttää hyvän suorituskyvyn omaavien virtalaskentaohjelmien toteuttamiseen. Jatkotutkimusta tarvitaan suorituskyvyn vertaamiseksi kilpailevien laskenta-alustojen suorituskykyyn.

Asiasanat:	virtalaskenta, digitaalinen signaaliprosessori, rinnakkaisuus, laitteistokiihdytetty ajoitus
Kieli:	Englanti

Acknowledgements

I would like to thank my supervisor, professor Heikki Saikkonen, for his invaluable feedback in the final stages of writing this thesis.

Besides my supervisor, I would like to express my sincere gratitude to my instructor D.Sc. (Tech) Vesa Hirvisalo, who provided me with support and feedback throughout the project and without whom this thesis would not have been possible.

My thanks goes to Jussi Hanhirova, who worked with me in solving many technical difficulties, provided me with insightful comments, and helped me direct the work when needed.

I want to thank my family for supporting me during the work on the thesis and in life in general.

Finally, I thank my good friend and fellow researcher Kristian Hartikainen for countless hours of coworking on the theses, his accurate feedback, and all the weird jokes about Finnish inflection of nouns.

Espoo, May 20, 2016

Risto Vuorio

Abbreviations and Acronyms

4CIF	4 x CIF
API	Application Programming Interface
CCS	Code Composer Studio
CIF	Common Intermediate Format
CPU	Central Processing Unit
DDF	Dynamic Dataflow
DDR	Double Data Rate
DMA	Direct Memory Access
DPDK	Dataplane Development Kit
DSP	Digital Signal Processor
EMIF	External Memory Interface
EO	Execution Object
FLOPS	Floating-point Operations Per Second
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
L1	Level 1 cache
L2	Level 2 cache
MCSDK	Multicore Software Development Kit
MPEG	Moving Picture Experts Group
MSM	Multicore Shared Memory
MSMC	Multicore Shared Memory Controller
MoC	Model of Computation
NSN	Nokia Solutions and Networks
NTSC	National Television System Committee
OpenEM	Open Event Machine
PCI	Peripheral Component Interconnect
PDSP	Packed Data Structure Processor
PKTDMA	Packet Direct Memory Access
PiSDF	Parameterized and Interfaced Synchronous Dataflow

QCIF	Quarter CIF
RGB	Red Green Blue
S-LAM	System-Level Architecture Model
SDF	Synchronous Dataflow
SoC	System on a Chip
TI	Texas Instruments
UI	User Interface
YCbCr	YCbCr color space
YUV	YUV color space

Contents

Abbreviations and Acronyms **5**

1 Introduction **10**
1.1 Problem statement 12
1.2 Contributions 13
1.3 Structure of the Thesis 14

2 Data Streams **15**
2.1 Stream Processing 16
2.1.1 Applications of Stream Processing 16
2.1.2 Stream Processing Platforms 18
2.1.3 Video Streams 18
2.2 Dataflow 19
2.2.1 Dataflow Models of Computation 19
2.2.2 Synchronous Dataflow 20
2.2.3 Dynamic Dataflow 22

3 Open Event Machine **24**
3.1 Parallel Computing 24
3.1.1 Task Parallelism 25
3.1.2 Event-driven Programming 25
3.2 OpenEM Framework 27
3.2.1 Events 28
3.2.2 Execution Objects 29
3.2.3 Queues 30
3.2.4 Scheduling 31
3.2.5 Error Handling 31
3.2.6 An Illustrative Example 32
3.3 Texas Instruments Implementation of OpenEM 33
3.3.1 Scheduling 34
3.3.2 Event Pre-loading 34

3.3.3 Hardware Acceleration in Texas Instruments OpenEM 35
3.3.4 Cache Coherency . . . 36
3.3.5 OpenEM Tracing . . . 36
3.3.6 Programming with TI OpenEM . . . 36
3.3.7 State of TI OpenEM Implementation . . . 37

4 Multi-core DSP in Video Stream Processing 38
4.1 Performance Analysis . . . 38
4.1.1 Analysing Software Systems . . . 39
4.1.2 Measuring Software Systems . . . 40
4.1.3 Analysis of the Measurement Data . . . 41
4.2 Texas Instruments TMS320C6678 . . . 41
4.2.1 Selection of the Hardware Platform . . . 42
4.2.2 TMS320C6678 Overview . . . 42
4.2.3 C66x DSP . . . 44
4.2.4 Memory Hierarchy . . . 44
4.2.5 Multicore Navigator . . . 45
4.3 PREESM . . . 46
4.3.1 PREESM Overview . . . 46
4.3.2 PREESM Internal Representations . . . 47
4.3.3 PREESM Scheduling . . . 49
4.3.4 PREESM Memory Allocation . . . 50
4.4 Video Streams . . . 51
4.4.1 YUV Format . . . 51
4.4.2 Common Intermediate Format . . . 51
4.5 Canny edge detector . . . 52
4.5.1 Gaussian filter . . . 53
4.5.2 Sobel filter . . . 53
4.5.3 Canny Edge Pruning . . . 53

5 Experiment Construction 55
5.1 Filter Application . . . 55
5.2 PREESM Filter Application . . . 56
5.2.1 Actor Model . . . 56
5.2.2 PREESM Schedule . . . 58
5.3 OpenEM Filter Application . . . 60
5.4 Instrumentation . . . 61
5.5 Description of Experiments . . . 62
5.5.1 Parameters and Factors . . . 63
5.5.2 Measurement Setups . . . 64

6 Experiment Results and Analysis **66**
6.1 Results 66
6.1.1 Comparing Dynamic and Static Scheduling 66
6.1.2 Investigating the Balance of Latency and Throughput . 68
6.1.3 Examining the Efficiency of Parallel Scheduling 70
6.2 Discussion 72
6.2.1 Discoveries 72
6.2.2 Future Work 75
6.2.3 Challenges 76

7 Conclusions **78**

Chapter 1

Introduction

Digital data is being created and copied over the Internet at an accelerating pace. A large fraction of this data is transferred and processed in the form of data streams such as video. In general, streaming data is processed as it flows by the processing node, without saving a copy of the complete stream on the node. For example, a video streaming application buffers some number of video frames on the viewing device, decodes and presents them, and then frees the memory for buffering more frames.

Stream processing programs consist of series of operations that are performed in parallel on the streams. The capability for parallelizing the programs is important for creating low-latency, high-throughput applications. In the past, single-threaded performance of computers was growing rapidly, which made hitting the latency and throughput constraints of the applications a question of waiting for the next generation CPUs, which would make all of the applications faster. For the past ten years or so, the single-threaded performance has remained roughly the same while the number of cores in CPUs has grown. This turn towards parallelism in hardware has emphasized the importance of parallel processing in streaming applications.

Traditional CPUs are not the only multi-core devices capable of stream processing. Different hardware architectures implemented by digital signal processors (DSP) and graphics processing units (GPU) have been successfully used for various forms of parallel computing. Each of the hardware architectures offer their own set of advantages: CPUs have highly optimized architecture for processing full-featured threads, GPUs excel in data parallelism, and DSPs offer low energy consumption in signal processing applications. DSPs have the potential for becoming an efficient platform for stream processing, as signal processing applications have similar structure to stream processing applications.

There are many forms of parallelism and one of them, task parallelism,

is of particular interest for this thesis. In task parallelism units of work called tasks that consist of code and data are distributed on the cores of the computer. Stream processing applications can be seen from the task parallel perspective with each of the operations performed on the streams making up a separate task. In this thesis, the stream processing performance of a task based programming model on a multi-core DSP is investigated. The task based programming model under study is the Open Event Machine (OpenEM).

Important aspect of the performance of task parallel programming models is how the tasks are scheduled on the cores. OpenEM features a dynamic scheduler, which means the execution order of tasks is decided in runtime. Another way of scheduling tasks would be static scheduling, where the execution order of the tasks is determined at the time of compilation. To put the OpenEM performance in perspective, it is compared to the performance of statically scheduled applications. The PREESM framework is used for creating the statically scheduled application. PREESM is a tool for creating statically scheduled parallel applications for DSPs following the synchronous dataflow model of computation.

The hardware platform used in the experiments conducted in this thesis is the Texas Instruments TMS320C6678 multi-core DSP. Out of other possible multi-core DSPs, the particular device was selected for the experiments most importantly because Texas Instrument provides an implementation of OpenEM for the device. Another advantage of the TMS320C6678 is that PREESM supports the processor as a target for code generation, which makes creating comparable applications simple.

Two stream processing applications were implemented: one using OpenEM and the other created with PREESM, using static scheduling. The main contribution of this thesis is the analysis of the performance of OpenEM in stream processing tasks.

The relevancy of studying OpenEM on multi-core digital signal processors for stream processing comes from a combination of factors. Firstly, stream processing is a computing paradigm that is receiving increasing attention from the industry and the research community due to the growing volume of data streaming in the Internet. Secondly, stream processing tasks have similar high-level structure as digital signal processing tasks. Examining the fit of digital signal processors for stream processing is thus a task relevant to the study of stream processing. Thirdly, multi-core coordination is non-trivial but parallel processing is required for efficient stream processing as stream processing tasks are highly data parallel. Therefore, the dynamic scheduler implemented in OpenEM has a large impact on the performance of the stream processing applications using it.

1.1 Problem statement

Multi-core DSPs have potential for high performance stream processing in terms of floating point operations per Joule. However, parallel programming for multi-core DSPs is more complicated than parallel programming for PC hardware, as commonly there is no operating system to handle the inter-core coordination and communication. The TI implementation of OpenEM is a runtime system that handles the inter-core coordination and communication. This thesis investigates the performance of the TI OpenEM implementation in stream processing.

The performance of a multi-core runtime system is dependent on the efficiency of the scheduling. TI OpenEM implements dynamic scheduling. The scheduler is hardware accelerated, running on a special processor that is connected to the hardware queuing system of the processor. The communication in TI OpenEM utilizes the hardware queues of the processor.

The dynamic scheduler can negatively affect the performance of the application versus a statically scheduled application if the scheduling requires a large overhead in terms of processor cycles or memory. In addition to the efficiency of the dynamic scheduler, the overall goodness of the schedule affects the application performance. If the scheduler (static or dynamic) places all the tasks sequentially, none of the benefits of parallelization are achieved. Another, more subtle effect the schedule has on the performance is the memory locality of the tasks. If tasks operating on the same data are executed on different processors or other tasks are run on the same processor between the tasks, the data has to be copied between the processors, whereas if the tasks execute sequentially on the same processor, no time is wasted in copying the data.

Three views to the performance of OpenEM in stream processing are taken through three different experiments. The performance of the dynamic scheduler is assessed by comparing an application implemented using OpenEM to a statically scheduled application implemented with PREESM in experiment 6.1.1. The capabilities of the dynamic scheduler are further investigated by looking at the balance of throughput and latency in experiment 6.1.2. Finally, an understanding of the parallelizing performance of the scheduler is sought out in the experiment 6.1.3 where the stream processing application is run on different numbers of cores and the results are compared.

1.2 Contributions

This thesis presents analysis of the performance of OpenEM in stream processing. The performance analysis is carried out by comparing the measured performance of a stream processing application implemented using OpenEM to a comparable stream processing application.

The application used for comparison was implemented using dataflow model of computation with a graphical tool called PREESM. The PREESM tool is able to generate executables for the Texas Instruments TMS320C6678 DSP from the graphical model of the dataflow application and source code for the processing kernels provided by the user. The dataflow graph, the processing kernels, and the instrumentation of the PREESM application were implemented for this thesis.

The main focus in the experiments is on the the OpenEM based stream processing application. The application was implemented from scratch for this thesis. The Texas Instruments Code Composer Studio was used to implement the stream processing application. The application was instrumented using the hardware performance counters found in the TI DSP.

In the analysis phase, the performance of the applications was compared based on the measurements made using the instrumentation. Three experiments were conducted to understand the performance of OpenEM based applications in stream processing.

The contributions of this thesis are summarized in the following listing:

- Implementation and instrumentation of an OpenEM based measurement system on TI DSP.
- Implementation and instrumentation of a comparable measurement system using PREESM.
- Analysis of the OpenEM scheduler performance in scheduling a stream processing application is conducted by comparing OpenEM dynamic scheduling with static scheduling on TI DSP.
- Examination of the adjustability of the OpenEM scheduler in terms of latency versus throughput is carried out.
- Analysis of the efficiency of the OpenEM scheduler in parallel processing is provided.

1.3 Structure of the Thesis

The structure of this thesis is presented in the following.

Chapter 2 introduces the context of this thesis. The relevance of stream processing to computer science research and the industry are explained. An overview to stream processing is given and a more focused look at one paradigm of stream processing, dataflow, is taken.

Chapter 3 takes an in-depth look at Open Event Machine. First, the context of parallel computing is introduced and the forms of parallelism implemented in OpenEM are described. Second, the Nokia Solutions and Networks specification of the framework is introduced. Third, the Texas Instruments implementation of OpenEM for TMS320C6678 multi-core DSP is examined.

Chapter 4 introduces the material and methods used in the experiments. In the beginning of the chapter, a look at the performance analysis of computer systems is taken. More specifically, measuring computer systems is described. Next, the hardware platform used in the experiments, the TI TMS320C6678 is described. Following the hardware description, PREESM is introduced as the tool that was used for creating the comparison point for the OpenEM stream processing application. After PREESM, video stream standards relevant to the experiments are explained. Finally, Canny edge detector that served as inspiration for the workload is described.

Chapter 5 describes the construction of the experiments. After the introduction to the experiments, the experiment workload is introduced. Next, the PREESM version and the OpenEM version of the workload application are described. Third, the instrumentation of the applications is explained. Finally, the experiments are presented.

Chapter 6 presents the results of the experiments. In the beginning of the chapter, the results are given and described. The rest of the chapter is dedicated for discussion of the results.

Chapter 7 is the conclusion of this thesis, bringing the results to the broader context of stream processing and parallel computing.

Chapter 2

Data Streams

The amount of digital data that is being created and copied is increasing at a massive pace. EMC Digital Universe study [67] estimates that the data we create and copy annually was 4.4 zettabytes in 2013 and is going to reach 44 zettabytes by 2020. A large portion of this data growth is due to increased volume of entertainment in the form of audio and video being streamed through the Internet. New data sources such as embedded sensors are contributing to the explosive data growth as well. Most of the data being created or copied is transient and thus does not require long term storage. [67]

Distributed data processing systems such as Hadoop [68] were developed for batch processing of Big Data. The batch processing systems are capable of processing massive amounts of data but the processing has high latency. As most of the growth in data creation and copying is coming from data that is not stored, batch processing is not the optimal processing method for data.

Consider live streaming of video as an example of modern application with specific data processing needs. A camera filming a live scene is generating the data and the users expect to access the video stream over the Internet with low latency. Many distinct processing steps are required in order to get the video frames from the camera to the viewer, all of which need to be performed in few milliseconds on the frames flowing past the processing units. In addition to video streams, many other kinds of data streams, which have similar processing needs are produced at an accelerating pace. Certain technologies developed for processing the streaming data are grouped under the term stream processing.

An introduction to stream processing and selected stream processing techniques are given in this chapter. A stream processing overview is given in 2.1. Selected streaming applications and their common features are described in 2.1.1. Platforms used for stream processing are introduced in 2.1.2. Video

streams are discussed as an example of a common type of streaming data in 2.1.3. Dataflow Models of Computation are discussed in 2.2.1 as an example of solution for stream processing. Variations of dataflow MoC are examined, synchronous dataflow in 2.2.2 and dynamic dataflow in 2.2.3.

2.1 Stream Processing

Stream processing is a method of computing over streaming data. Stream processing is often used to refer to the programming paradigm that follows the stream processing method but stream processing is not limited to the creation of software. Stream processing term has been used in the literature to describe different kinds of methods that deal with streaming data. In the context of this thesis, the broad definition of stream processing as a method of processing any kind of streaming data is used.

Stream processing paradigm defines modules that compute in parallel and communicate data via channels. The modules can be divided into three classes according to their placement and purpose in the process. The classes are sources, filters, and sinks. Sources act as the input points that pass data into the process. Filters perform atomic computations on the streams. Sinks are used to pass the data out from the process. [61]

In stream processing, the channels of communication between the modules are called streams. Streams can be described as infinite lists of elements taken from a dataset A. Mathematical formalization of a stream is a function $f : T \to A$ where $T = \mathbb{N}$ represents discrete time. [61] For example, the input stream of a signal processing application can be the sample based input sequence, which has been generated by sampling a sensor at fixed time intervals.

Systems built following the stream processing method can be categorized under stream processing systems. The stream processing survey by Stephens [61] categorizes *dataflow systems*, *reactive systems*, *synchronous concurrent algorithms*, *signal processing systems*, and some *real-time systems* as stream processing systems.

2.1.1 Applications of Stream Processing

The authors of the StreamIt language [65] have defined *streaming applications* as a class of programs, which commonly have many of the features defined in the following listing.

- *Large streams of data.* A streaming application operates on large, virtually infinite streams of data.

