ABSTRACT

Title of Document: **SYSTEM SYNTHESIS FOR IMAGE PROCESSING APPLICATIONS**

Dong-Ik Ko, Doctor of Philosophy, 2006

Directed By: Professor Shuvra S.Bhattacharyya
Department of Electrical and Computer Engineering

Over the past few decades, embedded systems have been widely infiltrated into our daily lives. Prominent examples are cellular phones, personal digital assistants, digital television set-top boxes, web-pads, and mp3 players. New kinds of embedded devices are being introduced continually for various purposes.

Embedded systems have different combinations and prioritizations of objectives and constraints for their proper design. With the increasing complexity in application functionality, implementation constraints, and optimization objectives, more effective techniques for modeling embedded applications, and for systematically synthesizing implementations become more and more desirable on one hand, and more and more challenging on the other.

In this thesis, we focus on the efficient design, implementation, and synthesis of signal processing applications, which form a broad and important class of embedded

systems. We place special emphasis in the thesis on the signal processing domain on image processing, a sector that has seen rapidly increasing demand in recent years, but for which present techniques for signal processing design are often lacking in modeling and optimization capability.

In this thesis, we propose novel models and algorithms for streamlining scheduling, memory management, and interprocessor communication in embedded multiprocessor implementations of signal processing applications, with the aforementioned emphasis on the image processing domain.

For application modeling, we propose two novel modeling techniques called blocked dataflow (BLDF) and dynamic graph topology (DGT). These modeling approaches capture within their respective formal frameworks the structure of block-based image processing operations and reconfigurable, multi-mode dataflow behaviors, respectively.

For scheduling, we develop a novel intermediate representation called the pipeline decomposition tree (PDT). The PDT provides efficient representation and analysis of alternative multiprocessing configurations for signal processing applications. We also develop an algorithm, called pipeline decomposition tree scheduling (PDT scheduling), which applies the PDT to systematically derive optimized multiprocessor schedules that employ coarse-grained (task-level) pipelining, which is an especially useful form of parallelism for signal processing. To optimize interprocessor communication, we develop two novel post-optimization techniques for hardware resource mapping and software synthesis.

The suite of techniques presented in this thesis address image processing system

optimization at key phases in the design process and lead to significant improvements in performance, cost, and predictability of implementations that are derived from them.

SYSTEM SYNTHESIS FOR IMAGE PROCESSING APPLICATIONS

By

Dong-Ik Ko

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee:

Professor Shuvra S. Bhattacharyya, Chairman/Advisor
Professor Rama Chellappa
Professor Gang Qu
Professor Manoj Franklin
Professor Chau-Wen Tseng

Dedication

To my parents, and to Gye-sun, Yu-jin, and Yu-min

Acknowledgements

Thank the Lord! He led me to the completion of this thesis.

It was a wonderful blessing to study under Prof. Shuvra Bhattacharyya.

His encouragement and kindness has kept me focusing on my research with great passion.

I also thank my friends, Ming-Yung, Vida, Mainak, Chung-Ching, Sankalita, Jerry, Sebastian, Ivan, Celine, Lin, Sadagopan and many other friends!

I also thank brothers and sisters in my church for their prayers and loves.

My lovely wife; Gye-Sun and two daughters; Yu-Jin and Yu-Min! I am sorry for not having so much time with you.

Praise the Lord !!!

Table of Contents

List of Tables

List of Figures

Chapter 1 : Introduction

As the complexity of functionality in modern embedded systems increases along with the rising demand for multimedia processing capabilities, embedded systems are increasingly incorporating image processing capabilities in various forms. Many image processing applications impose critical performance constraints, require high volumes of data processing, and also require tight resource usage due to cost considerations. System design factors such minimizing the amount of on-chip memory needed and the efficient configuration and utilization of digital signal processor cores become especially important and challenging under these considerations.

The decision on an appropriate system architecture is difficult due to conflicting requirements, such as the need for a cost- and power-efficient integrated circuit footprint, and the simultaneous need for extensive data management, high throughput, and low latency. As technology advances for integrating multiple cores on a single integrated circuit, embedded multiprocessor platforms become attractive for addressing these challenges of image processing system implementation.

For such embedded multiprocessor platforms, image processing tasks must be scheduled effectively onto the available processors in a manner that effectively exploits the various forms of available parallelism, and the memory architecture must be organized and utilized to support high volume data buffering and efficient interprocessor communication. Useful to both of these steps is the application of appropriate design representations based on image-processing-oriented models of computation. Such representations expose high level application structure that designer and design

tools can use to explore the design space more efficiently, and derive more optimized and more predictable implementations.

This thesis addresses key problems in the design and implementation of multiprocessor image processing systems. In this thesis, we divide the embedded multiprocessor implementation process into the three inter-related phases of application modeling, task scheduling, and communication optimization, and we provide a comprehensive, integrated approach to these phases.

In the remainder of this chapter, we provide an overview of relevant background concepts and technology considerations, along with brief, motivational overviews of the methods that are developed in the thesis.

1.1 Background

1.1.1 Modeling

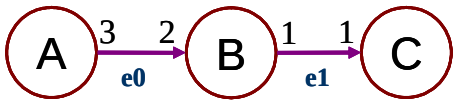
Modeling semantics based on dataflow graphs are used widely in design tools for digital signal processing (DSP). Dataflow is a directed graph called dataflow graph where vertices within the graph called actors represent computation and edges correspond to buffers between actors. These buffers hold data tokens which are delivered from the output port of one actor to the input port of another. An actor is ready for execution when all input ports of the actor have at least the minimum number of data tokens each input port requires for activation in the associated buffers. An actor consumes a certain number of tokens from its input ports and produces a certain number of tokens to its output ports when it is fired (executed).

Various kinds of dataflow models have been introduced for diverse purposes. Each dataflow model has different features and advantages in terms of expressivity and static (compile time) predictability of models. A common goal is to increase the flexibility of modeling an application in terms of expressivity while taking advantage of compile time predictability to reduce runtime overhead. Compile-time obtained information may include the estimation of a runtime memory usage and verification of valid schedule which guarantees the total number of data tokens produced within a dataflow graph is same as the total consumed number of data tokens within the same graph in one iteration.

1.1.1.1 Synchronous DataFlow (SDF)

Lee and Messerschmitt[63] have proposed the synchronous dataflow (SDF) model. SDF assumes that the number of tokens produced/consumed by each actor within a dataflow model is known at compile time. SDF enables us to predict bounded memory usage including code and data size statically and generate valid schedules at compile time. An optimal static schedule depends on the size of code and the size of data. Various valid schedules can be obtained based on the number of data tokens produced/consumed and the repetition vector. The repetition vector represents the number of firings of each actor. The repetition vector can be obtained through matrix computation with data tokens produced/consumed by each actor. Figure 1 shows an example of SDF graph. $e0$ represents the edge between actor A and actor B. $e1$ is the edge between actor A and actor B. A topology matrix of Eq 1 for a connected SDF graph can be built based on the number of tokens produced/consumed between actors within a SDF graph. The positive sign is set for the number of tokens produced and a minus

sign is set for the number of tokens consumed. A balance equation is built with a topology matrix as shown Eq2. In a balance equation matrix Eq3 of figure 1, columns of a topology matrix correspond to actors. Rows of a topology matrix correspond to edges. Finally, repetition vector q in eq 4 is obtained by solving eq3.



a) SDF graph

SAS : 2A3B3C, 2A3(BC)
MAS : ABABBCCC, ABABCCBC, ABCABCBC

b) valid schedules of a)

Figure 1. Example of SDF graph

$$T(e, v) = \begin{bmatrix} prd(e) & \text{if } v = src(e) \\ -cns(e) & \text{if } v = src(e) \\ 0 & \text{otherwise} \end{bmatrix} \quad (1)$$

$$T \bullet q = 0 \quad (2)$$

$$\begin{bmatrix} 3, -2, 0 \\ 0, 1, -1 \end{bmatrix} \bullet \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (3)$$

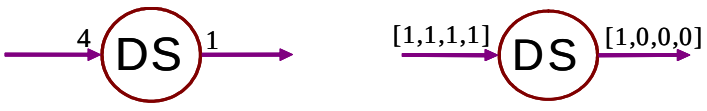
$$q = [2, 3, 3] \quad (4)$$

Figure 1 b) shows that figure 1 a) could have various valid schedules. For example, *SAS* in figure 1 b) represents a single appearance schedule where each actor appears only once in a schedule by exploiting looped schedule. SAS is good for reducing a code size. *MAS* in figure 1 b) is a multiple appearance schedule where each actor could appear multiple times to reduce buffer size between actors. For example, *SAS* schedule *2A3B3C* requires 6 tokens between actor A and actor B. *MAS* schedule *ABABBCCC* requires only 4 tokens between actor A and actor B. Thus, *MAS* is likely to be a better choice due to the advantage of further buffer size reduction at the expense of some code size increase when a buffer size dominates a total memory area

used. Despite the benefits of a static scheduling and a memory manage of SDF, as the need for the flexible expressivity for dataflow graphs increases, many other dataflow models are introduced.

1.1.1.2 Cyclo-Static DataFlow (CSDF)

As an extension of SDF. Cyclo-Static DataFlow (CSDF)[25] allows for modeling a dataflow graph whose actors can support a cyclic change of the number of data produced/consumed. Thus, over each iteration of a dataflow graph, actors under CSDF semantics can have different production and consumption rates in a cyclic and periodic pattern. Cyclo-Static DataFlow is more flexible than SDF in terms of the expressivity while maintaining a static predictability of a bounded buffer memory of SDF. For example[77], for the case of down-sampler actor by factor 4, in SDF semantics, the actor should wait for firing until the input port of the down-sampler actor holds at least 4 tokens. In CSDF semantics, the behavior of the down-sampler can be described in four different phases. The actor takes one token at the input port and produces one token through its output port for the first phase. And then the actor can take one token from its input port and produces zero token to the output port for the following three phases. Figure 2 shows the comparison of modeling of a down-sampler actor each under SDF and under CSDF semantic.



a) Under SDF b) Under CSDF

DS:Down sampler actor

Figure 2. Comparison of a down-sampler actor by factor 4 each under SDF and under CSDF

CSDF, as a generalization of SDF, increases the expressivity of dataflow model

but inevitably causes the complicated scheduling problem. As well, operational patterns of actors of dataflow under CSDF semantic are confined to be periodic. However, many image processing applications have the feature of unpredictable changes of the number of tokens produced/consumed in a non periodic manner. CSDF has the limitation to fully adopt the diverse needs of various complicated image processing applications.

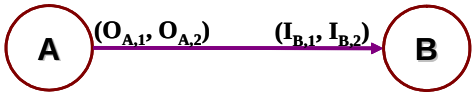
1.1.1.3 MultiDimensional Synchronous DataFlow (MDSDF)

SDF and other dataflow models takes only one-dimensional signal processing channel FIFO buffers and the associated one dimensional algorithms. As the demand for the multi-dimensional data processing increases, the efficient way of modeling two dimensional or higher dimensional data is necessary. As a generalized extension of SDF, multidimensional synchronous dataflow (MDSDF) is introduced. MDSDF extended the one dimensional FIFO queues used in SDF to array types of FIFO queues. Figure 3[70] shows the comparison of FIFO queues between a SDF model and a MDSDF model. In MDSDF, FIFO queue holds two dimensional data tokens. A balance equation for figure 3 a) is shown Eq 5. A balance equation of figure 3 b) under MDSDF can be extended to two balance equations for each dimension as shown in Eq 6. r represents repetition vector. O is the number of tokens produced. I is the number of tokens consumed.

$$r_A \cdot O_A = r_B \cdot I_B \tag{5}$$

$$\begin{matrix} r_{A,1} \cdot O_{A,1} = r_{B,1} \cdot I_{B,1} \\ r_{A,2} \cdot O_{A,2} = r_{B,2} \cdot I_{B,2} \end{matrix} \tag{6}$$

a) FIFO queue under SDF



b) FIFO queue under MDSDF

Figure 3. Comparison of FIFO queues under SDF and MDSDF model

MDSDF increases flexibility and expressivity while maintaining static schedulability of SDF model. However, as the data dimension and the complexity of an application graph under MDSDF increase, there is a high chance that unexpected errors can be smeared in the modeling process by a designer due to its dimensional complexity. As well, multidimensional distinction of data tokens leads to complicated scheduling problems even though MDSDF preserves data parallelism and functional parallelism through dimensional distinction of data tokens.

1.1.1.4 Boolean DataFlow (BDF)

Boolean dataflow (BDF) model by Buck[18] allows for each port to hold either a constant or a two-valued function for controlling a dataflow. This function is placed on a control port of an actor. A control token delivered through a control port of an actor controls the number of tokens transferred by a conditional data port. BDF extends the scheduling method for SDF graphs to process BDF actors with conditional ports, by associating symbolic expressions with conditional ports. By adding two simple control actors with a control port such as switch and select, conditional constructs like if-then-

else and do-while loops can be built under BDF.

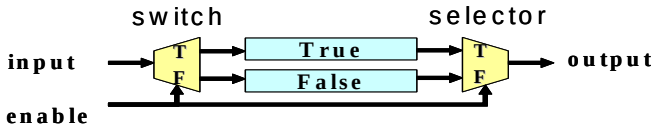


Figure 4. Control flow decision under BDF

Figure 4 shows how the switch actor and the selector actor under BDF semantic determine the number of tokens for an output port and an input port depending on a control token. In figure 4, the switch actor and the selector actor are BDF actors that take one token from the control input port and determine either a *True* route or *False* route depending on whether the value of the control token on *enable* in figure 4 is true or false.

A conditionally transferred data token allows for the runtime flow of a control to be determined based on the values of tokens on control ports. At compile-time, a scheduler analyzes the change of control flows based on values of control tokens. This enables us to build an annotated schedule which is a compile-time schedule where each firing of a BDF actor is linked with the runtime firing conditions.

BDF allows runtime change of a data flow while exploiting the benefit of compile time scheduling technique. However, BDF leads to the addition of redundant ports and paths for control token delivery. The change of token values of a BDF actor is limited to two cases. Building various conditional paths with multiple token values leads to a complicated graph topology with many switches and selectors.

1.1.1.5 Parameterized Synchronous DataFlow (PSDF)

A parameterized dataflow modeling emphasizes a hierarchical modeling of a dataflow and relates the underlying hierarchical dataflow to a subsystem. A parameter-

ized dataflow modeling framework allows a subsystem's behavior to be controlled by a set of parameters. These parameters can change at runtime by allowing the subsystem behavior to vary dynamically. Parameters can control the functional behaviors of subsystems as well as the token flow behavior of a dataflow graph. In parameterized dataflow model, the model can have different parameter configurations at each iteration of a graph. But, after parameters are configured, parameters are held during the corresponding iteration of a graph. Parameterized dataflow modeling is a meta-modeling technique which allows schedules of a graph to be expressed with meta variables of parameters enabling the use of quasi-static scheduling.

In quasi-static scheduling, the number of firings of actors could be annotated with meta-variable coefficients related to the values of parameters and those meta variable coefficients could be determined at runtime whereas firing orders of actors are determined at compile time. Thus, parameterized models allows dynamic reconfiguration of parameters.

Parameterized dataflow could be applied to any types of underlying dataflow graphs. As an extension of SDF semantic with parameterization, a parameterized synchronous dataflow (PSDF) is suggested.

PSDF adopts a hierarchical modeling of parametrization. A hierarchy represents an abstraction of subsystem. Parameters are used to control the functional behavior of hierarchical subsystems. PSDF specification consists of three distinct graphs: the init graph, the subinit graph and the body graph. Intuitively, the body graph models the main functional behavior of the subsystem, whereas the init and subinit graphs control the behavior of the body graph by appropriately configuring the body graph parame-

ters. The init graph is invoked prior to each invocation of the associated (hierarchical) parent subsystem while the subinit graph is invoked prior to each invocation of the associated body subsystem, thus allowing for two distinct reconfiguration of controls. Figure 5 shows an example of PSDF graph. Parent Φ has three sub graphs. Subinit graph sets parameters of the body graph before the associated body graph is fired. PSDF increases the expressivity by adopting parameterized modeling, and exploits a quasi-static schedule. PSDF model allows runtime reconfiguration of a dataflow model.

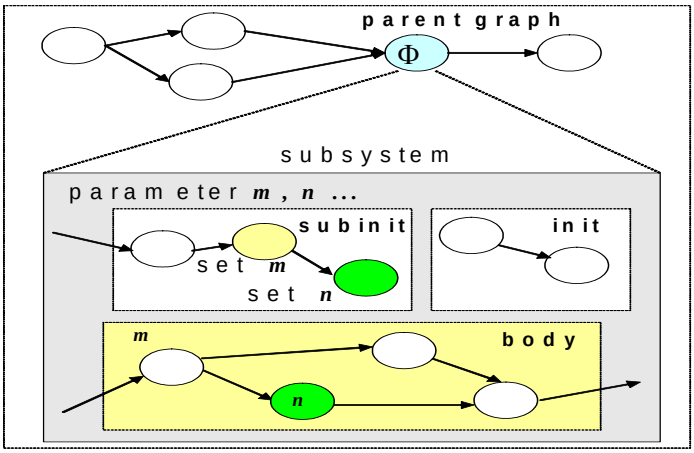


Figure 5. Example of PSDF model

1.1.2 Scheduling[94,95,96,97]

Mapping an application graph onto a multiprocessor architecture needs three major steps; **processor assignment**, **actor ordering** and **actor invocation**. The process assignment step corresponds to assignment of actors to processors. The actor ordering step is ordering the execution of tasks assigned to the same processor. The actor invocation step determines the time at which each actor starts execution. Actors are assumed to be non-preemptive. Once an actor is invoked on a processor, the pro-

cessor is allocated to the actor until the invocation completes. This is because preemption leads to a significant runtime context switch overhead and is of limited use in time-critical DSP embedded applications. These three steps can be performed at runtime (dynamic) or at compile time (static) depending on scheduling strategies.

Lee and Ha [64] suggested a scheduling taxonomy depending on scheduling strategies from a fully dynamic approach to a fully static approach. Performing as many of the three scheduling tasks as possible at compile reduces run time overhead specially for the applications with hard real-time constraints. Performing processor assignment and actor ordering at compile time is useful for a time-critical DSP applications. In general, runtime assignment and ordering allows a more flexible run time variations in terms of managing available hardware resources.

Depending on scheduling strategies, scheduling methods can be divided into four categories; fully static, self-timed, static assignment and fully dynamic scheduling.

In scheduling an application over multiprocessors, homogeneous SDF graph (HSDFG) is useful. In HSDF, every actor consumes and produces only one token from each of its inputs and outputs. A multirate SDF graph can be converted into an HSDF graph [61]. This conversion may lead to significantly increased number of actors in HSDF graph. However, this conversion process simplifies scheduling an application modeled by dataflow graph over multiprocessors. For algorithmic simplicity, HSDF graph can be converted into Acyclic Precedence Graph (APG) by removing edges with delays and replacing multiple edges between the same two actors in the same direction with a single edge. APG removes multiple edges leading to the identical precedence.

As a performance evaluation metric of schedules, the average iteration period (or makespan) is widely used. The average iteration period (or makespan) is time taken to execute all the actors in the graph once.

1.1.2.1 Fully static schedule.

In a fully-static strategy, assignment, ordering, and invocation are all performed at compile-time. The exact firing time of each actor is also determined at compile time. This technique is applied to scheduling VLIW processors [59] and synthesizing VLSI systems with guaranteed worst-case execution times[57].

Fully-static schedule can be expressed as a Gantt chart. In a Gantt chart the processors are arranged along the vertical axis. Elapsed times are marked along the horizontal axis. The actors are displayed as rectangles whose horizontal lengths correspond to the execution time of the actor. The left side of each rectangle in the Gantt chart corresponds to a starting time of the associated actor. Scheduling can be displayed by filling a Gantt chart with actors based on scheduling technique while minimizing the total schedule length and idle time slots.

Fully static schedules can be divided into two categories (blocked schedule and overlapped schedule) depending on the way of placing successive iterations of the HSDFG onto a Gantt chart.

1.1.2.1.1 Blocked schedule

In a blocked schedule, each iteration of the HSDFG is scheduled separately. Namely, executions of all actors in the previous iteration complete before the next iteration begins. Thus, dependencies between iterations are not considered. The schedule is

assumed to be repeated in a infinite periodic manner. Under a blocked schedule, the length of the critical path of the graph becomes a makespan.

1.1.2.1.2 Overlapped schedule

In an overlapped schedule, operations within a successive iteration of a graph can be overlapped with a previous iteration. To exploit an overlapped schedule, unfolding and retiming techniques are widely used. Unfolding schedules N iterations together where N is a blocking factor to improve a blocked schedule. However, unfolding leads to the increase of program size and complexity. Retiming manipulates delays in the HSDF graph to reduce the critical path in the graph[32,61].

Figure 6 [94] shows an example of a fully static schedule. Figure 6 c) shows a blocked schedule. Each iteration finish before the next one starts.

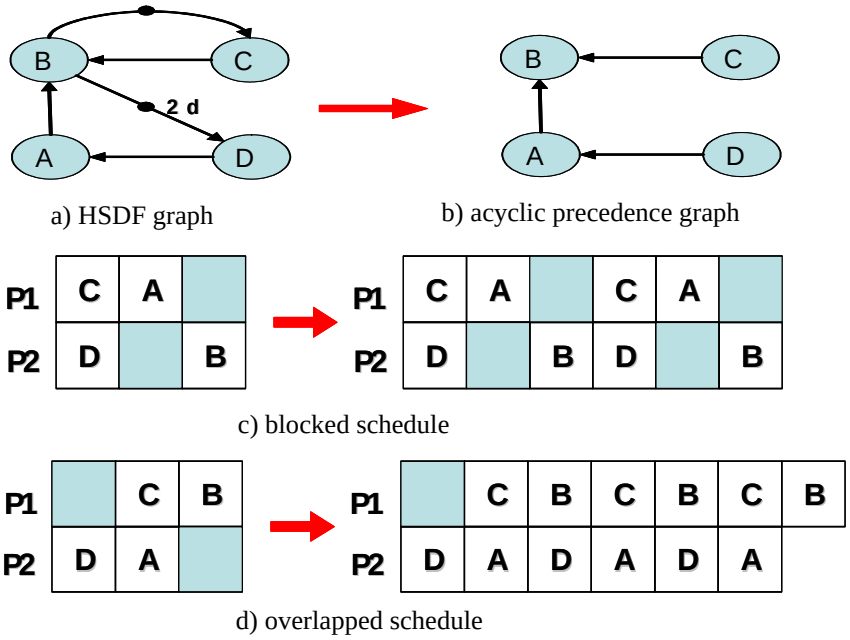


Figure 6. Fully static schedule

Figure 6 d) displays an overlapped schedule. Successive iterations in the HSDFG overlap. An overlapped schedule improves a makespan of a HSDF graph. The

makespan of the blocked schedule of Figure 6 c) occupies 3 time slots whereas the makespan of Figure 6 d) occupies 2 slots.

1.1.2.2 Self-timed schedule[60,61]

The fully-static strategy requires a precise estimation of actor execution times for processor communication synchronization and doesn't allow for the variations of execution times of actors. Self-timed schedule loosens this tight requirement by allowing the variations of execution times of actors. After the fully-static schedule, only the processor assignment and the firing orders of actors on each processor are retained while removing timing information among actors. Each processor holds a firing order of actors allocated to the process. Communication synchronization is performed at runtime by the associated processors. Runtime synchronization increases IPC cost and leads to a runtime bus arbitration. To reduce runtime communication cost, ordered transaction is introduced. Ordered transaction holds three scheduling information; the processors assignment, actor ordering and communication order at compile time. By making processors accesses to shared communication hardwares in an compile time obtained order, runtime arbitrations overhead can be alleviated.

1.1.2.3 Static assignment and dynamic scheduling

In a static assignment, only assignment of actors on processors is performed at compile-time but ordering and invocation of actors are performed at runtime. In fully dynamic scheduling, assignment, ordering, and invocation are all performed at runtime which is based on greedy approach and only guarantees locally optimal decisions. Dynamic scheduling also leads to resource contention problems at runtime. A static

scheduling approach may often lead to a better result.

This thesis provides an elaborate scheduling technique by applying a self-timed scheduling strategy to a pipelined processor manner while considering various constraints requirements.

1.1.3 Communication optimization

The communication optimization stage includes post optimization processes such as resource mapping or software communication optimization depending on application specific requirements and limitations. For example, after scheduling, a trade-off between resource costs and performance or between low power and high performance can be further exploited depending on priorities of an application's requirements. Applying the appropriate hardware or software communication optimization techniques can lead to reduced system cost or improved energy saving without sacrificing performance loss. This thesis studies two cases of an application specific post optimization technique each in terms of an efficient hardware mapping for resource cost reduction and a dataflow cutting technique for low power consumption.

In a hardware resource mapping study, this thesis contributes toward reducing hardware costs of FIFO buffers within a dataflow graph by analyzing data dependency of a dataflow graph without sacrificing performance loss. In a dataflow cutting technique, this thesis performed the case study of a sensor network application optimization in terms of power consumption minimization combined with the overall system performance improvement in conjunction with effects of communication traffic change on a sensor network.

1.2 Overview of the suggested techniques

In this section, brief descriptions of novel algorithms suggested in this thesis will be given in each category of system synthesis; modeling, scheduling and communication optimization. In modeling category, this thesis suggests two novel modeling techniques; Blocked DataFlow (BLDF) and Dynamically configured graph topology(DGT). In scheduling category, this thesis suggests a new multiprocessor based scheduling technique named Pipeline Decomposition Tree (PDT) scheduling. For communication optimization, this thesis suggests two new algorithms for communication optimization for a hardware and software mapping of a dataflow graph.

1.2.1 Modeling

1.2.1.1 Blocked DataFlow (BLDF)

In the digital signal processing (DSP) domain, rapid prototyping tools based on coarse-grain dataflow semantics are widely used [10]. One important requirement in these tools is support for block-based processing, such as that involved in image and video applications. A number of efforts have examined block processing at the level of individual actors. The scalable synchronous dataflow (SSDF) [53] model formalized this concept in the context of multirate dataflow graphs, and algorithms have been developed to extract the maximum vectorization potential from an SSDF graph [83]. More recently, retiming techniques have been explored for manipulating homogeneous dataflow graphs (graphs in which the production and consumption parameters are all equal to one) to improve vectorizability [58]. The objective in such vectorization is to improve throughput and reduce context-switching overhead by executing

actors many times in succession. BLDF(Blocked Dataflow) suggested in this thesis differs from these approaches in its applicability beyond the level of individual actors, and into arbitrary subsystems at any level of the modeling hierarchy. BLDF also differs in its close integration with parameterized dataflow semantics [9], which allows for powerful dynamic reconfiguration capabilities.

Modeling semantics based on dataflow graphs are used widely in design tools for digital signal processing (DSP). This thesis develops efficient techniques for representing and manipulating block-based operations in dataflow-based DSP design tools. In this context, a block refers to a finite-length sequence of data items, such as a sequence of speech samples, an image, or a group of video frames, as part of an enclosing data stream. We develop in this thesis a meta-modeling technique called blocked dataflow (BLDF) for augmenting DSP design tools with more effective blocked data support in an efficient and general manner. We compare BLDF against alternative modeling approaches through a detailed case study of an MPEG 2 video encoder system.

As dataflow modeling alternatives emerge further it is highly desirable to identify new modeling features that can be achieved through novel applications of existing models rather than defining a totally new dataflow variant for each new extension. This promotes reuse and integration rather than reinvention of the growing body of knowledge on established dataflow styles. BLDF adheres to this approach by defining general mechanisms that can be used to augment existing dataflow models with systematic data grouping capabilities. It is in this sense that we refer to BLDF as a meta-model. BLDF can be used with the well-known decidable dataflow models, SDF,

CSDF, MDSDF, and SSDF, as described above. Its use with other, more dynamic models such as boolean dataflow [17] and SBF [46] may be possible, although efficient application to such models requires further investigation.

Blocked data token delivery of BLDF enables us to reduce dimensions of MDSDF [70] by processing multi dimensional data tokens dimension by dimension with blocked data processing of nested BLDF subsystems. At the same time, BLDF can be used in conjunction with MDSDF, with BLDF parameter control used to define the boundaries of processing to be performed using MDSDF semantics.

We develop in this thesis a blocked dataflow (BLDF) modeling approach for efficient handling of block-based data in dataflow-based DSP design tools. BLDF combines meta-modeling, block-based processing, multidimensional representation, and dynamic parameter reconfiguration in a single, unified framework that leads to more efficient dataflow graphs for scheduling and software synthesis.

Blocked dataflow builds on parameterized dataflow semantics[9]. BLDF inherits most features of parameterized dataflow [9]. Thus, a BLDF specification (or subsystem) Φ also consists of three distinct graphs: 1) the *init* graph Φi; 2) the *subinit* graph Φs; and 3) the *body* graph Φb. Intuitively, the body graph models the main functional behavior of the subsystem, whereas the init and subinit graphs control the behavior of the body graph by appropriately configuring the body graph parameters. The init graph is invoked prior to each invocation of the associated (hierarchical) parent subsystem, $\texttt{parent}(\Phi)$, while the subinit graph is invoked prior to each invocation of the associated body subsystem Φb, thus allowing for two distinct "frequency levels" of reconfiguration control [9]. In a blocked dataflow subsystem, blocks of input

data are treated as subsystem parameters, and the initialization graphs (the subinit or init graphs, as described below) are used in-between processing of successive blocks to change the value of the associated block-parameter. Thus successive blocks of data are translated into successive reconfigurations of block-parameter values.

For example, consider an image processing system that performs a given filtering operation on a stream of input images. A blocked dataflow representation might define the processing of a single image using a dataflow graph G_c. The graph G_c operates on input from a special *image source* actor that is parameterized with an image I. The image source actor simply transfers its image parameter to its output according to the desired protocol. The transfer protocol involves both rasterization aspects, and may also involve *sub-blocking* (e.g., outputting the image as a sequence of row blocks). Such sub-blocking can be used to defined nested BLDF subsystems.

1.2.1.2 Dynamically configured graph topology(DGT)

Dataflow is widely used for designing DSP applications. Despite its intrinsic advantages, one weak point is its difficulty in flexible expression of applications with data dependent change in execution structure. To handle data driven changes in execution structure, several dataflow models such as CDDF [109], BDF [18], and BDDF [75], have been proposed. CDDF uses control tokens to determine the token transfer at an actor port. However, determination by a control token is applied to the actor in the next phase of execution, therefore, control tokens are not present at the moment that the actual phase is determined. BDDF introduces dynamic ports and an upper bound is provided for the data rate so that each dynamic port can keep the model bounded. However, control flow depends on FSMs. Using FSMs for minor changes of control

flow with dataflow graphs can make application models unnecessarily complicated and result in limited flexibility. BDF provides “SWITCH” and “SELECT” actors to determine control flow. For satisfying bounded memory and consistency, a symbolic function of probability is introduced. This function increases the complexity of solving the balance equations (for verifying sample rate consistency), and results in the possibility of “weak consistency,” which is less desirable in an implementation. This thesis suggests an approach to providing dynamically configured dataflow graph topologies using a new modeling and synthesis technique called DGT (Dynamic Graph Topology). DGT builds on PSDF semantics [84]. All possible graph topologies for a given graph are obtained at compile time and the corresponding graph based on parameters and data is dynamically set up in an efficient manner at runtime before the invocation of the associated graph.

To provide for more powerful and efficient data dependent execution related to application mode changes, where entire graphs or subsystem are replaced or reconfigured at run time, this thesis tackles dynamic set-up of dataflow graph topologies before the graphs are invoked. All configurations of possible graph topologies are pre-computed at compile time and stored for usage at run time. At runtime, the initialization step of DGT generates an appropriate graph topology based on parameters extracted from data being delivered and picks up a pre-computed schedule to fit the current parameter configuration.

However, not all configurations are valid or can be obtained at compile time. Some configurations may cause deadlock or inconsistency or may not be predictable at compile time. Reconfiguration of dataflow graphs is carefully considered in [73].

[73] analyzes the reconfiguration of a model based on behavioral types and extracts the *least change context* to check approximate semantic constraints. This thesis statically checks the validity of each configuration like [73] and keeps the scheduling results for use at run time.

The main distinguishing feature of DGT is that it efficiently supports multi-function applications by configuring graph topologies dynamically. There are two kinds of multi-function applications. The first, which we call type-I applications, are exclusive-or applications, where only one graph topology is selected from multiple sets of possible graph topologies for a given application. The other, which we call type-II applications, are concurrent applications where two or more applications with different graph topologies are running at the same time. This thesis focuses on type-I (exclusive-or) application for experimentation of DGT. For synthesis of type-I applications, [40] extracted *commonality measures* of each actor and used these values to determine a hardware bias of each actor by hardware oriented partitioning. This thesis focuses on software implementation, and applies novel scheduling techniques based on graph characteristics to reduce code and buffer size, which is critical for DSP software.

Systematic methods for reducing code and buffer size are applied based on characteristics of each configured graph. We have compared DGT against conventional modeling approaches through a detailed case study of an MPEG 2 video encoder system, and our experiments demonstrate the efficiency of the DGT approach. The DGT approach provides efficiency and flexibility in modeling applications with data driven change of graph topology from runtime parameter changes by using pre-computed information (information related to graph topology, scheduling, code/buffer size,

bounded memory, etc.).

1.2.2 Scheduling

1.2.2.1 Pipeline Decomposition Tree (PDT) scheduling

Scheduling an application under multiprocessors environment is a NP hard problem due to its complexity. Many heuristics or evolutionary[2][19][23][28][115] efforts have been proposed. Evolutionary algorithm can be used in case a deterministic algorithm cannot be easily applied. Under evolutionary approach, the manipulation of the effect of external constraints on the scheduling results is difficult due to its non deterministic optimization process. Besides an evolutionary approach, many heuristic algorithms have been exploited. Banerjee. [7] presented two-step approach by separating partitioning and process allocation under heterogeneous architecture. Hoang. [32] suggested a heuristic algorithm by providing detailed IPC cost model. Konstantinides. [53] tackled detailed issues in modeling I/O by subdividing I/O parts into sequential I/O parts and parallel I/O parts. However, these approaches overlooked the benefit of potential data parallelism that most DSP applications commonly have. Exploiting data parallelism contributes toward speed-up. Subhlok. [99] tackled data parallelism along with task parallelism for scheduling. However, this approach mainly focuses on a linearly chained dataflow. Applying data parallelism and task parallelism to an application with non-linearly connected dataflow paths causes more complicated and various difficult problems.

Modern embedded systems for digital signal processing (DSP) integrate more and more complicated functions in one system. As the complexity of functionality

increases, considering multiple processing units in one system is inevitable. The demand for the real-time response also grows along with various functionalities. Integration of multiple functions under tightly environmental constraints causes many complicated problems. Many efforts have been made for scheduling[29][37][69][78][93]and integrating an application over multiple processing units[10][11][12][13][14][29][71][85]. Researches mainly tend to focus on partial interactions of the overall problems environmental constraints may cause [3][65][68][74].

An application can be expressed as a dataflow graph of tasks. Many efforts tackling task dependencies of a graph have been widely taken to distribute the workloads of tasks over multiple processing units[85][86][93][97]. However, the internal operational features of each task was not widely exploited. Internal operations of a single task can be copied to multiple tasks and copied tasks can run in parallel over multiple processors. Finally a response time of the application can be reduced.

For this purpose, this thesis presents a deterministic scheduling method named **PDT scheduling** (Pipeline Decomposition Tree) by exploiting both heterogeneous data parallelism and task parallelism. In general, data parallelism allows multiple copies of a single task to run on multiple processing units. Operation of each task is independent of each other. Each copied task handles different sequences of data frames. Thus, a general data parallelism increase the overall buffer size since separate memory regions are required for holding different sequential data frames.

PDT scheduling suggests heterogeneous data parallelism model. Heterogeneous data parallelism is an extension of data parallelism. A single data frame can be divided

into smaller sub areas named copy-set. A sets of copied tasks can handle different copy-sets within a single data frame whose size can vary depending on available processors. Each copy-set can also be divided into sub regions. Thus, a single data frame consists of several copy-set-regions. Each copy set consists of sub regions. The size of a sub region is obtained by dividing the copy-set-region by the number of copied tasks allocated to the corresponding copy-set. Thus, all sub regions within a single copy-set are of the same size. But, the sizes of copy-set-regions may or may not be the same depending on available idle processors. The copy-set-region is an array of data tokens in a multi dimensional data stream frame, especially, two dimensional data tokens for most 2-D based image processing applications. Copied tasks can be allocated to different copy-sets whose sizes can vary. But, copied tasks allocated to the same copy-set handle the same size of sub regions within the corresponding copy-set-region. The number of tasks in a copy-set may vary from 1 to N depending on available idle processors. Ultimately, heterogeneous data parallelism allows for dynamic change of the size of sub regions and handles a single data frame by multiple processors without increasing the buffer size while exploiting the parallelism. The suggested technique tackles task parallelism by exploiting a pipelined architecture for the high throughput. The suggest scheduling technique provides constraints satisfactory solution by taking into consideration IPC communication cost model of a separate memory architecture[21][56][106][112] and a bus contention model of a shared memory architecture. Constraints could be the limitation of on/off chip memory size[100][108][113], latency, or throughput etc.

Most embedded systems for digital signal processing (DSP) integrate an image

processing application. The common feature of image processing applications is parallelism. The completion of the whole operations of a single task is based on an unit operation and each unit operation requires only a subset of neighboring data and each unit operation is independent of each other. This neighboring data can be a block or a window. The unit operation is called a **window based operation** in this thesis. The window based operation enables us to exploit potential parallelism by running a single task over multiple processors by task duplication[1][22][47][80]. This potential parallelism by a window based operation is called a **data parallelism** [55**]**[81**]**. Data parallelism hasn't been deeply exploited for a multi-processors based scheduling compared to **task parallelism**. Task parallelism exploits pipelined scheduling for improving throughput[4][16][20][26][36]. This thesis tackles heterogeneous data parallelism and task parallelism together for improving latency and throughput at the same time.

A lot of tasks in DSP applications have the feature of heterogeneous data parallelism due to their window based operation patterns. The representative application examples with a window based operation are image processing applications. We selected a complex image processing module consisting of multiple morphological operations like opening, closing, gradient, Laplacian, smoothing and top-hat simultaneously, Laplacian pyramid, Multi-resolution spline pyramid and MPEG2 encoder for experimentations.

Our scheduling algorithm basically chooses a pipelined architecture. Each stage of the pipeline can be mapped to multiple processing cores, which may or may not span over multiple DSP chips depending on the synthesis constraints. To determine the number of stages in a pipeline, this thesis suggests a new algorithm called **PDT**(**P**ipe-

line **D**ecomposition **T**ree) exploration process, which builds pipelines by a depth first search tree. By **PDT**, tasks are partitioned into stages of the pipeline[44][101]. Depending on a task dependency and relationship between neighboring tasks, different memory architectures and bus architectures are considered by **PDT scheduling**.

The suggested scheduling technique contributes toward finding a constraints satisfactory solution in consideration of memory architectures along with the studies of the associated communication models such as IPC model from a separate memory architecture or a bus contention model of a shared memory architecture.

1.2.3 Communication cost

1.2.3.1 Hardware communication optimization

Various efforts on dataflow graph mapping onto hardware implementations have been undertaken. For example, the approach of [30] exploits loop parallelism to map nested loop kernels onto a coarse-grained reconfigurable architecture. The approach of [33,34] uses direct mapping of each dataflow graph component (actor) onto the corresponding hardware resource. The approach of [38] uses shared resources and looped schedules. The approach of [40] analyzes a given set of applications to extract commonalities across nodes in different applications and uses them to bias the mapping of nodes in the partitioning process. For FPGA implementation, the approach of [92] provides a rapid system prototyping method through a component architecture and an associated set of software tools. The approach of [103] provides a pipelined asynchronous circuit mapping method. For pointer synthesis, the approach of [87] encodes pointer values and generates circuits that can dynamically access different locations

with each pointer reference. The approach of [105] points out that pointers can reference indices to RAM, registers or even wires in a hardware mapping. The approach of [8] applies an external memory for mapping FIFO buffers and implements real-time image convolution on an FPGA. The approach of [72] implements image processing applications on FPGAs and points out that such implementations lead to a large on-chip FIFO buffers that prevent flexible usage of FPGAs for image processing applications. The approach of [104] presents an elaborate technique for mapping global, static arrays to distributed communication structures while classifying four types of inter-process communication patterns. The approach of [110] studies memory optimization for embedded software, particularly the performance of cache-based systems. The approach of [107] presents a novel technique for background memory allocation in multi-dimensional signal processing applications based on dataflow analysis.

The efforts described above make useful contributions toward mapping application representations at various levels of abstraction into hardware implementations. However, the simultaneous analysis of both performance and cost implications when mapping image processing applications, which involve especially large volumes of data token delivery, has not been thoroughly investigated in previous work.

This thesis helps to bridge this gap by studying, in the context of mapping dataflow graphs into hardware, the relationship between token delivery methods (indirect, pointer-based token delivery vs. direct-reference, raw token delivery) and FIFO architecture. This thesis exploits pointer-based token delivery to reduce on-chip FIFO sizes, and also provides a range of efficient trade-offs between performance (latency and throughput) and FPGA resource cost through a novel FIFO mapping algorithm. This

thesis also shows how overall performance and cost vary in relation to the selected sub-frame size at which block processing is carried out. Finally, this thesis provides a new mapping algorithm for dataflow representations of image processing applications to reduce overall FPGA resource costs without significant performance loss.

1.2.3.2 Software communication optimization

This thesis studies a software communication optimization technique under the sensor network application domain in terms of power consumption minimization of a sensor network system and provides a dataflow graph cutting technique for mapping the divided graphs over multiple sensor nodes for minimizing communication traffics. In a sensor network, energy consumption of a sensor node is related to a network lifetime. To increase the network lifetime, low power friendly design of a sensor network is necessary. Many efficient approaches are suggested to reduce an energy consumption of a sensor network. [89] distributed FFT function over a master node and slave nodes to reduce energy consumption without consideration of data traffic change by moving FFT function from a cluster head node to slave nodes. [54] provides a trade-off of an energy and a latency by considering different computational capabilities for a master node and a slave node. However, [54] didn't consider the potential possibility of using a low computational micro controller by balancing functional workloads over sensor nodes. [66,91] suggested a hierarchical and physical layer driven sensor network design to reduce data traffic and energy consumption of a sensor node in connection with each physical function. However, the node optimization should be optimized in conjunction with a underlying protocol characteristics and change of data transmission method depending on specific characteristics of network related devices. This

thesis suggests an overall minimization of an energy consumption of a sensor network in connection with a trade-off of latency and network lifetime by balancing workload of each sensor node. This thesis exploits internal token flows of an application dataflow graph and divides the application over a master node and slave nodes by applying dataflow modeling technique. A sensor network application can be efficiently modeled under a dataflow semantics. By analyzing dataflow graph modeling an application[11,18,40], energy consumption and operational complexity of an application can be effectively estimated in a coarse grain level. Especially, parameterized dataflow semantic[9] is intrinsically friendly to reconfigurable demands of most sensor network applications. Parameterized dataflow allows for dynamic change of meta variables which can be mapped to internal parameters of an application. This thesis selects DGT[48] (Dynamic Graph Topology) method for modeling an application. DGT inherits from a parameterized dataflow and provides more efficiency by allowing for dynamic change of graph topologies based on runtime request. In DGT semantics, connection between nodes and the number of tokens produced/consumed by each node can be changed at runtime and be expressed along with reconfigurable parameters. This feature enables a master cluster to control slave nodes efficiently and allows each sensor node to support various graph topologies.

1.3 Contributions of this thesis

1.3.1 Modeling

In this thesis, we challenge new modeling techniques for image processing appli-

cations under a dataflow semantic while exploiting blocked processing and dynamic reconfigurability. This thesis suggests two new dataflow based modeling techniques named Blocked DataFlow (BLDF) and Dynamically reconfigurable Graph Topology (DGT), respectively.

1.3.1.1 Blocked DataFlow (BLDF)

This thesis suggests a new modeling technique named Blocked DataFlow (BLDF). Unlike other dataflow models, BLDF exploits a blocked processing feature of data tokens in a dataflow graph, which makes it possible to model most image processing applications. In BLDF, a blocked processing feature of multi dimensional data streams can be allowed in an automated manner. BLDF model enables the firing numbers of each actor within a dataflow graph to be expressed in meta variables. Meta variables are obtained through parameterization of blocked data tokens. Parameterized firing numbers allow for quasi-static schedule which can be reconfigured at runtime by the subinit sub system during the parameterization process of blocked data frames.

1.3.1.1.1 Iteration control

The major enhancement in BLDF is the delivery method of data tokens into body graphs. In BLDF, blocked data tokens such as sequential MPEG2 video streams are delivered via the parameter value updating process of init or subinit graphs so that an init or a subinit graph can extract information concerned for the associated body graph from raw data tokens delivered, and then convert raw data tokens as well as the information extracted into sets of new parameter values for the body graph. Thus, raw data

tokens are delivered to the associated body graph as parameters along with other parameters extracted from them before the body graph starts running.

Blocked tokens are transferred to the subinit graph and then converted into a block of parameters, which are set as parameters of each relevant actor in the associated body graph. Here, BLDF provides Dynamic configuration of parameters for the associated body graph such as image resolution and block size as basic processing units along with other provisional parameters at the stage of the subinit graph, which directs detailed operation of the associated body graph before that body graph starts an invocation of itself.

At the same time, iterations of each actor within a body graph can be obtained along with other parameters. Suppose, for example, that an init or a subinit graph takes a $\boldsymbol{Z}$ pixel frame from its input port. An init or a subinit graph can obtain $\boldsymbol{Z / N^2}$ iterations of the associated body graph actor by setting the *block size* parameter for the body graph as $\boldsymbol{N}$ by which image frames are divided into sub-image frames. Each actor within the body graph then operates on the basis of sub-image frames for high throughput and more parallelism. Iteration numbers may be used further as factors in a quasi-static looped schedule by a BLDF scheduler. Obtaining parameters relevant to the scheduling of the associated body graph before it runs and reconfiguring those parameters dynamically based on concerned payloads of tokens delivered at a runtime gives an application developer enhanced flexibility and efficiency in the design phase.

1.3.1.1.2 Token delivery

One of the advantages of BLDF is its efficiency in token delivery. First, in token delivery, BLDF enables us to reduce buffers required for delivering tokens among actors.

This is because tokens can be delivered from parent graphs to nested body graphs by parameterization. This parameterization process enables us to remove redundant connections and buffers between actors in BLDF.

1.3.1.1.3 Data tokens with nested headers

Most multimedia data tokens consist of a *header* part and a *payload* part. The header part has the information for handling the payload. However, the payload also may have sub-header and sub-payload components. Therefore, each level of composite actors implemented hierarchically or heterogeneously may process a different area of a packetized multimedia data token. BLDF provides an efficient way for delivering data tokens to composite actors of lower hierarchical levels by parameterization. Only the relevant part needs to be decoded for configuration and the remaining parts can be encapsulated as parameters for composite actors of lower hierarchical levels in the dataflow specification. Decoding headers sequentially according to the need for the associated header information allows us to implement each module within an application consistently, which is easy to understand for future code reuse. This approach also reduces the number of connections and buffers required between actors by parameterization.

1.3.1.2 Dynamically reconfigurable Graph Topology (DGT)

1.3.1.2.1 Modeling of separate dataflow graphs in a single dataflow semantic.

This thesis suggests a new modeling technique named Dynamically reconfigurable Graph Topology (DGT). Unlike other approaches challenging the change of data/control flow within dataflow models, DGT allows separate individual dataflow

graphs to be integrated in a single dataflow semantic. Under DGT semantic, vertices and edges within a dataflow graph can be categorized into two groups; fixed or varying. In DGT domain, Any vertex/edge whose topological behaviors are commonly constant among individual dataflow graphs can be marked fixed edge/vertex. Any vertex/edge not marked as fixed graphic components belongs to varying vertex/edge.

In DGT, the topological behaviors of varying edges/vertices can be dynamically changed based on the change of parameters or tokens being delivered while allowing for dynamic change of graph topologies and a single dataflow integration of separate individual dataflow graphs.

1.3.1.2.2 Minimization of resource usage among separate dataflow graphs

In modeling of separate dataflow graphs which share operational functionality or have topological similarity, separate modeling for each dataflow graph may lead to unnecessarily increased buffer/code size due to overlapped resources among the dataflow graphs modeled. DGT allows separate dataflow graphs to be integrated in a single dataflow semantic. By analyzing the shared functionalities and graph topological patterns among separate dataflow graphs, DGT minimizes an overall resource usage of dataflow graphs.

1.3.1.2.3 Dynamic reconfiguration of a graph topology

In DGT semantic, Memory usage and scheduling information of each possible graph topology are obtained at compile time. By inheriting the characteristics of PSDF semantic, DGT semantic consists of three sub graphs; init, subinit and body graphs.

Under DGT semantic, subinit system dynamically configures the graph topology

of the associated body graph and applies the corresponding precomputed scheduling information to the configured body graph before the body graph is invoked. Ultimately, DGT increases the expressivity and the flexibility of a dataflow graph model by allowing runtime reconfiguration of a graph topology based on runtime change of parametric variables.

1.3.2 Scheduling

As multiprocessors based scheduling technique, this thesis suggests a deterministic heuristic scheduling method named PDT(Pipeline Decomposition Tree)-schedule.

1.3.2.1 Pipeline Decomposition Tree (PDT) scheduling

1.3.2.1.1 Constraint aware multiprocessor scheduling for non-linearly linked dataflow graph

Unlike other existing approaches for multiprocessor scheduling methods, PDT scheduling considers complicated environmental constraints such as memory constraints, performance requirement or architectural limitation and provides influence of each individual constraint or interference of individual constraints on scheduling. PDT scheduling also provides an automatic shielding method, especially, for non-linearly configured application graph.

1.3.2.1.2 Exploitation of heterogeneous data parallelism with task parallelism

PDT scheduling exploits a pipelined processors architecture and tackles data parallelism and task parallelism together for improvement of both latency and throughput. To improve throughput, this thesis exploits a task parallelism which can be obtained

through a pipelined processing of an application. Besides task parallelism, this thesis suggests a novel concept of data parallelism named a heterogeneous data parallelism. A heterogeneous data parallelism improves both latency and throughput at the same time without buffer size increase. A general data parallelism usually increases the buffer size since duplicated tasks handle different sequential data frames and require separate memory areas for each sequential data frame. In heterogeneous data parallelism, duplicated tasks handles different divided regions within a single data frame without causing buffer size increase.

Thus, PDT scheduling contributes toward maximizing the performance of an application over multi processors environment under complicated environmental constraints such as resource usage limitation, performance requirements and architectural constraints.

1.3.2.1.3 Automatic pipelined multiprocessor architecture generation

Under PDT scheduling, multiple pipelines with different scheduling trade-offs are automatically generated through the PDT scheduling's pipeline exploration process. Pipeline exploration process recursively divides a given application graph into sub graphs by taking into account characteristics of a graph such as data dependency, execution time distribution of stages of a pipeline, operational characteristics of each actor in a depth first search way and finally generates pipelines with various potential performances and resource usage.

1.3.3 Communication optimization

This thesis suggests two novel post optimization techniques in terms of hardware and software communication optimization.

1.3.3.1 Minimization of FIFO buffer cost

In hardware communication optimization, this thesis tackles different features of memory devices in terms of performance and cost. Thus, FIFO buffers modeled within a dataflow graph could be mapped to memory devices with different performances and costs. This thesis reduces an overall hardware resource cost for FIFO mapping by analyzing data dependencies among actors (nodes) within a data flow graph. The suggested technique allows a maximal use of low cost memory devices for the synthesized system without performance loss.

1.3.3.2 Minimization of network communication cost

This thesis provides an efficient communication optimization technique for software code mapping of a dataflow graph for a sensor network application by redistributing a dataflow graph over multiple sensor nodes. This technique reduces a communication traffic and an overall power consumption of a sensor network, which are the most critical problems in a sensor network application design. This is a new approach in that this technique analyzes internal data token exchange rates of a dataflow graph for reducing communication cost between sensor nodes in consideration of response time change of an application. This is possible through finding a cutting line of a dataflow graph by tracking of edges with the lowest data token exchange rate within a dataflow graph. Based on the cutting line, a dataflow graph is divided into

two sub-graphs and sub-graphs are distributed to hierarchically clustered sensor nodes. The technique contributes toward increasing network lifetime by allowing a longer battery lifetime through reduced power consumption.

1.4 Outline of thesis

In chapter 1 (Introduction), the thesis introduced three sub categories of system synthesis process defined in this thesis; modeling, scheduling and communication optimization. In each description of modeling, scheduling and communication optimization, the thesis described major challenging issues and various present research efforts to solve these issues followed by a brief description of novel research studies suggested by this thesis belonging to each categories of system synthesis.

Chapter 2, as a modeling method of DSP systems, especially image processing applications, describes two individual novel modeling techniques suggested by this thesis separately. The first is named Blocked DataFlow (BLDF) which exploits a blocked processing feature most image processing applications commonly have. The other modeling technique is named Dynamic Graph Topology graph (DGT). DGT allows for runtime dynamic change of dataflow graphs under the variations of compile time obtained configurations. Chapter 3 describes a novel scheduling technique named PDT scheduling for scheduling image processing applications modeled under a dataflow semantic onto multiprocessors environment. PDT scheduling considers various system constraints such as memory usage limitation for on-chip or external memory, performance requirement (latency/throughput) in consideration of many architecture related challenges such as a shared memory architecture or a separate memory archi-

tecture and different communication costs (IPC or bus contention) related to memory models. Finally, PDT scheduling generates a pipelined architecture of processors through the suggested PDT(Pipeline Decomposition Tree) exploration process and exploits a data parallelism and a task parallelism together for improving latency and throughput at the same time. Chapter 4 describes two novel post optimization techniques as hardware/software communication optimization technique. The communication optimization technique exploits an application specific requirements in terms of power consumption, performance and resource cost. Finally, chapter 5 summarizes the results and discusses possible directions for the related future works.

Chapter 2 : Modeling of DSP applications

2.1 Introduction

In the previous chapter, we categorized the system synthesis technique into three areas; modeling, scheduling and communication optimization. We described the background technologies related to the system synthesis technique for DSP based embedded system in each category and briefly described motivations and the major contributions of the suggested techniques.

In this chapter, we describe major features and contributions of two suggested novel modeling techniques; Blocked DataFlow (BLDF) and Dynamically configured Graph Topology (DGT). A preliminary summary of part of this chapter is published in [48][49]

2.2 Blocked Dataflow Graph (BLDF)

2.2.1 Abstract

Modeling semantics based on dataflow graphs are used widely in design tools for digital signal processing (DSP). This thesis develops efficient techniques for representing and manipulating block-based operations in dataflow-based DSP design tools. In this context, a block refers to a finite-length sequence of data items, such as a sequence of speech samples, an image, or a group of video frames, as part of an enclosing data stream. We develop in this thesis a meta-modeling technique called blocked dataflow

(BLDF) for augmenting DSP design tools with more effective blocked data support in an efficient and general manner. We compare BLDF against alternative modeling approaches through a detailed case study of an MPEG 2 video encoder system.

2.2.2 Related work

In the digital signal processing (DSP) domain, rapid prototyping tools based on coarse-grain dataflow semantics are widely used [10]. One important requirement in these tools is support for block-based processing, such as that involved in image and video applications. We develop in this thesis a blocked dataflow (BLDF) modeling approach for efficient handling of block-based data in dataflow-based DSP design tools. BLDF combines meta-modeling, block-based processing, multidimensional representation, and dynamic parameter reconfiguration in a single, unified framework that leads to more efficient dataflow graphs for scheduling and software synthesis.

In this thesis, by a dataflow model of computation (dataflow MoC), we mean a programming model based on dataflow semantics. Programs in a dataflow MoC are thus represented as directed graphs in which vertices, called dataflow actors, represent computational tasks, and edges represent logical FIFO communication channels between tasks.

A decidable dataflow model is one in which deadlock and unbounded buffer accumulation can be determined in finite time for every specification in the model. Examples of decidable dataflow models are CSDF [99], SDF [63], MDSDF [70] and SSDF [53]. For consistent specifications in each of these models, there is a unique, integer-valued repetitions vector that is indexed by the graph actors and gives the number of

times each actor needs to be invoked to form a minimal periodic schedule for the graph.

A number of efforts have examined block processing at the level of individual actors. The objective in such vectorization is to improve throughput and reduce context-switching overhead by executing actors many times in succession. The scalable synchronous dataflow (SSDF) [53] model formalized this concept in the context of multirate dataflow graphs, and algorithms have been developed to extract the maximum vectorization potential from an SSDF graph [83]. More recently, retiming techniques have been explored for manipulating homogeneous dataflow graphs (graphs in which the production and consumption parameters are all equal to one) to improve vectorizability [58]. BLDF differs from these approaches in its applicability beyond the level of individual actors, and into arbitrary subsystems at any level of the modeling hierarchy. BLDF also differs in its close integration with parameterized dataflow semantics [9], which allows for powerful dynamic reconfiguration capabilities.

As dataflow modeling alternatives emerge further it is highly desirable to identify new modeling features that can be achieved through novel applications of existing models rather than defining a totally new dataflow variant for each new extension. This promotes reuse and integration rather than reinvention of the growing body of knowledge on established dataflow styles. BLDF adheres to this approach by defining general mechanisms that can be used to augment existing dataflow models with systematic data grouping capabilities. It is in this sense that we refer to BLDF as a meta-model. BLDF can be used with the well-known decidable dataflow models, SDF, CSDF, MDSDF, and SSDF, as described above. Its use with other, more dynamic models such

as boolean dataflow [17] and SBF [46] may be possible, although efficient application to such models requires further investigation.

2.2.3 Blocked dataflow

Blocked dataflow builds on parameterized dataflow semantics [9]. In a blocked dataflow subsystem, blocks of input data are treated as subsystem parameters, and the initialization graphs (the subinit or init graphs, as described below) are used in-between processing of successive blocks to change the value of the associated block-parameter. Thus successive blocks of data are translated into successive reconfigurations of block-parameter values.

For example, consider an image processing system that performs a given filtering operation on a stream of input images. A blocked dataflow representation might define the processing of a single image using a dataflow graph G_c. The graph G_c operates on input from a special *image source* actor that is parameterized with an image I. The image source actor simply transfers its image parameter to its output according to the desired protocol. The transfer protocol involves both rasterization aspects, and may also involve *sub-blocking* (e.g., outputting the image as a sequence of row blocks). Such sub-blocking can be used to defined nested BLDF subsystems.

BLDF inherits most features of parameterized dataflow [9]. Thus, a BLDF specification (or subsystem) Φ also consists of three distinct graphs: 1) the *init* graph Φi; 2) the *subinit* graph Φs; and 3) the *body* graph Φb. Intuitively, the body graph models the main functional behavior of the subsystem, whereas the init and subinit graphs control the behavior of the body graph by appropriately configuring the body graph parame-

ters. The init graph is invoked prior to each invocation of the associated (hierarchical) parent subsystem, parent(Φ), while the subinit graph is invoked prior to each invocation of the associated body subsystem Φb, thus allowing for two distinct “frequency levels” of reconfiguration control [9].

2.2.3.1 Iteration control

The major enhancement in BLDF is the delivery method of data tokens into body graphs. In BLDF, blocked data tokens such as sequential MPEG2 video streams are delivered via the parameter value updating process of init or subinit graphs so that an init or a subinit graph can extract information concerned for the associated body graph from raw data tokens delivered, and then convert raw data tokens as well as the information extracted into sets of new parameter values for the body graph. Thus, raw data tokens are delivered to the associated body graph as parameters along with other parameters extracted from them before the body graph starts running.

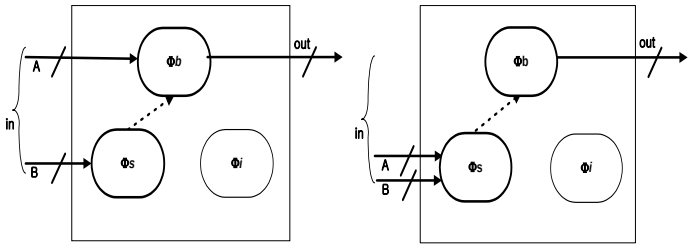
Figure 7 shows the mechanism by which BLDF builds on parameterized dataflow semantics.

Since the body graph of Figure 7(a) takes image frames directly from the outside without any parameterization process within an init or subinit graph, it is not possible to extract important information such as iterations of the associated body graph and also not possible to define detailed operation of each actor within that body graph by setting iteration limits.

On the other hand, in figure 7(b), image frames are transferred to the subinit graph and then converted into a block of parameters, which are set as parameters of each relevant actor in the associated body graph. Figure 7(b) allows dynamic configuration of

parameters for the associated body graph such as image resolution and block size as basic processing units along with other provisional parameters at the stage of the sub-init graph, which directs detailed operation of the associated body graph before that body graph starts an invocation of itself.

At the same time, iterations of each actor within a body graph can be obtained along with other parameters. Suppose, for example, that an init or a subinit graph takes a $\boldsymbol{Z}$ pixel frame from its input port. An init or a subinit graph can obtain $\boldsymbol{Z / N^2}$ iterations of the associated body graph actor by setting the *block size* parameter for the body graph as $\boldsymbol{N}$ by which image frames are divided into sub-image frames. Each actor within the body graph then operates on the basis of sub-image frames for high throughput and more parallelism. Iteration numbers may be used further as factors in a quasi-static looped schedule by a BLDF scheduler. Obtaining parameters relevant to the scheduling of the associated body graph before it runs and reconfiguring those parameters dynamically based on concerned payloads of tokens delivered at a runtime gives an application developer enhanced flexibility and efficiency in the design phase.



a) PSDF approach b) BLDF approach

A: major data tokens (e.g. image frames)
B: general data tokens for parameterization

Figure 7. PSDF and BLDF.

2.2.3.2 Token delivery

One of the advantages of BLDF is its efficiency in token delivery. First, in token delivery, BLDF enables us to reduce buffers required for delivering tokens among actors. This is because tokens can be delivered from parent graphs to nested body graphs by parameterization. Figure 8 shows how BLDF reduces buffering requirements in this way. In Figure 8, the “D” actor requires both “a” and “b” tokens, while the “A”, “B” and “C” actors require only token “a”. Here, suppose also that a sample rate change from “A” to “D” exists in the specification. Then in Figure 8(a), “A”, “B” and “C” actors must have additional input/output ports only for delivering token “b” to “D” without sample rate inconsistency. This in turn causes “redundant” or “extra” buffers between intermediate actors. However, in Figure 8(b), the subinit graph Φs converts input data into two parameters “a” and “b”, and then token “a” is set to actor “A” as a parameter while token “b” is set to the actor “D” directly as a parameter, while maintaining sample rate consistency. This parameterization process enables us to remove redundant connections and buffers between actors in BLDF.

2.2.3.3 Data tokens with nested headers

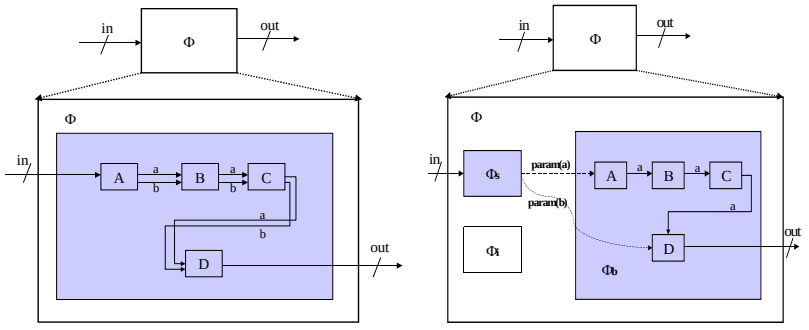
Most multimedia data tokens consist of a *header* part and a *payload* part. The header part has the information for handling the payload. However, the payload also may have sub-header and sub-payload components. Therefore, each level of composite actors implemented hierarchically or heterogeneously may process a different area of a packetized multimedia data token. BLDF provides an efficient way for delivering data tokens to composite actors of lower hierarchical levels by parameterization. Only the relevant part needs to be decoded for configuration and the remaining parts can be

encapsulated as parameters for composite actors of lower hierarchical levels in the dataflow specification. Figure 9 shows how data tokens with nested headers can be handled in BLDF. Decoding headers sequentially according to the need for the associated header information allows us to implement each module within an application consistently, which is easy to understand for future code reuse. This approach also reduces the number of connections and buffers required between actors by parameterization.