- *Independent stream filters.* The filters of a streaming application are generally self-contained. They perform atomic operations on the stream.
- *A stable computation pattern.* A streaming application has a steady state of operation during which the graph formed by the filters remains mostly constant.
- *Occasional modification of the stream structure.* A streaming application can occasionally modify the processing graph as a reaction to changed input or some other condition.
- *Occasional out-of-stream communication.* The high volume communication between the filters is handled through the streams but the filters may communicate small amount of control data outside the stream.
- *High performance expectations.* There often are real-time and power consumption constraints on streaming applications. For example, a streaming video decoder has to decode the stream at rate of input in order to avoid unbounded buffer growth and frame dropping.

Applications that handle streams of audio and video often have the above features and can be implemented following the stream processing paradigm. The multimedia domain is therefore full of examples of stream processing applications. Video conferencing is a commonly used example of a streaming application, streaming video decoding is used in online video streaming services, and audio streams are encoded and decoded in mobile devices for calls and also in streaming of music.

In addition to multimedia, streaming data is often encountered in other kinds of internet services as well. For example, efficient computation of analytics from search engine data can be implemented following stream processing paradigm. The total dataset may be quite large and a batch computing system computing over the complete data would provide results with high latency. A stream processing system, on the other hand, would be able to compute the analytics from the data as the data is being produced and update the analytics separately for each piece of data received. Google has developed the MillWheel framework for implementation of such analytics [3].

2.1.2 Stream Processing Platforms

A diverse variety of stream processing applications may run on different platforms ranging from phones to servers. Software stream processing systems may execute on arbitrary hardware, but to achieve good performance some

platforms are preferred over the others. Stream processing is well suited for designing applications for GPUs and DSPs. The modules of stream processing can often be executed in parallel allowing for efficient use of the multiple cores of GPUs, for example [27], makes use of GPUs for solving simulations involving partial differential equations using GPUs.

Digital signal processing applications are often designed in stream processing pattern and this is also reflected in the hardware making DSPs potentially powerful stream processing units [48]. Multi-core DSPs such as the Texas instruments TMS320C6678 used in this thesis have the potential for efficient stream processing, because they combine the DSP architecture suitable for stream processing with the parallel processing capabilities of the multiple cores.

In the software world, the expression power of stream processing paradigm has been recognized and many tools have been created for the development of stream processing applications. Examples of stream processing frameworks for distributed computing are Google MillWheel [3], Apache Storm [22], and Apache Spark Streaming [21]. StreamIT introduced in [65] is a programming language specifically for stream processing. Streaming Concurrent Collections [58] is a stream processing system based on the Concurrent Collections [12] programming model.

In addition to the software stream processing systems studied in this thesis, the increasing volume of streaming data has motivated research for hardware stream processors. Processor architectures that implement stream processing concepts in the hardware have been researched in Imagine [44] and Merrimac [15] projects at Stanford University.

2.1.3 Video Streams

Video streams are a prime example of streaming data. In the case of streaming service such as Netflix, the video data is downloaded from the server of the service provider and decompressed on the device of the consumer. In most use cases, the video streams are decompressed as they are downloaded and the complete video file is not necessarily stored on the device. Thus, stream processing is well suited for video stream decompression. The video streams are often accompanied by audio streams, which are processed similarly. [56]

The video conference use case is similar to the streaming video services but involves extra complexity with the compression of the raw data coming from the video camera and especially the requirement of low latency in the compression, decompression, and communication.

A large fraction of the growth of data creation can be attributed to video streams. More than billion hours of TV and movies are streamed through

the video streaming service Netflix [67]. Video streams are used for real-time communication through services such as Skype and Periscope. In addition to the number of users accessing the streams growing the bit rate of the streams is growing as well. The efficiency of video stream processing is thus a top priority in the industry.

2.2 Dataflow

Stream processing defines the high-level structure of how streaming data is to be processed but it does not define the stream processing methods in detail. On a high-level, the structure defined by dataflow models of computation corresponds well to stream processing. Looking at dataflow as the model of computation for stream processing is thus potentially useful. In this section the dataflow models of computation are examined.

2.2.1 Dataflow Models of Computation

Dataflow models of computation can be used to describe stream processing applications. In programs following a dataflow MoC, the computation can be described by a directed graph. The nodes of the graph are the computation kernels. The data being processed by the application is split into tokens, which flow from node to node along the directed arcs. The only means of communication between the nodes are the data tokens. This means the dependencies of a node are necessarily satisfied as soon as it has received its input tokens, and it may begin executing. The execution of the nodes is thus asynchronous. [48]

To allow asynchronous execution of the nodes, the tokens are buffered between the nodes. A common requirement of MoCs used in signal or stream processing is the capability of executing for an undetermined amount of time. Execution without determinate end is called unbounded execution and it means that the length of the input may be arbitrarily large. The token buffering combined with the required capability for unbounded execution leads to a possibility of unbounded buffer growth. Unbounded buffer growth is one of the main problems the different dataflow MoCs try to solve. The other main problem is the schedulability of the dataflow graph. If the graph has cycles, the model may deadlock when one of the nodes has an insufficient number of input tokens and cannot proceed execution. This keeps the nodes that come after it from receiving input and thus deadlocking the graph. [48]

The scheduling of dataflow graphs means determining when and where the nodes execute. The execution of the nodes is dependent on the availability of

data, which places restrictions on the possible schedules. The availability of data means that the node must have the specified number of tokens in each of its input queues in order to execute. Depending on the system there may be one or more cores available for execution of the nodes. For parallel systems the nodes may execute in parallel if a parallel schedule can be found that respects the data availability constraints of the graph. In order to generate efficient schedules, the scheduler must be aware of the overhead introduced by the communication between the cores. [46]

2.2.2 Synchronous Dataflow

One approach to solving the schedulability and the unbounded buffer growth of the dataflow MoC is fixing the production and consumption rates of the actors. These limitations are implemented in the family of dataflow models introduced by Lee and Messerschmit called synchronous dataflow (SDF) [46].

An SDF model imposes strict limits on the kinds of dataflow graphs allowed. The dataflow graphs in SDF models are fixed at compile time. No new actors may be added in run time to SDF graphs because all arcs of the model are fixed. The flow of tokens along the arcs is fixed as well, due to the fixed production and consumption rates of the model. Graphs created adhering to these limitations are fully schedulable at compile time and they execute the same way on each iteration. This means that the boundedness of the buffers is guaranteed and the deadlocking problems are avoided. Implementing such limitations reduces the expression power of SDF models compared to more lax dataflow models such as the dynamic dataflow, which is discussed in 2.2.3. [48]

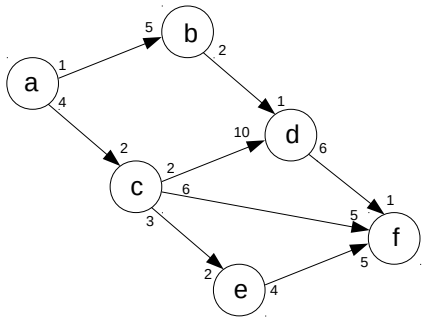


Figure 2.1: A SDF Graph from [2]. The production rate of each connection is marked at the beginning of the arcs connecting the nodes and the consumption rates are marked respectively at the end of each arc.

Arc	Buffer Size
ab	5
ac	10
bd	2
cd	10
ce	12
cf	30
df	6
ef	8

Table 2.1: The minimum required buffer sizes for the arcs in the SDF figure 2.1. The buffer sizes are presented as the maximum number of tokens the buffer fits. The buffers store the tokens that are waiting to be processed by the node in the receiving end of each arc.

The SDF MoC was specifically designed for digital signal processing applications. In domains such as digital signal processing the typical tasks can be presented as sequence of operations performed on an infinite stream of data, which makes them computable with SDF models. The SDF models provide good performance and guarantee deadlock-free execution and thus are a practical choice for DSP applications. [48]

Figure 2.1 presents an example acyclic SDF graph from [2]. The graph nodes are marked with letters from a to f and the production and consumption rates are marked at the ends of the arcs. The graph can be scheduled in many different ways, which yield different performance characteristics in terms of memory requirements of the buffers, latency, and throughput. The task of scheduling and allocating buffers for SDF graphs is non-trivial. Larger buffer sizes allow for more scheduling freedom but on certain targets such as FPGAs there are severe limitations on the number of registers available for the buffer allocation. The minimum required buffer sizes that guarantee the existence of deadlock-free schedule are represented in table 2.1. The buffer sizes were found using the algorithm from [2]. For some schedules that would yield improvements in throughput or latency, larger buffer sizes may be required.

Examples of practical implementations of the SDF MoC include the parameterized and interfaced synchronous dataflow (PiSDF) used by PREESM described in 4.3.2, the Ptolemy project [18] and the LUSTRE programming language [29].

2.2.3 Dynamic Dataflow

Synchronous dataflow may be a good fit for simple signal processing tasks such as signal filtering, but it is too restricted for efficiently describing more complicated algorithms. Multiple models of computation that share the basic structure of dataflow models have been developed for more advanced needs. Dataflow models of computation, which relax the constraints on token production and consumption are grouped under the name dynamic dataflow. Dynamic dataflow does not refer to a single model of computation but a group of models that differ in terms of expression power and analyzability. [8]

The dynamic dataflow models of computation do not constrain the number of tokens produced or consumed by an actor in a single firing. The actors may produce and consume different numbers of tokens on different firings. Relaxation of these constraints improves the expression power of the model but makes the analysis more difficult. In the class of dataflow models where these constraints have been lifted, the boundedness of the buffers and the deadlocks in cyclic graphs are undecidable [11]. Many dynamic dataflow models introduce other kinds of restrictions to replace the constraints on production of tokens. By limiting the types of the actors and available graph patterns dynamic dataflow models with decidable buffer growth and schedules are possible [8, 25]. Choosing which dynamic dataflow model to use is about finding the right tradeoff between analyzability and expression power for the specific application.

Dynamic dataflow is suitable for processing streams of complex data, such as compressed audio or video. A common example use case for DDF in the literature is the decoder of an MPEG video stream [8]. CAL actor language supports DDF MoC and it is used by the reconfigurable video coding working group to describe the MPEG encoder and decoder [7]. A more recent example of using DDF in practice is the TensorFlow machine learning framework developed by Google [1].

Chapter 3

Open Event Machine

This thesis investigates the performance of Open Event Machine (OpenEM) for stream processing. OpenEM is a programming framework for creating task parallel multi-core applications. Nokia Solutions and Networks (NSN) originally developed it for the networking dataplane [59].

In this chapter the OpenEM framework is introduced. To get a good view of the background of the framework, parallel computing is discussed in 3.1. The section about parallel computing contains descriptions of task parallelism, which is the form of parallelism implemented by OpenEM and event-driven programming, which is the model of concurrency used in OpenEM. After the context has been introduced two views to the OpenEM framework are provided. First, the OpenEM framework as specified by Nokia is described in 3.2. Second, the Texas Instruments implementation of the OpenEM framework for TMS320C6678 is explained in 3.3.

3.1 Parallel Computing

The single-thread performance of computer hardware grew rapidly, even exponentially, for many decades. This growth was reflected in software development where many software developers mainly focused on developing single-threaded programs. The main driver of single threaded performance was the increasing clock-speed of the CPUs. These increases in clock-speed slowed down around 2005 and hardware manufacturers turned more of their focus toward multi-cores. While parallel computing has been researched and practiced for decades, its importance has grown greatly after the proliferation of multi-core processors. [64]

The OpenEM model of parallelism is described in detail in this section. In subsection 3.1.1 a closer look is taken into task parallelism, which is the form

of parallelism implemented by the OpenEM framework. In subsection 3.1.2 event-driven programming, which is a model of concurrency used in OpenEM, is examined.

3.1.1 Task Parallelism

Task parallelism is a form of parallelism, in which units of work are distributed among multiple cores. The units of work in task parallelism are called tasks. The tasks consist of code and data in contrast to data parallelism where the same work is performed on multiple pieces of data. [30] Task parallelism can be flexibly used to construct parallel programs, as the different tasks can consist of work unrelated to each other. This makes task parallelism useful with handling irregular parallelism [5]. The coupling results in overhead compared to data parallel models, and thus the grain size of computation needs to be sufficiently large for efficient task parallelism [63].

Programming frameworks that provide task parallelism generally provide ways to declare dependencies between the tasks. By implementing task dependencies the programmer may avoid having to manually synchronize the threads of execution. The task scheduler is not aware of the contents of the tasks and thus accessing shared memory inside the tasks that do not have the appropriate dependencies leads to data races and acquiring locks may lead to deadlocks. For these reasons defining dependencies between tasks is the natural way of synchronization in task parallel programs. Introduction of dependencies between the tasks limits the possible orders of execution. With enough dependencies the only possible schedule may be executing the tasks sequentially. Therefore, the dependencies between the tasks have large impact on the performance of task parallel programs and therefore unnecessary dependencies should be avoided. [30]

Examples of task parallel libraries and programming languages include the Grand Central Dispatch by Apple [57], OpenMP implements tasks from version 3.0 onwards [5], Intel Thread Building Blocks [55], Cilk [10], .Net Parallel Extensions [49] and Habanero-Java [6].

3.1.2 Event-driven Programming

Along with parallelism, concurrency is becoming more and more common in modern computing. Reasons for increasing concurrency in programs are handling asynchronous IO and responding to external events such as user interactions.

In general, IO operations consist of three phases. In the first phase, CPU sets up the operation by deciding what is read or written and where. The

target hardware could be for example the hard drive of the device. In the second phase, the CPU waits for the operation to complete and has nothing to do with regards to the particular operation. In the third phase, the CPU is notified of the availability of results and begins processing them. Only phases one and three require active processing from the CPU. [24]

IO operations often take long compared to computation on the CPUs. In IO heavy applications multiple IO operations can be started concurrently and their results be processed in the order of completion. Such concurrent waiting increases the CPU utilization, as the CPUs are not spending time busy waiting for IO but instead processing the results of completed IO operations. [14]

A common way of enabling concurrency in programs is to split the execution of the program across multiple software threads. Using threads for concurrent programming is convenient because they allow interleaving IO and computation while preserving the appearance of a serial program. A separate thread could be spawned for each IO operation so that a non-blocked thread can execute while the IO thread is waiting. The use of threads has disadvantages; for example, they introduce concurrency even to sections of programs where it is not needed. Programming with threads requires explicit synchronization of the threads, which in practice yields data races and deadlocks. Spawning software threads consumes processing cycles and memory making full software threads often heavier than necessary for the task in hand. [14, 47]

Concurrent processing of IO and UI generated events does not necessitate the spawning of a thread for each operation. Another way of handling concurrent events is by registering an event handler with a central system that keeps track of completed operations. Such central system is commonly called the event loop. The event loop checks the completion of operations and calls the registered event handlers when the results are available. For example the Node.js [66] and Apple's Grand Central Dispatch [57] frameworks make use of such model of concurrency. The event-driven concurrent programming model leaves the mapping of event loops and details of how events are dispatched to its implementations. One or more threads can execute event loops and the events can even be dispatched from one thread to another depending on the implementation.

Dabek et al. argue for the use of events instead of threads to provide concurrency in IO heavy server environments. The benefits of events are that they provide comparable concurrency as threads in concurrent IO programs but are easier to program and tend to yield more stable performance under heavy loads. [14] This along with the other advantages of events in concurrent programming have generated enough interest that many event-

driven processing concepts have been developed. Here are some examples. Event-driven runtime with its own programming language called Eve has been developed by Fonseca et al. [20]. Node.js [66] implements event-driven processing. Majority of the UI libraries such as QT [9] are implemented in an event-driven pattern.

3.2 OpenEM Framework

The OpenEM framework provides a programming model for scalable and dynamically load balanced applications. OpenEM model of parallelism is task parallelism with the distinct feature that it separates the task code and data to different entities. The units of work in OpenEM are called events. The OpenEM events are scheduled using a dynamic scheduler that queues the events for one or more cores running the event loop. The Texas Instruments implementation of OpenEM uses hardware accelerators for the scheduling and inter-core communication.

The key components of the OpenEM programming model are events, execution objects, queues and the scheduler. OpenEM operates with so-called run-to-completion principle. The run-to-completion principle means that once an event starts executing it will not be interrupted. New events cannot be scheduled until a core has completed its current event, even if the events in execution had lower priorities than the events forced to wait. This implies limitations within which well performing applications must be designed. The program has to be divided to small events, so that the scheduler may efficiently distribute the computation and adhere to scheduling rules and priorities. Another limitation implied by the run-to-completion principle is that the application needs to be implemented lock-free for good performance. [60]

The OpenEM framework structure and functionality are loosely defined by the source code distribution available at [60]. Apart from the source distribution, there exists no public declaration of the structure and functionality of the framework. This lack of detail is explained by the OpenEM design principles, which are stated in the OpenEM source files at [60]. A selection of the design principles is presented here. The following list is not in any particular order.

- OpenEM has been designed to be *easy to implement on different multicore SoCs*.
- *Easy integration with modern hardware accelerators* is stated to have been a major driver in the OpenEM concept design.

- *All of the calls in the OpenEM API are multi-core safe* meaning no data structure gets broken if multiple cores make calls simultaneously. However the application designer needs to take the parallelism into consideration with regards to the application data structures and execution order.

- *The API attempts to guide the application designer towards portable architecture.*

- *The API is not defined for portability through recompilation.*

- *OpenEM does not implement a full software platform or a middleware solution.* OpenEM implements a driver-level layer of such a solution but can be used by the application directly for best performance.