2.2.4 Application example

2.2.4.1 Brief review of MPEG2 video streams

The MPEG2 specification has been widely selected as a standard for coding/decoding moving picture frames. Therefore, many modern embedded systems handling multi media integrate MPEG2 decoders. This thesis has selected MPEG2 as one example of a real field application for an embedded system. The MPEG2 specification roughly consists of three parts: the video, audio and system parts. In this thesis, we focus on the video part to show differences in efficiency, flexibility and extensibility among alternative modeling formats.



a) SDF b) BLDF

Figure 8. BLDF and SDF: **param(): parameterization; Φs: subinit graph, Φb: body graph; “a”, “b”: tokens being delivered.**

Moving pictures are made from combinations of consecutive image frames. Each image frame is composed of pixels and each pixel has its own value representing the degree of RGB or YCrCb. Pixel values are not independent but are correlated with their neighbors. Therefore, the value of a pixel is predictable, given the values of neighboring pixels. Image frames usually have redundant information in view of image compression, which can be categorized into two redundancies: spatial redundancy and temporal redundancy, based on whether they are exploited in relation with neighboring frames or not. Spatial redundancy is redundant information lying in an intra frame while temporal redundancy is redundant information lying between inter-frames.

The MPEG2 specification separates image frames into three different types (I, P and B frames). I frames exploit only spatial redundancy, while P and B frames exploit both spatial redundancy and temporal redundancy. Thus, an I frame does not refer to neighboring image frames for reducing redundant information within itself and plays a role of an anchor frame to separate groups of pictures from continuous image frames.

Even though the P and the B frames exploit both spatial redundancy and temporal

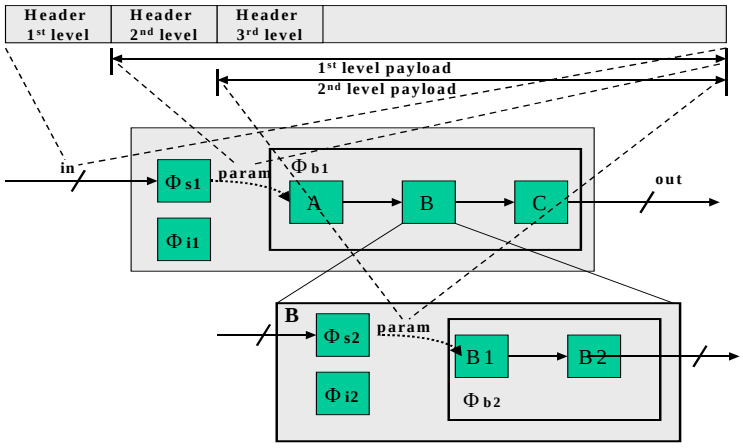


Figure 9. Data tokens with nested headers.

redundancy, there are different features between P and B frames in view of control flow. The P frame reduces redundant information by referring to a previous I or P image frame as a reference frame, differentiating pixel values between the current P frame and the reference frame, and exploiting spatial redundancy like the I frame. In contrast, the B frame requires two reference frames (a previous I or P frame and a future I or P frame) as reference frames for reducing temporal redundancy. The difference in the number of reference frames required among frame types makes it difficult to express an MPEG2 encoder in pure SDF form.

2.2.4.2 Problems in design of an MPEG video encoder with SDF

The problems from designing an MPEG2 video encoder using only SDF semantics occur from the dynamic change in MPEG2 video streams. Some actors inside the MPEG2 encoder dynamically change their operation based on the content of data tokens being delivered to them while other actors maintain their operation consistently. Also, motion compensation demands that image frames are encoded in different sequences from sequences transferred to the encoder. More specifically, problems in designing an MPEG2 video encoder under SDF are as follows.

- P1. *Control problem*. Every actor under SDF must consume and produce at least one token, which means that every connection between actors has to deliver at least one token during one invocation of the enclosing system. However, it is possible that some actors need special tokens from their input ports only in special cases and in other cases do not need any token. This situation arises in actors of an MPEG2 video encoder.
- P2. *Consistent schedule problem.* Data tokens can be categorized into two sub-

classes: major data tokens every actor is concerned with, and additional data tokens that are relevant for proper subsets of actors. Some actors of an MPEG2 video encoder require additional input or output ports that are only for delivering additional tokens. Those tokens have features of parameters and are usually used for setting internal state of actors. With such additional input or output ports only for delivering tokens to other actors, as the layout of applications get more and more complex, the possibility of introducing sample rate inconsistency into the dataflow signal processing increases. SPDF (Synchronous Piggybacked Data Flow) [76] suggested a piggybacked way to solve this problem. However, [76] also cannot avoid unnecessary and redundant delivery of the information, even if the methods of [76] are used to reduce buffers required by a piggybacked way, which delivers only a pointer of an entry in the global state table.

- P3. *Iteration counts.* Obtaining actor iteration counts at compile time is a major advantage in SDF. It reduces overhead of scheduling problems at a runtime. However, in general, the invocations of each actor can vary dynamically based on data being delivered. Such scenarios are not handled by SDF.

Also, an application developer may wish to manually set or dynamically change iteration numbers of special actors for low power requirements or quick user response time, which will affect iteration counts of subsequent actors. Such situations are also not permitted in SDF.

However, in BLDF, iteration numbers of subsequent actors can be determined at the “init” or “subinit” stage by extracting corresponding information from data tokens delivered and reconfiguring the associated parameters, while allowing for low over-

head quasi-static scheduling, as in parameterized dataflow [9]. This is possible through blocked parameter delivery in BLDF, which takes a block of input tokens, e.g. image frames at the init or subinit stage, and then converts them as blocked parameters along with other parameters. At the same time, important configuration information such as the resolution of an image frame and basic processing unit size (block size) can be used for dynamically calculating iteration counts of relevant actors in the associated body graph.

- P4. *Saving buffers and reducing unnecessary delivery.*

BLDF allows us to optimize data token delivery by "parameterization". By "parameterization", low overhead, "low frequency" connections between actors can be used. As mentioned in **P2,** we have two kinds of data tokens: tokens every actor requires and tokens that are relevant for individual actors. The second type of tokens can be directly delivered to the associated actors by parameter settings processed at the init or subinit stage. This allows us to remove unnecessary data delivery as well as unnecessary buffering requirements, as will be demonstrated in Section 2.2.5.

2.2.5 Experiments

We have prototyped a preliminary version of BLDF semantics in Ptolemy II [62], a widely-used tool for developing and integrating models of computation.

2.2.5.1 MPEG2 Video encoder implementation

We have implemented an MPEG2 Video encoder under the Ptolemy II environment in three different ways, including using BLDF, and have compared the resulting models in efficiency and flexibility.

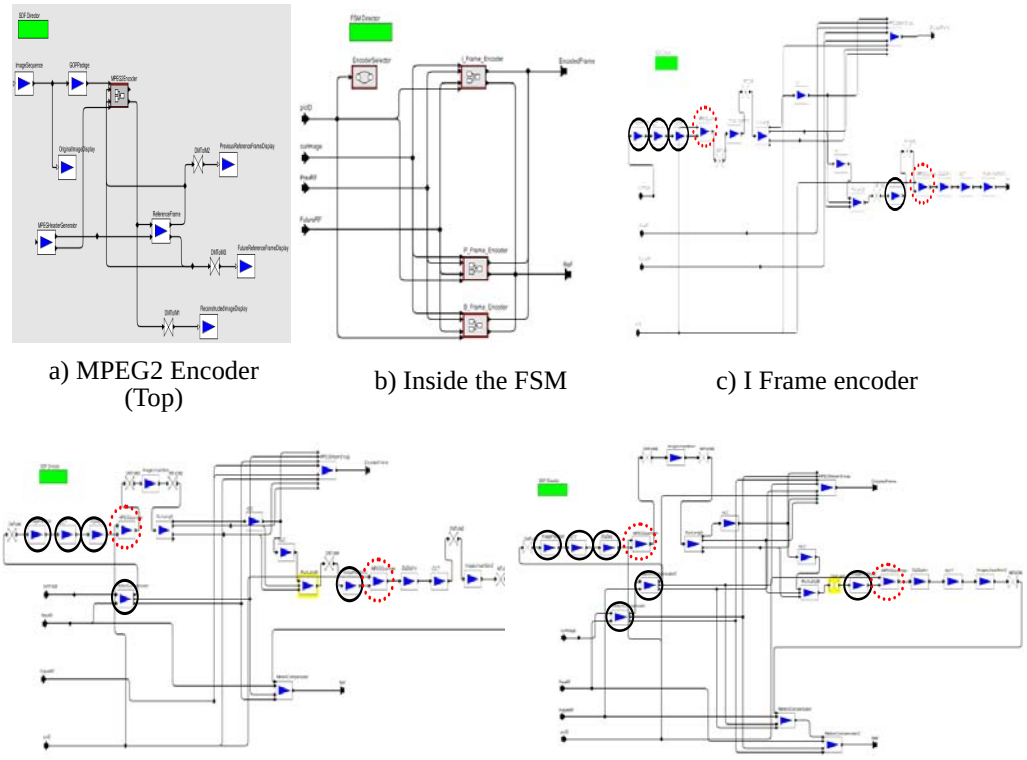
2.2.5.1.1 Method 1. FSM and SDF combination

An application developer often considers FSMs (Finite State Machines) when designing an application with nontrivial control flow. An MPEG video encoder clearly has features of dataflow, along with nontrivial control flow. In this method of implementation, we have used the two combined models of computation, SDF and FSM, in a heterogeneous and hierarchical way, using the heterogeneous modeling capabilities of Ptolemy II. Figure 10 illustrates our resulting design.

Our FSM representation within the MPEG2 video encoder has three states where each state is refined to three different SDF subgraphs, depending on the type of image frame: I, P or B. Since an I frame is coded by exploiting only spatial redundancy, the SDF graph shown in figure 10(c) for I frame processing does not have a motion compensator module. The SDF graph shown in figure 10(d) for P frame processing, which refers to only a previous I or P frame, has one motion compensator module, while the SDF graph shown in figure 10(e) for B frame processing, which refers to both a previous and a future I or P frame, has two motion compensator modules.

Here, it is useful to focus on two special functional blocks: *MPEGQuantizer* and *ReferenceFrame,* which help to distinguish our alternative encoder implementations.

MPEGQuantizer. This block needs a picture ID token to identify what image frames are delivered to it. MPEGQuantizer is placed after several preceding actors that are not concerned about the picture ID token. In implementation method 1 and method 2 (introduced below), the picture ID token must go through all preceding actors to the target actor, MPEGQuantizer, which, due to sample rate changes through the preceding actors, consumes that token to avoid an inconsistent schedule.



a) MPEG2 Encoder (Top)

b) Inside the FSM

c) I Frame encoder

d) P Frame encoder

e) B Frame encoder

Figure 10. FSM and SDF Combination

ReferenceFrame. This block operates differently, depending on the type of image frame delivered, and uses dummy tokens with “0” values:

Case 1: When an I frame comes, ReferenceFrame produces “0” values to output ports both for a previous and for a future reference frame. This is because an I image frame does not perform motion compensation. ReferenceFrame consumes I frame from its input port and updates its reference frame with the “I” frame. Here, ReferenceFrame has initial tokens as with a *delay* actor, for it is connected within a feedback loop.

Case 2: When a P frame comes, ReferenceFrame produces a previous I or P frame, which was saved in a previous cycle, for the previous reference frame and a “0” value for the future reference frame. Like when an I frame ID comes, a P frame is also

saved as a reference frame inside of ReferenceFrame.

Case 3: When a B frame comes, ReferenceFrame produces two saved reference frames (P and I frames) to the output ports. However, since a B frame is not used as a reference frame, it is discarded and not used for updating reference frames inside of ReferenceFrame.

In summary, this implementation method (Method 1) can satisfy problem P1; however, P2, P3 and P4 remain unsolved.

2.2.5.1.2 Method 2. SDF

In this method, we have implemented an MPEG2 Video encoder without integrating the FSM model of computation. All functional blocks inside are same as the method 1. However, method 2 does not have separated I, P and B sub-encoders so that all image frames go through two motion compensators with real values or dummy values depending upon the image frames. This implementation simplifies the design of an MPEG2 Video encoder. However, it still has the same problems (P2, P3 and P4) unsolved, as with method 1.

2.2.5.1.3 Method 3. BLDF

In this method, we separate the functional blocks of an MPEG2 video encoder into two parts: a subinit and a body graph. The actors configuring the body subsystem are placed in the subinit graph, and the actors actually processing image frames are placed in the body subsystem. First, the subinit graph obtains information required for configuring a body subsystem from data tokens delivered to itself and then converts image

data tokens, themselves, into blocked parameters for the body subsystem along with other parameters, such as block size and picture ID, obtained from image data tokens.

In parameterized dataflow, blocked data tokens such as image frames directly go to a body graph. An init or subinit graph manipulates only data tokens with parameter features for a body subsystem. Therefore, an init or subinit graph can not obtain parameters such as image resolution or block size for manipulating iteration numbers of the actors in the associated “body” graph.

Early knowledge of the iteration count of each functional block for a body subsystem gives more efficiency and flexibility in manipulating and predicting actors of the associated body graph. Above all, an iteration count acts as a factor in a looped schedule of quasi-static scheduling in BLDF. Thus, a more efficient quasi-static schedule of the associated body graph can be established, while keeping much of the advantage (the predictability) of SDF in the schedule. The name of BLDF is originates from this feature that a block of data tokens is packaged as parameters and then delivered to the associated body subsystem. Blocked data token delivery of BLDF enables us to reduce dimensions of MDSDF [70] by processing multi dimensional data tokens dimension by dimension with blocked data processing of nested BLDF subsystems. At the same time, BLDF can be used in conjunction with MDSDF, with BLDF parameter control used to define the boundaries of processing to be performed using MDSDF semantics.

Figure 11 shows iteration counts of the functional blocks in the associated body subsystem and how iteration counts are used for factors in a looped quasi-static schedule of the MPEG2 video encoder application. Here, the init subsystem contains the fol-

lowing three actors.

ImageFrameParameterizer. This actor delivers image frames to the ImagePropagator actor of the body subsystem as BLDF parameter values.

MPEGHeaderGenreator. This actor generates a picture ID for the associated body subsystem. The parameterized token delivery of a picture ID relieves the associated body graph of a complicated meshed layout of an MPEG2 video encoder and the inconsistent scheduling problem (P2).

BlockSize. This actor sets a block size parameter value for the associated body subsystem, which is the basic processing unit by which a full image frame is divided into groups of sub images for high throughput and more parallelism. Each functional block in the associated body subsystem processes an image frame on the basis of sub images defined in this manner.

In the body subsystem, it is useful to focus on two functional blocks: the *MPEGQuantizer* and *ReferenceFrame.* These two actors have additional input ports for a picture ID token in methods 1 and 2, but in BLDF, no additional input port for a picture ID token is required any longer since the tokens are delivered to these actors as

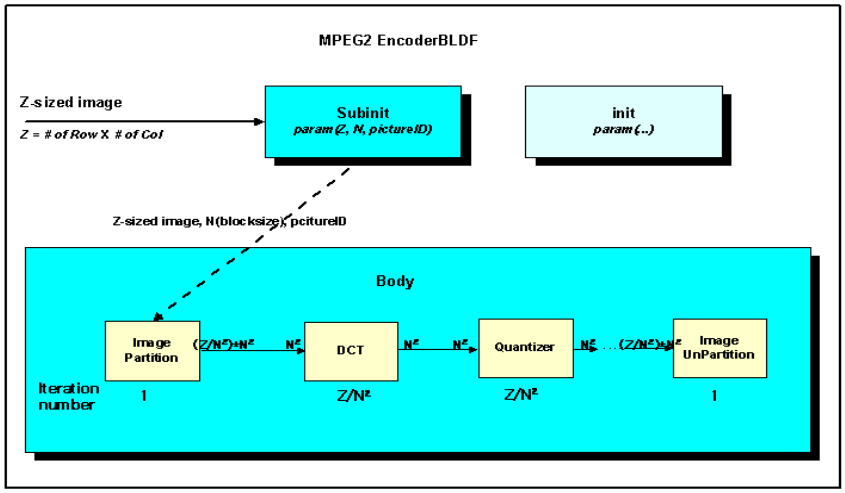


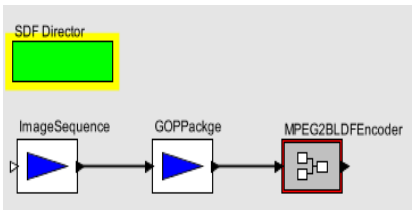
Figure 11. Blocked data delivery in BLDF

parameters, not tokens. The parameterized token delivery simplifies the layout of the MPEG2 video encoder and also removes redundant connections between all preceding actors to the target actor actually consuming that information without inconsistent schedule problem.

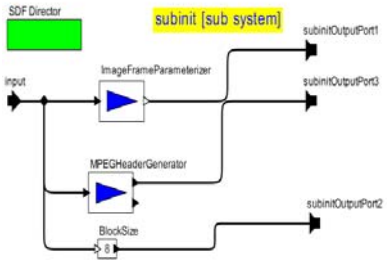
Also, this method allows dynamic configuration of parameters at a runtime. The subinit graph analyzes the tokens delivered to itself and then sets parameters of the associated body subsystem based on runtime need for parameter value delivery. Parameters maintain their value consistently during one iteration of the associated "body" graph. Figure 12 shows our implementation of the MPEG2 video encoder application under BLDF.

2.2.5.2 Comparison

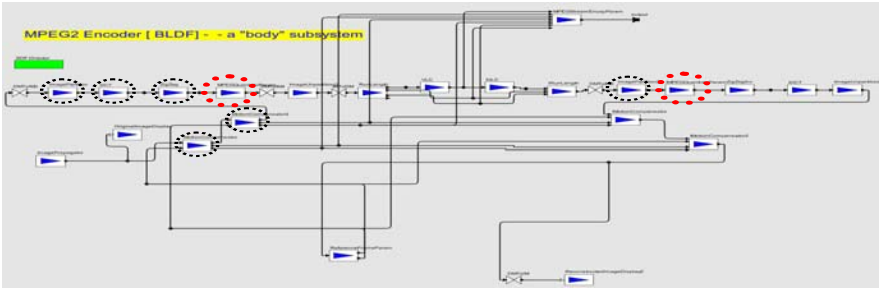
Method 1 (FSM + SDF Combination) has three different SDF graphs to which three states of the FSM are refined. However, each refined SDF graph shares most of its



a) MPEG2 Encoder in BLDF (Top)



b) "subinit" graph



c) "body" graph

Figure 12. MPEG2 Encoder under BLDF

actors with other refined graphs, so there is a problem with redundant copies of actors among each refined SDF graph.

Method 2 (SDF) simplifies three sub-encoders within method 1 into one common encoder. Thus, method 2 removes the problem of redundant (duplicated) actors. However, it still has problems of P2, P3 and P4 unsolved. Thus, unnecessary connections for picture ID delivery need to be established through preceding actors, most of which don't need a picture ID, in order to avoid an inconsistent schedule when the sample rate of tokens changes.

Method 3 (BLDF) has a similar layout as method 2, except that connections for delivering the picture ID are removed due to parameterized token delivery. This makes the layout of the encoder much simpler than method 2. Besides this, since parameters of the body subsystem are dynamically set by the subinit graph, method 3 provides more flexibility and extensibility in the design and maintenance of the application, especially by making room for future changes of the specification, along with improved efficiency in the design by reducing connections between functional blocks.

To illustrate this efficiency advantage, the following table shows how many buffers and connections in BLDF can be saved as the application complexity increases. In the MPEG2 application, we have two actors named MPEGQuantizer and InverseMPEGquantizer that require additional tokens for internal setting of values. The number of connections and the number of buffers required can be calculated by multiplying the number of preceding actors and the number of tokens for parameters.

Number of preceding actors: *n*

Number of tokens for parameters: *m*

Number of connections: *n***m*

Number of buffers required: *n***m*

Therefore, generally, *n***m* unnecessary connections and buffers between preceding actors can be saved in BLDF, compared with alternative modeling formats.

2.2.6 Conclusions of BLDF

This thesis has developed a blocked dataflow (BLDF) modeling semantics for augmenting dataflow-based DSP design tools with integrated capabilities for meta-modeling, block-based processing, multidimensional representation, and dynamic parameter reconfiguration. BLDF builds on parameterized dataflow semantics, and is compatible with decidable dataflow models such as CSDF, MDSDF, SDF, and SSDF. This thesis

Table 1. Comparison of three methods in "Buffer memory" and "Token delivery"

	Total ***#B***: Number of buffers required ***#W*** : Number of words required ***#W*** = ***#B*** * ***#WpB*** ***#WpB*** : Number of words per buffer cf) *Picture ID* : 1 word per buffer is required.(***#WpB* = 1)**	"MPEGQuantizer" actor < # of preceding actors > SDF+FSM : 3(I), 4(P), 5(B) SDF, BLDF : 5 # of tokens for parameters : 1		"Inverse MPEGQuantizer" actor # of preceding actors : 1 # of tokens for parameters : 1	
		Number of connections	Number of buffers required	Number of connections	Number of buffers required
SDF + FSM	***#B*** : = (3+4+5)+(1+1+1) = **15 buffers** ***#W*** = *#B* * *#WpB* : =15 * 1 = **15 words**	I subencoder: = 3*1 = **3** P subencoder: = 4*1 = **4** B subencoder: = 5*1 = **5**	I subencoder : = 3*1 = **3** P subencoder : = 4*1 = **4** B subencoder : = 5*1 = **5**	I subencoder : = 1*1 = **1** P subencoder : = 1*1 = **1** B subencoder : = 1*1 = **1**	I subencoder : = 1*1 = **1** P subencoder : = 1*1 = **1** B subencoder : = 1*1 = **1**
SDF	***#B*** : = (5)+(1) = **6 buffers** ***#W*** = *#B* * *#WpB* : = 6 * 1 = **6 words**	5*1 = **5**	5*1 = **5**	1*1 = **1**	1*1 = **1**
BLDF	***#B*** : **0 buffers** ***#W*** : **0 words**	**0**	**0**	**0**	**0**

has described the semantics of BLDF, and illustrated its efficiency through a case study of an MPEG 2 video encoder system. Useful directions for further study include optimized synthesis, hardware/software partitioning algorithms, and automated verification from BLDF specifications.

2.3 Dynamically configured graph topology (DGT)

2.3.1 Abstract

Dataflow is widely used for designing DSP applications. Despite its intrinsic advantages, one weak point is its difficulty in flexible expression of applications with data dependent change in execution structure. This thesis suggests an approach to providing dynamically configured dataflow graph topologies using a new modeling and synthesis technique called DGT (Dynamic Graph Topology). DGT builds on PSDF semantics [84]. All possible graph topologies for a given graph are obtained at compile time and the corresponding graph based on parameters and data is dynamically set up in an efficient manner at runtime before the invocation of the associated graph. Systematic methods for reducing code and buffer size are applied based on characteristics of each configured graph. We have compared DGT against conventional modeling approaches through a detailed case study of an MPEG 2 video encoder system, and our experiments demonstrate the efficiency of the DGT approach.

2.3.2 Related Work

To handle data driven changes in execution structure, several dataflow models such as CDDF [109], BDF [18], and BDDF [75], have been proposed. CDDF uses control tokens to determine the token transfer at an actor port. However, determination by a control token is applied to the actor in the next phase of execution, therefore, control tokens are not present at the moment that the actual phase is determined. BDDF introduces dynamic ports and an upper bound is provided for the data rate so that each dynamic port can keep the model bounded. However, control flow depends on FSMs. Using FSMs for minor changes of control flow with dataflow graphs can make application models unnecessarily complicated and result in limited flexibility. BDF provides “SWITCH” and “SELECT” actors to determine control flow. For satisfying bounded memory and consistency, a symbolic function of probability is introduced. This function increases the complexity of solving the balance equations (for verifying sample rate consistency), and results in the possibility of “weak consistency,” which is less desirable in an implementation.

To provide for more powerful and efficient data dependent execution related to application mode changes, where entire graphs or subsystem are replaced or reconfigured at run time, this thesis tackles dynamic set-up of dataflow graph topologies before the graphs are invoked. All configurations of possible graph topologies are pre-computed at compile time and stored for usage at run time. At runtime, the initialization step of DGT generates an appropriate graph topology based on parameters extracted from data being delivered and picks up a pre-computed schedule to fit the current parameter configuration. However, not all configurations are valid or can be obtained

at compile time. Some configurations may cause deadlock or inconsistency or may not be predictable at compile time. Reconfiguration of dataflow graphs is carefully considered in [73]. [73] analyzes the reconfiguration of a model based on behavioral types and extracts the *least change context* to check approximate semantic constraints. This thesis statically checks the validity of each configuration like [73] and keeps the scheduling results for use at run time. The main distinguishing feature of DGT is that it efficiently supports multi-function applications by configuring graph topologies dynamically. There are two kinds of multi-function applications. The first, which we call type-I applications, are exclusive-or applications, where only one graph topology is selected from multiple sets of possible graph topologies for a given application. The other, which we call type-II applications, are concurrent applications where two or more applications with different graph topologies are running at the same time. This thesis focuses on type-I (exclusive-or) application for experimentation of DGT. For synthesis of type-I applications, [40] extracted *commonality measures* of each actor and used these values to determine a hardware bias of each actor by hardware oriented partitioning. This thesis focuses on software implementation, and applies novel scheduling techniques based on graph characteristics to reduce code and buffer size, which is critical for DSP software. The DGT approach provides efficiency and flexibility in modeling applications with data driven change of graph topology from runtime parameter changes by using pre-computed information (information related to graph topology, scheduling, code/buffer size, bounded memory, etc.).

2.3.3 Dynamic Graph Topology

2.3.3.1 DGT (Dynamic Graph Topology) specifications

As applications for embedded systems grow more complicated, the requirement of dynamic on/off of actors and ports of actors as well as the change of transfer rates(production and consumption rates) on dataflow edges is unavoidable. To support dynamic change of graph topologies, actors, ports of actors and transfer rates should be considered to be adaptable based on the delivered data. Dynamic change of a graph topology requires run-time scheduling, which potentially causes problems of execution time overhead. To alleviate this overhead, this thesis provides for dynamic change of graph topologies through schedules that are pre-computed at compile time. DGT is based on PSDF semantics [84],[48], but is significantly more flexible than PSDF in that it allows graph actors and edges to be treated as dynamic parameters as well as the more standard types of parameters supported in the dynamic reconfiguration capabilities of PSDF. Therefore, in DGT, the transfer rate of each port of a graph, itself, is determined by a special subgraph, called the *init* graph, as in PSDF [84], so that the consumption rate and production rate of each port of the graph can be determined before the invocation of the associated DGT graph. However, in DGT, the *subinit* graph Φs controls the behavior of the associated body graph by determining the graph topology of the associated body graph before the invocation of the body graph. The number of possible graph topologies is predicted at compile time.

Figure 13 shows that how a subinit graph can extract appropriate header information and set up parameters (X:param) with the required information for the associated body graph. An appropriate graph is selected from a set of possible graphs($\{G_1, G_2, G_3\}$ by

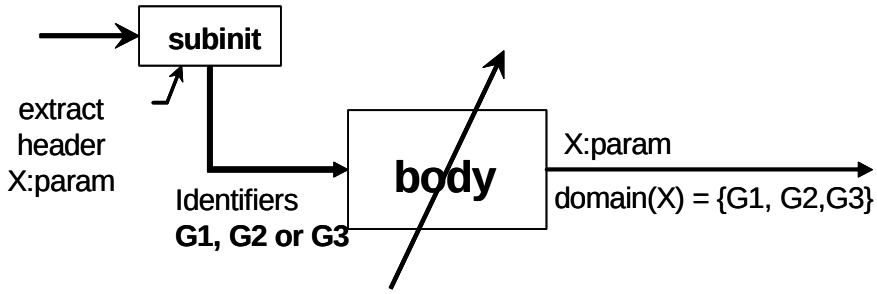


Figure 13. DGT (Dynamic Graph Topology)

the subinit graph with (X:param). This mechanism is effective because many data tokens for modern DSP applications are delivered as frames with a header part and a payload part.

Here, we classify actors and ports into two categories based on the presence or absence of data driven change of their behaviors. Actors and ports that are not changed in a graph topology are called fixed actors (a_f) and fixed ports (p_f), respectively, while actors and ports having potential dynamic changes are named as varying actors (a_v) and varying ports (p_v). Here, one point that requires careful consideration is that a fixed actor(a_f) can have a varying port (p_v) since a fixed actor (a_f) can appear with different types of ports. The *subinit* graph Φs dynamically sets up varying actors and varying ports based on data being delivered and produces an appropriate graph topology for the associated body graph. Consistency and bounded memory for each possible set of graph topologies are verified at compile time. At runtime, the *subinit* graph Φs sets up an appropriate graph topology for the associated body graph and picks up an appropriate pre-computed schedule that also contains code and buffer size minimized for the configured graph. Code and buffer size minimization is obtained by a scheduling technique appropriately chosen depending on graph characteristics. In DGT, verification of validity of schedules can be performed at compile time and valid

schedules can be guaranteed and can be ready to be used at runtime without the overhead of dynamic scheduling. At runtime, *the subinit* graph Φs looks up pre-computed schedules in a table with the appropriate parameter values.

Figure 14 shows an example of how DGT is applied to configure a body graph. Here, $\{p_v^o(i, a_k)\}$ represents all the possible sets of ports to which the i_{th} varying output port of the actor a_k can be connected. $\{p_v^{in}(i, a_k)\}$ represents a counterpart of an input port. In figure 14, dotted line represents varying edges while solid lines represents fixed edges. Also, a dash filled actor represents a varying actor while a white blank actor represents a fixed actor. Each actor can have varying ports and fixed ports together. The transfer rates or connections of varying edges are data dependent while the transfer rates and connections of fixed edges are fixed. Varying edges and varying actors can be turned on or off based on the data tokens delivered.

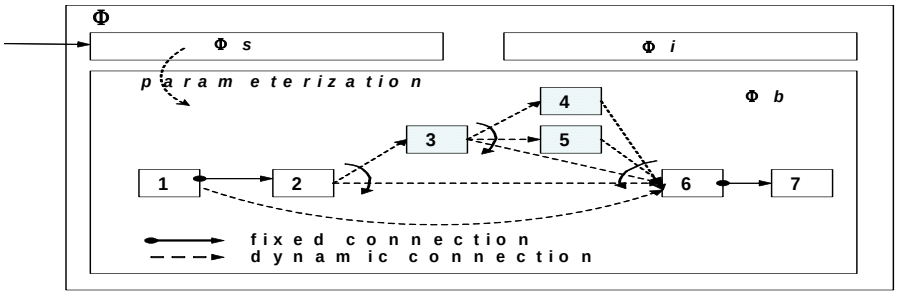


Figure 14. An example of a graph under DGT

The following equation represents a general case where the i_{th} varying output or input port of the actor a_k connects to the j_{th} input or output port of another actor a_n or does not connect to anything.

$$\{p_v^o(i, a_k)\} = \{p_v^{in}(j, a_n), \perp\}, (\{p_v^{in}(i, a_k)\} = \{p_v^o(j, a_n), \perp\})$$

This is an example of the $1st$ input port of a_6 in Figure 14.

$$\{p_v^{in}(1, a_6)\} = \{\{p_v^o(1, a_4), p_v^o(1, a_5)\}, p_v^o(1, a_3), p_v^o(1, a_2)\}$$

Here, $\perp$ means there are no edges from or to the associated port. The graph G ($G = G_f \cup G_v$) is made up of G_v (a graph with varying graph components) and G_f (a graph with fixed graph components). By separating from G_f parts that are common across different subsystems, possible overlapping of resources among different subgraphs can be removed.

2.3.3.2 Scheduling of DGT specifications

A DGT subsystem produces various sets of configurations for the associated body graph Φb. For each graph generated, checking of both synchrony (synchronous dataflow [63] behavior) for the duration of the configuration and bounded memory is performed. For this purpose, a graph is considered as a general fixed graph after the subinit graph configures the graph topology. All of the major configurations for the corresponding graph are captured at the compilation stage and are kept for use at runtime. The *subinit* graph Φs extracts parameters from the header part of data being processed and then sets appropriately the associated body graph Φb. For many applications, such as those involving a few to several or even dozens of different modes, the number of combinations of DGT configurations is manageable for reasonable implementation platforms. Here, the transfer rate of every port of each actor within a body graph under DGT can be changed by the associated graph Φs.

A useful restriction in the use of DGT is that when a DGT graph is embedded within a dataflow model other than DGT or PSDF, the transfer rates of interface ports of a DGT graph must generally be fixed even though the graph topology inside the DGT subsystem can be vary dynamically. This assumption allows DGT graphs to be

embedded easily in other dataflow models with the external appearance of simple SDF actors. Therefore, the transfer rates of input/output ports of the DGT graph, itself, should be set by the *init* graph Φi before the DGT graph is invoked and should be kept invariant during the entire iteration of the graph.

Figure 15 shows an example that illustrates DGT scheduling within SDF. The DGT graph Φ takes two tokens and produces two tokens. Therefore, the schedule for Figure 15 will be like $2s \cdot \Phi \cdot 2e$. However, by looking into the DGT graph Φ, we see that the actor b is a varying actor that can be removed by the subinit graph Φs on demand. Also, the transfer rates of actor b are not fixed. The actor a has one output port, which is a varying port. Therefore, the actor a can be connected to either the actor b or the actor c. The actor c has one varying input port and one fixed output port. The actor c consumes one token either from actor a or actor b and produces two tokens to a fixed output port. The schedule of the DGT graph Φ can be either

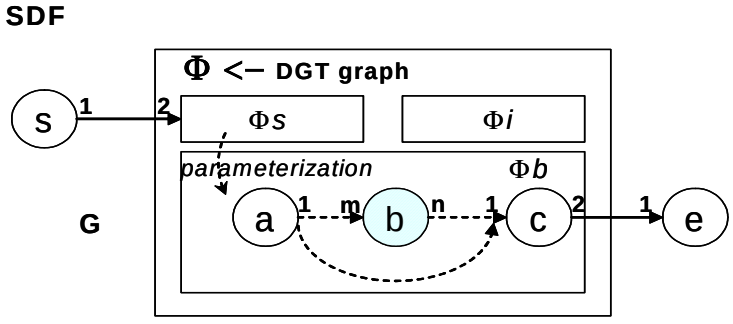


Figure 15. DGT graph under SDF

$ma \cdot b \cdot nc$ or $a \cdot c$. The schedule for the graph G is $2s \cdot \Phi \cdot 2e$ and the schedule for the graph Φ is either $ma \cdot b \cdot nc$ or $a \cdot c$. The schedule for each graph is hierarchically maintained in this manner. Here, the two schedules for the graph Φ are SAS (Single Appearance Schedule)[11] where each actor appears only once. The following section

shows how different scheduling techniques are applied systematically based on characteristics of the configured graphs.

2.3.3.3 Minimization of code and buffer requirements

According to graph characteristics and the granularity (complexity) of each actor, efficient scheduling considering both code size and buffer memory requirements is important when synthesizing implementations. Since a DGT system supports runtime adjustment of pre-computed schedules, decisions on the methods for minimizing code and buffer requirements can be made statically. For an application graph, the ratio of code size vs. buffer size as well as graph characteristics are important factors to select an appropriate technique for efficient minimization of both code and buffer size. For example, for an application with a very small code size but requiring high buffer size, minimizing code size by SAS (Single Appearance Schedule) is not likely to lead to a cost-effective solution. Instead, a carefully-constructed MAS (Multiple Appearance Schedule) is likely to be a better choice due to the advantage of further buffer size reduction at the expense of some code size increase. In our DGT synthesis approach, for efficient multiple appearance schedule generation, we have adapted the MAS approach of [52], and for SAS generation, techniques from [84], [10] and [11] are applied. For selection between MAS and SAS implementation, we have formulated a normalized criterion (SS:Schedule Selector) to determine the most appropriate technique.

$$SS = \gamma_{\mu} \times \mu + \gamma_{\tau} \times \tau$$

μ is the uniformity metric of [52] (explained below) and τ is the ratio of total code size to the average data frame size obtained based on simulation. γ_{μ} and γ_{τ} are user-

defined weight values and are chosen based on simulation. μ is proportional to the number of edges whose transfer rates are multiples of one another. A high value of μ reflects potentially low opportunity for buffer size reduction using the techniques of [52]. τ suggests which factor between code size and buffer size is more important to reduce the overall memory requirements. A graph with a higher τ suggests that a scheduling technique that is more efficient in reducing code size produces a better result rather than a buffer-oriented technique. Consequentially, a high SS value suggests that an SAS is appropriate for the graph.

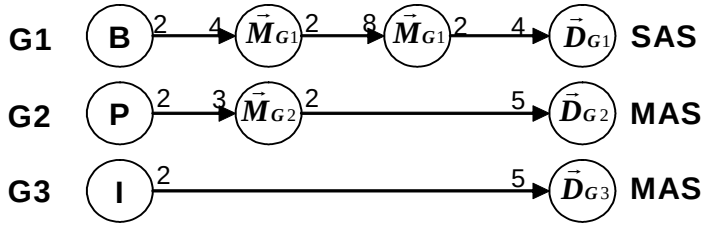


Figure 16. Part of an MPEG2 video encoder

Figure 16 shows part of an MPEG2 encoder modeled using our DGT technique. Some of the actors can operate with different parameters and transfer data at rates depending on the graph(G) in which the actor is included. Those actors are symbolized as $\overrightarrow{X_G}$. In Figure 16, $\vec{M}$ represents MC (motion compensators) and $\vec{D}_G$ represents a DCT (Discrete Cosine Transform). In MPEG2, the B frame requires two MCs and the P frame requires one MC, while the I frame does not need a MC. Therefore, three different graph topologies are required within the application, and the particular topology to use at a given time depends on the picture frame type (I, P, or B).

Each graph topology has different SS values depending on the characteristics the graph. For G1 of B frame, SAS implementation is selected, while for G2 of P frame

and G3 of I frame, MAS implementation is selected. In Figure 16, the behaviors of the actor $\vec{M}_G$ and the actor $\vec{D}_G$ can be changed depending on the graph characteristics and the change of parameters, while other actors are invariant.

From a DGT representation, we can often reduce code size by removing overlapping graph components across graph sets. If m is the number of common actors in graphs with different configurations, and λ_i is the number of graphs (G_i) including the i_{th} common actor (C_i).

$$ReducedCodeSize = \sum_{i=1}^{m} \sum_{j=1}^{\lambda_i - 1} codeSize(C_i)$$

2.3.3.4 Operational semantics of DGT

Figure 17 shows the operational semantics of DGT operating with any type of data-flow model. Because of its ability to operate with different types of dataflow models, DGT is more accurately characterized as a meta-modeling technique. Each hierarchical actor (Φ) in a DGT system also can be viewed as an independent graph and can have its own schedule. In our implementation of DGT, we maintain schedules in a hierarchical manner. Therefore a graph (G) has the schedule for itself and also maintains schedules for each hierarchical actor(Φ) under the graph (G). Each hierarchical actor Φ under G also maintains the schedule for itself and schedules for graphs representing every hierarchical actor Φ_{sub} inside Φ. This way, the schedule for the graph G and schedules for sub graphs of Φs inside G are maintained in a hierarchical way until graphs in the lowest level of the hierarchy are scheduled.

The function $scheduleX(G)$ is a function to schedule a graph G. For all general hierarchical actors (Φ) inside G except Φs of DGT, $scheduleX$ is applied. The function

scheduleDGT is applied for Φ of DGT within G. Then *linkSchedList* is applied to have the schedule for the graph G, itself and schedules for Φs in G kept linked together. The function *setGraphTopology* in *scheduleDGT* generates the corresponding graph with given parameters. Ultimately, *schedulerXDF* in a *scheduleX* generates an appropriate schedule based on the graph topology along with code and buffer size suitable for each graph. For each configured graph, type checking of the given graph is performed and then if SS is bigger than $Threshold_{SS}$ for selecting an scheduling technique, the chosen SAS based technique (*SASTechnique*) is applied. Otherwise, the chosen MAS based technique (*MASTechnique*) is chosen.

2.3.4 Experimental results

In our experiments, we developed MPEG2 encoder, Laplacian pyramid, Multi resolution spline pyramid, Pyramid complex application which is a combined model of Laplacian pyramid and Multi-resolution spline and an image complex application consisting of several individual morphological applications (Top-Hat, Smoothing, Laplacian and Gradient).

For MPEG2 encoder, an MPEG2 video encoder has some different operational blocks depending on the picture frame, but shares most of the blocks across picture frames (I, B or P frame). We compared the total memory usage of a DGT graph implementation with a conventional separate-graph approach. A separate graph approach uses a combination of SDF and FSM in all experiments (table 2~6). Each SDF graph processes a different picture frame. The DGT method selects different scheduling methods (SAS or MAS) depending on graph characteristics.

```
function scheduleX(G) {
  for each Φ_i of DGT in G
    sched_Φi = scheduleX(Φ_i[i])
  end for
  shed_G = scheduleXDF(G)
  for each Φ in G
    if(Φ under DGT)
      sched_Φ[i] = scheduleDGT(Φ[i])
    else
      sched_Φ[i] = scheduleX(Φ[i])
  end for
  sched_G = linkSC(linkSC(shed_G, sched_Φ), sched_Φi)
  return sched_G
}
function schedulerXDF(G) {
  if(SS > Threshold_SS)
    sched_G = SASTechnique(G)
  else
    sched_G = MASTechnique(G)
  return sched_G
}
function scheduleDGT(Φ) {
  sched_Φs = scheduleX(Φ_s)
  setUpPortTransferRate(Φ) //in & out port of Φ
  C_b = getParamConfigSets(Φ_b)
  for each C_b in Φ_b
    Φ*_b = setGraphTopology(C_b, Φ_b)
    sched_Φ*b[i] = scheduleX(Φ*_b)
  end for
  sched_Φ = linkSC(sched_Φ*b, sched_Φs)
  return sched_Φ
}
function setGraphTopology(Φ, C) {
   determineG_vTopology(Φ, C)
  for each a in Φ
    for each p_v^o in a
    {e_v} = connectEdge(p_v^o(i, a_v))
    end for
    for each p_v^in in a
      {e_v}= {e_v} ∪ connectEdge(p_v^in(i, a_v))
    end for
  end for
  Φ_v = {a_v} ∪ {e_v}
  Φ = Φ_v ∪ Φ_f
  return Φ
}
```

$Threshold_{SS}$ is obtained based on simulation

Figure 17. Operational semantics of DGT operating with any type of dataflow model

For each Laplacian and Multi resolution pyramid application, we compared the

total memory usage of DGT modeling method with the conventional separate dataflow model. Laplacian pyramid and Multi resolution pyramid may need different dataflow graphs depending on the depth of an image pyramid. Finally, the change of an image pyramid depth requires the variation of a dataflow graph topology. These variation can be modeled under SDF and FSM refinement with overlapping of partial graph topology among dataflow graphs. Under DGT semantics, an image pyramid application with different pyramid depth can be efficiently modeled within a single dataflow graph domain while avoiding redundant resource usage.

For a pyramid complex and an image complex application, each application (a pyramid complex and an image complex application) consists of individual sub applications for different purposes while sharing partial operational functionalities (or actors). Thus, a pyramid complex or an image complex application can be configured for multiple sub applications at runtime while changing the combination of each sub applications. Under SDF and FSM refinement approach, every combination of individual sub applications may correspond to separate dataflow graph models. However, under DGT semantic, these individual sub applications can be modeled within a single dataflow semantic and can be reconfigured at runtime while setting up a combined single graph topology for multiple individual applications while avoiding unnecessary resource overlapping among applications.

For obtaining the code size, we used the Texas Instruments Code Composer simulator of the 64XX series processor. In the experiments, as the frame size increases, the impact of buffer size on total memory usage becomes larger than the impact of code size. We applied SAS, MAS and a combination of SAS and MAS to each case.

Table 2. Memory usage comparison(MPEG2 encoder)

Frame Size		DG			SG		
		C1	C2	C3	C4	C5	C6
128 * 128	Code	26,469	31,946	26,469	63,341	79,773	63,341
	Buffer	1,557	1,429	1,557	4,667	4,283	4,667
	Total	28,026	33,375	**28,026**	68,008	84,056	**68,008**
256 * 256	Code	26,469	31,946	26,469	63,341	79,773	63,341
	Buffer	6,173	5,661	6,173	18,515	16,979	18,515
	Total	32,642	37,607	**32,642**	81,856	96,752	**81,856**
480 * 720	Code	26,469	44,903	31,393	63,341	118,645	94,180
	Buffer	52,852	19,991	21,788	158,551	59,967	65,364
	Total	79,321	64,894	**53,181**	221,892	178,612	**159,544**
768 * 1024	Code	26,469	58,074	44,564	63,341	158,157	133,692
	Buffer	130,680	45,320	49,397	392,035	135,955	148,192
	Total	157,149	103,394	**93,961**	455,376	294,112	**281,884**
1080 * 1920	Code	26,469	58,074	50,041	63,341	158,157	150,124
	Buffer	1,817,064	100,940	100,937	5,451,187	302,815	302,524
	Total	1,843,533	159,014	**150,978**	5,514,528	460,972	**452,648**

. DG: DGT approach, SG: Separate graph approach (FSM+SDF), C1: SAS, C2: MAS,
. C3: SAS+MAS, C4: SAS, C5: MAS, C6: SAS+MAS

In C3 and C6, (see Table 2) while SAS is selected for both 128*128 and 256*256, either SAS or MAS is selected for each picture frame (I, B and P) dynamically for a frame larger than 256*256. This is because a trade-off between code size and buffer size exists in the vicinity of 480*720 size. In Multi resolution spline (see Table 3) and Laplacian pyramid (see Table 4) experiments, this pattern (SAS to MAS migration) appears around 480*720 resolution. In a pyramid complex (see Table 5) and an image complex application (see Table 6), the migration of scheduling method from SAS to MAS for minimization of total memory usage appears around 768*1024 resolution. However, this pattern of scheduling method migration (SAS to MAS) is common to all image processing benchmark applications (table 2 to 6). It's because the minimization of the buffer size is more effective than code size reduction as image size increases.

The experiment (table 2~6) shows that the DGT approach reduces total memory usage from 60% to 72% compared with a separate graph approach through shared

code and the streamlining of scheduling methods to fit graph characteristics. The runtime overhead for finding a proper schedule for each graph topology is only $\Omega(N) + \Omega(m)$, where m is the number of varying graph components (varying actors and varying edges) and N is the number of possible schedules for each DGT graph depending on the topology, which is relatively modest compared with the complexity of typical signal/image processing actors.

Table 3. Memory usage comparison (Multi resolution Spline Pyramid)

Frame Size		DG			SG		
		C1	C2	C3	C4	C5	C6
128 * 128	Code	38,821	263,941	38,821	100,397	775,757	100,397
	Buffer	11,661	1,293	11,660	34,977	3,873	34,976
	Total	50,482	265,234	**50,481**	135,374	779,630	**135,373**
256 * 256	Code	38,821	263,941	38,821	100,397	775,757	100,397
	Buffer	46,635	5,163	46,634	139,899	15,483	139,898
	Total	85,456	269,104	**85,455**	240,296	791,240	**240,295**
480 * 720	Code	38,821	263,941	85,049	100,397	775,757	239,081
	Buffer	210,171	23,547	147,962	630,507	70,635	443,882
	Total	248,992	287,488	**233,011**	730,904	846,392	**682,963**
768 * 1024	Code	38,821	263,941	263,941	100,397	775,757	775,757
	Buffer	497,411	55,043	55,042	1,492,227	165,123	165,122
	Total	536,232	318,984	**318,983**	1,592,624	940,880	**940,879**
1080 * 1920	Code	38,821	263,941	263,941	100,397	775,757	775,757
	Buffer	1,259,067	139,323	139,322	3,777,195	417,963	417,962
	Total	1,297,888	403,264	**403,263**	3,877,592	1,193,720	**1,193,719**

Table 4. Memory usage comparison (Laplacian Pyramid)

Frame Size		DG			SG		
		C1	C2	C3	C4	C5	C6
128 * 128	Code	22,515	128,387	22,515	51,479	369,095	51,479
	Buffer	5,721	537	5,720	17,157	1,605	17,156
	Total	28,236	128,924	**28,235**	68,636	370,700	**68,635**
256 * 256	Code	22,515	128,387	22,515	51,479	369,095	51,479
	Buffer	22,875	2,139	22,874	68,619	6,411	68,618
	Total	45,390	130,526	**45,389**	120,098	375,506	**120,097**
480 * 720	Code	22,515	128,387	44,311	51,479	369,095	116,869
	Buffer	103,071	9,759	71,966	309,207	29,271	215,894
	Total	125,586	138,146	**116,277**	360,686	398,366	**332,763**
768 * 1024	Code	22,515	128,387	128,387	51,479	369,095	369,095
	Buffer	243,971	22,787	22,786	731,907	68,355	68,354
	Total	266,486	151,174	**151,173**	783,386	437,450	**437,449**
1080 * 1920	Code	22,515	128,387	128,387	51,479	369,095	369,095
	Buffer	617,547	57,675	57,674	1,852,635	173,019	173,018
	Total	640,062	186,062	**186,061**	1,904,114	542,114	**542,113**

Table 5. Memory usage comparison (Pyramid Complex)

Frame Size		DG			SG		
		C1	C2	C3	C4	C5	C6
128 * 128	Code	32,719	221,674	32,719	57,405	435,315	57,405
	Buffer	8,723	947	8,723	17,444	1,892	17,444
	Total	41,442	222,621	**41,442**	74,849	437,207	**74,849**
256 * 256	Code	32,719	221,674	32,719	57,405	435,315	57,405
	Buffer	34,886	3,782	34,886	69,770	7,562	69,770
	Total	67,605	225,456	**67,605**	127,175	442,877	**127,175**
480 * 720	Code	32,719	221,674	32,719	57,405	435,315	57,405
	Buffer	157,520	17,552	157,520	315,038	35,102	315,038
	Total	190,239	239,226	**190,239**	372,443	470,417	**372,443**
768 * 1024	Code	32,719	221,674	221,674	57,405	435,315	435,315
	Buffer	372,098	40,322	40,322	744,194	80,642	80,642
	Total	404,817	261,996	**261,996**	801,599	515,957	**515,957**
1080 * 1920	Code	32,719	221,674	221,674	57,405	435,315	435,315
	Buffer	941,870	102,062	102,062	1,883,738	204,122	204,122
	Total	974,589	323,736	**323,736**	1,941,143	639,437	**639,437**

Table 6. Memory usage comparison (Image Complex)

Frame Size		DG			SG		
		C1	C2	C3	C4	C5	C6
128 * 128	Code	13,049	17,001	13,049	28,097	43,905	28,097
	Buffer	406	322	406	1,612	1,276	1,612
	Total	13,455	17,323	**13,455**	29,709	45,181	**29,709**
256 * 256	Code	13,049	17,001	13,049	28,097	43,905	28,097
	Buffer	1,612	1,276	1,612	6,436	5,092	6,436
	Total	14,661	18,277	**14,661**	34,533	48,997	**34,533**
480 * 720	Code	13,049	17,001	13,049	28,097	43,905	28,097
	Buffer	9,049	7,159	9,049	36,184	28,624	36,184
	Total	22,098	24,160	**22,098**	64,281	72,529	**64,281**
768 * 1024	Code	13,049	17,001	14,873	28,097	43,905	35,393
	Buffer	20,104	15,904	17,704	80,404	63,604	70,804
	Total	33,153	32,905	**32,577**	108,501	107,509	**106,197**
1080 * 1920	Code	13,049	17,001	17,001	28,097	43,905	43,905
	Buffer	54,274	42,934	42,934	217,084	171,724	171,724
	Total	67,323	59,935	**59,935**	245,181	215,629	**215,629**

2.3.5 Conclusions of DGT

This thesis develops efficient support for dynamic graph topologies for dataflow graphs requiring different execution structures based on dynamic parameters, and data being processed. In addition to providing efficient and flexible support for multiple modes of system operation, DGT allows us to reduce overall memory size by systematically sharing code and applying tailored scheduling methods across the different graph topologies that make up a DGT application. Useful directions for future work

Table 7. Memory usage comparison

Frame Size		DG			SG		
		C1	C2	C3	C4	C5	C6
128 * 128	Code	26,469	31,946	26,469	63,341	79,773	63,341
	Buffer	1,557	1,429	1,557	4,667	4,283	4,667
	Total	28,026	33,375	**28,026**	68,008	84,056	**68,008**
256 * 256	Code	26,469	31,946	26,469	63,341	79,773	63,341
	Buffer	6,173	5,661	6,173	18,515	16,979	18,515
	Total	32,642	37,607	**32,642**	81,856	96,752	**81,856**
480 * 720	Code	26,469	44,903	31,393	63,341	118,645	94,180
	Buffer	52,852	19,991	21,788	158,551	59,967	65,364
	Total	79,321	64,894	**53,181**	221,892	178,612	**159,544**
768 * 1024	Code	26,469	58,074	44,564	63,341	158,157	133,692
	Buffer	130,680	45,320	49,397	392,035	135,955	148,192
	Total	157,149	103,394	**93,961**	455,376	294,112	**281,884**
1080 * 1920	Code	26,469	58,074	50,041	63,341	158,157	150,124
	Buffer	1,817,064	100,940	100,937	5,451,187	302,815	302,524
	Total	1,843,533	159,014	**150,978**	5,514,528	460,972	**452,648**

DG: DGT approach, SG: Separate graph approach (FSM+SDF), C1: SAS, C2: MAS, C3: SAS+MAS, C4: SAS, C5: MAS, C6: SAS+MAS

include integrating DGT with other dataflow models as a meta-modeling technique, and implementation of concurrent applications through DGT semantics under resource and performance constraints.

Chapter 3 : Scheduling of DSP applications onto multiprocessors

3.1 Introduction

In the previous chapter, we described two new modeling techniques; Blocked Data-Flow (BLDF) and Dynamically configured Graph Topology (DGT). Blocked Data-Flow (BLDF) tackled blocked processing feature of image processing applications. BLDF improved the expressivity of a dataflow model by a quasi-static scheduling with meta-variables and provides an efficient way of modeling applications with a blocked processing pattern by exploiting parameterized token delivery. Dynamically configured Graph Topology (DGT) provided runtime reconfiguration of a graph topology while taking advantage of static scheduling information. DGT enabled us to model an application with various graph topologies depending on the change of parameters in a single dataflow domain.

In this chapter, we introduce a novel scheduling technique for mapping a dataflow graph over multiprocessors environment and describe the major features and contributions of the suggested scheduling technique.

3.2 Pipeline Decomposition Tree scheduling

3.2.1 Abstract

Modern embedded systems for image processing involve increasingly complex levels of functionality under real-time and resource-related constraints. As this complexity increases, the application of single-chip multiprocessor technology is attractive. To address the challenges of mapping image processing applications onto embedded multiprocessor platforms, this paper presents a novel data structure called the *pipeline decomposition tree* (*PDT*), and an associated scheduling framework, which we refer to as *PDT scheduling*. PDT scheduling exploits both heterogeneous data parallelism [55][81] and task-level parallelism [4][16][36], which are important considerations for scheduling image processing applications, and systematically derives customized pipelined architectures that are streamlined for the given implementation constraints.

3.2.2 Introduction

The proliferation of embedded systems that involve image processing, such as digital cameras and video-conferencing systems, exhibits trends towards the integration of multiple image processing operations to provide diverse functionalities, and the application of embedded multiprocessor technology to provide the required performance.

This paper presents a novel data structure called the *pipeline decomposition tree* (*PDT*), and an associated scheduling framework, which we refer to as *PDT schedul-*

ing, for mapping image processing applications onto embedded multiprocessor systems. PDT scheduling is based on a model of the target implementation as a coarse-grained (task-level), pipelined architecture. PDT scheduling spreads functional operations over the underlying pipeline through construction and iterative analysis of the PDT. Intuitively, the PDT can be viewed as a kind of depth first search tree whose nodes are mapped to stages of the targeted pipeline. Any number of nodes of the PDT can be mapped to a single stage of the pipeline. PDT scheduling ultimately generates schedules with different latency/throughput trade-offs to effectively explore the multi-dimensional space of signal processing performance considerations. Furthermore, the PDT scheduling process can take into consideration various scheduling constraints, such as constraints on the number of available processors, and the amounts of on-chip and off-chip memory, as well as performance-related constraints (i.e., constraints involving latency and throughput).

The PDT scheduling approach places special emphasis on distinguishing and taking into account different modes of parallelism — task-level parallelism, as well as homogeneous and heterogeneous data parallelism — that must be exploited carefully to achieve efficient implementation of image processing applications. Data parallelism is a specialized form of parallel processing that allows multiple copies of a single task to execute simultaneously on multiple processing units. Heterogeneous data parallelism is an extension of data parallelism that allows for variability in the sizes of the memory regions to which data parallelism is applied. Under heterogeneous data parallelism, each copy of a task handles different sizes of blocks from the input data stream.

Although concepts related to the PDT and PDT scheduling can be applied to vari-

ous domains of signal processing, including speech processing, high fidelity audio processing, and digital communications, the emphasis in PDT on data parallelism considerations makes the technique especially well suited to image processing.

Throughout the process of PDT scheduling, different interprocessor communication (IPC) architectures (point-to-point communication links or shared buses), and memory architectures (shared-memory or distributed memory architectures) are considered in an effort to achieve the most effective balance under the given constraints and available modes of parallelism.

3.2.2.1 Related Work

In most practical contexts, scheduling applications onto multiprocessors environments is NP hard. Many deterministic heuristics and evolutionary algorithm techniques have been proposed in this area (e.g., see [2][19][23][28][115]). In some cases, evolutionary algorithms are used in conjunction with deterministic approaches to yield their complementary advantages, and systematic methods have been developed also to perform such integration between evolutionary and deterministic approaches [6]. In particular, evolutionary approaches provide robust, easily adaptive methods for global search, while deterministic approaches are effective at exploiting application-specific insights that often provide for derivation of good solutions very rapidly, as well as effective local optimization. The PDT approach can be viewed as a deterministic approach that can be used in isolation as a fast, effective heuristic, and can also be combined with evolutionary algorithms when more thorough, computationally-intensive optimization is desired. This paper focuses on the former application of PDT scheduling; integra-

tion with evolutionary algorithms or other randomized search methods is a useful direction for further investigation.

A number of important deterministic techniques have been proposed in previous work related to embedded multiprocessor implementation of signal processing applications. Banerjee, Hamada, Chau, and Fellman[7] presented a two-step approach for coarse-grain pipeline scheduling by separating partitioning and process allocation for heterogeneous architectures. Hoang and Rabaey[32] developed a heuristic algorithm by innovative modeling and incorporation of interprocessor communication costs into the framework of coarse-grain pipelining. Konstantinides, Kaneshiro, and Tani[53] tackled detailed issues in modeling input/output (I/O) operations by decomposing I/O into sequential and parallel components. PDT scheduling is different from these approaches in its deep integration of data parallelism configurations with task-level parallelism and coarse-grained pipeline implementation. Our PDT approach is motivated by the fundamental importance of data parallelism in performance optimization of image processing applications.

Subhlok and Vondran[99] have previously considered the integration of data parallelism with task-level parallelism for multiprocessor scheduling. However, this work focuses mainly on applications that can be represented as linearly-chained dataflow graphs. Applying data parallelism and task parallelism to applications that have more general dataflow topologies causes various complications that are not addressed by the techniques of Subhlok.

In contrast, this paper targets general application dataflow topologies, including those with linear and non-linear data dependencies, and configures data parallelism

and task parallelism appropriately based on the dataflow topology as well as the given implementation constraints. To demonstrate our proposed methods, we have applied them to complex morphological operations, Laplacian pyramid computation, Gaussian pyramid computation, and multi-resolution splines, which are all important image processing subsystems. The morphological operations that we have considered include opening, closing, gradient, Laplacian, smoothing and top-hat.

3.2.3 PDT(Pipeline Decomposition Tree) based scheduling

3.2.3.1 Assumptions of PDT scheduling

PDT scheduling is applied based on the following constraints and architectural assumptions.

- **Assumption 0**:

PDT scheduling operates under HSDF (Homogeneous Synchronous Dataflow Graph). For an application modeled under non HSDF, conversion to HSDF is required before applying **PDT scheduling**.

- **Assumption 1**:

On-chip memory are dumped down to an external memory or filled up from the external memory based on a window (or block) size to reduce a data transfer overhead between *on-chip* memory and *external* memory. This way, *on-chip* memory can be efficiently managed by placing relating data onto neighboring block within *on-chip* memory.

- **Assumption 2**:

Tasks in a graph are mapped to clusters based on task dependency[27]. Tasks

sharing a predecessor are mapped to the same cluster. These clusters are named **TG-Cluster(Task Grouped Cluster)**. In a graph, the point preceding TG-Cluster is named *branch point*. Mapping tasks following *branch point* onto the same cluster allows for exploiting the benefit of a shared memory architecture and leads to reducing a memory size since tasks in TG-Cluster share input data. Other tasks are mapped to individual different clusters. Figure 22 d) shows an example of *branch point* and **TG-Cluster(Task Grouped Cluster).**

- **Assumption 3**:

Tasks in a TG-Cluster(Task Grouped Cluster) could run in parallel depending on available processing units.

- **Assumption 4**:

A couple of processing cores can be integrated in a single chip; “DSP chip”. Each core holds its own separate internal cache. Processor cores within each DSP chip hold a shared *on-chip* memory. This thesis considers only a shared architecture for *on-chip* memory to reduce the size of memory area in a DSP chip. PDT scheduling challenges scheduling an application under limited *on-chip* and *external* memory size by monitoring a peak memory usage of the application. Figure 18 shows how *on-chip* memory and an internal cache are integrated in each DSP chip.

- **Assumption 5**:

External memory is located outside a DSP chip. There are two different architectures available for an *external* memory: a shared external memory architecture and a separate external memory architecture. In a shared architecture, an *external* memory can be accessed by all DSP chips sharing it whereas in a separate architecture, each

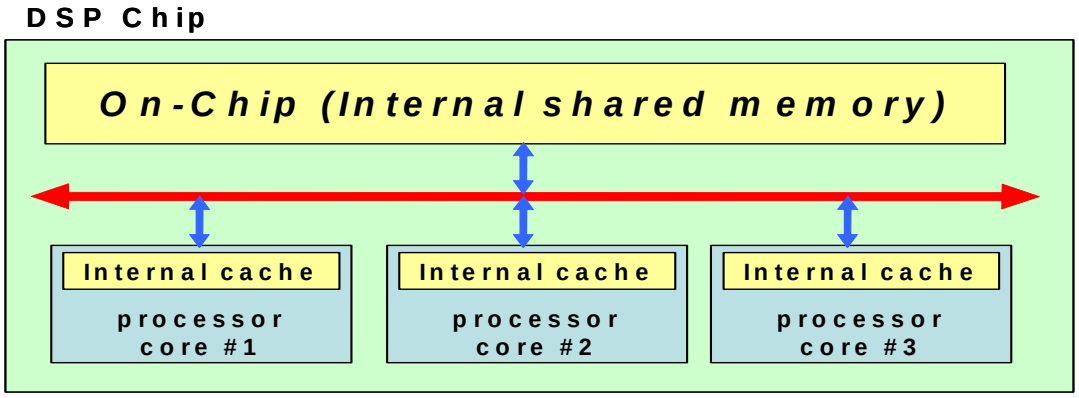
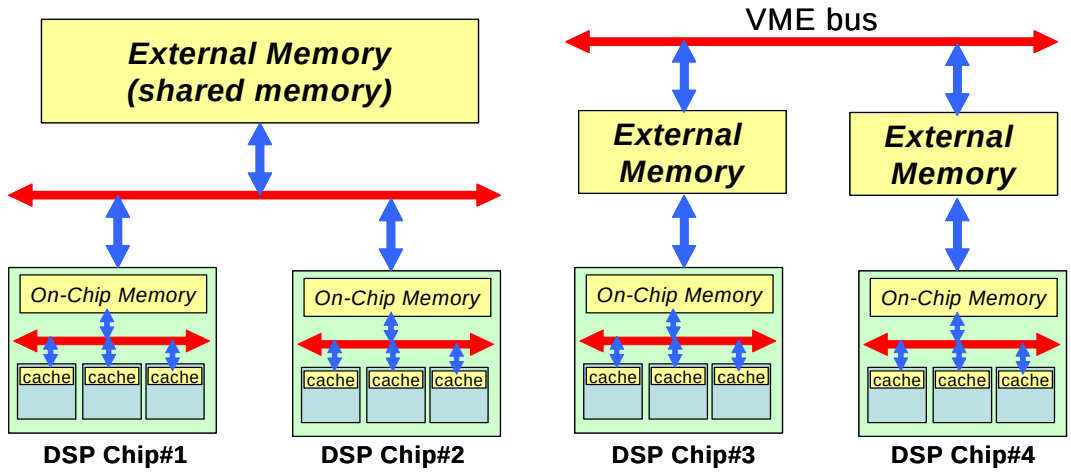


Figure 18. An "*on-chip*" memory and an internal cache of DSP chip

DSP chip has its own *external* memory and can access only the associated *external* memory.