Another factor explaining the loose definitions is how NSN views the status of its own public OpenEM implementation targeted for Intel DPDK. The disclaimer in the NSN source distribution at [60] states: "The implementation of OpenEM for Intel CPUs in this package should NOT be considered a 'reference OpenEM implementation', but rather an 'example of an all-SW implementation of OpenEM for Intel CPUs'." This helps put the differences between the OpenEM API specification and the TI OpenEM implementation introduced in 3.3 in context.

The next sections will introduce the key concepts of OpenEM framework as described in the OpenEM source distribution at [60]. The unit of work in OpenEM is the event described in 3.2.1. Events are sent to Queues (3.2.3). Queues are connected to Execution Objects (3.2.2). The scheduler (3.2.4) chooses an event from a suitable queue based on scheduling rules and schedules it on a core. Finally, an abstract example of using OpenEM for network packet processing is given in 3.2.6.

3.2.1 Events

In the core of the OpenEM framework design is the event. Events are simply pointers to application specific data structures that define the communication between the different parts of the application. The application creates events and tells the framework where the event is to be passed by enqueueing it in a queue. The scheduler observes the state of the queues in the application and schedules execution objects for execution when there are events available in the queues connected to the execution objects. [59]

OpenEM does not specify the event content to allow for the OpenEM implementations for different hardware platforms to organize the communication in the most efficient way available. The communication may be handled for example by hardware accelerated inter-core communication units or through simple shared memory. Usually events carry pointers to messages but they may also represent tokens for the scheduler with no data payload. [59] Event memory is managed by OpenEM, meaning the application allocates events from OpenEM and freeing the events returns the control of the memory to the framework [59].

OpenEM includes a fork-join helper called **event groups**. Event groups track the completion of the events sent to the group and send a notification event once a predetermined number of events have completed. The events belonging to an event group execute independently of each other and depending on other scheduling rules may execute on different cores. [59]

3.2.2 Execution Objects

The execution objects contain the application logic. The application is built from execution objects connected by queues. Multiple queues can be connected to an execution object and an execution object may execute on multiple cores if the queueing rules allow. There are multiple possible queue configurations, which allow the execution objects to be executed on multiple cores, the main method is through the use of parallel queues. [59]

The application logic for each execution object is implemented in three functions, namely start, receive, and stop. Execution object construction and destruction are handled in the application defined start and stop functions. An execution object can be scheduled on a core if there are events in the queues attached to it. When an execution object is scheduled for execution, one of the events in the queues attached to it is dequeued and passed to the receive function. Once the execution object has been scheduled on a core, it will run until the receive function returns. [59]

OpenEM passes a pointer to a user defined context when calling the receive function. The application designer has complete freedom over the execution object context contents as the OpenEM runtime only passes a user defined pointer. [59] If the execution object is connected to a parallel queue or multiple atomic queues, the execution object may execute on multiple cores simultaneously and thus the execution object context is shared between the cores. This limits the use of execution object context for content that has to be updated by the execution object, as the use of locks may cause the application to deadlock and is thus discouraged.

3.2.3 Queues

The communication between the execution objects of an OpenEM application is handled using events. The execution objects send events to first in first out queues, which are attached to other execution objects. The scheduler selects which queue to pick events from for execution based on queue properties. The properties of the queues that affect scheduling are queue priorities, queue types, and queue groups. [59] The queues define what can be executed in parallel and in which order the different execution objects are executed for a given event, therefore constituting the high-level schedule of the application.

A queue is always attached to a single execution object and it cannot be attached to anything else. The behavior of queues left unattached to execution objects is not defined in the OpenEM framework. An exception to this is the unscheduled queue, which cannot be attached to execution objects, or anything else. An execution object may have one or more queues attached to it. [59]

There are four types of queues: atomic, parallel, parallel ordered, and unscheduled [59]. Only one event from an **atomic queue** may be scheduled at a time. As the use of locks is discouraged, atomic queues are the preferred way to control the access to shared memory locations. Events from **parallel queues** can be scheduled on any core at any time according to other scheduling rules explained in section 3.2.4. Events received from a **parallel ordered queue** are scheduled like events from parallel queues but OpenEM will restore the event ordering before events are forwarded to other queues even if the processing of the events ends out of order. The **unscheduled queues** are special in that they are not scheduled automatically. The events are sent to unscheduled queues in the same way they are sent to the other queue types, but they need to be explicitly dequeued from the unscheduled queues. [60]

Queues belong to **queue groups**. Queue groups define the set of cores the events from the queues in the group can be scheduled on. Queue groups can be used, for example, to separate application layers, to control load balancing or to help guarantee that quality of service or latency targets are reached. [59] Queue groups can be modified at initialization of the queues or later during the execution [60]. Possibility for modifying the queue groups during execution can be used to implement dynamic load balancing of the computation.

3.2.4 Scheduling

The purpose of the OpenEM scheduler is to choose, which events to execute on which cores. The scheduler does this by dequeueing events from the queues in order defined by the scheduling rules. The scheduling rules are defined by the queue properties described in 3.2.3. The efficiency of the scheduler is an important factor to the overall performance of the OpenEM applications. [60] OpenEM implementations may target hardware platforms, which provide different facilities for the efficient implementation of the scheduler. For example, some hardware platforms, such as the TMS320C6678, have hardware accelerated communication facilities for the communication between the cores. To let the OpenEM implementations leverage the hardware as much as possible, the OpenEM API does not specify anything about the scheduler implementation. Thus the schedulers are always implementation specific.

Scheduling an execution object to a core is a two-phase process, where the scheduler selects which event to schedule on which core and the dispatcher passes one event at a time to an execution object. Each core executes the dispatcher. The dispatcher can run either in an OS thread or on bare metal. Dispatcher checks which queue the event was dequeued from and calls the receive function of the execution object associated with that queue. [59]

3.2.5 Error Handling

OpenEM provides two mechanisms for handling errors and exceptions: return values and an error handler. Most of the OpenEM API functions return status codes, which can be checked by the application to implement simple error handling. [60] Optionally an error handler can be registered with the OpenEM runtime.

The error handler is an application specified function, which will be called by the OpenEM implementation when the framework encounters an error or the application explicitly invokes an OpenEM error. The error handler can be used to centralize the error management. The application can register a global error handler and additionally one error handler for each execution object. If an error occurs in the scope of an execution object and there is an error handler registered with it, the runtime will call the error handler. If there is no error handler registered with the execution object, the runtime will check if there is a global error handler registered. If there is no applicable error handler registered, the runtime just returns an error status code from the API function. [60]

The error handler can modify the return value of the original API call.

This can be useful if centralized error handling is desired. The error handler can free the reserved memory, do the bookkeeping for the error and return a non-error status code from the original API call, leaving no need for the application code outside the error handler to check for OpenEM errors. [60]

3.2.6 An Illustrative Example

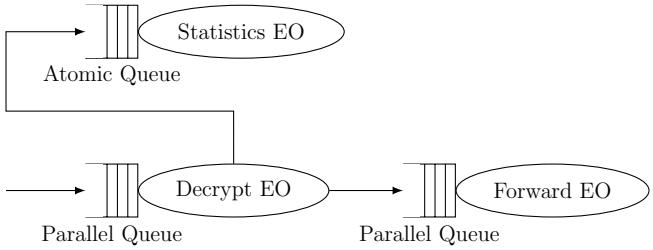


Figure 3.1: The OpenEM packet processing application.

An abstract example of a network packet processing application demonstrating the key capabilities of the OpenEM framework is presented in this subsection. In a network packet processing application, network packets are received and transmitted at variable intervals. The example application has to perform some actions for every received packet and decide if the packet has to be forwarded. When a packet is received it is moved into memory accessible by the processor executing the packet processing application and the application is notified of its arrival. The notification is handled by generating an event corresponding to the packet and enqueueing it in the first queue of the application. The event is generated in either hardware or software depending on the OpenEM implementation. A schematic of the queues, execution objects and the connections between them is presented in the figure 3.1

The queues of the application may be atomic or parallel. For this example the first queue is parallel. The events in the parallel queue may be scheduled on any core not executing an event currently. The scheduler adheres to the scheduling rules and its optimization parameters, trying to minimize the need to move execution objects from one core to another. The execution object the parallel queue is connected to decrypts the packet contents, which can be done in parallel for many packets at once. After decryption of the packet contents, the execution object decides if the packet needs to be forwarded or dropped. If the packet is forwarded the event is passed forward to a parallel queue connected to another execution object. The second execution object

updates the network packet headers and sends it forward. If the packet is dropped, the first execution object frees the event, otherwise it is freed by the second execution object.

The application keeps statistics about the number of packets decrypted, dropped, and forwarded. Since the tasks the statistics are collected for can be done in parallel, the statistics need to be updated in a thread-safe context. To avoid the use of locks in the application code the statistics events are sent to an atomic queue by the first execution object. The atomic queue is connected to the execution object that accesses the shared memory to update the statistics. The update execution object frees the statistics events after it has finished the update.

3.3 Texas Instruments Implementation of OpenEM

Texas Instruments (TI) provides an implementation of the OpenEM framework for its Keystone I family of multi-core digital signal processors. The OpenEM programming model is suitable for developing DSP applications that utilize the multiple cores of the DSP as well as the various accelerators available. There are many multi-core runtime systems available for multi-core DSPs, including embedded operating systems and operating system agnostic frameworks such as OpenMP. According to TI, the motivation for introducing OpenEM for their multi-core DSPs is that the existing options are not designed for heterogeneous platforms and lack support for features that are often required in DSP applications such as real time constraints [51].

The TI OpenEM library is delivered as a part of the TI Multicore Software Development Kit (MCSDK), which is available for download at [40]. MCSDK distribution contains source files for the OpenEM components defined in the OpenEM API but no source code is available for the scheduler. In the TI implementation of OpenEM all of the cores run an identical runtime component, which implements most of the framework functionality. The scheduler is not part not part of this component. The scheduler is deployed on a separate PDSP core.

This section describes the Texas Instruments OpenEM implementation version 1.0.0.2. The features specific to the TI implementation of the OpenEM framework are presented in the subsections 3.3.1 Scheduling, 3.3.2 Event Preloading, 3.3.3 Hardware Acceleration, 3.3.4 Cache Coherency and 3.3.5 Tracing. After the descriptions of the features, a look at programming with the TI OpenEM framework is taken in 3.3.6. Finally the state of the TI

OpenEM implementation is examined in 3.3.7.

3.3.1 Scheduling

The TI OpenEM implementation follows the scheduling rules defined by the general OpenEM queue properties as explained in 3.2.3 and 3.2.4. In addition to the scheduling criteria defined by the OpenEM framework, the TI OpenEM scheduler considers the execution locality. Locality criterion means that the scheduler tries to schedule events so that the cores can execute the same execution object consecutively as much as possible. [51] In DSP applications context switching between execution objects may become expensive if, for example, the program cache does not fit both the execution object ending its execution on the core and the execution object to replace it. By considering locality the scheduler may decrease the overhead caused by the context switching.

The Texas Instruments OpenEM documentation specifies two alternative scheduling modes, namely synchronous and asynchronous scheduling. The asynchronous scheduler is deployed on one of the PDSP cores of the Multicore Navigator described in 4.2.5. Scheduling requests sent by the c66x cores described in 4.2.3 trigger the scheduling operations of the asynchronous scheduler, but the actual event selection is performed on the PDSP core. The synchronous scheduler is deployed on the c66x cores. The synchronous scheduler interleaves performing the scheduling decisions with executing the execution objects. The synchronous scheduler is not available in the TI OpenEM version 1.0.0.2. [51]

The part of the framework responsible for scheduling consists of two parts: the scheduler and the dispatcher as explained in 3.2.4. The dispatcher is deployed on the c66x cores. The dispatcher checks if there are events scheduled for execution on the core it was called from. The calls to the dispatcher are non-blocking and are made from the application code, typically from within a dispatch loop. The dispatcher returns immediately if no events are available for dispatching. If an event is available the dispatcher will call the receive function of the execution object connected to the queue the event was received from. [51]

3.3.2 Event Pre-loading

The global event buffers, containing the event payload are located in the shared memory of the device. The buffers can be located in the MSMC RAM or in the DDR RAM depending on the space requirements of the buffers. If the event buffer resides in the shared memory when the event has

been dispatched, reading the buffer will require accessing the shared memory, which will cause read stalls. Using event pre-loading can minimize these stalls in the TI OpenEM implementation. [51]

Event pre-loading means that moving the event buffers to a core local L1 or L2 memory is started before the event has been dispatched. When an event with event pre-loading enabled is scheduled, one of the Packet DMA engines of the Multicore Navigator will begin to move the event buffers to the local memory. Event pre-loading is enabled separately for every event by the user. [51]

3.3.3 Hardware Acceleration in Texas Instruments OpenEM

TI OpenEM utilises Multicore Navigator extensively for the scheduler and inter-core communication. Multicore Navigator consists of features that enable hardware accelerated communication between the on-chip devices, including hardware queues and separate cores for queue management. It is described more detail in 4.2.5.

The main hardware components used by the OpenEM framework apart from the DSP cores and the different types of memory available, are the hardware queues and one of the PDSP cores of the Multicore Navigator. The hardware queues are used for implementing the software queues of the OpenEM framework. The exact mapping of the software queues to the hardware queues is not documented. The asynchronous scheduler is run on one of the PDSP cores. OpenEM does not use all of the Multicore Navigator queues or PDSP cores, leaving some of the Multicore Navigator resources are available for the user. TI OpenEM provides a mechanism in the framework initialization to divide the Multicore Navigator resources between the application and the framework. [39]

In OpenEM applications the computational work is done in the execution objects. The execution objects are always deployed on the DSP cores, but they may distribute parts of the communication on hardware accelerators as well. Deeper interaction of the OpenEM application and the hardware accelerators is possible with certain devices that support interfacing with Multicore Navigator. These hardware devices may produce and consume events without software intervention. [51]

The initialization of the OpenEM framework and the required hardware components is handled by the software abstraction layer provided with the example application in [39]. The software abstraction layer is a useful source of information for details about the use of hardware accelerators in OpenEM.

3.3.4 Cache Coherency

The Keystone DSP devices support caching of data in the higher levels of memory to the L1 and L2 local memories but they do not have hardware support for cache coherency [37]. In the general case the application programmer is responsible for managing the cache coherency of the application. TI OpenEM provides optional automatic cache coherency management for event buffers. The cache coherency mode can be set for each event individually. Automatic coherency management is also partially supported for queue contexts. [51]

3.3.5 OpenEM Tracing

TI OpenEM includes a built-in tracing feature that provides data about the runtime behaviour. The trace API is simple to use. The application has to register a trace handler with the OpenEM runtime and link the application with a trace-enabled version of the runtime library. [37]

The trace handler will be called every time the runtime or the application makes calls to OpenEM functions and it will be passed information about the type of the call made. The programmer should note that multiple cores may call the handler at overlapping times and therefore race conditions are possible. [37] Tracing can be used for example to track the number of events in each queue. This type of tracking may help debug problems with congestion and many other types of problems with OpenEM.

3.3.6 Programming with TI OpenEM

The OpenEM framework was not designed by NSN to be a full software platform or a middleware solution as explained in the OpenEM design principles available at [60], which are reproduced in 3.2. However the level of abstraction is suitable for DSP programming, which is done close to the hardware without an operating system. The low abstraction level means that the developer using TI OpenEM for developing applications for DSPs has to pay attention to the hardware initialization, memory allocation, and cache coherency for their own code as well as for the framework. [39]

The hardware initialization required by TI OpenEM is implemented in the example application described in [39]. The initialization code is not a part of the OpenEM framework, but the initialization code provided in the example application works without modification or with minor modifications for most purposes. The initialization layer provides an API for coordinating the hardware resource use between the application and the framework. [39]

The framework initialization documentation is incomplete and the application developer has to look at the source code of the example initialization layer to find out about the resource use of the framework if there exists a possibility of conflicting use.

3.3.7 State of TI OpenEM Implementation

The TI OpenEM library version 1.0.0.2 does not implement the complete OpenEM API as specified by the NSN implementation of OpenEM described in 3.2. The following listing presents the unimplemented features as listed in the OpenEM library version 1.0.0.2 release notes [38] and the TI OpenEM white paper [51].

- **Event Groups**, The TI implementation of event groups is functional but lacks the functionality to delete event groups.
- **Distributed Scheduling**, A synchronous scheduler is described in [51]. Distributed Scheduling is not part of the NSN specification of OpenEM.
- **Co-operative dispatcher**, The co-operative dispatcher is described in [51] provides services for suspending and resuming events. The co-operative dispatcher only works with the synchronous scheduler.
- **Execution Object context**
- **Parallel Ordered Queue**
- **Unscheduled Queue**

In addition to the limitations listed above the Queue Group implementation appears incomplete in the version 1.0.0.2. Queue Groups can be defined and modified as described in 3.2.3 but only one queue group can exist at a time. This could be a limitation of the hardware platform but there is no mention of the limitation in the TI OpenEM implementation documentation or the header files.

Chapter 4

Multi-core DSP in Video Stream Processing

The objective of this thesis is to understand the performance of the Texas Instruments implementation of Open Event Machine in stream processing. The objective is achieved by implementing a video stream processing application using OpenEM, measuring its performance and comparing that to a comparable, statically scheduled application. The comparable application is implemented using the PREESM framework. Both of the applications are instrumented in similar manner.