- **Assumption 6**:

In case of separate external memory architecture, each DSP chip is assumed to be connected through VME (Versa Module Europa). The IPC cost is modeled for estimating communication cost between processors. For a shared external memory architecture, bus contention among DSP chips sharing a memory area is considered instead.



a) shared external memory architecture b) separate external memory architecture

Figure 19. Comparison of a shared external memory architecture and a separate external memory architecture

3.2.3.2 Heterogeneous data parallelism

In many image processing operations, the overall operation can be performed by iteratively executing a lower-level operation on different parts of the input image. Usually, this lower level operation requires only a subset of neighboring pixels for any given invocation, and furthermore different invocations of the lower level operation are usually independent of one other. The neighboring data items for each invocation is called a “window” or “block” of image pixels.

Keinert, Haubelt, and Teich have studied the formal modeling of such window-based image processing operations, and have developed novel extensions of the synchronous dataflow model for effectively representing this important class of operations [43]. Keener’s work is limited to the constraint of static scheduling.

In contrast, the blocked dataflow modeling technique [48] that we present in this thesis provides for more flexible quasi static scheduling. This is achieved by parameterizing windowed (“blocked”) data, and dynamically adjusting the associated parameter values as necessary before executing a dataflow subsystem. This feature of windowed data representation allows us to flexibly exploit data parallelism when mapping image processing applications onto embedded multiprocessor platforms.

Data parallelism allows multiple copies of a single task to run on multiple processing units by task duplication. An operation of each task is independent. Each copied task processes a sub region of the whole data frame. The whole data frame can be divided into sub regions with different offsets. Finally the whole data frame is processed by each copied task in parallel. The sizes of sub areas are same for all copied tasks in a general data parallelism. Heterogeneous data parallelism is an extension of

data parallelism. Heterogeneous data parallelism allows for dynamic change of the sub region size depending on the availability of resource. In heterogeneous data parallelism, the whole data frame consists of copy-sets. The size of copy-set may or may not be the same depending on available idle processors. Each copy-set consists of the same size of sub regions. Each copied task are allocated to handle different copy-set areas. Inside each copy set area, each task handles different sub regions. The size of sub regions within a copy-set is same and can be obtained by dividing the size of the copy-set by the number of tasks assigned to the copy-set. The number of tasks within a copy-set may vary from 1 to N depending on available idle processors. Each copied task corresponds to each invocation of the task. So each invocation of a single task processes different sub regions and is allocated to different copy-sets.

Figure 20 a) shows copy-set 1 has a single task which is the first invocation of task A, $I_1(A)$ and processes a half($=\Re/2$) of the whole data frame($\Re = W \times H$). Each invocation of the task can process different data frames, different copy-sets or different sub regions. In figure 20, the size of a sub region of copy-set 1 becomes $\Re/2$ since the number of task invocation($=1$) within the copy-set ia 1. Copy-set 2 has two copied tasks which lead to two different invocations of the task A; $I_2(A)$ and $I_3(A)$. Each invocations $I_2(A)$ and $I_3(A)$ processes copy-set 2. The size for copy-set 2 is $\Re/2$. The size of each sub region of copy-set 2 is $\Re/4$ as copy-set 2 has two invocations. Figure 20 b) shows how the whole data frame $\Re$ is divided into copy-sets and, in turn, sub regions within each copy-set. Figure 20 c) shows how the execution time for task A is reduced by filling the idle processor $P2$ under an idle interval; $idleInterval$ due to exploiting heterogeneous data parallelism. In Figure 20 c),

EarliestEndTimeForSchedule is the earliest end time among processors within the stage. *EarliestEndNextTimeForSchedule* is the next earliest end time. *idleInterval* is an interval where no tasks are available for scheduling due to task dependency between *EarliestEndTimeForSchedule* and *EarliestEndNextTimeForSchedule*. Task *E* can be invoked after receiving data from task *A* due to data dependency between task A and task E.

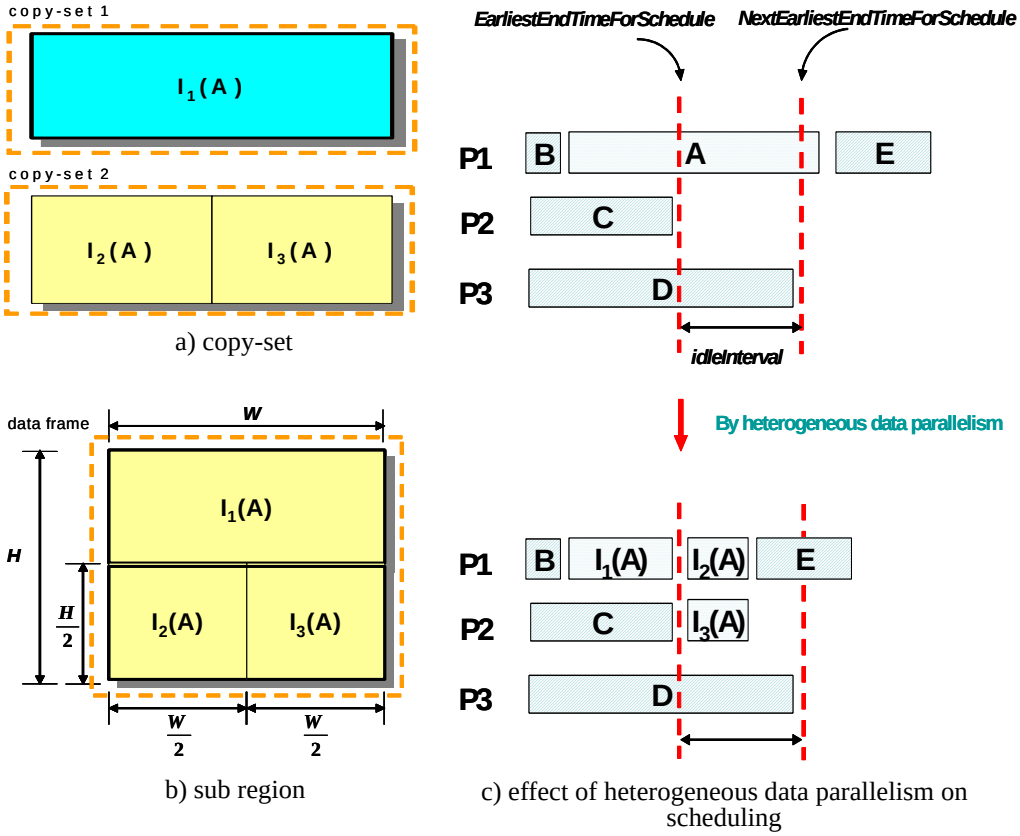


Figure 20. Heterogeneous data parallelism.

Task duplication under general data parallelism allows for each invocation of copied tasks to process different sequential data frame in parallel. Thus, task duplication under general data parallelism contributes toward improving throughput, but causes an

increased buffer size since each copied task processes multiple sequential data frames at the same time. However, task duplication under heterogeneous data parallelism contributes toward reducing execution time of the corresponding stage without increase of buffer size. Each invocation of a single task processes different sub regions within a single data frame.

Figure 21 compares task duplication of task A_k each under general data parallelism and under heterogeneous data parallelism. In figure 21 a), each invocation of task A_k processes different sequential data frames. The first invocation of task A_k, $I_1(A_k^{n-1})$ processes $n-1$ th data frame whereas $I_2(A_k^{n})$ and $I_2(A_k^{n+1})$ process each n th and $n+1$ th data frames respectively. Therefore, as the number of invocation increases, the buffer size between A_{k-1} and A_{k+1} increases too. In figure 21 b), the whole data frame is divided into several copy-sets. Each copy-set consists of different size of sub regions and is processed by different invocations of task A_k. Decision on the number of invocations of a task in each copy-set, the size of a copy-set and the size of sub regions within each copy-set are based on available idle processors.

For task A_k, the relationship between the size of copy-sets and each invocation of tasks is in equation 7-9.

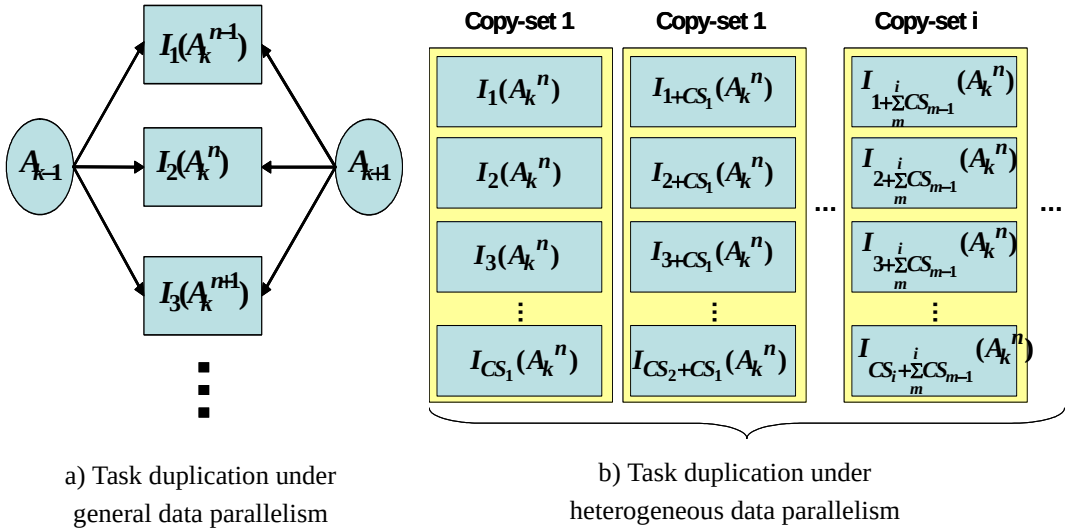
$$SR_i = region(I_n(A_k)),\ n = j + \sum_{m=1}^{i} CS_{m-1} \tag{7}$$

$$region(CS_i) = CS_i \times SR_i \tag{8}$$

$$frame = \sum_{i=1}^{M} region(CS_i) \tag{9}$$

SR_i returns an area processed by invocation n_{th} of task A_k within i th copy set.Here, SR_i is same for all j within the associated coy set. $region(CS_i)$ is the area processed within i th copy set. CS_i is the number of task invocations within i th copy-

set($CS_0 = 0$). M is the total number of copy-sets for processing the data frame, $frame$.



a) Task duplication under general data parallelism

b) Task duplication under heterogeneous data parallelism

Figure 21. Task duplication under general data parallelism and under heterogeneous data parallelism

3.2.3.3 memory usage

As CMOS technology progresses, the effective usage of *on-chip* memory becomes a key issue in integration of DSP chip. This section shows how in general a shared memory architecture and task duplication[1][22][47][80] influence the buffer memory size both in *on-chip* memory and in *external* memory. This comparison is based on an accumulative usage of memories used by tasks. The memory usage model allows for predicting the effect of task duplication on memory usage by a linear memory consumption pattern.

Task duplication is considered only if the associated task has the feature of heterogeneous data parallelism and processors are available for duplication. The number of duplications of a single task can be dynamically changed depending on the number of available processors. This section also shows how task duplication each under data parallelism and under heterogeneous data parallelism influences the size of used mem-

ory region.

3.2.3.3.1 Memory usage comparison

3.2.3.3.1.1 “Without Task Duplication”

Equation 4 shows memory usage under a separate memory architecture. Here, $|\tau_i|$ is the number of tasks within ith cluster, τ_i. $c(i,j)$ is the code size for task j within τ_i. $b(i,j,k)$ is the buffer size for kth input port of task j within τ_i. Equation 11 shows memory usage under a shared memory architecture, $|\tau_i|$ for buffer memory becomes 1 since all tasks within τ_i shares buffer memory.

- 1. A separate memory architecture

$$Mem_{sp} = \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|} c(i,j) + \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{k=1}^{|In_i|} b(i,j,k) \tag{10}$$

- 2. A shared memory architecture

$$Mem_{sh} = \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|} c(i,j) + \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|=1}\sum_{k=1}^{|In_i|} b(i,\mathbf{1},k) = \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|} c(i,j) + \sum_{i=1}^{|T_C|}\sum_{k=1}^{|In_i|} b(i,k) \tag{11}$$

3.2.3.3.1.2 “With Task Duplication”

A shared memory architecture both under general data parallelism and under heterogeneous data parallelism allows for reducing buffer size compared to a separate memory architecture whereas code size under both architectures is same. Task duplication under general data parallelism increases buffer size proportional to $|D_j|$ whereas task duplication under heterogeneous data parallelism doesn’t increase buffer size. It’s because copied tasks by task duplication under heterogeneous data parallelism process different offsets within the same data frame. Code size is assumed to include a stack

size. Thus, code size increases proportional to $|D_j|$ for both general data parallelism and heterogeneous data parallelism.

3.2.3.3.1.3 Task duplication under general data parallelism

- 1. A separate memory architecture

$$Mem_{sp,task\ duplicatio\ n,DP} = \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|}\sum_{k=1}^{|In_i|} b(i,j,m,k) \tag{12}$$

- 2. A shared memory architecture

$$Mem_{sh,task\ duplicatio\ n,DP} = \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|=1}\sum_{m=1}^{|D_j|}\sum_{k=1}^{|In_i|} b(i,j,m,k) = \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_C|}\sum_{m=1}^{|D_j|}\sum_{k=1}^{|In_i|} b(i,m,k) \tag{13}$$

3.2.3.3.1.4 Task duplication under a heterogeneous data parallelism

- 1. A separate memory architecture

$$Mem_{sp,task\ duplicatio\ n,HDP} = \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|=1}\sum_{k=1}^{|In_i|} b(i,j,\mathbf{1},k) = \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{k=1}^{|In_i|} b(i,j,k) \tag{14}$$

- 2. A shared memory architecture

$$Mem_{sh,task\ duplicatio\ n,HDP} = \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|=1}\sum_{m=1}^{|D_j|=1}\sum_{k=1}^{|In_i|} b(i,\mathbf{1},\mathbf{1},k) = \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_C|}\sum_{k=1}^{|In_i|} b(i,k) \tag{15}$$

3.2.3.3.2 Memory usage ratio

3.2.3.3.2.1 A separate memory architecture vs. A shared memory architecture[Without task duplication]

$$\frac{Mem_{sp}}{Mem_{sh}} = \frac{\sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|} c(i,j) + \sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|}\sum_{k=1}^{|In_i|} b(i,j,k)}{\sum_{i=1}^{|T_C|}\sum_{j=1}^{|\tau_i|} c(i,j) + \sum_{i=1}^{|T_C|}\sum_{k=1}^{|In_i|} b(i,k)} \tag{16}$$

3.2.3.3.2.2 A separate memory architecture vs. A shared memory architecture [With

task duplication]

- Under general data parallelism

$$\frac{Mem_{sp,task\ duplication,DP}}{Mem_{sh,task\ duplication},DP} = \frac{\sum_{i=1}^{|T_c|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_c|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|}\sum_{k=1}^{|In_i|} b(i,j,m,k)}{\sum_{i=1}^{|T_c|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_c|}\sum_{m=1}^{|D_j|}\sum_{k=1}^{|In_i|} b(i,m,k)} \tag{17}$$

- Under heterogeneous data parallelism

$$\frac{Mem_{sp,task\ duplication,HDP}}{Mem_{sh,task\ duplication},HDP} = \frac{\sum_{i=1}^{|T_c|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_c|}\sum_{j=1}^{|\tau_i|}\sum_{k=1}^{|In_i|} b(i,j,k)}{\sum_{i=1}^{|T_c|}\sum_{j=1}^{|\tau_i|}\sum_{m=1}^{|D_j|} c(i,j,m) + \sum_{i=1}^{|T_c|}\sum_{k=1}^{|In_i|} b(i,k)} \tag{18}$$

Remark 1 : Definitions for PDT scheduling - 1

Definition 1: t : a task.

Definition 2: τ_i : i th cluster.

Definition 3: $|\tau_i|$: the number of tasks; $\{t_1, t_2, \ldots, t_{|\tau_i|}\}$ within i th cluster, τ_i .

Definition 4: T_c : a set of total clusters $\{\tau_1, \tau_2, \ldots, \tau_{|T_c|}\}$.

Definition 5: $|T_c|$: the total number of clusters.

Definition 6: D_j : a set of invocations of j th task, t_j by task duplication, $\{I_1(t_j), I_2(t_j), \ldots, I_{|D_j|}(t_j)\}$.

Definition 7: $|D_j|$: the number of invocations j th task, t_j by task duplication.

Definition 8: In_i : a set of input ports of i th cluster, τ_i .
$In_i = \{p^{in}((1, \tau_i), p^{in}(2, \tau_i), \ldots, p^{in}(|In_i|, \tau_i))\}$.

Definition 9: $p^{in}(k, \tau_i)$: k th input port of i th cluster, τ_i .

Definition 10: $|In_i|$: the number of input ports of i th cluster, τ_i .

Definition 11: $b(i, j, k)$: a buffer of k th input port of j th task, t_j , within i th cluster, τ_i .

Definition 12: $b(i, j, m, k)$: a buffer of k th input port of m th invocation of j th task, t_j by task duplication, within i_{th} cluster, τ_i .

e.g.) $b(i, j, m, k) = b(i, 1, m, k) = b(i, m, k)$, $b(i, j, k) = b(i, 1, k) = b(i, k)$ when $|\tau_i| = 1$.

Definition 13: $c(i, j)$: a code of j th task, t_j , within i th cluster, τ_i .

Definition 14: $c(i, j, m)$: a code of m th invocation of j th task, t_j by task duplication, within i th cluster, τ_i .

Definition 15: w : window size. N : data frame size.

Clustering process groups tasks depending on task dependencies and possible sharing of buffers. More detailed explanation of clustering is given in the section 3.2.4. Suppose we have an application graph(named a **task dependency graph**) as shown in figure 22 a). Figure 22 b) is the graph after clustering, which is called a **cluster dependency graph**. After clustering, by seeing buf, the number of input ports of some nodes are changed, which affect memory usage of a cluster. Table 8 shows how buffer memory usage of figure 22 b) is changed depending on task duplication and a shared memory. Here code size is not influenced by task duplication or memory architecture. Table 8 provides an example of figure 22 with real numbers for a clear understanding of relationship between heterogeneous data parallelism and memory usage depending on architectures. In table 8, the value of $|T_c|$ is 8 and the value of $|\tau_3|$ is 3 and the value of $|\tau_4|$ is 2. For cases except τ_3 and τ_4, the value of $|\tau_i|$ is 1. The values of $|D|$ is assumed to be constant for all tasks. Figure 22 e) shows the relationship between w and N (frame size). w is window size and N is data frame size. We assume the value of w is $64 \times 1B(Byte) = 64B$ and the value of $N = 256 \times 256\,(Total\ pixels)$. The buffer size between clusters is assumed to be N. In case task duplication is applied, D is assumed to be 4. We also assume that task duplication is performed for only task 3 of the cluster τ_3 like figure 22 c). By seeing figure 22 b), $|In_i|$ is 1 for all clusters except $|In_8| = 3$.

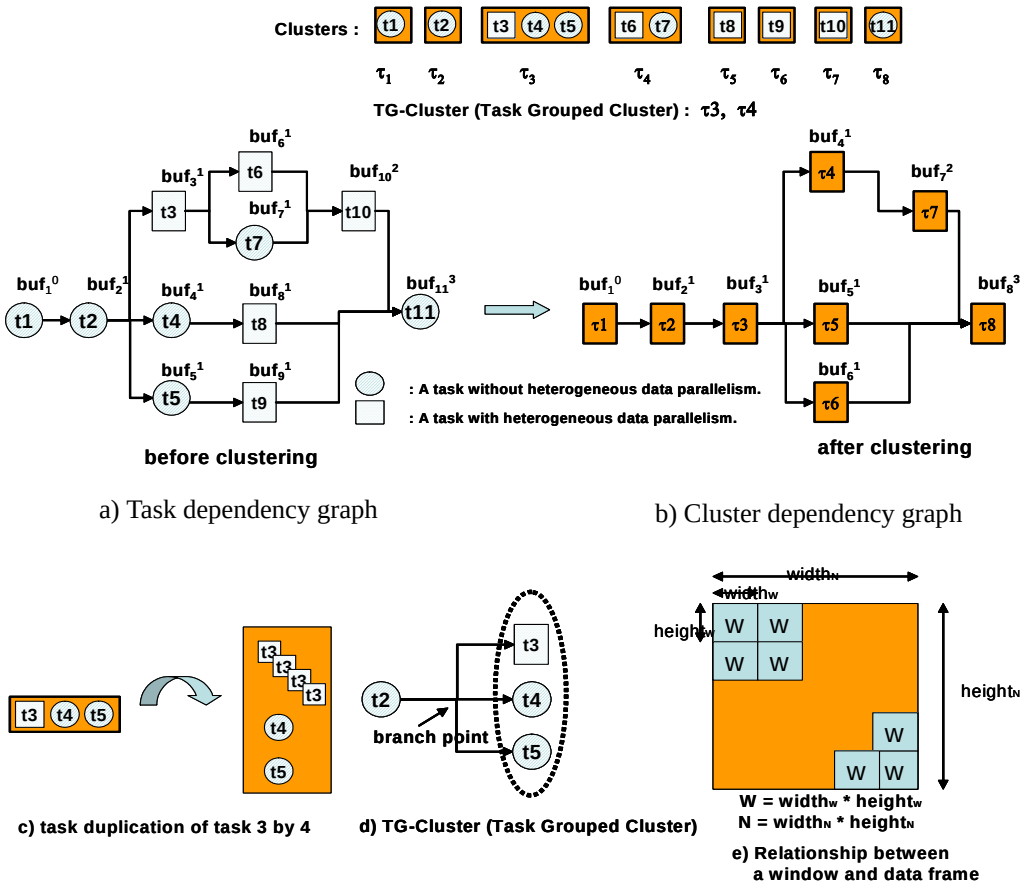


Figure 22. Examples of tasks, clusters, clustering and window

3.2.4 Scheduling

Our heuristic scheduling algorithm on multiprocessors tackles both heterogeneous data parallelism and task parallelism together in a pipelined way while considering user given constraints. Tasks immediately following a *branch point* are mapped to a TG-Cluster(Task Grouped Cluster) for exploiting the benefit of a shared memory architecture. By clustering, each task except tasks in TG-Cluster is mapped to the corresponding cluster one to one. After clustering, a new graph(a cluster dependency graph) is generated based on dependencies of clusters. The cluster dependency graph is used for partitioning clusters into stages of a pipeline. Each partition consists of a

Table 8. An example of comparison of buffer memory usages depending on task duplication and a memory architecture both under general data parallelism and under heterogeneous data parallelism.

		Buffer memory usage
Without Task Duplication	Separate Memory architecture	4[->#of normal clusters]*1[->|τi|]*1[->|Ini|]*64K -> normal clusters + 3[->|τ3|]*1[->|In3|]*64K -> τ3 + 2[->|τ4|]*1[->|In4|]*64K -> τ4 + 1[->|τ8|]*3[->|In8|]*64K -> τ8 = 12*64K = 768KB
	Shared Memory Architecture	6[->#of normal clusters]* 1[->|τi|]*1[->|Ini|]*64K -> normal clusters + 1[->|τ8|]*3[->|In8|]*64K -> τ8 =9*64K = 576KB
With Task Duplication under DP	Separate Memory architecture	3[->#of normal clusters]* 1[->|τi|]*1[->|Ini|]*1[->|Di|]*64K-> normal clusters + 2[->|τ3|]* 1[->|In3|]*1[->|D3|]*64K -> task4 and task 5 in τ3 + 1[->|τ3|]* 1[->|In3|]*4 [->|D3|]*64K –> task3 in τ3 + 2[->|τ4|]*1[->|In4|]*1[->|D4|]*64K -> τ4 + 1[->|τ6|]* 1[->|In6|]*4[->|D6|]*64K -> τ6 + 1[->|τ8|]*3[->|In8|]*4[->|D8|]*64K -> τ8 = 27*64K = 1728KB
	Shared Memory Architecture	4[->#of normal clusters]* 1[->|τi|]*1[->|Ini|]*1[->|Di|]*64K-> normal clusters + 1[->|τ3|]* 1[->|In3|]*1[->|D3|]*64K -> task4, task 5 in τ3 + 1[->|τ3|]* 1[->|In3|]*4[->|D3|]*64K –> task3 in τ3 + 1[->|τ6|]* 1[->|In6|]*4[->|D6|]*64K -> τ6 +1[->|τ8|]* 3[->|In8|]*4[->|D8|]*64K -> τ8 = 25*64K = 1600KB
With Task Duplication under HDP	Separate Memory architecture	3[->#of normal clusters]* 1[->|τi|]*1[->|Ini|]*1[->|Di|]*64K-> normal clusters + 2[->|τ3|]* 1[->|In3|]*1[->|D3|]*64K -> task4 and task 5 in τ3 + 1[->|τ3|]* 1[->|In3|]*1 [->|D3|]*64K –> task3 in τ3 + 2[->|τ4|]*1[->|In4|]*1[->|D4|]*64K -> τ4 + 1[->|τ6|]* 1[->|In6|]*1[->|D6|]*64K -> τ6 + 1[->|τ8|]*3[->|In8|]*1[->|D8|]*64K -> τ8 = 12*64K = 768KB
	Shared Memory Architecture	4[->#of normal clusters]* 1[->|τi|]*1[->|Ini|]*1[->|Di|]*64K-> normal clusters + 1[->|τ3|]* 1[->|In3|]*1[->|D3|]*64K -> τ3 + 1[->|τ6|]* 1[->|In6|]*1[->|D6|]*64K -> τ6 + 1[->|τ8|]* 3[->|In8|]*1[->|D8|]*64K -> τ8 = 9*64K = 576KB

DP: general data parallelism. HDP: Heterogeneous data parallelism.
In each notation([->X]), the number ahead of [->X] represents the value of X. e.g. 3[->$|\tau_3|$] means the value of $|\tau_3|$ is "3". The notation is given to provide a clear understanding of where each number comes from.

group of clusters. By partitioning clusters are allocated to stages of the pipeline correspondingly. Clusters in each partition build a new cluster dependency graph within the corresponding partition. After partitioning, for scheduling tasks within each stage of the pipeline, the original task dependency graph is used not to violate data depen-

```
FindSchedule() {
  CurrentBest = ∞
  for(Param = setUpParam(); Param<Param.searchRegion){
    BestSchudule = PDT–schedule(G, P, Param){
    if(BestSchudule better than CurrentBest)
      CurrentBest = BestSchudule;
  }
}


PDT–schedule(G, P, Param){
  ScheduleList = Φ;
  π_i = G;
  partitionDB = ϕ
  PDT(G, P, Param, π_i, i, partitionDB);
  Pipeline[] = partitionDB.buildPipelines();
  for(i=0;i<Pipeline.num;i++) {
    for(j=0;j<Pipeline[i].Pa.length;j++) {
      Pipeline[i].S[j] = HDEST(Pipeline[i].Pa[j], P, Cth, Param);
    }
    if(Pipeline[i].schedule < C_th)
      ScheduleList.put(Pipeline[i].schedule);
  }
  BestSchudule = ScheduleList.getBestSchedule();
  return BestSchudule;
}
```

partitionDB: keeps all partitions produced by **PDT**. By *partitionDB*.*buildPipelines*() builds pipelines by searching partitions in different depth of **PDT**.
π_i: an initial partition and initially set as the original input graph.
Pipeline[]: pipelines in every search depth level of ***PDT***
Pipeline[*i*]: *i* th pipeline.
Pipeline[*i*].*Pa*[*j*]: holds *j* th partition's information of *i* th pipeline.
Pipeline[*i*].*S*[*j*]: holds *j* th stage's information of *i* th pipeline.
P: processors
C_{th}: threshold of given constraints.
BestSchudule: the best solution produced by ***PDT–schedule*** with a given parameter values.
CurrentBest: current best solution held by ***FindSchedule***.
HDEST(): A processor allocation algorithm based on EST(Earliest Start Time) with a *HDP* (Heterogeneous Data Parallelism).
PDT(): produces pipelines based on **PDT**(Pipeline Decomposition Tree) algorithm.

Figure 23. FindSchedule() algorithm

dency. Figure 22 a) and b) show how the task dependency graph is converted into a cluster dependency graph. A cluster dependency graph satisfies a topological sort within each partition. A parent partition is divided into two sub partitions while making clusters in each partition have weak cluster dependencies. A weak cluster depen-

dency in each partition allows for more potential parallelism. Here, partitioning is applied to a cluster level[44][85][101] while processor allocation and task scheduling are applied to a task level inside each cluster. Finally, each stage can have evenly divided estimated execution time and potential parallelism[55][81]. Potential parallelism is exploited through the scheduling of each stage. For partition, this thesis provides a heuristic method named **CPAP**(**C**ritical **PA**th based **P**artitioning). **CPAP** partitions clusters into two sub-partitions by cutting a critical path of the associated cluster dependency graph evenly in terms of estimated execution time of clusters. A critical path is the longest dependency chain. By **CPAP**, the possibility of an over-loaded or an under-loaded stage can be prevented. This procedure is performed recursively by a depth first search tree until an appropriate number of stages in a pipeline is obtained. This recursive partitioning by a depth first search tree generates a tree named **PDT**(Pipeline Decomposition Tree). Each node within **PDT** corresponds to a stage in the pipeline.

PDT produces several sets of pipelines with different number of stages by choosing partitions in different tree depth correspondingly. For tasks within each partition of the associated pipeline, a precise process allocation and a scheduling process named **HDEST** (**H**eterogeneous **D**ata parallelism **E**arliest **S**tart **T**ime)[89] is applied. We named our heuristic algorithm **PDT scheduling**. Here, heterogeneous data parallelism and task parallelism are simultaneously considered along with IPC cost, memory usage and bus contention. The scheduling algorithm has specific parameters which influence the scheduling outcome. The values of these parameters may vary depending on the change of applications. **PDT scheduling** is applied in an iterative way by

changing the values of parameters appropriately. The objective is to find the best schedule satisfying given constraints for a given application. Figure 23 roughly shows the top level function of scheduling algorithm.

Remark 2 : Definitions for PDT scheduling - 2

Definition 16: $executeTime(\tau_i) = \sum_{j=1}^{|\tau_i|} executeTime(t_{i,j})$

Definition 17: π_i : i th partition.

Definition 18: $|\pi_i|$: the number of clusters; $\pi_i = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,|\pi_i|}\}$ within a i th partition, π_i .