In this chapter the material and methods used in the experiment are introduced. The performance and behavior of OpenEM is evaluated using quantitative analysis. The analysis methods are explained in section 4.1. The hardware platform used in the experiments is the Texas Instruments TMS320C6678, which is described in 4.2. PREESM provides the tools for implementing the comparable workload. PREESM is introduced in section 4.3. The workload application processes video streams. An overview to video streams is provided in section 4.4. Finally, the Canny edge detection algorithm is introduced in 4.5. The algorithms computed in the stream processing application are part of the Canny edge detection algorithm.

4.1 Performance Analysis

To understand the behavior and performance of software systems, quantitative data about the system execution is required. Precisely what data is needed depends on the purpose of the analysis. A couple of popular methods for acquiring quantitative data about software systems are introduced in this section. First, an overview to the performance analysis of software

systems is provided in the subsection 4.1.1. Second, a closer focus to measuring software systems is taken in subsection 4.1.2. Finally, the analysis of the acquired data is described in subsection 4.1.3.

4.1.1 Analysing Software Systems

According to Jain [43] the performance analysis of software systems can be split into three categories, which are analytical modeling, simulation, and measuring of software systems. Analytical modeling and simulation use mathematical models for the analysis. Abstract mathematical models of systems can be constructed without access to the systems under study. To measure a software system, access to the specific system under study is always required. There are many methods for modeling and simulation of software systems, but in most cases they require less work to implement than the actual system under study. These properties make the analytical modeling and simulation attractive for explorative study of software systems that do not exist yet. [43]

The key difference between analytical modeling and simulation is the notion of time present in simulation. Analytical modeling solves the system state at a fixed point in time, in contrast to simulators where the system state is computed iteratively at multiple points in time. [43]

Performance analysis is used for many different purposes. For example, performance analysis can be used to help choose the best performing hardware platform for certain application, or to explore different configurations of an application. Successful analysis requires careful experiment design. First step to successful performance analysis is method selection. Using simulation or analytical modeling can yield results quickly, but they are not as accurate as measuring the real world system. [43]

The execution of a computer program is a complex interaction of hardware and software components and thus the number of parameters of the analysis grows large. Factors are parameters that are varied in the analysis. Factors are selected from among all parameters of the system [43]. Every factor increases the time it takes to complete the analysis and only few of the parameters are relevant for the result of the analysis. Thus, the factor selection requires clear goals for the analysis and a good understanding of the problem space so that the most relevant factors are chosen. The factor selection of the experiments conducted in this thesis is discussed in section 5.5.1.

4.1.2 Measuring Software Systems

In this thesis, a measurement system consisting of workload, applications, and execution hardware is constructed and its performance is measured. In this subsection a closer look at measuring software systems is taken.

Successful comparison of software systems requires meaningful and reasonably accurate measurements of the systems under study. Measurements are obtained by monitoring the system while it is being subjected to a particular workload [43]. Often the monitored applications are built for the comparison purpose only and therefore any workload they are subjected to is an approximation of the real world workload that would be processed by their real world application counterparts. These approximate workloads are called synthetic workloads. The use of a synthetic workload gives more control over the test conditions and most importantly makes the experiments repeatable. [43]

Synthetic workload creation requires care because it needs to mimic its real world counterpart with high accuracy to provide any useful information. Performance analysis is often conducted to understand the performance or feasibility of a software component or a system that does not exist yet. In such situations synthetic workloads need to be used out of necessity.

The workload is the target of the measurements but it does not define the exact measurements that are to be conducted. Selection of metrics is equally important as the selection of the workload for successful analysis. The best metrics for a given analysis are determined by what is the goal of the analysis. [43] For example, measurements can be used for comparison of throughput of two comparable software systems. In that case a good measure of throughput is clearly needed.

The software measurement tools are called monitors, which can be implemented both in hardware and in software. Monitors are classified to software monitors, hardware monitors, firmware monitors or hybrid monitors depending on the implementation level of the monitor. The implementation level of the monitor affects the level of events that are convenient to measure with it. For example, hardware monitors can monitor the state of registers and hardware counters but have difficulties in observing the status of software constructs such as the execution of functions. The software monitors on the other hand can be used to monitor the status of software components but gathering information about the status of the hardware is difficult and in some cases impossible. [43] For example, it is very complicated to determine whether a memory operation hit a given level of cache using software alone but many hardware platforms offer hardware counters to monitor the cache hits and misses.

4.1.3 Analysis of the Measurement Data

The goal of the performance analysis is to get actionable results about the systems under study. The data obtained from the models or measurements is not in itself enough for making well-grounded decisions. The models and measurements may yield millions of values for the observed variables and the analyst needs to decide how to best represent the data so that the phenomena behind the data are explained [43].

Statistical methods are used to analyze the numerical data and expose the causation and correlation between the factors and the results. Often in the literature, simple statistical tools such as mean, mode, and standard deviation are used to represent the data in only a few numbers. Such simple statistics are enough if they capture the relevant information about behavior of the system. For example, if the goal of the analysis is to compare the latency of two non-realtime systems, an average of the latency and its standard deviation over a reasonable measurement period can be enough. A more thorough look at the statistical tools is provided in [43].

The results of the analysis are often easiest to understand when presented in graphical form. Graphical representations of the data such as histograms, line charts, and bar charts are commonly used. These graphs are very generic and used in many fields to present many kinds of data. There are also more domain specific visualizations of data such as the Gantt charts used to represent schedules in computer context and elsewhere. The visualizations of data are designed to be faster to understand than the corresponding numerical views to the same data but they have their limitations. The graphical representations are inaccurate and if they are not carefully prepared they may present a biased view to the real data. Due to these limitations the visualizations should be prepared carefully and the numerical data they are based on should be also made available. [43]

4.2 Texas Instruments TMS320C6678

This section describes the hardware platform used in the experiments. The experiments were conducted on a Texas Instruments TMS320C6678 multicore digital signal processor. First, the selection of the TMS320C6678 as the hardware platform for the experiments in this thesis is explained in subsection 4.2.1. Second, an overview of the hardware platform is given in subsection 4.2.2. After the overview, the key features of the platform are described in subsections 4.2.3 C66x DSP, 4.2.4 Memory Hierarchy and 4.2.5

Multicore Navigator.

4.2.1 Selection of the Hardware Platform

This thesis investigates stream processing with Open Event Machine on a multi-core DSP. The Texas Instruments Keystone I family of multi-core DSPs has an advanced support for multi-core programming, including a Texas Instruments implementation of OpenEM [19]. The multi-core programming support has not always been a design priority of multi-core DSPs, for example, the TMS320C647x DSPs were looked at as, "multiple single-core DSP in a single package" by the programmers [51].

The hardware platform used in the experiments in this thesis is the Texas Instrument TMS320C6678. The TMS320C6678 is a fixed and floating point digital signal processor based on the Texas Instruments Keystone I architecture [35]. The Keystone I architecture was selected because there exists an OpenEM implementation for the processors implementing the architecture. Out of the Keystone I devices TMS320C6678 was selected because Advantech provides an evaluation module TMDXEVM6678L for the specific processor, which makes the experimentation more straightforward than building an evaluation platform from scratch. Another benefit of the TMS320C6678 is that the PREESM rapid prototyping tool for dataflow applications has support for it [54].

4.2.2 TMS320C6678 Overview

The TMS320C6678 is based on the Keystone I architecture. The Keystone I architecture specifies a set of hardware elements which enable integration of C66x DSP cores, application specific co-processors, and IO [35]. The Keystone I hardware modules and their connections are presented in the figure 4.1. The points of interest in the figure in the scope of this thesis are the C66x CorePac cores depicted in the middle of the figure, the memory subsystems and its components in the top-left of the figure, and the Multicore Navigator depicted in the right edge of the figure.

In the Keystone I architecture there are multiple ways for the C66x cores to communicate with each other, the memory, and the peripherals. The methods of communication of specific interest for the experiments in this thesis are communication through shared memory discussed in subsection 4.2.4 and communication through packet based communication manager Multicore Navigator introduced in subsection 4.2.5.

The development board used for development of the experiment applications and measurements is an Advantech TMDXEVM6678L. The board

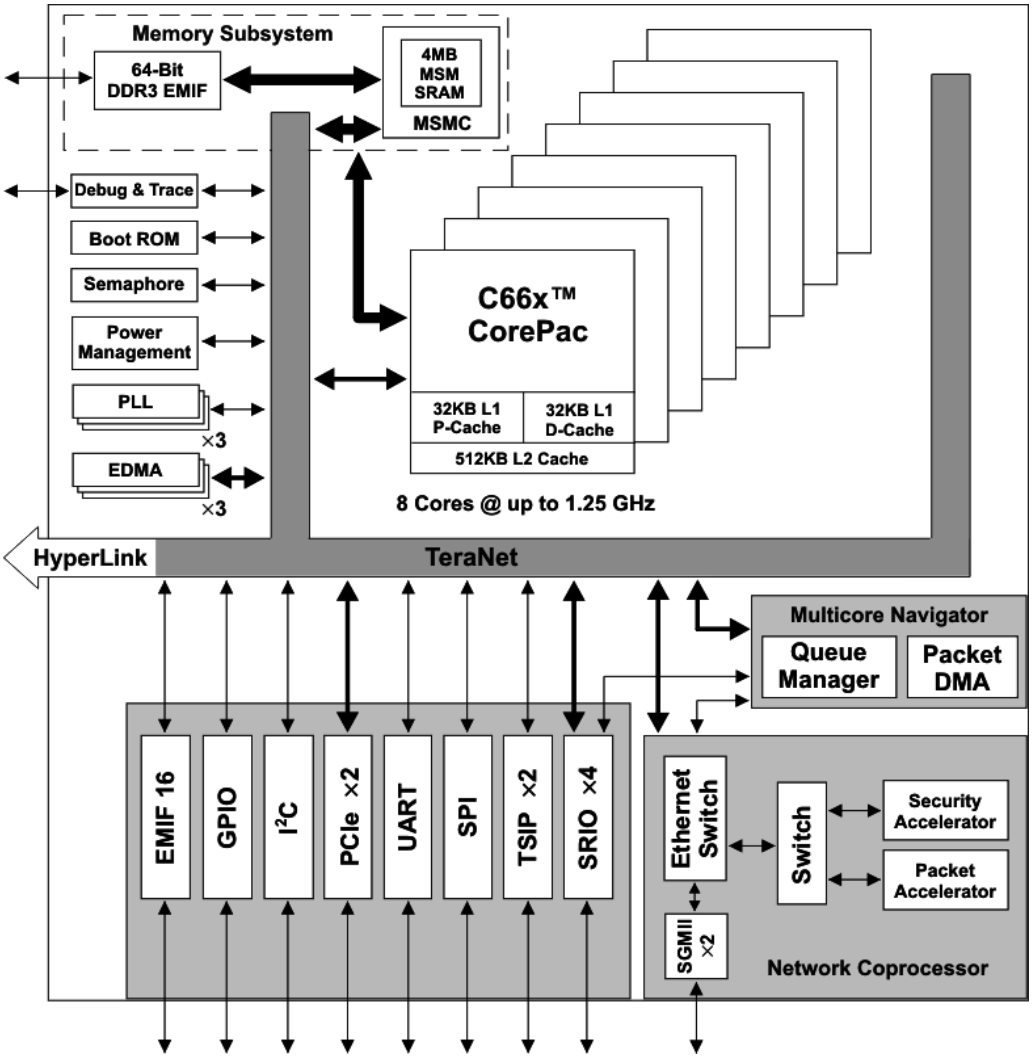


Figure 4.1: High-level schematic of the TMS320C6678 architecture. Figure from [35].

is an evaluation module for the TMS320C6678 multi-core DSP. The evaluation module has 512 megabytes of DDR3 memory which is sufficient for stream processing applications studied in this thesis. An important feature of the evaluation module is the emulator module with USB connectivity. The emulator together with the Code Composer Studio IDE (CCS) make the programming and debugging the experiment programs for the DSPs simple. [36] CCS version 5.2 is distributed with the hardware evaluation module and was used for development of the experiments in this thesis.

4.2.3 C66x DSP

The hardware platform used in this thesis is the TMS320C6678 multi-core digital signal processor. The processor consists of eight C66x DSP cores. The C66x is based on the Texas Instruments TMS320C66x instruction set architecture. The TMS320C66x is a very long instruction word architecture, which allows for high amount of instruction level parallelism. The C66x has a total of eight functional units, which operate in parallel. This means that the C66x can dispatch up to eight instructions per cycle. The instructions dispatched in parallel move through pipeline stages simultaneously. Pipelining helps eliminate CPU stalls while waiting for memory operations or other CPU instructions taking multiple cycles complete. [34]

Keeping the utilization of the wide pipeline high, the CPU needs to have enough registers to prevent excessive memory access stalling, for this purpose the CPU has 64 32-bit general purpose registers [34]. As a high-end processor the C66x has native support for 32-bit and 64-bit floating point instructions and capability of clock speeds up to 1.4 GHz [34].

4.2.4 Memory Hierarchy

The TMS320C6678 contains a multi-layer memory hierarchy, which can be configured to a large extent by the user. The memory hierarchy in the device consists of L1 and L2 memories for each core, Multicore Shared Memory (MSM), and additionally external memory provided, for example, by the evaluation module. In the figure 4.1 the memories are placed inside the subsystems they are part of, MSM can be found in the upper-left corner as part of the memory subsystem.

Each c66x CPU has 32 KB level 1 program cache (L1P) and 32 KB level 1 data cache (L1D). Each CPU also has 512 KB of level 2 cache. Both of the L1 caches and the L2 cache can be found in the figure 4.1 as part of the C66x CorePac box. Initially after bootup both L1P and L1D are configured as cache but they can be reconfigured as addressable memory by software. The L2 memory is always configured as addressable memory after reset but can be configured as cache by software. [35] L2 SRAM addresses are always cached with L1P and L1D whereas external memory addresses are configured noncacheable by default [33].

Configuring the state of the L1 and L2 memories as well as other memory configurations can be handled with software [34]. CCS automatically handles a lot of the memory mapping needed for applications and provides tools for creating custom configurations.

In PC hardware cache coherence is usually handled automatically. How-

ever, in c66x that is not the case. Each c66x core maintains cache coherence between its L1 caches and the L2 cache automatically but programmer needs to manage coherence in most other cases. For example, if caching is enabled for an external memory region shared by two cores, explicit cache coherence operations need to be performed before each core can read from or write to the shared region [33].

The evaluation module has 512 MB of DDR3 memory [36]. The memory in the evaluation module, as any external memory in other hardware configurations, is accessible through the Multicore Shared Memory Controller (MSMC). The MSMC itself contains 4096KB of shared memory accessible by all cores. In the figure 4.1 the memory subsystem contains the EMIF link to the external memory and the MSMC.

4.2.5 Multicore Navigator

The multi-core programmability of the TMS320C6678 makes it interesting for this thesis. The device is designed to allow simple cooperation of the DSP cores and provides the required hardware support for that purpose. The core features enabling the multi-core programmability are grouped under the name of Multicore Navigator.

Multicore Navigator is the name for a collection of features in Keystone I and II devices, which enables hardware-accelerated, packet-based communication between on-chip devices. Texas Instruments claims the use of specialized hardware for on-chip communication results in significant performance gains when implemented carefully. The design goals stated for the Multicore Navigator in [41] are minimizing host interaction and maximizing memory use efficiency. [41]

In Keystone I devices such as the TMS320C6678, the Multicore Navigator provides a hardware queue manager, a special direct memory access for different subsystems called Packet DMA (PKTDMA), and multi-core host notifications via interrupts. [41] The Texas Instruments OpenEM implementation heavily utilizes the features provided by Multicore Navigator.

The Queue Manager in Keystone I architecture devices is a hardware module that manages 8192 queues. Packets are queued and dequeued from the queues by the applications. The Queue Manager is responsible for accelerating the packet communication. In addition to the Queue Manager the Queue Management Subsystem contains two Packed Data Structure Processors (PSDP), which perform tasks related to the queue management and packet communication. For example, the PDSP processors can be used to perform accumulation of packets. The accumulation program is given a list of queues to poll. Whenever it finds a descriptor from one of the queues

it is watching, it will pop the descriptor and place it in a buffer provided by the application. After a pre-determined number of descriptors, or after reaching its time limit, the accumulator program notifies the host processor about the descriptors in the buffer via an interrupt. Use of such firmware offloads the burden of queue polling from the host processors. [41] The TI implementation of OpenEM provides its own firmware for the PDSP cores, which is utilized by the OpenEM runtime for event scheduling [51].

The PKTDMA is a special DMA utilized by the Multicore Navigator to transfer packet buffers between memory locations. When a packet is sent to a queue the PKTDMA reads the address of the data to be transferred from packet descriptor, transfers the data in one or more data moves, and writes the pointer to the data queue specified as the receiver of the packet. The PKTDMA is useful because it allows the program running on a PDSP core to move data in the memory without interrupting the host processors. [41] OpenEM uses PKTDMA to move event buffers to the caches of the core, which is about to receive the event [51].