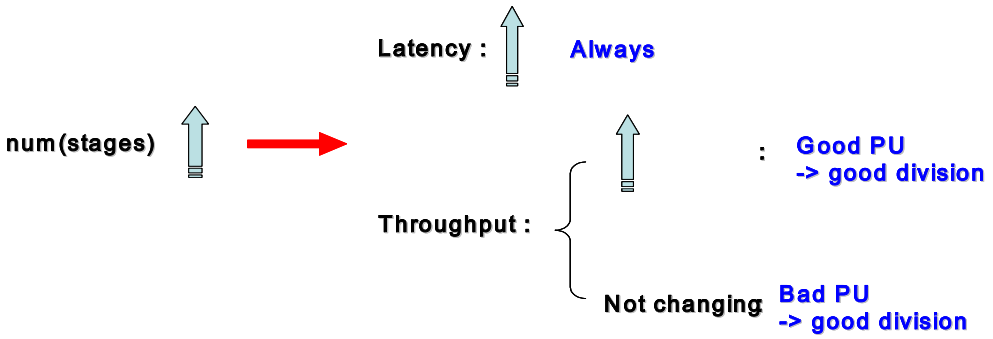
Definition 19: $\tau_{i,k}$: k th cluster within i th partition, π_i . Here, k is a local index within partition, π_i and is different from a global index in Definition 2.

Definition 20: $executeTime(\pi_i) = \sum_{j=1}^{|\pi_i|} executeTime(\tau_{i,j})$

3.2.4.1 "PDT()" - Pipeline Decomposition Tree

Latency is inversely proportional to one per throughput (latency of bottleneck stage in a pipeline). The schedule could be obtained based on a trade off between throughput and latency while satisfying resources constraints. Ideally, the throughput can be assumed to be improved by simply increasing the number of stages in a pipeline by sacrificing latency. However, improperly divided pipeline with poor PUs(Processor Utilization) deteriorates throughput as well as latency in spite of increased number of stages of a pipeline. Figure 24 shows the relationship between the latency and the throughput in a pipeline.

The number of stages is a critical factor influencing both throughput and latency in a pipelined multiprocessor based scheduling. However, deciding an appropriate number of stages in a pipeline under given constraints is not trivial. This thesis provides a new way named **PDT(Pipeline Decomposition Tree)** for generating pipelines. **PDT** is a modified depth first search algorithm. Starting from the whole graph, **PDT** divides the



$num(Stages)$: the number of $Stages$ in a pipeline

Figure 24. Relationship between Latency, Throughput and Number of stages

graph into two sub partitions while satisfying a topological sort in each partition. The objective is that clusters in each partition have weak cluster dependencies so that, consequently, each partition has more potential parallelism. Here, the cluster dependency becomes highest when all clusters in a partition are linearly linked in a row. On the other hand, the cluster dependency is weakest when all clusters in a partition are independent. A relatively weak cluster dependency in a partition gives more potential parallelism in scheduling. Equation 19 and equation 20 are to divide a partition into two sub partitions so that cluster dependency in each partition is evenly distributed and each sub partition π_1 and π_2 has the similar level of an execution time of partition, $executeTime(\pi)$.

$$min(|Dep(\pi_1) - Dep(\pi_2)|) \tag{19}$$

$$min(executeTime(\pi_1) - executeTime(\pi_2)) \tag{20}$$

Here, clusters in each partition should satisfy the following condition.

$$if\ \tau_i \subset \pi_1,\ successors(\tau_i) \subset \pi_1 \text{ or } successors(\tau_i) \subset \pi_2. \tag{21}$$

$$if\ \tau_i \subset \pi_2,\ successors(\tau_i) \subset \pi_2. \tag{22}$$

$$Dep(\pi) = e_{\pi_{cp}} + i \bullet e_{\pi_{\overline{cp}}} \tag{23}$$

$$|Dep(\pi)| = \sqrt{e_{\pi_{cp}}^{\ 2} + e_{\pi_{\overline{cp}}}^{\ 2}} \tag{24}$$

$$e_{\pi_{cp}} = \sum_{i=1}^{|\pi_{cp}|} j \tag{25}$$

$$e_{\pi_{\overline{cp}}} = \sum_{j=1}^{\left|CH_{\pi_{\overline{cp}}}\right|} \sum_{k=1}^{\left|CH_{j,\pi_{\overline{cp}}}\right|} k \quad (26)$$

$Dep(\pi)$: dependency degree of clusters in partition, π.
$e_{\pi_{cp}}$: the weighted sum of edges of clusters in a critical path within partition, π.
$e_{\pi_{\overline{cp}}}$: the weighted sum of edges of clusters outside a critical path within partition, π.
π_{cp}: the number of clusters in a critical path within partition π.
$\pi_{\overline{cp}}$: the number of clusters not in a critical path within partition π.
CH : a **C**luster c**H**ain(CH) which is connected with two or more clusters in a row within partition, π. Each π can have one or more isolated CHs.
$CH_{\pi_{\overline{cp}}}$: a set of CHs not in a critical path within partition, π.
e.g. $CH_{\pi_{\overline{cp}}} = \left\{ CH_{1,\pi_{\overline{cp}}}, CH_{2,\pi_{\overline{cp}}}, \ldots, CH_{\left|CH_{\pi_{\overline{cp}}}\right|,\pi_{\overline{cp}}} \right\}$.
$\left|CH_{\pi_{\overline{cp}}}\right|$: the number of CHs not in a critical path within partition, π.
$CH_{k,\pi_{\overline{cp}}}$: kth CH not in a critical path within partition, π.
$\left|CH_{k,\pi_{\overline{cp}}}\right|$: the length of $CH_{k,\pi_{\overline{cp}}}$.

$Dep(\pi)$ is dependency degree of clusters within partition π. $Dep(\pi)$ is a complex number. The real number of $Dep(\pi)$ represents the weighted sum of edges of a critical path in partition π while the imaginary number represents the weighted sum of edges of clusters which are not involved in a critical path within partition π. The real number potentially corresponds to the lowest bound of latency of the partition π. This bound can be further decreased by exploiting HDP(Heterogeneous Data Parallelism). The imaginary number shows dependency degree of clusters outside the critical path within partition π. The low number provides more parallelism during scheduling.

3.2.4.1.1 CPAP (Critical PAth based Partition)

To divide a partition into two sub partitions while satisfying both equation 19 and equation 20, this thesis provides a heuristic method named **CPAP**(**C**ritical **PA**th based **P**artition) which uses estimated execution times of clusters as well as the critical path of the graph. **CPAP** groups clusters depending on heterogeneous data parallelism and cluster dependency. Figure 26 shows $PDT()$ algorithm with a basic criterion for decid-

ing on further progress of **PDT** make-up process of a given node within **PDT**. Precise criteria for $PDT()$ are introduced in the section 3.2.4.1.3.

```
CPAP(G, CutTh) {
  CPAPDB = ⊥;
  LongestPath = ⊥;
  π_left = ⊥; π_right = ⊥; G_left = ⊥; G_right = ⊥;
  while(TRUE) {

    Step 1 => Find the longest critical path in a given graph.
    for(i=0;i<G.length;i++) {
      CP[i] = FindCP(G);
      if(LongestPath.length < CP[i].length)
        LongestPath = CP[i];
    }

    Step 2 => Add a half of clusters in current longest path, LongestPath to the left partition
only if
    each node in LongestPath satisfies equation 21 and 22.
    for((i=0;i<LongestPath.length;i++) {
      predeNodes[]=predecessor(LongestPath.node[i]);
      if(∀predeNodes = ⊥ || ∀predeNodes ⊂ π_left) {
        π_left += LongestPath.node[i];
        G_left.add(LongestPath.node[i]);
      }
      else
        break;
    }
    G = G - LongestPath;

    Step 3 => continue until executeTime(π_left) reaches CutTh.
    if(executeTime(π_left) ≥ CutTh or G = ⊥)
      break;
  }
  π_right = G.nodes();
  G_right.add(G.nodes());
  CPAPDB = {π_left, π_right, G_left, G_right};
  return CPAPDB;
}
```

FindCP: return the critical path, $LongestPath$ from graph, G.
$predeNodes[]$: predecessors of a given node.
$predecessor(node)$; return predecessors, $predeNodes[]$ of a given node, $node$.
$CPAPDB$: a data base of partitions and graphs(π_{left}, π_{right} G_{left} and G_{right}) grouped by $CPAP$.

Figure 25. CPAP algorithm

- $findCutThreshold$

```
PDT(G, P, Param, π, depth, partitionDB){
  AveExTime = partitionDB.getAveTime(depth);
  if(checkBasicCriterion(π, AveExTime, num(P)) == continuePDTDivision) {
    switch (checkPreciseCriterion(π, π_left, π_right))
    begin
      case PDTDivisionContinue:
      begin
        CutTh = findCutThreshold(G);
        CPAPDB = CPAP(G, CutTh);
        π_left = CPAPDB.π_left; π_right = CPAPDB.π_right;
        G_left = CPAPDB.G_left; G_right = CPAPDB.G_right;
        DivideProcessors(P_left, P_right, P, G);
        partitionDB.put(depth, π_left, OnGoingNode);
        PDT(G_left, P_left, Param, π_left, depth++);
        partitionDB.put(depth, π_right, OnGoingNode);
        PDT(G_right, P_right, Param, π_right, depth++);
      end
      case PDTDivisionStop:
      begin
        partitionDB.put(depth, π, TerminalNode);
      end
      case PartitionDuplication:
      begin
        partitionDuplication(π);
      end
    end
  }
  else {
    partitionDB.put(depth, π, TerminalNode);
  }
}
```

findCutThreshold(): find a *CutTh* which is used to be used for dividing a graph of a parent partition into two sub partitions evenly and is a half of execution times of a given graph.
G_{left}: a graph pruned to the left partition, π_{left}.
G_{right}: a graph pruned to the right partition, π_{right}.
DivideProcessors: divide a given number of processors for each sub partition based on each $excuteTime(\pi_{left})$ and $excuteTime(\pi_{right})$.
maxT: the longest execution time of a task which is a lower bound for the throughput.
AveExTime: an average value of *executeTime*()s of partitions in a given level of depth within a pipeline.
This value is set by $partitionDB.getAveTime(depth)$. The $partitionDB.getAveTime(depth)$ calculates *AveExTime* by referring to partitions in neighboring depths around a given level of *depth*.
$checkBasicCriterion(\pi, AveExTime, num(P))$: check if partition, π satisfies a basic criterion for **PDT** division. The function will be described in the following section.
$checkPreciseCriterion(\pi, \pi_{left}, \pi_{right})$: For partition, π satisfying a basic criterion, checking with a precise criterion for PDT division is performed. The function will be described in the following section.
TerminalNode: a node whose further division in **PDT** is not possible.
OnGoingNode: a node whose division in **PDT** can be exploited further.

Figure 26. PDT() algorithm

This function is to find a $CutTh$ which is a threshold for cutting a parent partition into two sub partitions. A half of the total execution time of a given graph is used for a threshold value for cutting.

$$CutTh = \frac{\sum_{i=1}^{|\pi|} executeTime(\tau_i)}{2}$$

- Precise way to divide a parent partition into two sub partitions.

In many cases, ideally and evenly dividing a partition into two sub partitions in terms of the execution time is not possible due to inequity of the graph's internal dependency. Thus, specially, for the cluster in a boundary position precise cutting needs to be considered.

$$Z_{out} = \min_{i=1}^{|\pi|} \left[\sum_{j=1}^{i} executeTime(\tau_i) - CutTh \right] \tag{27}$$

$$Zout = \begin{cases} Zmin & \text{if (cutflag == accepted) a cut point = i;} \\ Zmin\text{-}1 & \text{if (cutflag == excluded) a cut point = i-1} \end{cases}$$

$|\pi|$: the number of clusters in a given partition, π.
Z_{out} : the values subtracting $CutTh$ from accumulated $executeTime$ () up to i_{th} cluster.
Z_{min} : the minimum value of Z_{out}.
$cutflag$: the flag for zigzagging a cut point in a graph, G_π of a given partition, π.
$accepted$: accept the cluster just over $CutTh$ in a $executeTime(\pi_{left})$ to the left partition.
$excluded$: excluded the cluster just over $CutTh$ in a $executeTime(\pi_{left})$ from the left partition.

Here, the $cutflag$ needs to zigzag between $accepted$ and $excluded$ so that execution-times of partitions are evenly distributed along with increase of tree depth in spite of uneven pattern of data dependency. Otherwise, either left or right partition always gets bigger than the other counterpart. It causes undesirable deviation increase between partitions which in turn results in unbalanced execution time distribution. By seeing

figure 27, if the *cutflag* is set to *excluded*, then the right partitions are always bigger than the left partitions like figure 27 a). Or like figure 27 b), the left partitions are always bigger than the right partitions. figure 27 c) shows a precise partitioning by zig-zag cutting.

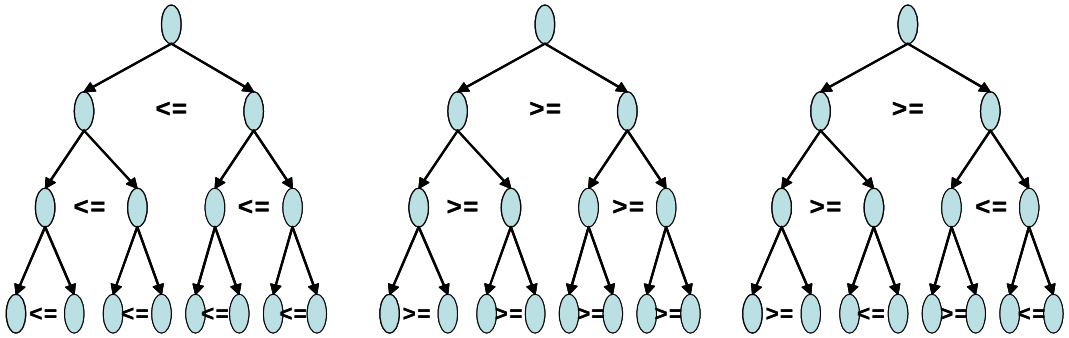


Figure 27. A variation of the size of partitions depending on the *cutflag*.

Figure 28 shows how **CPAP** tracks down a global and a local critical path to divide a parent partition into two sub partitions.

3.2.4.1.2 Effects of *executeTime*() and cluster dependencies.

An execution time of a cluster is a major factor for dividing partitions. Finally, partitions relate to stages in a pipeline. The objective of partitioning is that each partition have evenly divided execution time. So a pipeline provides the best throughput under a given number of processors after processor allocation and then scheduling is applied to the partition. Therefore, decision on how many stages(or partitions) are suitable for a pipeline is a critical factor influencing the final schedule result. However, various cluster dependency patterns between partitions and different operational features of tasks within a cluster can result in unexpected execution time distribution among partitions. It is because the proportion of heterogeneous data parallelism tasks in each cluster or different patterns of cluster dependency in each partition causes an unbalanced

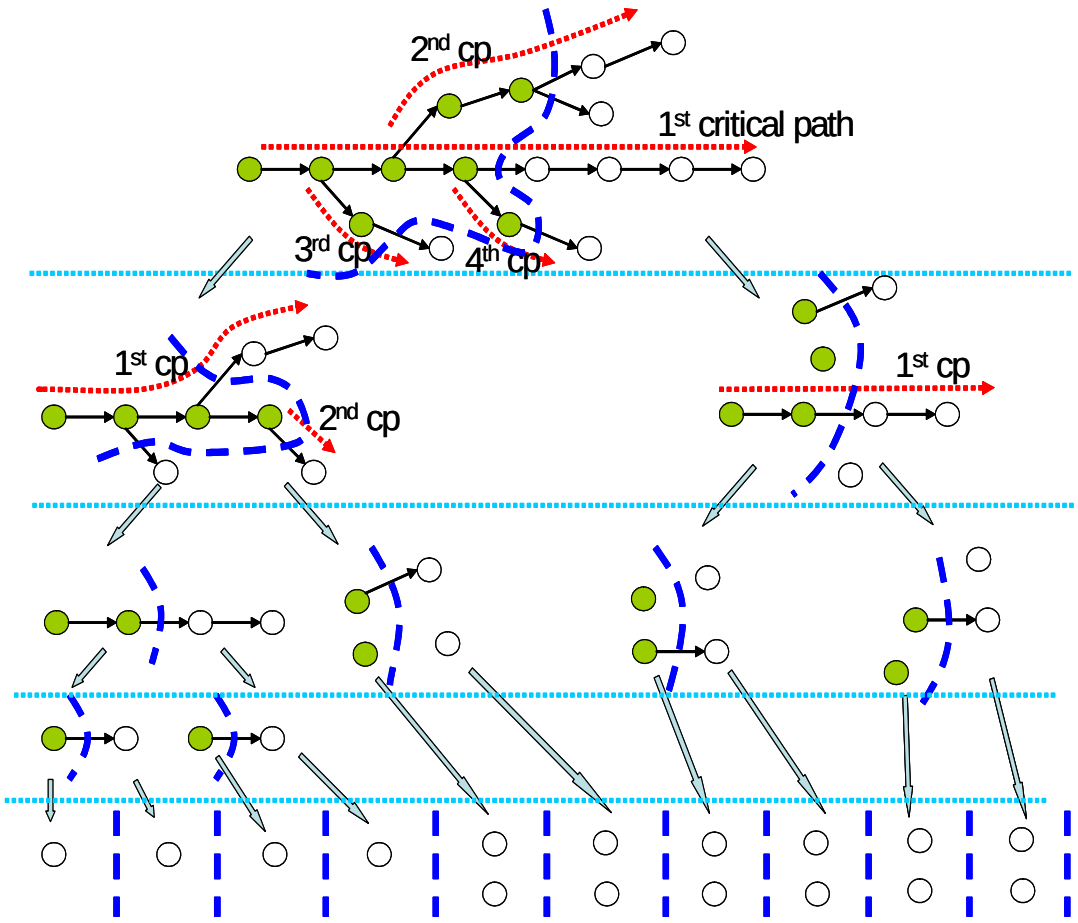


Figure 28. An example of usage of **CPAP** in **PDT**

execution time between stages. While building up **PDT**, these potential factors must be precisely considered by applying various criteria based on operational feature of tasks in each cluster and cluster dependency pattern in a partition. Classification of clusters depending on existence or nonexistence of heterogeneous data parallelism of tasks in each cluster and a cluster dependency pattern in each partition allows for more precisely divided workload for stages in a pipeline.

Remark 3 : Definitions for PDT scheduling - 3

Definition 21: THD : A task with HDP (Heterogeneous Data Parallelism).

Definition 22: $TNHD$: A task without HDP (Heterogeneous Data Parallelism).

Definition 23: $executeTime(THDs)$: the execution time of THDs

Definition 24: $executeTime(THNDs)$: the execution time of TNHDs

Definition 25: $executeTime$ (*Tasks in the longest critical path*): the execution time of tasks in the longest critical path.

Definition 26: $executeTime$ (*Other tasks not included in the longest critical path*): the execution time of other tasks not included in the longest critical path.

Observation 1: Effects of $executeTime(THD)$ and $executeTime(TNHD)$.

Figure 29 shows that the final execution times of each partition obtained by **HDEST** is different from predicted execution times due to tasks with heterogeneous data parallelism which allows for further exploitation of hidden parallelism. **HDEST** is the linked list based greedy scheduling method suggested in this thesis. **HDEST** adopts heterogeneous data parallelism. **HDEST** is described in detail in the section **3.2.4.2.1**. In figure **29**, **PDT scheduling** produces partition 1 to partition 4 initially. Here, both partition 3 and partition 4 have bad PUs(Processor Utilization) while partition 1 and partition 2 have good PUs by exploiting HDP(Heterogeneous Data Parallelism). Exploitation of $executeTime(THD)$ and an $executeTime(TNHD)$ without consideration of PU(Processor Utilization) results in undesirable execution time distribution among partitions. By considering PU(Processor Utilization) of each partition, a further division is applied both to partition 3 and to partition 4. Finally, six partitions(partition 1 to partition 6) with evenly divided execution times are obtained.

Observation 2: Effect of $executeTime$ (*Tasks of clusters in the longest critical path*) and $executeTime$ (*Tasks of clusters not in the longest critical path*).

An $executeTime$ (*Tasks of clusters in the longest critical path*) within a graph is a major factor determining the latency of the graph. Therefore, partitioning focuses on dividing the longest critical path of the corresponding graph evenly in each level of tree depth. A critical path based division allows for evenly distributed execution times of

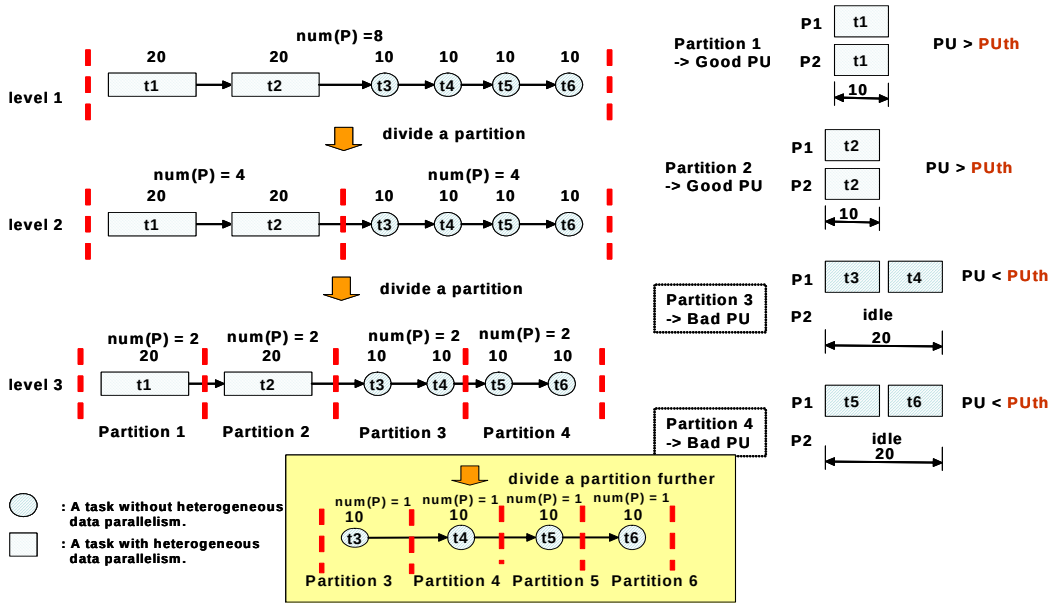


Figure 29. Effect of *THD*s and *TNHD*s in scheduling

partitions which can be exploited later for further reducing execution time of stages in a pipeline and increased potential parallelism in each partition. However, figure 30 shows that the execution time of partition A is almost twice as large as the one of partition B after applying **HDEST** to them. It is because partition B has more potential parallelism than partition A, which was not detected during **PDT** process. An improved schedule is obtained through a further division in conjunction with PU (Processor Utilization) and $PUth$.

3.2.4.1.3 Division criteria

Partitioning of **PDT** is determined by referring to $executeTime()$s of clusters based on division criteria of **PDT**. Division criteria is classified by considering a predicted execution time of each partition. These criteria relate to coefficient values. Appropriate values for these coefficients vary depending on graph characteristic. This thesis

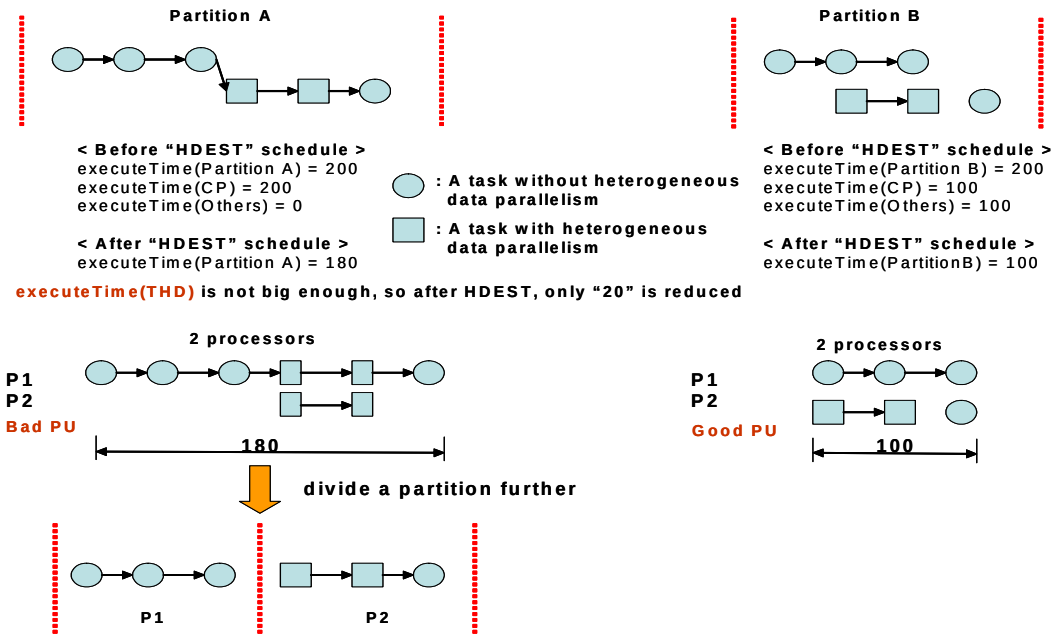


Figure 30. Effect of *executeTime*(*Tasks in the longest critical path*) and *executeTime*(*Other tasks not included in the longest critical path*) in scheduling

applies **PDT scheduling** algorithm in an iterative way with different coefficient values.

- Basic criterion

During division of a partition of **PDT**, $executeTime(\pi)$ of partitions should be evenly distributed to prevent a bottleneck stage in a pipeline since the bottle neck stage results in degrading the throughput. Therefore, the decision on division of a certain partition is based on the average value of $executeTime()$s of other terminal node partitions. Each partition of PDT tree can be classified into two groups, a terminal node partition, $TerminalNode$ and an on-going node partition, $OnGoingNode$. $TerminalNode$ is a node whose further division of **PDT** is not allowed whereas $OnGoingNode$ is a node whose division of **PDT** can be exploited further. **PDT** keeps track of every partition in each level of **PDT** make-up process so that **PDT** can provide multiple pipelines with different trade-off between latency and throughput. To satisfy various graph characteristics,

α_{basic} varies depending on a graph characteristics. the algorithm uses coefficient, α_{basic} for comparison of $executeTime(\pi)$ and $AveExTime$. The following condition shows the basic criteria for division. First, $executeTime(\pi)$ must be bigger than $AveExTime$. Second, $executeTime(\pi)$ should be bigger than $\boldsymbol{maxT}$ for dividing partition, π further. Third, of course, the number of processors given for partition, π should be larger than or equal to at least two for further division.

< Basic criterion >

```
checkBasicCriterion(π, AveExTime, P) {
  if(executeTime(π) > α_basic × AveExTime && executeTime(π) > maxT && num(P) ≥ 2)
    return continuePDTDivision;
  else
    return stopPDTDivision;
}
```

α_{basic} : a coefficient for a basic criterion, which allows adaptive comparison of $executeTime(\pi)$ and $AveExTime$ to determine stopping condition of **PDT**-make up process.

Figure 31 shows **PDT**(Pipeline Decomposition Tree) and how a basic division criterion is used for dividing partitions.

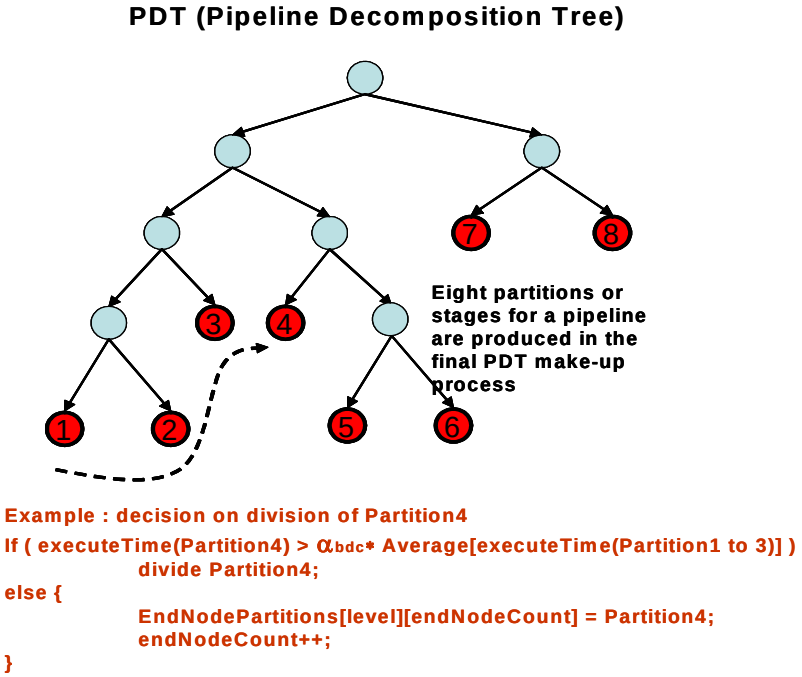


Figure 31. PDT(Pipeline Decomposition Tree) and division by basic division criterion

- Precise criterion

In PDT, criteria for partitioning are hierarchically applied so that graphs of unusual patterns are filtered out for precise analysis. Therefore, when the deviation of $executeTime(\pi)$ between two partitions is over a threshold, more precise criterion of equation 28 is applied. This usually happens when a cluster dominating an executing time of parent partition is placed in a boundary between two sub partitions. Figure 32 shows two cases where a cluster dominating an execution time of a given partition is located in a boundary. Despite extreme differences in an $executeTime(\pi)$, case 1 produces evenly divided $executeTime(\pi)$ of partitions. However, case 2 can't be divided any further. So the corresponding partition becomes the terminal node partition, $TerminalNode$, as a further division of the partition deteriorates execution time distribution among partitions.

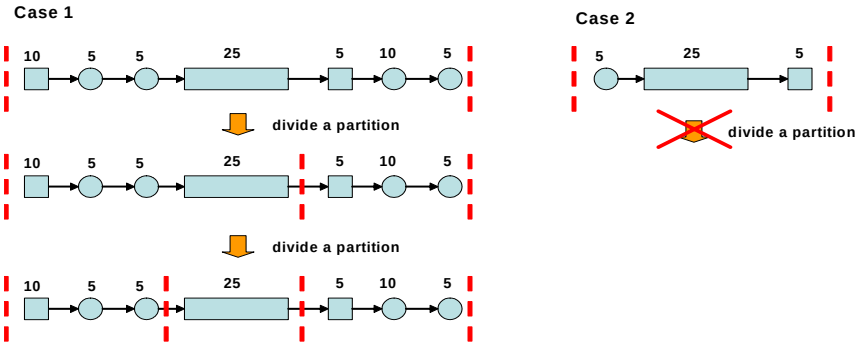


Figure 32. Examples with a large difference in *executeTime(Partition)*s between two sub partitions

$$if(executeTime(\pi_{left}) \gg executeTime(\pi_{right}) \parallel executeTime(\pi_{left}) \ll executeTime(\pi_{right})) \quad (28)$$

$$if(executeTime(\tau_{boundary}) \leq \alpha_{precise_1} \times executeTime(\pi)) \quad (29)$$

$$if(executeTime(THD_{\pi}) \geq \alpha_{precise_2} \times executeTime(\pi)) \quad (30)$$

Equation 29 checks a ratio between an execution time of a cluster in a boundary and an overall execution time of the partition. So for partitions satisfying equation 29, further division is considered. However, for any partition violating equation 29, two solutions

< Precise criterion >

```
checkPreciseCriterion(π, π_left, π_right) {
  if(executeTime(π_left) » executeTime(π_right) || executeTime(π_left) « executeTime(π_right)) {
    if(executeTime(τ_boundary) ≤ α_precise1 × executeTime(π)) {
      return PDTDivisionContinue;
    }
    else {
      if(executeTime(THD_π) ≥ α_precise2 × executeTime(π)) {
        return PDTDivisionStop;
      }
      else {
        return PartitionDuplication;
      }
  }
  else{
      return PDTDivisionContinue;
  }
}
```

π : a given partition which will be divided into two sub partitions (π_{left} and π_{right}).
$\tau_{boundary}$: Cluster placed in a boundary of π.
$\alpha_{precise_1}$: is initially set around $1/3$, however, this value is precisely reconfigured by iteratively applying appropriate parameter values.
$\alpha_{precise_2}$: is initially set around $1/2$, however, this value is precisely reconfigured by iteratively applying appropriate parameter values.
THD_π : THD in a partition, π.
PARTITION DUPULICATION : duplication of a whole π up to a given number of processors

are possible in conjunction of an $executeTime(THD_\pi)$. If the partition satisfies equation 30, then further division is not allowed and **HDEST** performs task duplication to reduce the execution time of a given partition. For the partition violating equation 30, **partition duplication** can be considered. **Partition duplication** is different from task duplication. **Partition duplication** is performed by copying the whole partition up to the number of processors available. **Partition duplication** can improve the throughput by having each copied partition handle different sequential data frames. However, **partition duplication** causes an increase in the buffer usage due to intrinsic feature of data parallelism. It is desirable to consider **partition duplication** only if the partition finally becomes a bottleneck stage in a pipeline. Figure 33 shows an example of a cluster dominating most of an $executeTime(\pi)$ of a partition. Case 1 exploits heterogeneous

data parallelism for scheduling and obtains stage latency(=25). Stage latency is the execution time of a given stage. Case 2 copies the whole partition by two. If this partition in a pipeline is a bottleneck, throughput can be improved to 1/(40/2) even though a stage latency of case 2 is still 40. It is assumed that an application run infinitely and handle different sequential data frames in each iteration. In case 2, the original partition and the copied partition handle different sequential data frame.

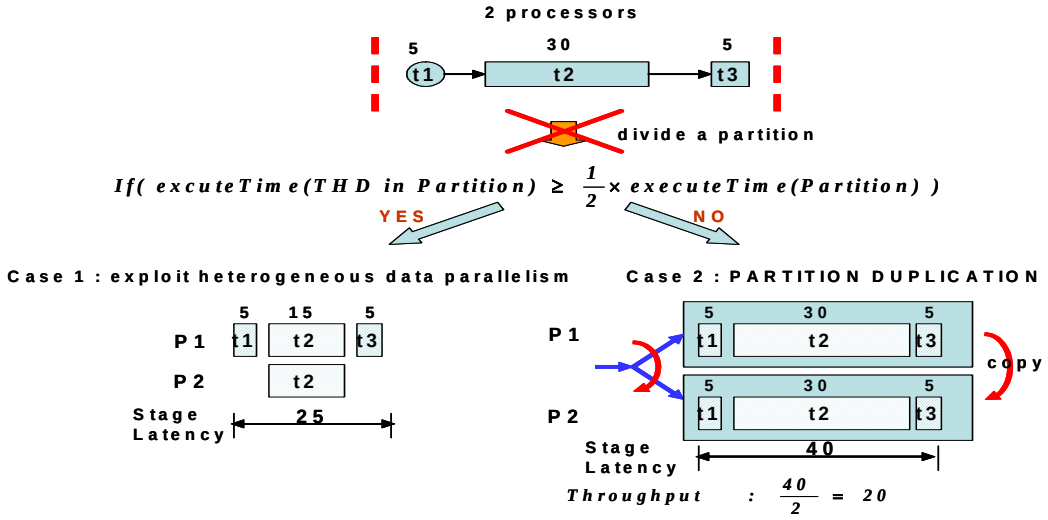


Figure 33. Handling of the case of one task dominating most of *excuteTime (Partition)*

3.2.4.1.4 Trade-off between Latency and throughput in PDT.

During **PDT** make-up process, information about all partitions in intermediate levels is stored up. Partitions in intermediate levels provide various pipelines with various trade-offs between latency and throughput while satisfying given constraints. Here, partitions in each intermediate levels of **PDT** are mapped to different stages of various pipelines. The way of mapping partitions to stages is based on the distribution of $executeTime(\pi)$ s. Therefore, partitions in different depths of **PDT** can be picked up to

generate pipelines. Figure 34 shows how each pipeline is made of intermediate partitions of **PDT** in different depth. Pipeline 2 is made up of partitions in different levels of depths while pipeline 1 is made up with partitions in the same tree depth. Pipeline 1 has a better latency than both pipeline 2 and pipeline 3 even though throughput of pipeline 1 is worse than both pipeline 2 and pipeline 3. Pipeline 3 provides the best throughput for a given application.

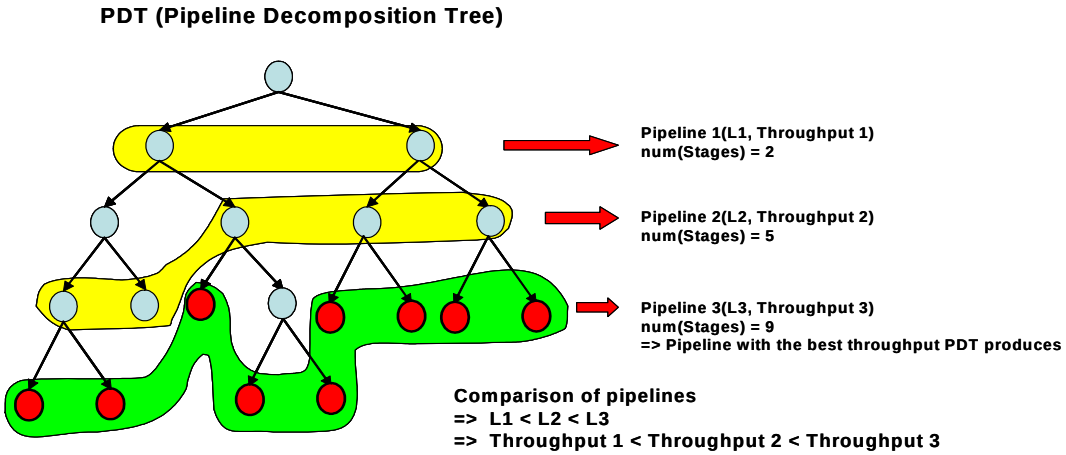


Figure 34. An example of making up pipelines with different trade-offs between latency and throughp from **PDT**

3.2.4.2 - Processor allocation, communication model and memory model

Assignment of processors and scheduling of tasks[35][39][82][85] inside each partition are performed for each partition produced by $PDT()$. This thesis suggests a heuristic processor allocation and scheduling algorithm named **HDEST** (Heterogeneous Data parallel Earliest Start Time)[89]. **HDEST** is a kind of a greedy algorithm which allocates tasks with earliest start time in a $RL(ReadyList)$ first. However, **HDEST** applies dynamic scheduling policies depending on existence/nonexistence of heterogeneous data parallelism of tasks in a RL. This approach enables us to reduce the latency of the associated stage in conjunction with PU(Processor Utilization) and heteroge-

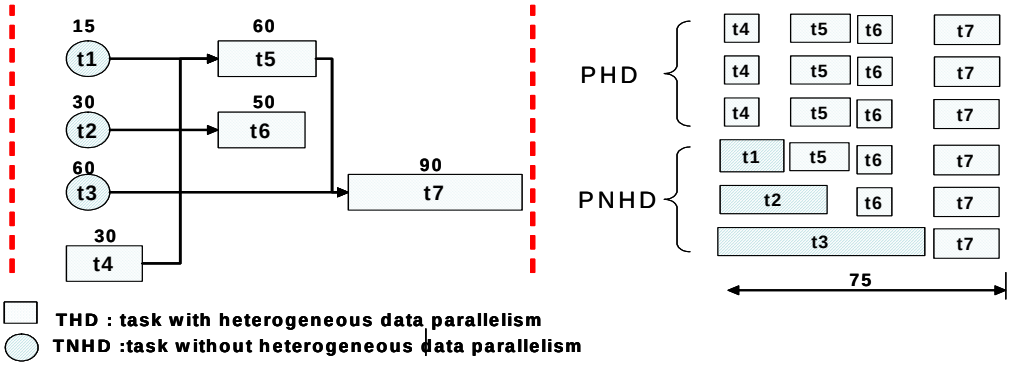
neous data parallelism. A processor allocation is restricted by memory usage status of both *on-chip* memory and *external* memory along with consideration of communication cost in connection with consideration of communication cost depending on architectures. After **HDEST**, each stage in a pipeline can be mapped to multiple processing cores, which may or may not span multiple DSP chips depending on memory usage of that stage and the number of processors allocated to that stage. Each DSP chip is assumed to have up to $P_{DSP-Chip_{max}}$ processor cores. Thus, each stage with more than $P_{DSP-Chip_{max}}$ processors in a pipeline can be mapped to multiple processing cores, which may or may not span multiple DSP chips in synthesis processors. Or multiple stages with less than $P_{DSP-Chip_{max}}$ processors can be merged to a single DSP chip only if they satisfy memory requirement of a single DSP chip (refer to Assumption 1 to Assumption 6.). $P_{DSP-Chip_{max}}$ is the maximum number of processors which can be synthesized in a single chip. In this section, setting of communication model, memory model and processor allocation based on resource constraints and high performance is introduced.

3.2.4.2.1 HDEST (Heterogeneous Data Parallelism Earliest Start Time)

This algorithm is an extension of EST(Earliest Start Time). While EST puts the same priority to tasks in a RL (Ready List), **HDEST** applies different priorities to tasks in a RL based on depth of a critical path of succeeding tasks. Thus, a task with the longest critical path of succeeding tasks has the highest priority in a RL. **HDEST** also looks up all tasks in a RL and classifies them based on existence or nonexistence of heterogeneous data parallelism into two groups. A task with heterogeneous data parallelism is named THD whereas a task without heterogeneous data parallelism is named $TNHD$.

Idle processors can be utilized by task duplication in conjunction with *THD*. **HDEST** tackles heterogeneous data parallelism for increasing *PU* (Processor Utilization). In case every task in a stage is *THD*, all processors in the associated stage of a pipeline can be fully utilized 100%. In general, task dependencies and *TNHD*s prevents ideal exploitation of *PU* (Processor Utilization). After **HDEST**, by checking *PU*s(processor utilization) of stages of a pipeline, stages with poor *PU* (Processor Utilization) are repartitioned by refining process. Refining process redistributes workloads of stages. Figure 35 shows one example about how **HDEST** fills available processors when *THD* and *TNHD* coexist in the *RL*. *PHD* is the number of processors allocated to *THD*s. *PNHD* is the number of processors to *THND*s. Because of data parallelism feature of *THD*, *THD* can utilize an idle time of processors allocated to *TNHD* by task duplication. If all tasks in a given stage are *THD*, all processors can be fully utilized by task duplication. In case *THD* and *THND* coexist in a given stage, an idle time of processors caused by task dependency can be filled with *THD*s. Figure 36 shows **HDEST** algorithm. **HDEST** exploits heterogeneous data parallelism with EST(Earliest Start Time). First, **HDEST** finds all tasks in *RL* and then classifies them depending on existence or nonexistence of heterogeneous data parallelism into two groups; *THD* and *TNHD*. Second, **HDEST** schedules *TNHD* by considering priority and communication cost. The rule to set up priorities of tasks in *RL* and the method to measure communication cost will be explained in detail in the following sections. Here, *TNHD* is scheduled before *THD*. It's because execution time of *THD* can be reduced by exploiting heterogeneous data parallelism and *THD* also fills idles processors by task duplication flexibly. When no tasks in *RL* are available, task duplication is considered for tasks of

THD which have already been scheduled, but are still running over an $idleInterval$ so that idle processors in an idle interval, $idleInterval$ can be filled with copied tasks of THD. Here, *EarliestEndTimeForSchedule* is the end time of the first available processor among processors being scheduled in a stage. *nextEarliestTimeForSchedule* is the end time of the second available processor among processors being scheduled in a stage. An idle interval, $idleInterval$ is an interval in which RL is empty due to task dependency.



PHD : processors used for scheduling THD s
$PNHD$: processors used for scheduling $TNHD$ s

Figure 35. An example of a schedule by **HDEST**

3.2.4.2.1.1 Setting up priorities of tasks in RL.

Even though tasks in RL (Ready List) are ready to activate, tasks have different priorities based on depth of the critical path of succeeding tasks. Thus, task with the longest critical path of succeeding tasks has the highest priority in RL. Equation 31 is to return the task with the highest priority. $A_{highest}$ is the task with the highest priority in RL. N_{RL} is the number of tasks in RL. $CriticalDepth_{successor}(A)$ is to return the critical path of succeeding tasks of task A.

$$A_{highest} = \underset{i=1}{\overset{N_{RL}}{Max}}[CriticalDepth_{successor}(A_i)] \tag{31}$$

* Descriptions for terminologies used in HDEST algorithm of figure 36
<Data structure description >
tasksInStage: tasks in the corresponding stage.
processorsInStage: processors in the corresponding stage.
EarliestEndTimeForSchedule: the end time of the first available processor among processors being scheduled in a stage.
nextEarliestTimeForSchedule: the end time of the second available processor among processors being scheduled in a stage.
readyTasks[]: tasks in a ready list satisfying task dependency at "***EarliestEndTimeForSchedule***" time. ***idleInterval***: an interval in which RL is empty due to task dependency.
THDTasksInIdleInterval[]: THD tasks scheduled over an idle interval.
processorsForTaskDuplication[]: processors available for task duplication in a stage.
< Function description >
pickUpTasksEST(): return tasks with EST(Earliest Start Time) in ***RL*** (Ready List).
setTaskPriotity(): set priority of each task based on a critical path of successors of the task in terms of the execution time.
returnHighestPriorityTask(): return the task with the highest priority.
returnProcessorMinimumCost(): return a processor which provides a minimum communication cost for a given task.
allocateTaskToProcessor(): allocate a given task to the processor returned by the **returnProcessorMinimumCost**().
updateReadyList(): update a *RL* (Ready List) with the remaining tasks in the corresponding stage.
returnNextEarliestTimeForSchedule(): return the next "***EarliestTimeForSchedule***".
pickUpTHDTasksInIdleInterval(): return THD tasks in an idle interval.
returnProcessorsForTaskDuplication(): return processors available for task duplication of a given THD task.
taskDuplication(): perform task duplication of a given THD task up to "***processorsForTaskDuplication***".

Figure 37 shows how tasks in *RL* have different priorities based on the length of the critical path of succeeding tasks. Task *A* has the highest priority due to the longest critical path and then task *B* is next and finally task *C* has the lowest priority.

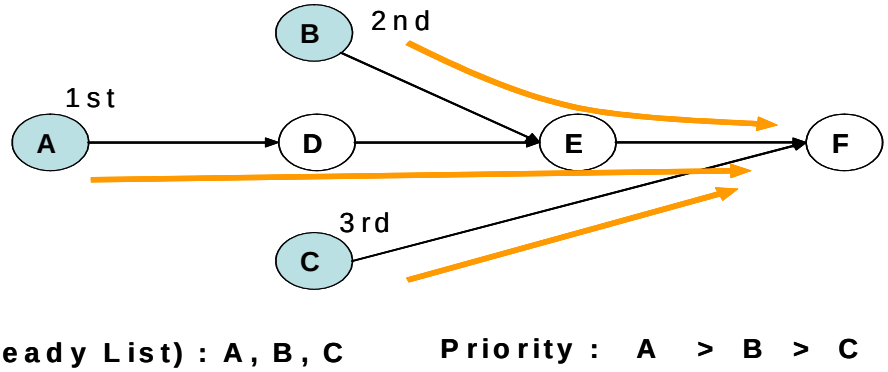


Figure 37. Priority setting of tasks in *RL (Ready List)* based on a critical path of succeeding tasks.

3.2.4.2.1.2 Communication cost in scheduling.

HDEST allocates tasks to the processor with the minimum communication cost by monitoring data dependency between tasks. We refer to Banerjee's model[7] for IPC cost estimation and then extend the communication model by adding bus contention problem caused by a shared memory architecture.[21][56][106][112]

$$Dij(s) = C0ij + C1ij \times s$$

```
HDEST(tasksInStage, processorInStage) {
  EarliestEndTimeForSchedule = updateReadyList(RL, tasksInStage);
  While(tasksForSchedule != empty) {
    if(RL != empty) {
      readyTasks[] = pickUpTasksEST(RL, EarliestEndTimeForSchedule);
      setTaskPriotity(readyTasks);
      //allocate TNHD tasks first in readyTasks
      for(i=0;i<readyTasks.TNHDtasks.length;i++) {
        taskTNHD = returnHighestPriorityTask(readyTasks.TNHDtasks);
        processor = returnProcessorMinimumCost(taskTNHD);
        allocateTaskToProcessor(processor, taskTNHD);
      }
      //then allocate THD tasks first in readyTasks
      for(i=0;i<readyTasks.THDtasks.length;i++) {
        taskTHD = returnHighestPriorityTask(readyTasks.THDtasks);
        processor = returnProcessorMinimumCost(taskTHD);
        allocateTaskToProcessor(processor, taskTHD);
      }
      tasksInStage.remove(readyTasks);
      EarliestEndTimeForSchedule = updateReadyList(RL, tasksInStage);
    }
    //exploit Heterogeneous Data Parallelism
    else {
      nextEarliestTimeForSchedule = returnNextEarliestTimeForSchedule(processorsInStage);
      idleInterval= nextEarliestEndTimeForSchedule - EarliestEndTimeForSchedule;
      THDTasksInIdleInterval[]
            = pickUpTHDTasksInIdleInterval(processorsInStage, idleInterval);
      while(true) {
        taskTHD = returnHighestPriorityTask(THDTasksInIdleInterval);
        THDTasksInIdleInterval.count--;
        processorsForTaskDuplication[] = returnProcessorsForTaskDuplication(idleInterval);
        taskDuplication(processorsForTaskDuplication, taskTHD);
        if(idleInterval == filled or THDTasksInIdleInterval.count == null)
          break;
      }
      EarliestEndTimeForSchedule = updateReadyList(RL, tasksInStage);
    }
  }
}
```

Figure 36. **HDEST** algorithm

Here, $Dij(s)$ is the communication delay from processor *i* to processor *j*. $C0ij$ is the fixed communication delay between processor *i* and processor *j*. $C1ij$ is the communication delay per a unit data size communication.

$$CC_{k,j} = IPC_{k,j}(s) + BC_k(p) \tag{32}$$

$$IPC_{k,j}(s) = \gamma \times \left(\sum_{i=1}^{\eta} D_{i,j}(s) \right) \tag{33}$$

$$BC_k(p) = (1-\gamma) \times bc_{delay}(p) \times s \tag{34}$$

In equation 34, $BC_k(p)$ is the delay from bus contention among p processors sharing memory region when task k runs on p. In equation 33, $IPC_{k,j}(s)$ is IPC cost when task k runs on processor j and η tasks running other processors except processor j send data to task k. $bc_{delay}(p)$ is delay per unit data size from bus contention among p processors sharing a memory region. Here, we select a linear model of p. $bc_{delay}(p) = p \times \lambda$, λ is a constant delay factor for bus contention, as operation patterns of tasks allocated to each processor using a shared memory architecture are very similar. $CC_{k,j}$ is the communication cost when task k runs on processor j either when η tasks running other processors except processor j send data to task k or when p processors share a memory region with processor j. $CC_{k,j}$ depends on the memory architecture chosen by a value of γ. If $\gamma = 1$, in equation 32, $BC_k(p)$ for bus contention from a shared memory architecture is ignored whereas if $\gamma = 0$, $IPC_{k,j}(s)$ of equation 32 for IPC cost is ignored. η is the number of tasks running on processors except processor j sending data to task k. The following inequality shows how memory architecture influences task activation time.

when $\gamma = 1$, $\max(\Psi_i + executeTime(t_i) + D_{i,j}) \le \Psi_j$, i for $\forall$

when $\gamma = 0$, $\max(\Psi_i + executeTime(t_i) + BC_i(p)) \le \Psi_j$, i for $\forall$

Ψ_i and Ψ_j are starting times of each task i and task j. $executeTime(t_i)$ is the execution time of i_{th} task, t_i. For task k, $HDEST$ allocates task k to a process producing minimum communication cost in terms of both IPC and bus contentions so that the overall communication cost for the schedule is minimized.

$$p_{\min CC} = \underset{j=1}{\overset{P_{stage}}{Min}}(CC_{k,j})$$

P_{minCC} is the processor providing the minimum communication cost for task k. P_{stage} is the number of processors in a stage. Figure 38 shows how $HDEST$ allocates task D when a separate memory architecture is applied. Task D is allocated to processor 2, as preceding tasks, task B and task C are allocated to processor 2 which leads to a lower communication cost than allocated to processor 1 due to data dependency with task B and task C.

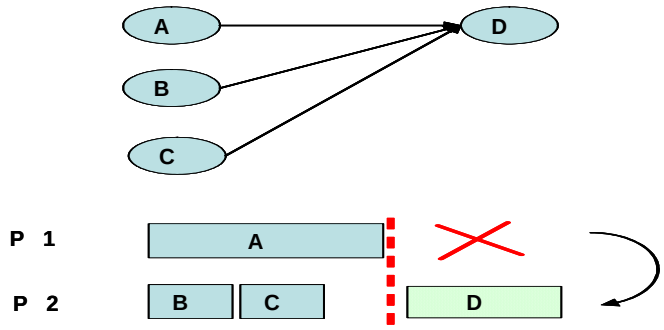


Figure 38. Example of consideration communication cost of *HDEST* in scheduling.

3.2.4.2.1.3 Examples of how $HDEST()$ operates.

Figure 39 shows how $executeTime(THD)$ and $executeTime(TNHD)$ influence scheduling of each stage. From case 1 to case 3, HDP (Heterogeneous Data Parallelism) is exploited with different configurations. These cases produce better latencies than a schedule without consideration of HDP (Heterogeneous Data Parallelism).

Here, case 3 produces the best result by reducing idle times of processors with *HDP*. In case 3, task 4 starts running on processor 2 in parallel with task 3, When task 3 finishes on processor 1, the remaining portion of task 4 is performed on both processor 1 and processor 2 in parallel by task duplication. Figure 39 shows that *THDs* have hidden potential parallelism which can be exploited further by **HDEST**. Usually, the execution time of a partition can't be smaller than the longest critical path within the partition. Thus, the longest critical path within the partition becomes a lower bound for latency. However, by exploiting *HDP* (Heterogeneous Data Parallelism), the execution time of a partition can be smaller than execution time of the longest critical path.

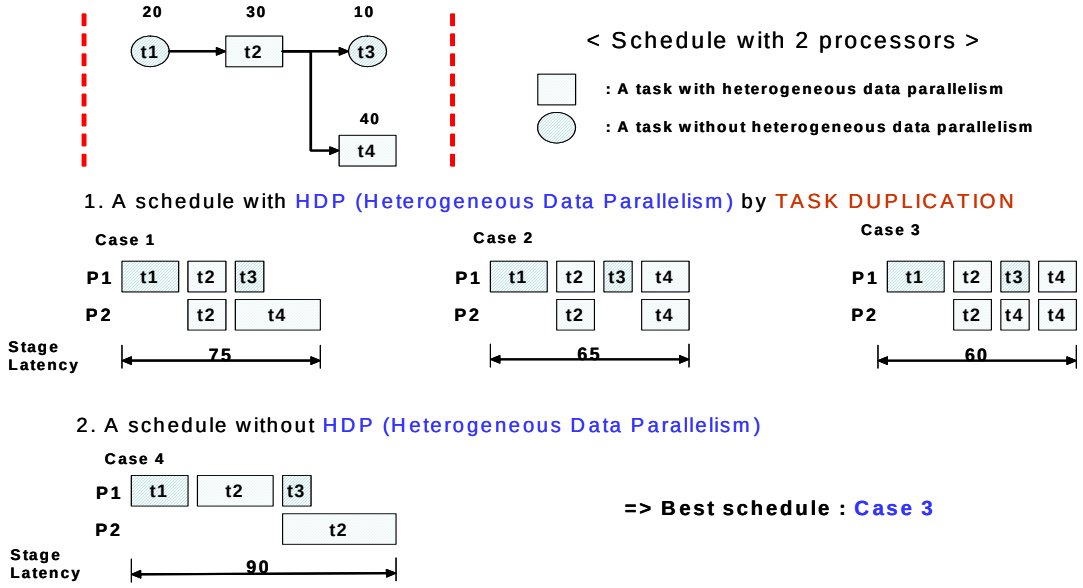


Figure 39. An example of how *THD*s reduce the execution time of a given stage

3.2.4.2.1.4 Verification of the number of processors allocated by *PU* (Processor Utilization).

The number of processors allocated to a partition is based on an *executeTime*(). However, while applying **HDEST**, some partitions result in poor *PU*s. In general, the more processors are allocated, the faster latency of the corresponding stage is. However,

stages with poor PU (Processor Utilization) produce overall poor throughput and bad latency since other candidate stages which could produce better latency along with extra processors lose potential chances. Thus, verification of the number of processors allocated to each partition is necessary in terms of PU (Processor Utilization). Figure 40 shows the relationship among latency, the number of processors and processor utilization. By seeing figure 40 a), the latency drops slowly after a specific point. Therefore, in the process of processor allocation, a scheduler needs to investigate how the latency varies depending on the number of processors. Here, a search region from P_{min} to P_{max} is chosen based on P_{given}.

$$\underset{p=P_{\min}+1}{\overset{P_{\max}+1}{Max}} \left[executeTime_{p}(\pi) - executeTime_{p-1}(\pi) \right],\ search\ region: P_{min} \le P_{given} \le P_{max} \tag{35}$$

Equation 35 returns the point where execution time of partition, π change slowly as the number of processors, p is changed. Thus, if $i_{sel}-1$ is equal to P_{given} and $PU(P_{given})$ is larger than or equal to $PUth$, it means that a given number of processors, P_{given} for a partition is proper. Here, $executeTime_{p}(\pi)$ is execution time of partition, π under the number of processors, p.

In both figure 41 a) and figure 41 b), three processors are given initially for scheduling. In figure 41 a), schedules of both two cases with one processor and two processors satisfy $PUth$, but the schedule with $P_{given} = 3$ is below $PUth$. In this case, "P_{given} -P_{sel}" is stored in a P_{stored}. Processors in a P_{stored} can be used to reduce the latency of a bottleneck partition later. In figure 41 b), the schedule with P_{given} satisfies $PUth$. Here, $P_{given}+1$ also satisfies $PUth$ by improving latency of a given partition. In this case, if processors are available in a P_{stored} and a given partition is a bottleneck partition in a pipeline, extra processors can be applied for a better schedule in addition to P_{given}.

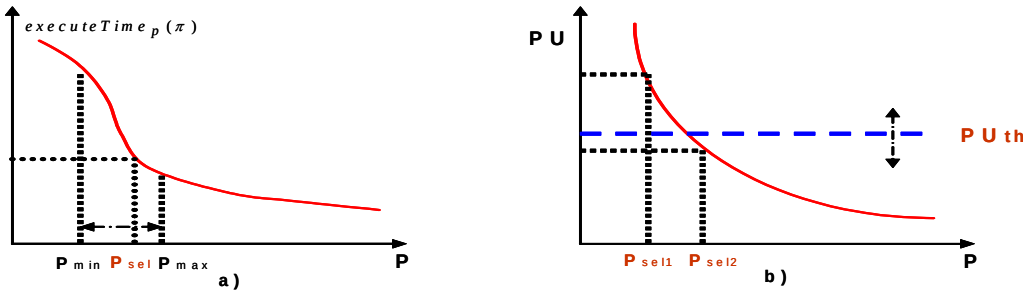
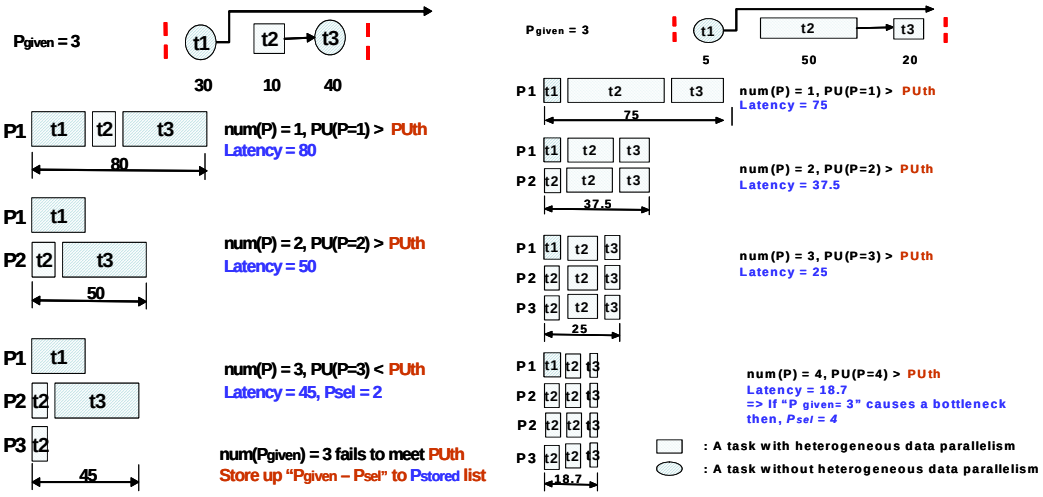


Figure 40. Relationship among execution time of partition, π, the number of processors, *p* and processor utilization, *PU*.



P_{stored} : the number of processors stored up which can be used for a bottleneck partition.
$num(P)$: the number of processors.

Figure 41. Examples of verification of P_{given}

3.2.4.2.2 Memory model

This thesis considers two different memory architectures; a shared memory architecture vs a separate memory architecture. For *on-chip*, only a shared memory is considered since multiple processing cores in a single DSP chip lead to the limited chip size. For *external* memory, both a shared and a separate memory architecture are exclu-

sively considered depending on task dependency of an application graph(refer to Assumption 1 to Assumption 6.). Intuitively, tasks immediately following a *branch point* is assumed to be integrated to the same stage for maximizing the effect of a shared memory architecture.

Since *on-chip* memory is shared by processor cores, monitoring of runtime memory usage within a *on-chip* memory usage is necessary for appropriately allocating tasks with different window sizes to the associated processor core. $MO_{th,\,single}$ is *on-chip* memory threshold for a single DSP chip and $P_{DSP-Chip_{max}}$ is the maximum number of processor cores to be embedded within a single DSP chip. Thus, in equation 36, MO^{n}_{th}, the total *on-chip* memory threshold for stage, n with more processors than $P_{DSP-Chip_{max}}$ is readjusted to $P_{stage}/P_{DSP-Chip_{max}} \times MO_{th,\,single}$. Here, P_{stage} is the number of processors allocated to a pipeline stage.

Runtime usage of *external* memory linked to each processor also limits allocation of tasks to available processors. Figure 42 a) shows the case that $task4$ can't run on idle $P3$ due to the shortage of available *on-chip* memory. Figure 42 b) shows that the shortage of available memory for P_1 prevents $task6$ running on P_1.

$$MO^{n}_{th} = P_{stage}/P_{DSP-Chip_{max}} \times MO_{th,\,single} \tag{36}$$

$$\sum_{i=1}^{\mu} \sum_{k=1}^{I_{t_i}} w_{k_{t_i}} \le MO^{n}_{th} \tag{37}$$

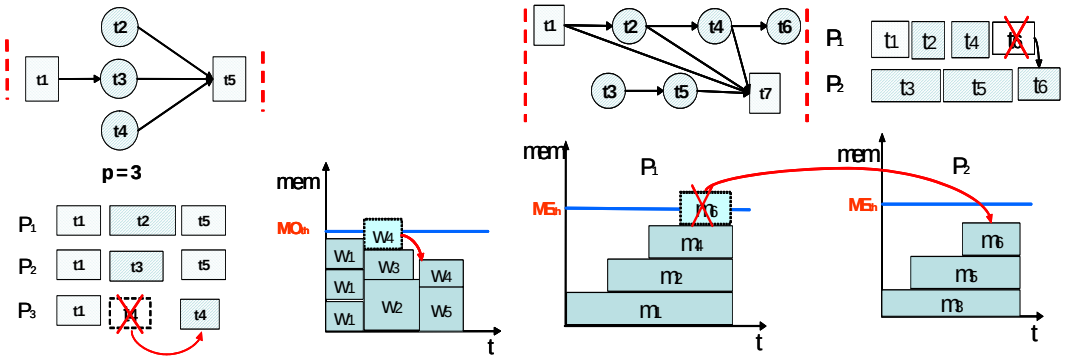
$$\sum_{i=1}^{\mu_j} \sum_{k=1}^{I_{t_i}} b_{k_{t_i}} \le ME^{n}_{th}(j) \tag{38}$$

$ME^{n}_{th}(j)$ is *external* memory threshold of a processor j within stage n. μ is the number of tasks currently running on the associated stage, n. t_i is a task running on the associated stage, n. I_{t_i} is the number of input ports of task, t_i. $w_{k_{t_i}}$ is the window of k_{th} input

port of task, t_i. μ_j is the number of tasks currently running on processor j. $b_{k_{t_i}}$ is the buffer of k_{th} input port of task, t_i. Both *on-chip* memory usage and *external* memory usage of each stage within a pipeline are examined by equation 37 and 38 during scheduling.