4.3 PREESM

PREESM is a rapid prototyping framework for multi-core development. For understanding the OpenEM framework, a way to construct comparable programs with statically scheduled multi-core runtime was needed. PREESM provides a way to quickly construct multi-core applications for PC as well as for the Texas Instruments multi-core DSP used in the experiments. In this chapter the PREESM framework is introduced. First, an overview of the framework is given in subsection 4.3.1. Second, the framework overview, the internal representations used by the framework are described in subsection 4.3.2. Third, scheduling in the PREESM framework is explained in section 4.3.3. And finally, the memory allocation and code generation in PREESM are explained in section 4.3.4.

4.3.1 PREESM Overview

PREESM is a collection of tools for rapid prototyping multi-core applications. The tools include a graphical editor for the hardware and the software models, code generators for multiple hardware platforms, and an automated generator for fixed schedules. The PREESM tools are used through the Eclipse IDE based environment available at [31].

In PREESM, prototypes of applications are constructed by combining hardware and software models with manually created source code and auto-

matically generated schedule. The software model used in PREESM is based on dataflow models of computation and it is discussed in detail in subsection 4.3.2. To create a software model in PREESM the application is divided into actors. The actors contain manually created code, which is often written in side-effect free style to enable parallel execution of the actors, but the framework does not enforce this. The graphical editor for the software model allows the user to create actors and connect them together using first in first out queues. [31]

The model of the target hardware platform is created using a similar graphical tool as the software model. The hardware model is described in subsection 4.3.2. PREESM parses the graphs of the models and creates a static schedule for the executable. The schedule, the actor implementations, and the graphs are inputs for the code generator, which creates the multicore executable. [54] The graphical tools allow the PREESM user to create dependencies between the actors. They also allow a more fine grained control over the code generation by setting estimated actor execution times and selecting the cores on which each actor is allowed to execute.

4.3.2 PREESM Internal Representations

PREESM applications are created by combining inputs of two different internal representations and manually created source code. The PREESM internal representations are described in this subsection. First, a look is taken at the PREESM algorithm representation PiSDF and second, to the PREESM hardware model.

In PREESM the applications are modeled using a synchronous dataflow based representation of the software called parameterized and interfaced synchronous dataflow model of computation (PiSDF) [54]. PREESM provides graphical tools for editing the dataflow diagram. An example of a dataflow diagram created in PREESM is presented in figure 4.2. The biggest benefit of using synchronous dataflow based model of computation is that deadlock free static schedules can be generated from them [54]. The static schedules generated by PREESM are guaranteed to be deadlock free [31].

PREESM uses an extended version of SDF called PiSDF [54]. PiSDF extends SDF by providing hierarchical graphs based on interfaces and by introducing parameterized actors. An actor in PiSDF can be replaced by a subgraph, which has input and output interfaces. The interfaces insulate the hierarchy levels in terms of schedulability analysis, meaning that schedules can be generated for the subgraphs without knowledge of the higher-level models. The parameters are used to configure the production and consumption rates of the actors. [17]

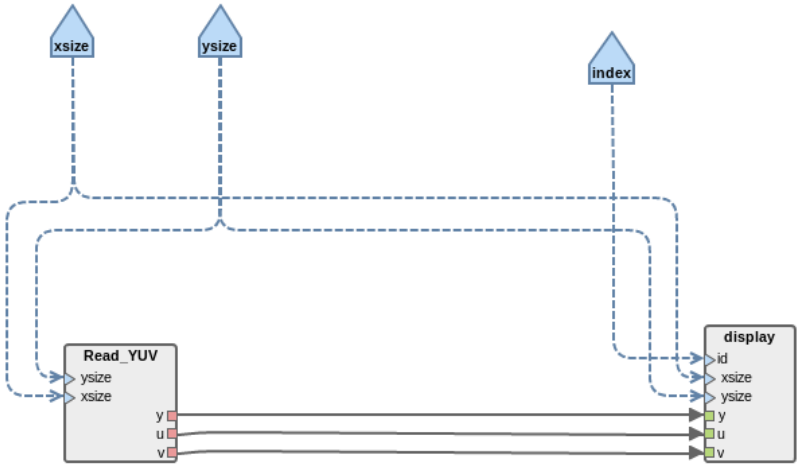


Figure 4.2: A simple dataflow diagram created with PREESM. On the left, the *Read_YUV* actor acts as the data source for the algorithm represented by the graph. On the right, the *display* actor acts as the end point of the algorithm. The pentagons at the top of the figure are configuration parameters of the model.

An example of a PiSDF graph is presented in figure 4.2. The parameters of the PiSDF model are presented at the top of the figure as pentagons connected to the actors with dashed lines. The actors have input and output ports for parameters and data. The ports define the input and output interfaces of the actors. The actual data paths are the arcs connecting the actors in the bottom of the figure.

The PREESM code generation is aware of the target hardware platform. PREESM uses an internal representation called the System-Level Architecture Model (S-LAM) [53] to describe the target architecture. S-LAM is designed specifically to provide architecture models of high abstraction level for rapid prototyping purposes. S-LAM has good expressive power and it is suitable for modeling heterogeneous architectures as a S-LAM can contain different types of computational resources and communication links. [53] In the PREESM context however, the typical S-LAM models are simple. For example, the S-LAM representing the TMS320C6678 used in experiment 6.1.1 is quite simple consisting of eight c6678 cores connected through shared memory.

In PREESM the user inputs the speeds of the hardware components of the S-LAM and the speeds of the connections between them. This information

is used by the PREESM scheduler to generate a static schedule, that takes communication delays and the component speeds into account, trying to minimize the time spent waiting for communication. [53]

4.3.3 PREESM Scheduling

Scheduling multi-core applications is not a trivial task, because the execution on a single core is sensitive to what other cores are doing. An unsuccessful schedule may result in the application deadlocking or other kinds of problems. The approach PREESM takes to overcome the complexity of scheduling multi-core applications is the use of a highly analyzable model of computation PiSDF and the creation of a static schedule based on on the model. PREESM uses the *List* and *Fast* scheduling methods described in [45] for generating a static schedule for multi-core platforms. Using these methods PREESM is able to generate a static schedule and guarantee that it is deadlock free.

The reason why multi-core applications are created in the first place is the performance increase available through parallelizing the execution. To get a performance increase from parallelizing the application the scheduler must be able to efficiently utilize the multiple cores. The critical measure of performance can be throughput or latency. The designer of the PREESM scheduler has made the decision to focus on so-called latency dominated systems [54]. Latency dominated system is defined in [26] as a system where respecting the latency constraint placed upon the system automatically guarantees the satisfaction of the throughput constraint. In other words, in the systems PREESM is designed for, each iteration of the application has to fulfill some latency constraint. Fulfilling this latency constraint yields satisfactory throughput. This frees the PREESM scheduler from considering throughput in the scheduling decisions and yields a simpler schedule. The iterations of the algorithm are not interleaved, because the scheduler only considers the latency of the execution. Instead, all cores are synchronized between the iterations with a barrier. [54]

The execution of the different iterations of the algorithm is not interleaved by the PREESM scheduler. However, intra-iteration interleaving is supported. In intra-iteration interleaving the actors belonging to the same iteration of the algorithm are executed in parallel. [54] An example of the intra-iteration interleaving is given in the software pipelining tutorial available at [31].

The PiSDF model of the application defines the dependencies between the actors. These dependencies are enough to generate ordering of the actor firings, but to get an efficient schedule the execution times of the actors

are needed. The user provides the estimated execution times for each actor. The time required for communication between the actors is approximated from the amount of data to be transferred and the speed of the communication links. The data amounts are defined in the actor model and the communication speeds in the S-LAM. With the actor ordering and the timing information, the framework generates a static schedule. [54] PREESM framework visualizes the generated schedule with a Gantt chart. An example Gantt chart is presented in figure 5.2.

4.3.4 PREESM Memory Allocation

The output of the PREESM framework is a multi-core executable. Before the schedule and the user created actors can be combined into an executable, PREESM has to determine how much memory is required for the communication between the actors. The sizes of the data transfers are determined by the input and output interfaces of the actors and they do not change during the execution. Because the size of the buffers is static, the buffers can be allocated before the execution. [16]

The data in the communication buffer is considered reserved from the beginning of the execution of the producing actor to the end of execution of the consuming actor. This choice is made to enable custom production rates of the actors. Due to this choice the actors cannot reuse their input buffers as their output buffers. [16] This approach keeps the resulting executable simple but it results in extra copying of buffers between the actors.

In the simple case where the producing and the consuming actors execute on the same core and are not dependent on any other actors, simple copying of buffers is enough. However when the producing actor executes on a different core than the consuming actor, the cores need to synchronize before the consuming actor can start to execute. If the user provided estimates for the execution times of the actors are accurate, the synchronizations should not cause long delays in the execution. [54]

With the memory allocations computed, the framework can create an executable. The structure of the complete application follows the template provided by the framework. The memory allocations are handled in the initialization. After the initialization, the cores enter loops where the static schedule for each core is laid out as interleaved calls to actor implementations, memory operations to move the data between the buffers, and synchronization barriers.

4.4 Video Streams

Many different video stream formats are needed to capture, transfer, and store videos. A lot of space is required for storing of raw video material and a high bandwidth is required for its transmission. Despite the high requirements, the part of the video stream that is being processed has to be in unpacked format to enable efficient processing. In this section two standards related to video streams are explained: first, the YUV color specification and second, the CIF frame resolution format.

4.4.1 YUV Format

YUV color space is used in encoding colors of images and videos. The YUV color system is used, for example, in encoding the colors of television broadcasts using the PAL or NTSC systems. The YUV color space is designed with the human perception in mind, meaning the channels are selected so that compression artifacts and other errors are more likely masked by human perception. This allows for reduced bandwidth compared to RGB encoded colors. The Y channel corresponds to the luminance of the image. Luminance means the perceived brightness. In a black and white image or video only the Y channel is used. The U and V channels are called the chrominance channels and they encode the color component of the image. [42]

In digital media YUV term is commonly used to refer to YCbCr, which is a way of encoding RGB color information. Y' (Y prime) is called luma and is distinct from the Y channel of the analog YUV system. The luma channel is a non-linear encoding of the light intensity. Cb and Cr are the blue-difference and red-difference components respectively. It is common to store the luma channel at a higher resolution than the chroma channels to save bandwidth in a process is called chroma sub-sampling. Humans are more sensitive to the brightness of the image than the color of the image and thus the lower resolution of the chroma channels causes less noticeable artifacts. [42]

4.4.2 Common Intermediate Format

Common Intermediate Format or CIF is a standardization of the horizontal and vertical resolutions of pixels in video signals. The CIF pixels are non-square with an aspect ratio of approximately 1.222:1. The standard defines multiple resolutions such as the CIF 352x288 and QCIF for quarter CIF 176x144. CIF resolutions were designed to be easily convertible to the PAL and NTSC systems. [52]

4.5 Canny edge detector

Edge detection is an important tool in image processing and computer vision. Some image processing and computer vision algorithms operate on detected edges. The quality of the detected edges is important for the algorithms using them as input. In his 1986 paper John F. Canny [13], lays out the mathematical criteria for successful edge detection and presents an algorithm, which achieves decent edge detection performance. The algorithm is suitable for implementation on DSPs. The Canny edge detection algorithm consists of five steps presented in the following list.

1. Noise reduction
2. Finding the intensity gradient of the image
3. Non-maximum suppression
4. Double thresholding
5. Edge tracking by hysteresis

In the Canny edge detector the image is first filtered with a Gaussian filter to reduce the amount of noise in the image. Second, the changes in the intensity of light in the image are detected using an image gradient operator such as the Sobel operator. The third step improves the accuracy of the edge detection by suppressing all but the strongest responses to the detected edges, in practice "thinning" the edges. The fourth step classifies the edge pixels to three classes separated by empirically determined threshold values. The pixels with gradient value above the high threshold are marked strong pixels and the pixels with gradient value below the low threshold are suppressed. In the fifth step the remaining weak pixels with gradient values below the high threshold are preserved or suppressed according to the presence of strong pixels in their neighborhood. Detailed description of the algorithm is presented in the original paper by Canny [13], information about implementing a Canny edge detector is available in [28], and comparison of its performance to other edge detectors can be found in [50].

In this section the phases of Canny edge detector are described in the order they are used in Canny filter. First, the Gaussian filter is introduced in subsection 4.5.1. The Sobel filter is looked at next in subsection 4.5.2. After the Sobel filter, the edge responses are pruned in three phases. The three phases are presented in subsection 4.5.3.

4.5.1 Gaussian filter

Gaussian filtering is used for multiple purposes in digital image processing. In the Canny edge detector the Gaussian filter is used to reduce noise in the processed images. The Gaussian filter works by convolving a Gaussian function with the input signal. Gaussian function is non-zero everywhere, which means it would theoretically require an infinite convolution window. Since the function decays rapidly, it is often reasonable to truncate the function and use small windows. [28]

Calculating the convolution with a truncated function means in practice that every pixel in the filtered image has an intensity value computed by taking a weighted average of the neighboring pixels in the input image. The weights are pre-calculated from the Gaussian function, giving the highest weight to the pixel in the center of the window. The Gaussian filter displayed in figure 4.3 was calculated with $\sigma = 1.3$. The filtered image has a smoothed appearance compared to the original image.

$$B = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * A$$

Figure 4.3: Convolution with a Gaussian kernel computed with $\sigma = 1.3$. Asterisk denotes the convolution operation.

4.5.2 Sobel filter

The actual edge detection in the Canny edge detector begins with determining the changes in the intensity of the image. Applying the Sobel operator to the input image does this. The Sobel operator is a discrete differentiation operator. It consists of two 3x3 kernels, which are convolved with the image to approximate the derivatives. The two kernels represent horizontal and vertical changes. At each point in the image the resulting gradient approximations are combined giving an approximate gradient magnitude. [28] The convolution operations are presented in figure 4.4.

4.5.3 Canny Edge Pruning

Canny edge detector is designed to detect edges accurately and as unambiguously as possible. To make unambiguous detections, each edge in the input

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * A$$

Figure 4.4: The 3x3 Sobel kernels used in the application to compute the gradient approximation. G_x is the horizontal gradient approximation at given pixel and G_y is the vertical gradient approximation. The asterisk denotes the convolution operation.

image should produce only one edge response. For this purpose Canny edge detector employs **non-maximum suppression**. Non-maximum suppression works on the gradient image that was calculated in the previous phase. For each pixel in the gradient images, non-maximum suppression compares the gradient value to the adjacent pixels in positive and negative gradient directions. The pixel is preserved only if it has the largest gradient value compared to its neighbors. [28]

The non-maximum suppression outputs an image where the edge responses have been pruned to single response per edge, but there still remain gradient pixels that correspond to noise or other uninteresting variations in the image. The detection quality is further improved by applying two thresholds to the output of the non-maximum suppression phase. In the literature this phase is called **double thresholding**. If the gradient value of a pixel is higher than the high threshold, the pixel is marked as a strong pixel. If the value is lower than the high threshold but higher than the low threshold, the pixel is marked as weak pixel. Pixels with gradient values lower than the low threshold are automatically discarded. The threshold values are determined empirically. [28]

In the final phase of the Canny edge detector, called **edge tracking by hysteresis**, the strong pixels are used to determine which of the weak pixels to keep and which to discard. A weak pixel is preserved if there is at least one strong pixel among its eight neighboring pixels. If there are no strong pixels in the neighborhood the weak pixel is discarded. [28]

Chapter 5

Experiment Construction

This chapter describes the implementation of the applications studied in the experiments. The workload application was inspired by the Canny edge detector. Two versions of the workload were implemented and both were instrumented for measurement. Three experiments were designed for comparison of the applications.

To begin this chapter the general idea of the workload application is introduced in 5.1. Then, the PREESM implementation of the application is described in 5.2. Next, the OpenEM implementation is explained in 5.3. After the applications have been described, the instrumentation explained in 5.4. Finally the experiments introduced in 5.5.

5.1 Filter Application

The objective of this thesis is to study the suitability of OpenEM framework for building stream processing applications. Video streams make up an increasing proportion of the data being transferred in the Internet. The processing of video streams requires a lot of computing power, as the streams have increasingly high bit rates and they have to be compressed for transfer and storage. The high performance requirements of processing video streams make them interesting for research and for this reason they were selected as the stream format used in this thesis.

Three requirements for the workload applications were specified to ensure that the results of the experiments help answer the research problem. The requirements are presented in the following list.

1. Variable input bit rates
2. Comparability the PREESM and OpenEM applications

3. High analyzability

Variable input bit rates were required of the workload applications to make it possible to study the performance of the OpenEM framework in processing dynamic workloads. Varying the bit rates of the video stream inputs is simple as the frame size of the stream is easy to change and it doesn't affect the algorithms used in any unforeseeable way. **Comparability of the PREESM and OpenEM applications** was required so the PREESM workload could be used as the baseline implementation. Last requirement for the applications was that they needed to be **highly analyzable** in order to enable the study of OpenEM performance. An application satisfying these requirements was designed using the PREESM Sobel example at [32] as the starting point and adding another processing component to it.

It is important to notice that the actual performance of the filter application in its nominal video filtering task was mostly disregarded. For example, the data is copied to a new buffer in every processing stage and the filter algorithms are not optimized for the specific platform. Therefore, comparing the performance of the implemented applications to real world implementations of similar applications is not useful.

The idea for using Sobel and Gauss filters in the same application came from the Canny edge detector. However, the filters in the workload applications are executing independently of each other on separate video streams, unlike how they are connected in the Canny edge detector.

5.2 PREESM Filter Application

An actor network that represents the video filter application is constructed in PREESM. The final PiSDF model of the PREESM video filter application is presented in figure 5.1. The PREESM filter application is adapted from the PREESM example in [32] by adding another processing path for the Gaussian filter and making the necessary modifications to the shared parts of the application.

5.2.1 Actor Model

To keep the model simple and the program well analyzable, both of the processing paths in the network are independent. The first actor on both of the processing paths, the *Read_YUV* and the *Read_YUV2* actors in the figure 5.1, loads the video frames from memory and passes them to splitting actors. The *Split* actors split the frames to a suitable number of slices to

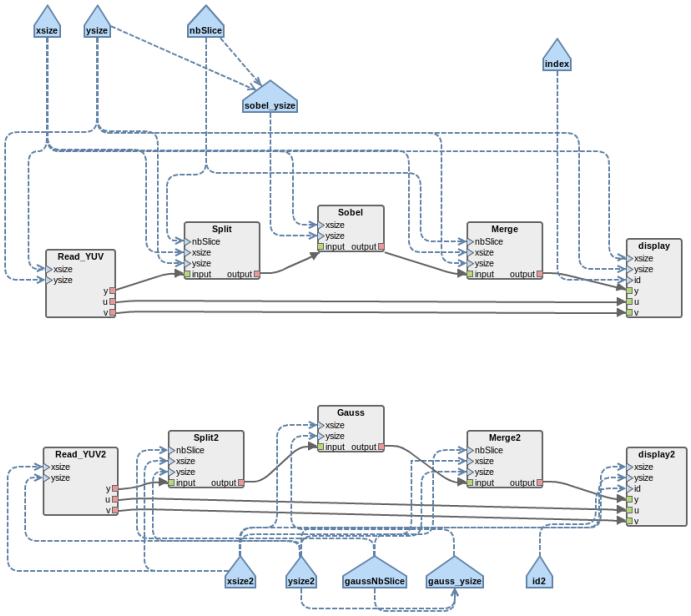


Figure 5.1: The PiSDF graph of the PREESM filter application

enable processing of the same video stream on multiple cores. The filter actors, *Sobel* and *Gauss* in the figure, follow the *Split* actors. Partial frames filtered in the filter actor are merged back into whole frames in the *Merge* actors. The last actors on both of the processing paths, called *display* and *display2*, are dummy actors.

The pentagons in the top and the bottom of the figure represent the parameters of the application. The `xsize`, `ysize`, `xsize2` and `ysize2` parameters specify the size of the input frames. `nbSlice` and `gaussNbSlice` determine the numbers of slices the frames are split to. The `sobel_ysize` and `gauss_ysize` are the heights of the slices after splitting.

The first actors *Read_YUV* and *Read_YUV2* call the `readYUV` function defined in the PREESM example at [32]. IO is omitted from the application, as it is not in the scope of the experiments. The processing starts with loading the frames from memory. In a real world application the frames would be written to the memory, for example, with DMA by a packet processor processing a stream of network packets. The `readYUV` function copies the Y component of the input frame to the output address while the U and V components are ignored. The U and V are omitted because the *Sobel* and *Gauss* actors only operate on the Y channel in the experiment setup.

The *Split* actor operates on the output of the *Read_YUV* actor. The *Split* actor preprocesses the frame for the *Sobel* and *Gaussi* actors. Since the Sobel and Gauss filters involve convolution with 3x3 and 5x5 matrices respectively, they will need to access image data outside the part of the image they are processing. To enable parallel processing of a single frame on multiple cores, the frame is split in to slices. These slices will also need to contain a bit of extra data so that the filters can operate correctly. Black lines, meaning lines with the Y value of 0, are added to the top and the bottom of the frame. One black line is enough for the Sobel frames but two black lines are needed for the Gaussian frames, corresponding to the filter kernel sizes. After the padding, the slices overlap each other for one or two lines again corresponding to the size of the filter kernel. With the black lines added the frame is copied to the output buffer one slice at a time.

The *Sobel* actor calculates the convolution of the Sobel kernels presented in 4.4 on the Y component of the input frame. The *Sobel* actor from the PREESM example at [32] is used in this experiment. The convolution is computed by looping over the pixels in the input frame, calculating the G_x and G_y components of the Sobel operator and combining them by computing the average of absolute values of G_x and G_y. After looping over the image the actor overwrites the edges of the frame with black pixels.

The *Gauss* actor operates similarly to the *Sobel* actor but instead of closed expressions, the value of the filter function at each points is calculated by looping over the neighboring pixels and multiplying the intensity values by the corresponding weight from the Gaussian kernel presented in 4.3.

The last actor processing the frame is the *Merge* actor. The *Merge* actor copies the processed data from its input buffer to its output buffer, overlaying the slices so that the output frame does not contain the extra lines created in the *Split* actor. Since there is no real IO in the application, the merged frame is not processed further. In a real world application the processed frame would be copied for example to a network packet. As there is no IO the *Display* actors do not do any computations. In the experiments the *Display* actor is used as the endpoint of the processing and it starts the measurement data export after predetermined number of iterations has completed.

5.2.2 PREESM Schedule

The scheduler of the PREESM framework is described in 4.3.3. The scheduler creates a block schedule from the actor model using user provided estimates for the actor durations. To get reasonably accurate estimates, the application was first scheduled with the default values and the actor durations were measured on the target device. The resulting timings are presented in the

Cycles	Actor
600	Gauss
150	Merge
150	Merge2
150	Read_YUV
150	Read_YUV2
150	Sobel
150	Split
150	Split2
1	display
1	display2

Table 5.1: PREESM Actor Timings.

table 5.1. The timings from the measurements were approximately the same for all the actors except for the *Gauss* actor. A Gantt chart representing the schedule, created using the values in the table, is presented in figure 5.2. The mutable parameters in the application, presented in the graph 5.1, affect the static memory allocations of the PREESM application. This makes it necessary to execute the PREESM codegen workflow every time the parameters are changed. For this reason, the PREESM schedules of the different measurement setups are slightly different.

The PREESM framework does not support changing the timing of the implode and explode operations, which was found in the experiments to start affecting the schedule considerably with larger frame sizes. The duration estimates used by PREESM for the implode and explode operations are very short compared to durations of the actors measured from the application. The explode operations for both streams are represented in the Gantt chart by the small time slices preceding *Gauss* actors on cores 4 and 7. The implode operation for Gauss stream follows the *Gauss* actor execution on core 7 and the implode operation of Sobel stream follows the latter *Sobel* actor on core 5.

A block schedule could be generated, which would yield a higher throughput by introducing multiple instances of the actors for each repetition of the schedule. The successive repetitions of the schedule would be interleaved and the overhead of synchronizing the cores would be reduced. The current version of PREESM does not support the described interleaving of the repetitions [54].

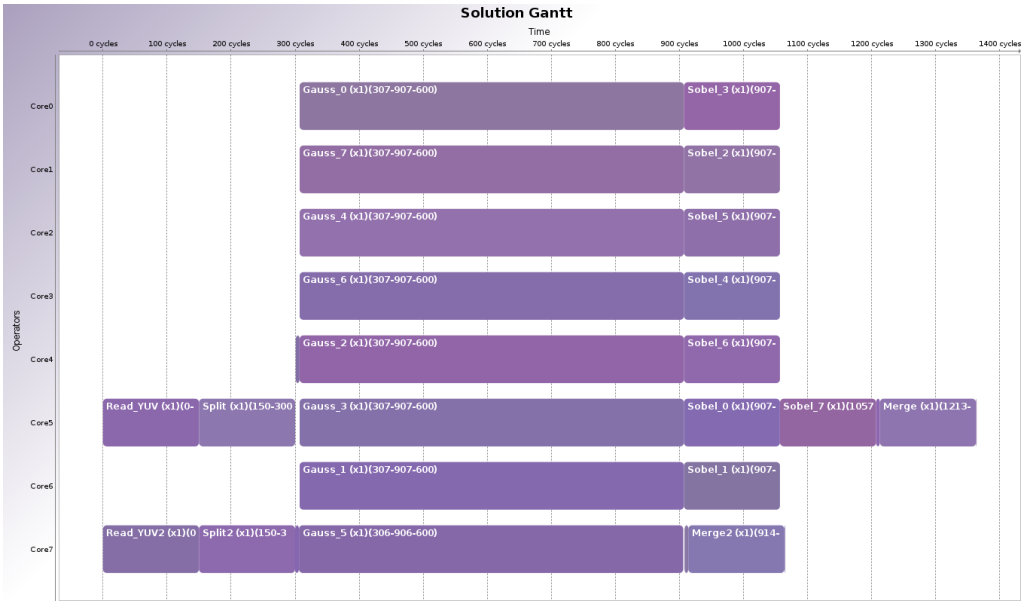


Figure 5.2: Gantt chart representing the schedule of a PREESM Filter Application.

5.3 OpenEM Filter Application

The OpenEM implementation of the filter application was designed using the PREESM filter application as the starting point. Specifically, the OpenEM application has to process the frames in a similar manner, so that ideally only the scheduling policies between the two programming models would differ. The high-level structure of the OpenEM filter application is presented in the figure 5.3. The application consists of two atomic queues, a parallel queue, and three execution objects. Each queue is connected to only one execution object. The execution runs in cycles. A new cycle is started every time the previous cycle finishes. There are multiple cycles running in parallel, but since the read and merge EOs are atomic, only filter EO is running on multiple cores in parallel.

The design principles introduced in the beginning of this chapter apply to the OpenEM application as well. The OpenEM application was not designed for maximum performance but to demonstrate the suitability of OpenEM as a platform for stream processing applications. The performance of the application could be improved in many ways, for example, by minimizing the amount of redundant copying of the frame slices and introducing parallelism to frame reading and merging.

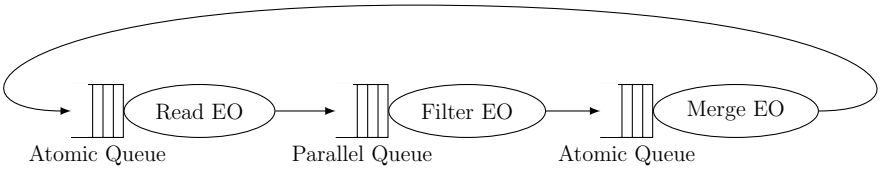


Figure 5.3: The OpenEM filter application.

The application execution starts with initializing the OpenEM framework and the application data structures. The OpenEM initialization is explained in 3.3.6. After the queues and the execution objects have been initialized, the application creates a number of initial events that are sent to the read queue. Once the events have been queued the application synchronizes all cores and the actual execution starts.

The Read EO combines the functionality of the *Read* and *Split* actors of the PREESM filter application presented in 5.1. A frame is read from the input video memory and split into a number of slices. The functions called are exactly the same as in the PREESM application. The frame slices are copied to event buffers of new events. The new events are then sent to the filter queue. There is only one filter queue and one filter execution object. The same execution object will compute either Sobel or Gauss filter on the frame slice according to its type.

The PREESM application splits the frames into slices and processes the slices separately before merging them back into one frame. Similar fork-join mechanism was implemented in the OpenEM application. Frames are accumulated simply in a merge buffer located in shared memory, which is referenced through queue context pointers. The bookkeeping for frame completion is handled in the shared memory as well.

The filtered frame slices are accumulated in the merge buffer, which is accessed through the merge queue context. The merge queue is atomic to avoid the need for synchronization between the cores. After a frame has been merged the merge EO creates a new event, which is sent to the read queue. The ratio of Sobel filtered frames to Gauss filtered frames is always one to one.

5.4 Instrumentation

To understand the behavior and performance of the applications their execution needs to be measured. The measurements conducted in the experiments are based on cycle counts. The TMS320C6678 exposes cycle counts on each

core through two registers, namely TSCL and TSCH, which store the low 32 bits and the high 32 bits of the cycle count respectively. Comparing the values of the TSCL register in the beginning and the end of processing on each core gives accurate local timings and would be enough for measuring the execution on single core. However, for accurate measurements of global cycles, the cycle offsets to the global cycle count need to be stored. The functionality for counting cycles is included in the Texas Instruments OpenEM initialization and utilities code. The cycle counts are stored in the DDR memory.

The latency of both of the filters is measured from the time the frame is loaded from the memory to the time the frame is merged after filtering. The throughput is measured as frames per second processed in total by the application. Since both of the applications process frames at the same rate from both streams, there is only one FPS measurement for each of the measurement setups.

In the PREESM application the cycle counts are saved in the beginning and the end of each user defined function of the application. In addition to this total cycle count, another count was measured in the `readYUV` function. In the OpenEM application cycle counts are measured over the same functions as in the PREESM application and in the beginning and in the end of each EO. The latter measurements are carried out to get a better understanding of the OpenEM runtime overhead.

The measurements are exported through the debug connection of the TMS320C6678 using IO library provided by Texas Instruments.

5.5 Description of Experiments

In these experiments, the OpenEM and PREESM filter applications are loaded with similar workloads and their execution is measured. The idea of the experiments is to examine the dynamic scheduling capabilities of the OpenEM scheduler and the overhead of the OpenEM framework in stream processing. The OpenEM scheduler is hardware accelerated, running on a separate processor in the TMS320C6678 chip. The scheduling is explained in detail in chapter 3. To achieve this objective the OpenEM filter application is measured under different loads and compared to a similar application implemented using PREESM. The PREESM application should be considered a baseline, which demonstrates how a statically scheduled application behaves under dynamic workload.

The static schedule of the PREESM application is regenerated between every measurement setup due to the limitations of the code generation in

PREESM framework. The specific limitation is that the parameters of the actor model are translated into static memory allocations in the code generator and manually changing the generated allocations would be complicated and prone to error. As a result the actors are scheduled slightly differently between each scenario. The schedules differ in the mapping of actors to cores but not in the ordering of the actors. The actor ordering is defined by the dependencies between the actors and availability of computing resources. The estimated actor timings of the PREESM application are not modified when changing the frame size.

In the experiments the applications are loaded with three different workloads and measured. In addition the OpenEM application is measured under the same load but different numbers of available cores. The experiments are described in the following subsections. In the first subsection the parameters and factors of the experiment are introduced. Second the different measurement setups are described and third the results of the experiment are presented.

5.5.1 Parameters and Factors

Dynamic workload conditions are emulated by repeating the measurements with different factors. To keep the measurement setup simple the video streams are not dynamically switched at runtime. The measurement parameters are presented in the following listing.

- **Video Frame Size** - Changing the frame sizes of the video streams differentiates the workloads.
- **OpenEM Core Masks** - The OpenEM application is measured with different core masks of the Execution Objects.
- **Number of Frames Processed Simultaneously** - The OpenEM application processes variable number of frames simultaneously.

The different video frame sizes used are presented in table 5.2. The frame sizes used are selected from among the Common Intermediate Format frame sizes. YUV video format is used in the applications, but only the Y channel is processed by the applications. The Y channel in the YUV frame contains $R_x * R_y$ bytes where R is the resolution. In the YUV format used the U and V channels have reduced bit rates of $\frac{1}{4} * R_x * R_y$ per frame.

Throughout its lifetime, the OpenEM application circulates the events created initially. The number of initial events determines the maximum number of frames that can be processed simultaneously. The effect of the

Name	X resolution	Y resolution
QCIF	176	144
CIF	352	288
4CIF	704	576

Table 5.2: CIF frame sizes

Sobel Resolution	Gauss Resolution
CIF	CIF
4CIF	CIF
CIF	4CIF
QCIF	QCIF

Table 5.3: PREESM and OpenEM measurement setups

number of the simultaneously processed frames to the latency and throughput of the application is examined in the experiments.

OpenEM core masks are used to limit the number of cores available to the filter application. The behavior of OpenEM is examined under limited resources. With the use of core masks the performance improvement through increased parallelism is investigated.

5.5.2 Measurement Setups

The filter applications process two video streams simultaneously as described in chapter 5. One video stream is processed with a Sobel filter and the other is processed with a Gaussian filter. The dynamic behavior of the applications is investigated using different workloads.

- *Comparing Dynamic and Static Scheduling*, The workloads used in the first experiment are presented in the table 5.3. The purpose of the different bit rates used for each video stream is to expose the behavior of the OpenEM scheduler in handling dynamic workloads. The static schedule in the PREESM application will provide a baseline to reflect the OpenEM performance to, but the performance of the applications should not be directly compared due to the difference in the runtime systems.

- *Investigating the Balance of Latency and Throughput*, To examine the capabilities of the dynamic scheduler the number of simultaneously

processed events is varied in the second experiment. The maximum number of frames that can be processed simultaneously corresponds to the number of initial events created in the initialization. Both filters are loaded with CIF streams and the measurements are run with 1 to 24 initial events. The number of simultaneous frames changes the throughput versus latency balance of the OpenEM application.

- *Examining the Efficiency of Parallel Scheduling*, In the third experiment the OpenEM application is measured with different core masks to investigate the ability of the dynamic scheduler to utilize the increased parallel resources. Both filters of the OpenEM application are loaded with CIF streams. Different numbers of cores are used, but all actors can run on all available cores. The experiment is run with core masks allowing one to eight cores be used for processing the streams. The experiment is run with 16 initial events.

Chapter 6

Experiment Results and Analysis

The results of the experiments conducted in this thesis are presented in this chapter. The section 6.1 presents and describes the results. A more in-depth look at the results is taken in 6.2 where the results are discussed and their significance assessed.

6.1 Results

In this section the results of the three experiments described in 5.5.2 are described. The results of the comparison of the application using the OpenEM dynamic scheduler and the statically scheduled application created with PREESM are given in 6.1.1. The results of the experiment investigating the balance of latency versus throughput are given in 6.1.2. Finally, the results of the third experiment examining the efficiency of increasing the parallelism in the dynamically scheduled application are given in 6.1.3.

6.1.1 Comparing Dynamic and Static Scheduling

In the first experiment the bitrates of the two video streams are varied. The latency and throughput of the OpenEM application measurements are summarized in the table 6.1. The corresponding summaries for the PREESM application are presented in the table 6.2

The latencies of the two streams of the PREESM application in all measurement setups in the table 6.2 are similar. The largest difference in the latencies is in the case where the Sobel filter filters a CIF stream and the Gauss filter filters a 4CIF stream, where the Gauss latency is 60% higher than

Sobel latency	Gauss latency	FPS	Sobel frame	Gauss frame
3,59	10,93	256	CIF	4CIF
3,75	2,95	527	4CIF	CIF
1,42	2,80	889	CIF	CIF
0,32	0,71	3534	QCIF	QCIF

Table 6.1: OpenEM latency and throughput with 2 frames processed simultaneously. The latencies are measured in milliseconds.

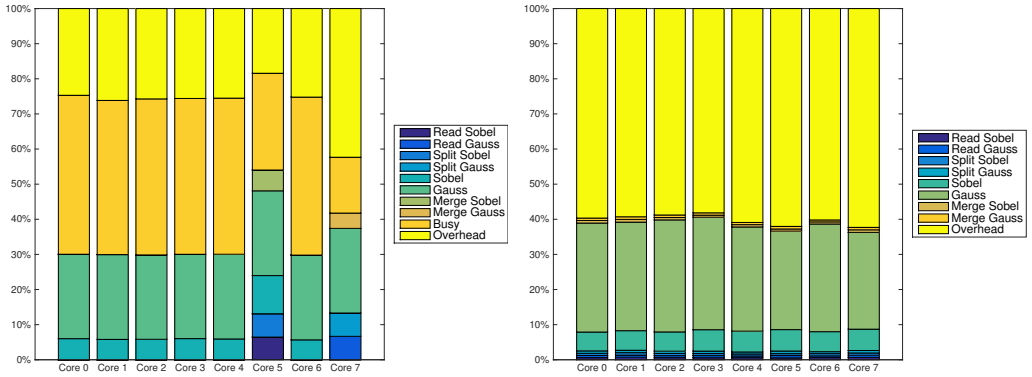
Sobel latency	Gauss latency	FPS	Sobel frame	Gauss frame
5,41	8,78	223	CIF	4CIF
4,65	3,54	334	4CIF	CIF
2,15	2,51	668	CIF	CIF
0,61	0,71	2004	QCIF	QCIF

Table 6.2: PREESM latency and throughput. The the latencies are measured in milliseconds.

the Sobel latency. Larger differences in the latencies are observed with the OpenEM application with two initial events presented in table 6.1, where the largest difference in latencies is also measured between the CIF Sobel stream and 4CIF Gauss stream. The Gauss latency in the particular OpenEM measurement is 200% higher than the Sobel latency. The other OpenEM measurements show larger differences as well compared to PREESM.

The throughput of the application is determined by how much time the application spends processing the streams versus the time spent doing something else. For example the synchronization of all cores before each repetition of the PREESM schedule consumes a lot of CPU cycles as is readily observed from the figure 6.1a. The portion of the bars marked as busy corresponds to the cycles spent in the synchronization between the repetitions of the schedule. The percentage of cycles spent in the synchronization varies from 16% on Core 7 to 45% on Core 0. The OpenEM dynamic scheduler seems to spread out the work more evenly in this case where the total overhead cycles are approximately 60% for all cores. The core utilization per function is presented in figure 6.1b. Most of the overhead cycles in the OpenEM application are spent waiting for more frames for processing.

The overhead portions in the figures 6.1a and 6.1b contain all of the data copying from buffer to buffer outside the measured functions, but they also contain different amounts of cycles spent in communications between the



(a) PREESM (b) OpenEM

Figure 6.1: In the PREESM graph the busy portion of the bars correspond to the cycles spent synchronizing the cores between the repetitions of the block schedule. The overhead corresponds to cycles spent outside the measured functions and the synchronization. In the OpenEM application the overhead cycles are the cycles spent outside the compared functions.

cores. The measured functions are the functions, which are explicitly called in the PREESM actor model. Other functions not included in the measured functions contain runtime specific communication and some copying of buffers.

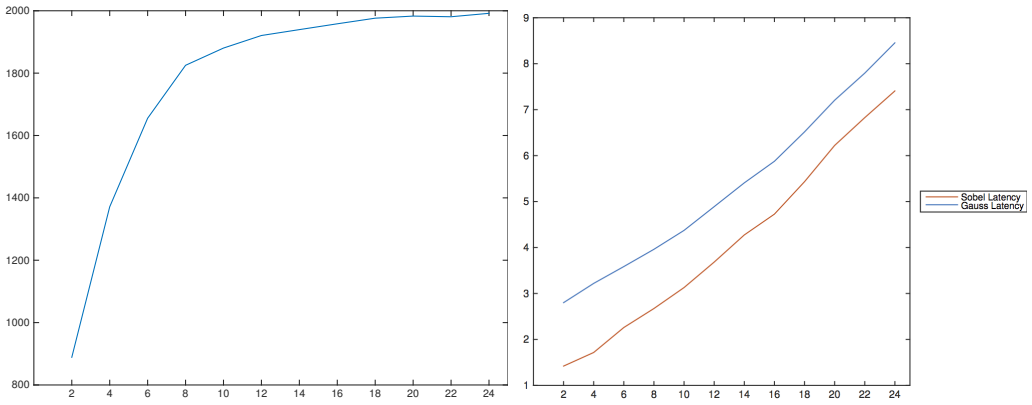
6.1.2 Investigating the Balance of Latency and Throughput

In this set of measurements the OpenEM application is configured to process different numbers of frames simultaneously. The PREESM latencies in the table 6.2 are consistently smaller than the latencies of the throughput optimized OpenEM application in the table 6.3. The throughput optimized application is processing 16 frames simultaneously. The throughput of the OpenEM application is almost doubled compared to the configuration presented in 6.1 where two frames processed simultaneously.

The effect of increasing the number of simultaneous frames was further examined by measuring the application with two CIF streams using 2 to 24 initial events. The results of the measurements are presented in table 6.4. The throughput is increased when the number of frames processed simultaneously increases. This behavior is visualized in figures 6.2a and 6.2b. The throughput grows rapidly up to 8 simultaneous frames, after which the growth slows down. The latencies grow with each added frame steadily.

Sobel latency	Gauss latency	FPS	Sobel frame	Gauss frame
15,82	22,85	599	CIF	4CIF
4,85	3,67	895	4CIF	CIF
4,91	5,96	1955	CIF	CIF
1,33	1,62	7819	QCIF	QCIF

Table 6.3: OpenEM latency and throughput with 16 frames processed simultaneously. The latencies are measured in milliseconds.



(a) FPS as a function of simultaneous frames.

(b) Latencies as functions of simultaneous frames. The latencies are measured in milliseconds.

Figure 6.2: The effect of increasing the number of frames processed simultaneously is presented in the figures. The throughput increases rapidly up to 8 simultaneous frames after which the growth slows down. The latency grows more steadily across all measured setups.

Sobel latency	Gauss latency	FPS	Simultaneous Frames
1,42	2,80	889	2
1,72	3,22	1371	4
2,26	3,59	1655	6
2,67	3,96	1825	8
3,13	4,37	1880	10
3,68	4,89	1921	12
4,27	5,40	1940	14
4,73	5,88	1958	16
5,43	6,52	1976	18
6,22	7,21	1983	20
6,83	7,80	1981	22
7,41	8,45	1992	24

Table 6.4: OpenEM measurements with different numbers of frames in processed simultaneously.

6.1.3 Examining the Efficiency of Parallel Scheduling

In the third experiment the number of cores available for the OpenEM application was varied. Both filters were processing CIF streams. The measurements were repeated using one to eight cores. The resulting latencies and throughputs are presented in the table 6.5. The increase of the throughput of the OpenEM application is presented in figure 6.3. The actual FPS measured with eight cores is approximately 90% of the linear growth calculated from the FPS using a single core.

Sobel latency	Gauss latency	FPS	No of cores
57,05	57,11	263	1
22,59	23,15	510	2
15,09	15,84	768	3
10,91	11,85	1014	4
8,41	9,53	1268	5
7,05	8,07	1500	6
5,74	6,83	1731	7
4,91	5,96	1955	8

Table 6.5: OpenEM measurements with number of cores varied

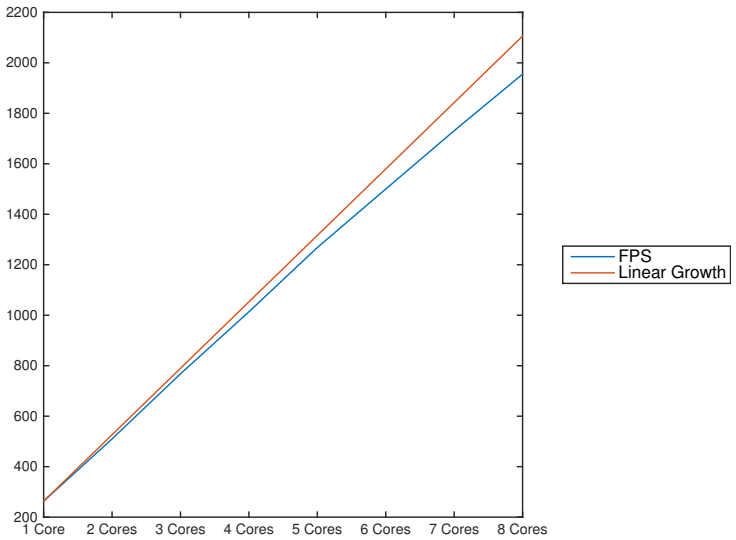


Figure 6.3: FPS increase as the function of cores vs. linear growth

6.2 Discussion

Three experiments were conducted to understand the performance of Open Event Machine based multi-core DSP applications in stream processing. The studied stream processing application was implemented with OpenEM and a comparable application was implemented using PREESM. The performance of the applications as a part of a stream processing system was not studied. Therefore, IO was omitted from the stream processing task that both of the applications implemented. The experiments focused on understanding the performance of the OpenEM scheduler in handling stream processing tasks.

In this section the results of the three conducted experiments are discussed. First, the discoveries made from the experiments are described in 6.2.1. Second, the possible directions of future work are discussed in 6.2.2. Finally, the challenges faced conducting the experiments are explained in 6.2.3

6.2.1 Discoveries

The first experiment where the throughput and latency of OpenEM based application were compared to those of the statically scheduled application showed that the runtime systems are roughly comparable. The statically scheduled application created with PREESM potentially had the advantage here, since with static schedule no cycles are wasted at any point in the execution in deciding which actor to execute next on which core. OpenEM achieved very similar latencies with the dynamic scheduler. This suggests that the OpenEM scheduler utilizes the hardware resources efficiently and causes relatively small overhead.

The PREESM authors state that PREESM is a prototyping tool [31], which suggests that it does not necessarily aim to create latency or throughput optimized applications. The Gantt chart in figure 5.2 suggests that manually optimizing the schedule, a lower latency and higher throughput could be achievable. This is supported by the core utilization graph in figure 6.1a where a large portion of the execution time is spent waiting for other cores. Nevertheless, the OpenEM scheduling performance can be considered good as it roughly matches the statically generated schedule.

In the second experiment the balance of latency versus throughput with the OpenEM scheduler was investigated. The experiment where the number of simultaneously processed frames was varied suggested that by configuring the application the balance between throughput and latency can be controlled. The adjustability of the scheduler is a useful feature to have in real

applications, as the throughput can be increased for tasks, which are less latency sensitive without modifying the program code. The extent of control the application designer has depends on the complexity of the application. The application measured in the experiments was simple and it is possible that the results overemphasize the adaptivity compared to more complex applications.

The third experiment was conducted to understand the performance improvement gained from making more processing units available for the runtime. The FPS improvement from one to eight cores is close to 90% of linear improvement. This means that the sequential portion of the application was small and the scheduler was able to utilize the increased parallelism efficiently.

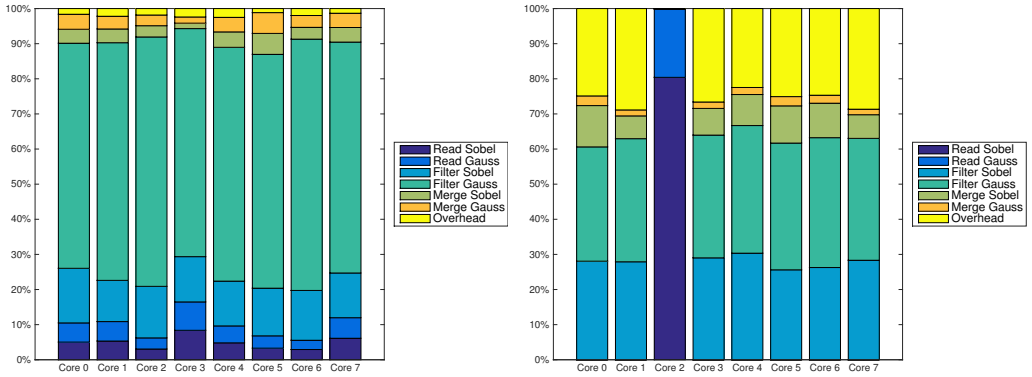
Since two frames are processed simultaneously the execution of the serial part of the program performed in the read and merge execution objects is interleaved with the execution of the filter execution object of the other frame. This behavior seems to yield a large latency decrease from execution on one core to execution on two cores. Unfortunately this complicates the analysis, since the proportion of the serial and parallel parts of the program is ambiguous. This complication prevents useful comparisons with the theoretical improvement defined by Amdahl's law [4].

The result of the parallel scheduling experiment was not surprising but it is reassuring of the OpenEM scheduler performance. As the total number of events in circulation was quite small compared to the example applications provided with the TI OpenEM distribution [39], the scheduler should have no trouble keeping up with the pace of the application even with eight cores.

The throughput optimized application processing 16 events simultaneously exhibits formation of a bottleneck in the atomic execution object. In the table 6.3 an improvement of latencies is observed when the workload is made heavier by moving 4CIF stream from Gauss filter to the Sobel filter while the other stream is kept at CIF resolution.

The probable cause of the improvement of the latencies when the workload is made heavier by increasing the Sobel stream resolution is the reduced interleaving of the processing of the subsequent frames. The reduction in the interleaving is caused by the read execution object, which is only connected to an atomic queue. When the frame size of the Sobel stream is increased the read EO starts limiting the throughput of the application. Fewer frames are processed in parallel, which decreases the time from reading each individual frame to the completion of that frame.

The formation of the bottleneck can be observed by comparing the core utilization in the figure 6.4a to the core utilization in the figure 6.4b where seven cores need to wait for the read operations on the Core 2 and the overall



(a) Sobel CIF, Gauss CIF

(b) Sobel 4CIF, Gauss Cif

Figure 6.4: The bottleneck forming due to the atomic read operation can be observed by comparing the core utilization when both of the streams are at CIF resolution to the case where Sobel resolution is increased to 4CIF. In these graphs the application is processing 16 simultaneous events.

overhead is increased. The other cores do not receive any events from the OpenEM scheduler while waiting.

When the Gauss stream is increased to 4CIF and the Sobel stream is kept at CIF the bottleneck does not form. This is likely because computing the Gaussian filter for the 4CIF frames is consuming approximately 80% of the cycles on all cores. In this case the read EO only consumes approximately 5% to 13% of the cycles on all cores. Compared to the case of two CIF streams, the latencies in this case are increased by factors of approximately 3 and 4 for Sobel and Gauss correspondingly. The resulting core utilization graph is presented in figure 6.5.

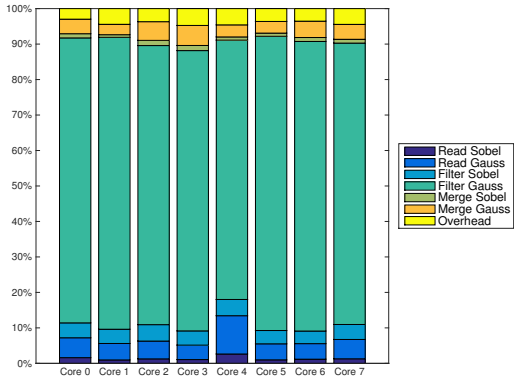


Figure 6.5: OpenEM cycles spent per execution object for CIF Sobel frames and 4CIF Gauss frames

6.2.2 Future Work

In this thesis the use of OpenEM to provide task management and scheduling in a stream processing application was investigated. The results suggest that OpenEM scales well to the needs of a simple stream processing application and it simplifies the implementation of a multi-core application on the DSP platform without operating system.

The workload studied in this thesis was simple and lacked dynamic nature of processing graphs common to stream processing applications. To better understand the performance of OpenEM, two future research approaches could be taken. First, studying the performance of OpenEM scheduler with another artificial workload application could still strengthen the case for using OpenEM as the runtime for stream processing. Instead of a simple, linear processing graph like the one used in the experiments of this thesis, a more dynamic graph with optional processing paths could be studied. Studying this kind of graph would show if the scheduler is capable of hitting the latency and throughput constraints of the application consistently under varying load. Second, a more complex application could be constructed and measured. For example implementing an MPEG decoder using OpenEM as the runtime system would provide a more complete view to the dynamic performance of OpenEM. An MPEG decoder would be an interesting workload because it has been used as something of a standard benchmark in dataflow literature and thus there exists many implementations based on different technologies. Having points of comparison for the workload would help put the performance in broader context.

The idea of using DSPs for stream processing has been researched to some

extent but the idea has not made it to the mainstream of computing industry. To make the case for DSPs as stream processors, a thorough benchmark should be created that would compare the streaming performance versus energy consumed for similar stream processing applications implemented with DSPs, GPUs, and CPUs. Making such comparison is not trivial, since to the author's knowledge there does not exist a single widely accepted measure of streaming performance. The question is complicated further by the need for measuring the energy consumption, which itself is a complex task.

Before conducting such research, three problems would have to be solved. First, selection of the measured variables. Comparing floating point operations per second per Joule between the platforms would be the simple solution, but FLOPS is not necessarily good measurement for streaming performance. For example, for video stream, frames per second and latency would provide a better understanding of the streaming performance than FLOPS. Second, accounting of the energy consumption fairly with all measured processing units is not trivial, since they are commonly installed in different forms of devices. CPUs are installed in sockets on motherboards, GPUs are installed on PCI cards or increasingly as part of integrated chips, and DSPs are installed in various devices including PCI cards. Comparing the vendor provided energy specifications is one way to deal with this problem, but it might miss how the devices are actually deployed. Third, the workload would have to be implemented in different patterns for each of the platforms making use of the strengths of each platform equally.

Two papers conduct comparisons that partially answer the questions above, but do not actually provide the complete comparison between the processors. Stotzer et al. conduct an experiment in [62] where a comparison between the different processing units is conducted in the context of OpenMP, but the comparison does not include energy measurements. The performance of FPGAs and GPUs in processing what the author calls "sliding-window applications" is compared in terms of Joules per frame in [23]. The "sliding-window applications" consist of applying convolutions on frames of data in the stream, similar to how the workload in this thesis computes.

6.2.3 Challenges

Analysing the performance of OpenEM based stream processing applications is a broad task, requiring the use of many distinct technologies. First, a suitable workload, which is representative of stream processing tasks had to be designed. Next, two versions of the workload application had to be implemented, one for the evaluation of OpenEM and the other to provide a point of comparison. In order to get usable data from the applications, they

had to be instrumented and the measurement data analysed. Most of the work required was straightforward but some complications were faced with the tools.

The baseline application was implemented using PREESM. PREESM generates static schedules for the application from provided the dataflow graph and the source code. To get specific points of comparison for each of the measurement setups working with the PREESM automatic code generation was complicated. The tool would create different schedules on each iteration and thus yield applications with dissimilar core usage between the measurement setups.

Texas Instruments OpenEM version 1.0.0.2 that was used for the implementation of the OpenEM workload has some unimplemented features. Most of the unimplemented features were documented and did not cause unnecessary work. However, the implementation of Queue Groups did not work as expected. Creation of multiple queue groups did not work as specified in the documentation, and was left unexplored after multiple trials at working with them.

The documentation of the Texas Instruments implementation of OpenEM was lacking. The documentation provided with the source code was incomplete and included a sketchy version of the OpenEM white paper [51], which was the most important source for understanding the use of hardware acceleration in the runtime. The documentation also lacked details for the initialization layer without which the initialization of the hardware components would have been difficult. The initialization code was provided as a part of a scarcely documented example application.

Chapter 7

Conclusions

This thesis set out to investigate the performance of OpenEM based stream processing applications, with specific focus on scheduling. Scheduling is an important factor of the streaming performance. A good scheduler is capable of utilizing parallelism to improve throughput and latency of the application, whereas worse schedulers do not distribute work on multiple cores as efficiently and thus execute more of the work sequentially leading to increased latency and decreased throughput.

The problem was approached by comparing the performance of an Open Event Machine based streaming application to the performance of a similar, statically scheduled application. The workload application designed for the experiments is a video stream processing application inspired by the Canny edge detector. The baseline application was implemented using PREESM, which generates statically scheduled multi-core applications based on synchronous dataflow graphs. The applications were instrumented so that they could be compared in terms of latency, throughput, and core utilization. The results of the comparison experiment suggest that the OpenEM dynamic scheduler does not cause unmanageable overhead compared to statically scheduled applications.

The scheduling performance of OpenEM was further investigated with two experiments carried out using the same measurement system. In one experiment the balance between throughput and latency was examined by increasing the number of frames processed concurrently. This experiment showed that at least simple OpenEM applications are adjustable in terms of throughput and latency, which may be useful in stream processing tasks. In the other experiment the parallelizing performance of the scheduler was examined by computing the same tasks using variable number of cores. The growth in performance was 90% of linear growth from one to eight cores, which is decent improvement considering that in addition to the parallel

actors, the workload had sequential actors, which did not benefit from the parallelization.

The results of the experiments suggest that Texas Instruments OpenEM runtime is flexible enough to support the implementation of streaming applications and it was found that the dynamic scheduler is capable of making reasonable scheduling decisions with small overhead. Further research is required to understand the overall performance of multi-core DSPs versus GPUs, FPGAs, and CPUs in stream processing.

As data streaming is becoming more widespread in the Internet, high performance stream processing solutions are in demand in the industry. Exploring the use of DSPs for stream processing is interesting because they achieve high ratio of floating point operations to energy consumption and they implement an architecture that is well suited to stream processing in theory. As multi-core DSPs are not as commonly used for stream processing as for example GPUs, the toolset for parallel programming is not as standardized as on other platforms. Therefore, runtime systems such as OpenEM have the potential of becoming the standard tools for stream processing on DSPs.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Online, Accessed 17 May 2016.

[2] Marleen Adé, Rudy Lauwereins, and JA Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on dsp-fpga targets. In *Proceedings of the 34th annual Design Automation Conference*, pages 64–69. ACM, 1997. doi: 10.1109/DAC.1997.597118.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013. doi: 10.14778/2536222.2536229.

[4] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967. doi: 10.1145/1465482.1465560.

[5] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *Parallel and Dis-*

tributed Systems, IEEE Transactions on, 20(3):404–418, 2009. doi: 10.1109/TPDS.2008.105.

[6] Rajkishore Barik, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, et al. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736. ACM, 2009. doi: 10.1145/1639950.1639989.

[7] Shuvra S Bhattacharyya, Johan Eker, Jörn W Janneck, Christophe Lucarz, Marco Mattavelli, and Mickaël Raulet. Overview of the mpeg reconfigurable video coding framework. *Journal of Signal Processing Systems*, 63(2):251–263, 2011. doi: 10.1007/s11265-009-0399-3.

[8] Shuvra S Bhattacharyya, Ed F Deprettere, Rainer Leupers, and Jarmo Takala. *Handbook of Signal Processing Systems*. Springer Science & Business Media, 2013. ISBN 9781461468592.

[9] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall Professional, 2006. ISBN 9780131872493.

[10] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996. doi: 10.1006/jpdc.1996.0107.

[11] Joseph T Buck, Edward Lee, et al. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 429–432. IEEE, 1993. doi: 10.1109/ICASSP.1993.319147.

[12] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010. doi: 10.3233/SPR-2011-0305.

[13] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986. doi: 10.1109/TPAMI.1986.4767851.

[14] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002. doi: 10.1145/1133373.1133410.

[15] William J Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J Knight, et al. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35. ACM, 2003. doi: 10.1145/1048935.1050187.

[16] Karol Desnos, Maxime Pelcat, Jean-Francois Nezan, and Slaheddine Aridhi. Memory bounds for the distributed execution of a hierarchical synchronous data-flow graph. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 160–167. IEEE, 2012. doi: 10.1109/SAMOS.2012.6404170.

[17] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya, and Slaheddine Aridhi. Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 41–48. IEEE, 2013. doi: 10.1109/SAMOS.2013.6621104.

[18] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003. doi: 10.1109/JPROC.2002.805829.

[19] Tom Flanagan, Zhihong Lin, and Sneha Narnakaje. Accelerate multicore application development with keystone software, 2013. URL `http://www.ti.com/lit/wp/spry231/spry231.pdf`. Online, Accessed 21 August 2015.

[20] Alcides Fonseca, João Rafael, and Bruno Cabral. Eve: A parallel event-driven programming language. In *Euro-Par 2014: Parallel Processing Workshops*, pages 170–181. Springer, 2014. doi: 10.1007/978-3-319-14313-2_15.

[21] Apache Software Foundation. Apache spark streaming, 2016. URL `http://spark.apache.org/streaming/`. Online, Accessed 17 May 2016.

[22] Apache Software Foundation. Apache storm, 2016. URL `http://storm.apache.org/`. Online, Accessed 17 May 2016.

[23] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 47–56. ACM, 2012. doi: 10.1145/2145694.2145704.

[24] Jeff Friesen. Asynchronous i/o. In *"Java I/O, NIO and NIO.2"*, pages 387–415. Springer, 2015. doi: 10.1007/978-1-4842-1565-4_13.

[25] GR Gao, R Govindarajan, and Prakash Panangaden. Well-behaved dataflow programs for dsp computation. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, volume 5, pages 561–564. IEEE, 1992. doi: 10.1109/ICASSP.1992.226558.

[26] Amir Hossein Ghamarian, MCW Geilen, Sander Stuijk, Twan Basten, AJM Moonen, Marco JG Bekooij, Bart D Theelen, and Mohammad Reza Mousavi. Throughput analysis of synchronous data flow graphs. In *Sixth International Conference on Application of Concurrency to System Design, 2006*, pages 25–36. IEEE, 2006. doi: 10.1109/ACSD.2006.33.

[27] Dominik Göddeke. *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*. Logos Verlag Berlin GmbH, 2011. ISBN 9783832527686.

[28] Rafael C Gonzalez and Richard E Woods. *Digital Image Processing*. Prentice Hall, 2008. ISBN 9780131687288.

[29] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. doi: 10.1109/5.97300.

[30] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011. ISBN 9780123838735.

[31] INSA. Preesm, 2015. URL `http://preesm.sourceforge.net/website/`. Online, Accessed 22 April 2015.

[32] INSA. Preesm tutorials, parallelize an application on a multicore cpu, 2015. URL `http://preesm.sourceforge.net/website/index.php?id=code-generation-for-multicore-dsp`. Online, Accessed 3 September 2015.

[33] Texas Instruments. Tms320c66x dsp cache user guide, 2010. URL http://www.ti.com/lit/ug/sprugy8/sprugy8.pdf. Online, Accessed 27 August 2015.

[34] Texas Instruments. Tms320c66x dsp cpu and instruction set reference guide, 2010. URL http://www.ti.com/lit/ug/sprugh7/sprugh7.pdf. Online, Accessed 22 April 2015.

[35] Texas Instruments. Tms320c6678 multicore fixed and floating-point digital signal processor, 2010. URL http://www.ti.com/lit/ds/symlink/tms320c6678.pdf. Online, Accessed 21 August 2015.

[36] Texas Instruments. Tmdxevm6678l evm technical reference manual version 1.0, 2011. URL http://wfcache.advantech.com/www/support/TI-EVM/download/TMDXEVM6678L_Technical_Reference_Manual_1V00.pdf. Online, Accessed 24 August 2015.

[37] Texas Instruments. Open event machine library api specification, 2012. URL http://software-dl.ti.com/sdoemb/sdoemb_public_sw/bios_mcsdk/latest/index_FDS.html. Distributed with OpenEM library. Accessed 20 August 2015.

[38] Texas Instruments. Open event machine library release notes, 2012. URL http://software-dl.ti.com/sdoemb/sdoemb_public_sw/bios_mcsdk/latest/index_FDS.html. Distributed with OpenEM library. Accessed 20 August 2015.

[39] Texas Instruments. Open event machine library user guide, 2012. URL http://software-dl.ti.com/sdoemb/sdoemb_public_sw/bios_mcsdk/latest/index_FDS.html. Distributed with OpenEM library. Accessed 20 August 2015.

[40] Texas Instruments. Mcsdk download link, 2015. URL http://software-dl.ti.com/sdoemb/sdoemb_public_sw/bios_mcsdk/latest/index_FDS.html. Online, Accessed 2 September 2015.

[41] Texas Instruments. Keystone architecture multicore navigator, user's guide, 2015. URL http://www.ti.com/lit/ug/sprugr9h/sprugr9h.pdf. Online, Accessed 22 April 2015.

[42] Keith Jack. *Video Demystified: a Handbook for the Digital Engineer*. Elsevier, 2011. ISBN 9780080553955.

[43] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991. ISBN 9780471503361.

[44] Ujval J Kapasi, William J Dally, Scott Rixner, John D Owens, and Brucek Khailany. The imagine stream processor. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 282–288. IEEE, 2002. doi: 10.1109/ICCD.2002.1106783.

[45] Yu-Kwong Kwok. High-performance algorithms for compile-time scheduling of parallel processors, 1997. URL `http://hdl.handle.net/1783.1/1515`. Online, Accessed 17 May 2016.

[46] Edward Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. doi: 10.1109/PROC.1987.13876.

[47] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. doi: 10.1109/MC.2006.180.

[48] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. Lee & Seshia, 2014. ISBN 9781312427402.

[49] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Acm Sigplan Notices*, volume 44, pages 227–242. ACM, 2009. doi: 10.1145/1639949.1640106.

[50] Raman Maini and Himanshu Aggarwal. Study and comparison of various image edge detection techniques, 2009. URL `http://www.cscjournals.org/manuscript/Journals/IJIP/Volume3/Issue1/IJIP-15.pdf`. Online, Accessed 20 May 2016.

[51] Filip Moerman. Open event machine: A multi-core run-time designed for performance. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 41–45. IEEE, 2014. doi: 10.1109/EDERC.2014.6924355.

[52] Telecommunication Standardization Sector of ITU. Itu-t recommendation h. 261: Line transmission of non-telephone signals: Video codec for audiovisual services at px 64 kbits, 1993. URL `https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-H.261-199303-I!!PDF-E&type=items`. Online, Accessed 20 May 2016.

[53] Maxime Pelcat, Jean Francois Nezan, Jonathan Piat, Jerome Croizer, and Slaheddine Aridhi. A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems, 2009. URL `https://hal.archives-ouvertes.fr/hal-00429397/document`. Online, Accessed 20 May 2016.

[54] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 36–40. IEEE, 2014. doi: 10.1109/EDERC.2014.6924354.

[55] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008. ISSN 1937-4771.

[56] Iain E Richardson. *Video Codec Design: Developing Image and Video Compression Systems*. John Wiley & Sons, 2002. ISBN 9780471485537.

[57] Kazuki Sakamoto and Tomohiko Furumoto. Grand central dispatch. In *Pro Multithreading and Memory Management for iOS and OS X*, pages 139–145. Springer, 2012. doi: 10.1007/978-1-4302-4117-1_6.

[58] Dragoş Sbîrlea, Jun Shirako, Ryan Newton, and Vivek Sarkar. Scnc: Efficient unification of streaming with dynamic task parallelism. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on*, pages 58–65. IEEE, 2011. doi: 10.1109/DFM.2011.13.

[59] Nokia Solutions and Networks. Open event machine introduction, 2013. URL `http://sourceforge.net/projects/eventmachine/files/Documents/EM_introduction_1_0.pdf`. Online, Accessed 1 September 2015.

[60] Nokia Solutions and Networks. Open event machine an event driven processing runtime for multicore, 2015. URL `http://sourceforge.net/projects/eventmachine/`. Documentation included in the source files. Online, Accessed 1 September 2015.

[61] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34 (7):491–541, 1997. doi: 10.1007/s002360050095.

[62] Eric Stotzer, Ajay Jayaraj, Murtaza Ali, Arnon Friedmann, Gaurav Mitra, Alistair P Rendell, and Ian Lintault. Openmp on the low-power ti keystone ii arm/dsp system-on-chip. In *OpenMP in the Era of Low*

Power Devices and Accelerators, pages 114–127. Springer, 2013. doi: 10.1007/978-3-642-40698-0_9.

[63] Jaspal Subhlok, James M Stichnoth, David R O'hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Notices*, volume 28, pages 13–22. ACM, 1993. doi: 10.1145/173284.155334.

[64] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software, 2005. URL `http://www.drdobbs.com/web-development/a-fundamental-turn-toward-concurrency-in/184405990`. Online, Accessed 20 May 2016.

[65] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002. doi: 10.1007/3-540-45937-5_14.

[66] Stefan Tilkov and Steve Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6): 80, 2010. doi: 10.1109/MIC.2010.145.

[67] Vernon Turner, John F Gantz, David Reinsel, and Stephen Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things, 2014. URL `http://www.emc.com/leadership/digital-universe/index.htm`. Online, Accessed 17 May 2016.

[68] Tom White. *Hadoop: The Definitive Guide*. "O'Reilly Media, Inc.", 2012. ISBN 9781449311520.