- **Assumption 7**:

For both *on-chip* and *external* memory, a memory region allocated by a sender-task is held up until all receiver-tasks are activated. Here, a sender-task is a task sending data to a receiver-task which consumes the data.



a) on-*chip* memory b) *external* memory

Figure 42. Usage of *on-chip* and *external* memory

3.2.4.3 Iterative change of parameters

Parameters in **PDT scheduling** influence an output schedule. Appropriate values for parameters can be intuitively predicted due to the deterministic feature of our algorithm. However, calibrated values of these parameters may slightly vary depending on a given application[5]. The initial values of parameters are obtained from arbitrary generated application graphs. Starting from these initial values, **PDT scheduling** algorithm is applied in an iterative way by changing values of those parameters until the best schedule under given constraints is obtained.

Figure 43 shows how **PDT scheduling** algorithm is applied along with varying values of parameters in an iterative search way.

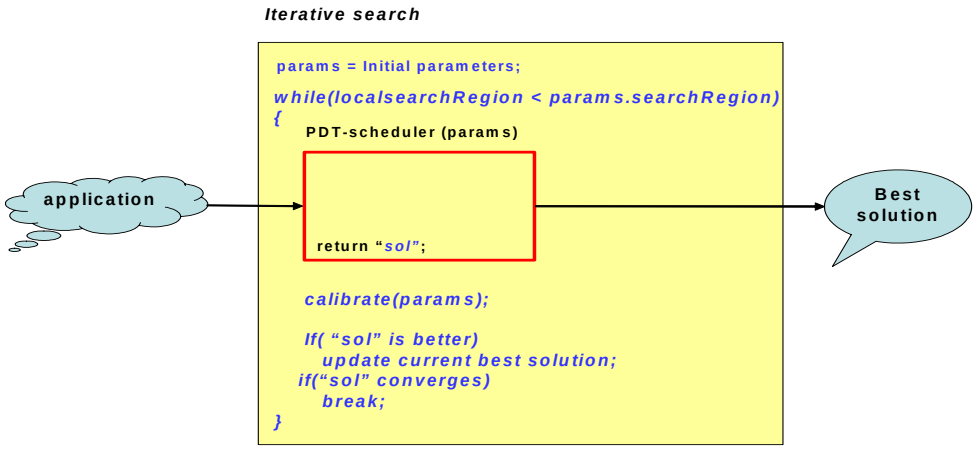


Figure 43. Adaptation of PDT scheduling algorithm with varying parameters to an iterative search approach

3.2.5 Application examples

This thesis mainly focuses on applications consisting of both *THD* tasks and *TNHD* tasks. Since *THD* tasks operate on the basis of a window, they provide important information at the compile stage for the resource management and the scheduling. Examples of those features are image processing applications. We selected a complex image processing module based on morphological operations, Laplacian image pyramid and Multi-resolution spline. Figure 44, 45 and 46 show graphs of application examples.

Figure 44 shows an application integrating major morphological image processing modules. This application produces the outputs of several applications of morphological operation modules (Top-hat, Gradient, Laplacian and Smoothing). Table 9 shows functional descriptions of each blocks in figure 44.

Figure 45 and 46 show an application performing Laplacian pyramid and Gaussian Pyramid. An image in level $i+1$ of Gaussian Pyramid is obtained by convolution of

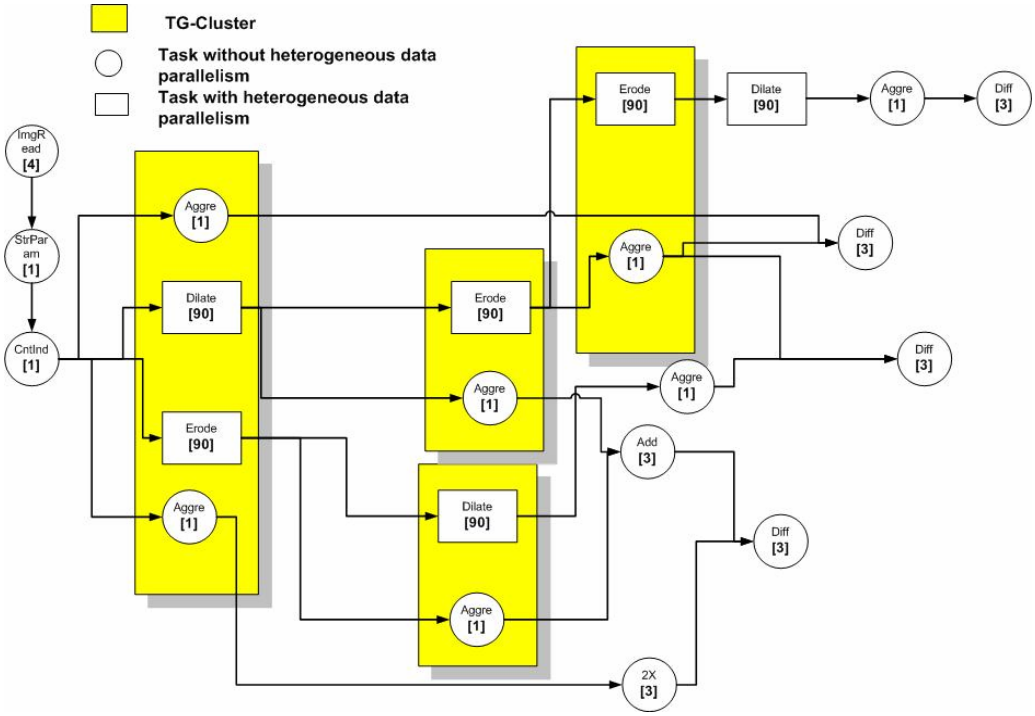
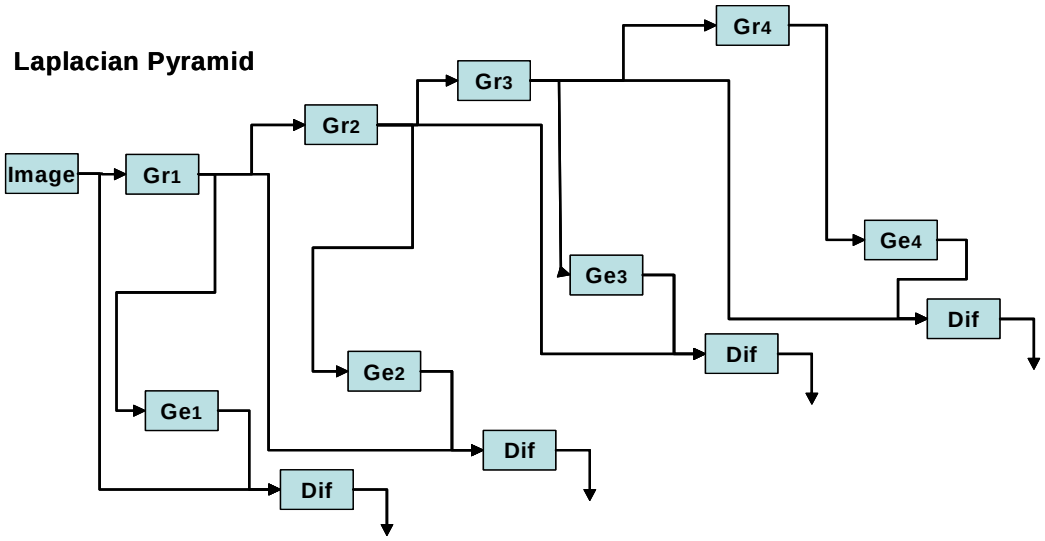


Figure 44. A graph of a complex module of morphological operations



Gri : reduction of i h level image by Gaussian Pyramid
Gei : expansion of i h level image by Gaussian Pyramid
Dif : produce difference between two inputs

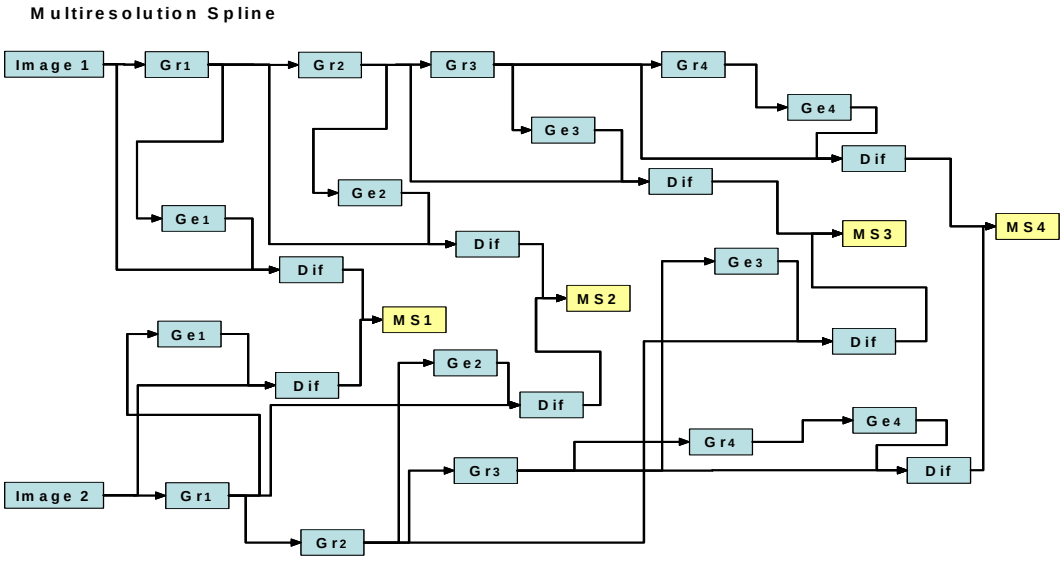
Figure 45. Laplacian Pyramid as an application example.

an image in level i with Gaussian filter and sub-sampling. An image in each level of Laplacian Pyramid is obtained by differentiating the original image in each level and a

Table 9. Function description of each block of figure 44

	Function description of each task
Image Reader	Provide the original image to an application
StrParam(Stream Parameterizer)	Convert an image frame and frame information accompanied into parameters for the body sub-system of BLDF
CntIn-dex(CountIndex)	Produces indices to which each task of the associated body subsystem refer to access image frame
Dilate	Perform dilation operation
Erode	Perform erosion operation
Aggre(Aggregate)	Aggregate triggers each task produces to check if each task operating in parallel is finished
Diff(Differentiate)	Produce the difference from two input frames
2X	Produce an output by multiplying an input with 2

reconstructed image from an image in the next level of Gaussian Pyramid. Table 10



MS_i : produce a multi resolution spline of i th level images

Figure 46. Multi resolution Spine as an application example.

shows functional descriptions of each block of figure 45.

Figure 46 shows an application performing Multi-resolution spline. Multi-resolution spline(MS) produces a merged image from two different images by Laplacian Pyra-

$G_0, G_1, ... G_i$ = the levels of a Gaussian Pyramid
$L_i = G_i - G_i'$
L_i : an image in a level i of Laplacian Pyramid.
G_i : an image in a level i of Gaussian Pyramid.
G_i' : a reconstructed image from an image G_{i+1} by expanding operation.

Table 10. Function description of each block of figure 45

	Function description of each task
Image	Provide the original image
G_r	Produce an image reduced by convolution an original image with Gaussian Filter and Sub-Sampling
G_e	Produce an image by convolution and Zero-padding
Dif	Produce the difference from two input frames

mid. MS creates a new Laplacian Pyramid generated by combining two different Laplacian Pyramids.

$$L_{img3}(i,j) \begin{cases} L_{img1}(i,j) & \textit{if } i < width/2 \\ (L_{img1}(i,j) + L_{img2}(i,j))/2 & \textit{if } i = width/2 \\ L_{img2}(i,j) & \textit{if } i > width/2 \end{cases}$$

$L_{img3}(i,j)$: a Laplacian Pyramid of Multi-resolution spline.

$L_{img1}(i,j)$: a Laplacian Pyramid of img_1 .

3.2.6 Experimental results

This thesis uses TMS320C64x DSP simulator of Texas Instruments' code composer to measure estimated execution time of each task within a dataflow graph by assuming each task running on a single processor. We used complex morphological application, Laplacian Pyramid, Multi-resolution spline and MPEG2 encoder for scheduling over multi processors with the suggested technique. The application was scheduled under different constraints and architectures. We assumed each DSP chip can integrate up to

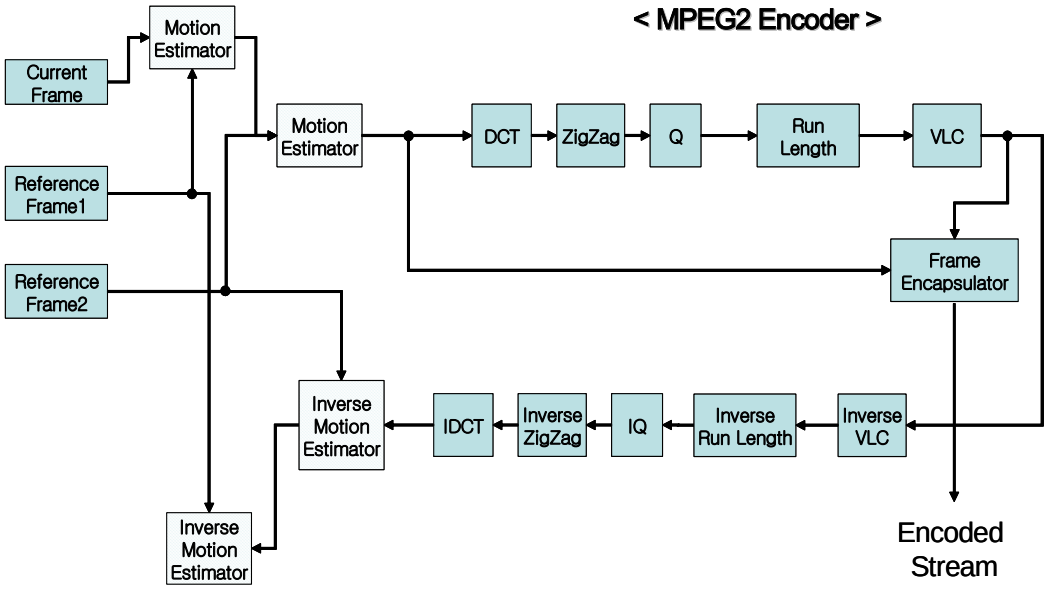
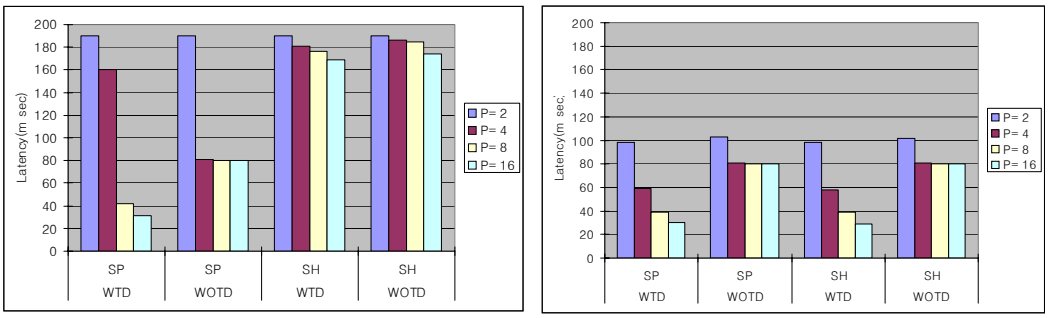


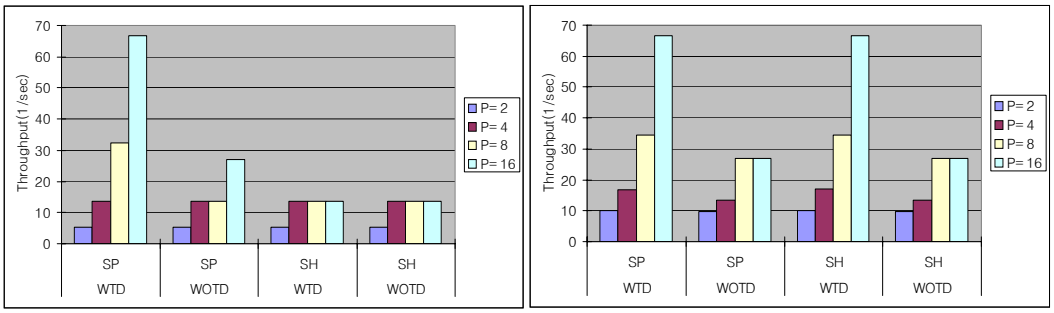
Figure 47. MPEG2 Encoder

4 processor cores. Each DSP chip has on-chip memory and external memory. Each stage of a pipeline consist of one or more DSP chips with different number of processors cores depending on data dependency. We assumed that external memory for each processor core within DSP chip can be configured in either a separate memory architecture (SP) or a shared memory architecture (SH) whereas only a shared memory was considered for on-chip memory due to the size issue of DSP chip. We applied 10% reduction for on-chip and 50% memory reduction for an external memory compared to peak memory usage of each processor core. We observed the effect of memory constraints on performance in each architecture configuration. We compared the suggested technique with EST(Earliest Start Time) algorithm. We performed the experimentation with 2, 4, 8 and 16 numbers of processors. Figure 48 through Figure 51 show the comparison of latency and throughput for Multi-resolution Spline, Laplacian pyramid, Image complex and MPEG2 encoder benchmark applications under either memory constraint or unconstraint environment with different numbers of pro-



a) Latency (Constrained)

b) Latency (Unconstrained)



c) Throughput (Constrained)

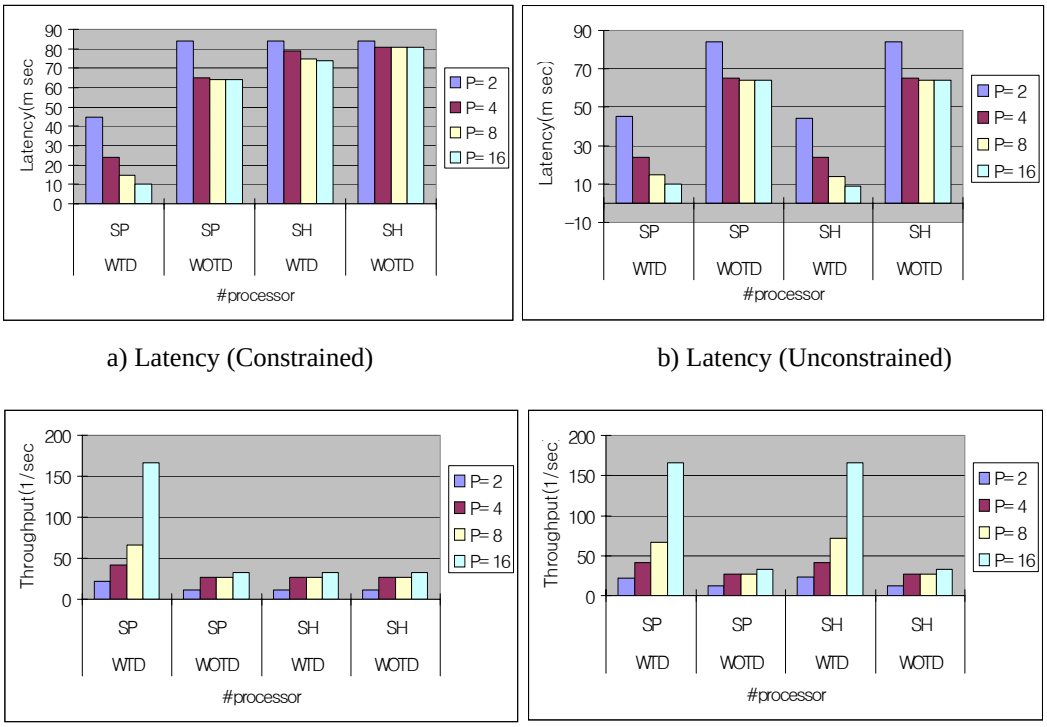
d) Throughput (Unconstrained)

SP: separate memory.
SH: shared memory.
WTD: With Heterogeneous data parallelism.
WOTD: Without Heterogeneous data parallelism.
< Constrained >
[SH: On-Chip 3.6KB, EX-MEM: The number of stages*64KB].
[SP: On-Chip 3.6KB, EX-MEM: The number of processors*64KB]
< Unconstrained >
[SH: On-Chip 4KB, EX-MEM: The number of stages*2*64KB].
[SP: On-Chip 4KB, EX-MEM: The number of processors*2*64KB]

Figure 48. Latency and throughput comparison (Multi-Spline)

cessors ($Processors = 2, 4, 8$ and 16). Figure 48 shows that scheduling results under memory constraints lead to each 80% (WTD) and 51%(WOTD) performance degrade in terms of throughput under a shared memory architecture with 16 processors. Shared memory architecture can save up to 37.5% memory usage under an unconstrained memory and 16 processors environment while providing 25% faster latency than separate memory architecture. Heterogeneous data parallelism of the suggested technique provides 2.46 times better throughput and 62.5% reduced latency than scheduling of without heterogeneous data parallelism ($WOTD$) with 16 processors. In figure 50 and

figure 51 a), latencies for *WTD* and *WOTD* under memory constrained scheduling with a shared memory architecture are not changed along with increased number of processors. This shows that memory constraint for the application is close to a low boundary of memory usage, which prevents heterogeneous data parallelism or more available processors for scheduling improving performance. In figure 51 b), latencies under both a separate memory architecture and a shared memory architecture without memory constraints are not improved even though more processors are given for scheduling. This result shows that the critical data dependency prevents the scheduler taking advantage of idle processors. In this case, heterogeneous data parallelism by HDEST reduces 84.6% of latency of *WOTD* configuration (P=16). This means heterogeneous data parallelism is not sensitive to data dependency. Further exploitation of available idle processors by heterogeneous data parallelism can be possible.,



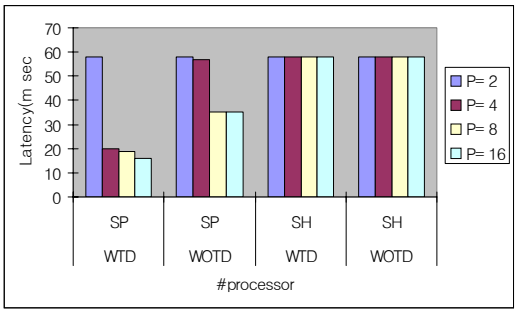
a) Latency (Constrained)

b) Latency (Unconstrained)

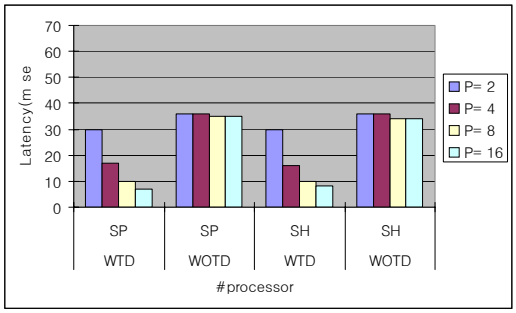
c) Throughput (Constrained)

d) Throughput (Unconstrained)

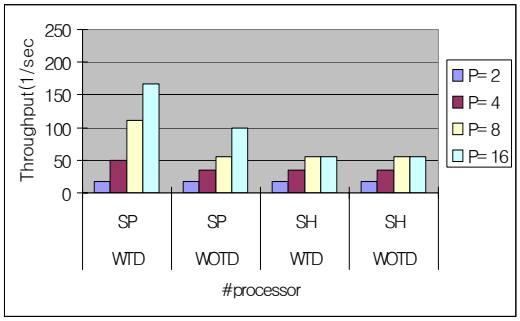
Figure 49. Latency and throughput comparison (Laplacian)



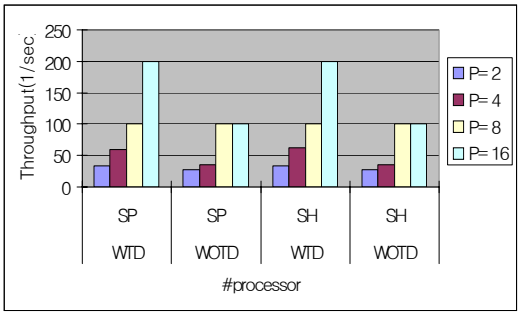
a) latency (Constrained)



b) latency (Unconstrained)

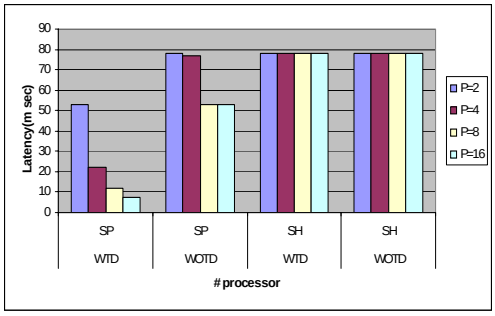


c) Throughput (Constrained)

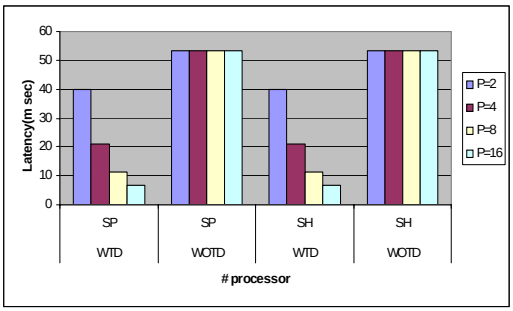


d) Throughput (Unconstrained)

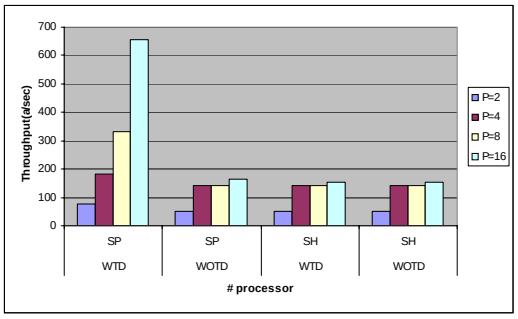
Figure 50. Latency and throughput comparison (Image Complex)



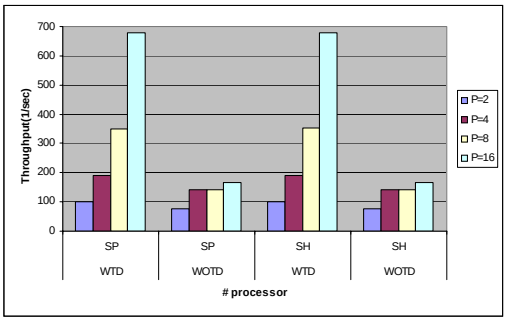
a) latency (Constrained)



b) latency (Unconstrained)



c) Throughput (Constrained)



d) Throughput (Unconstrained)

Figure 51. Latency and throughput comparison(MPEG2 Encoder)

Figure 52 shows comparison between EST and PDT technique for experiments with Multi-Spline, Laplacian, Image Complex and MPEG2 applications. The suggested technique provides 63.8% reduced latency and 4.94 times fast throughput compared to EST approach under an unconstrained memory configuration for Multi-Spline application. The graph shows that tight memory constraint makes the results between the suggested technique and EST less obvious compared to an unconstrained memory environment. Especially, in latency of $A3$ (Image Complex) and $A4$ (MPEG2 encoder) under memory constraints(C configuration), a relatively tight memory constraint compared to the minimum memory usage for scheduling prevents PDT exploiting idle processors with heterogeneous data parallelism. Figure 53 shows that pipelines generated by PDT have different latencies and throughputs. $P1$ provides the lowest latency(31 msec) whereas $P3$ has the best throughput (66.7 frames per sec). In figure 53, $P2$ (Latency: 52m sec, Throughput: 43.5 frames per sec) is chosen under given latency (lower than 60m sec) and throughput (over 40 frames per sec) requirement boundary.

3.2.7 Conclusion

Effective, coarse-grained (task-level) pipelined scheduling of an application over

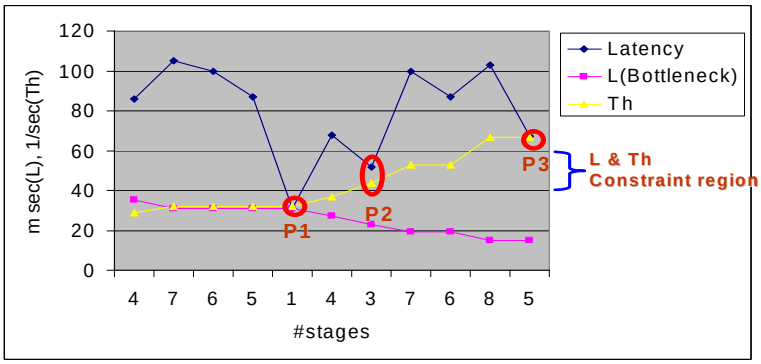
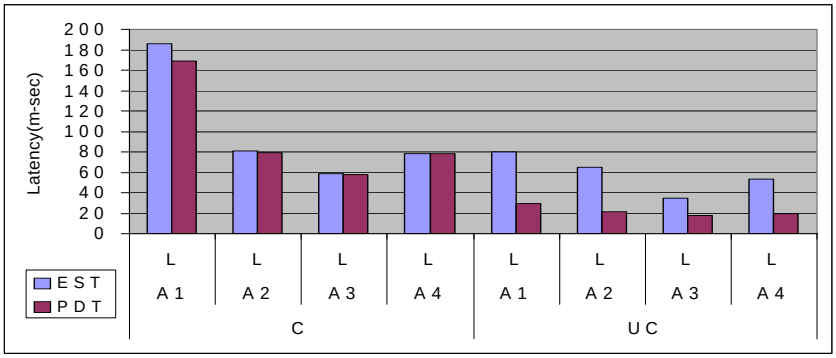
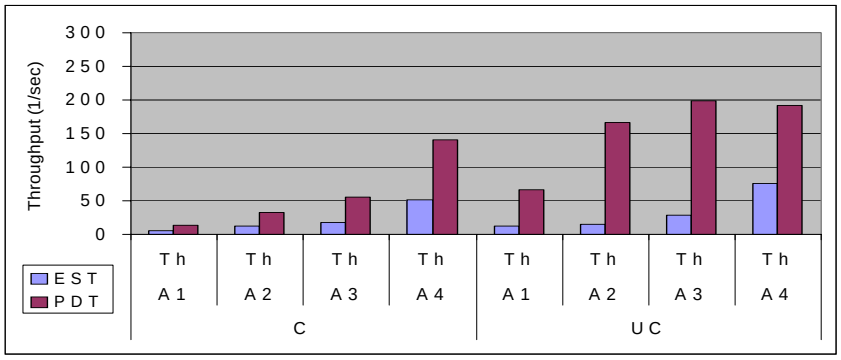


Figure 53. Latency vs Throughput trade-off (Multi-resolution Spline, P=16, Unconstrained, Shared memory)



a) Latency comparison (EST vs PDT)



b) Throughput comparison (EST vs PDT)

A1:Multi-Spline, A2:Laplacian, A3:ImageComplex, A4:MPEG2
C: Constrained memory
UC: Unconstrained memory

Figure 52. EST vs PDT comparison

multi processors generally provides increased throughput. However, pipelined scheduling can significantly increase latency. Furthermore, pipelined scheduling of image processing applications requires careful and flexible consideration of data- and task-level parallelism. This paper provides a new approach to generating coarse-grained pipelines for image processing applications in a manner that simultaneously considers latency/throughput trade-offs; memory and performance constraints; task-level parallelism; and homogeneous and heterogeneous modes of data parallelism. The approach is based on a novel data structure called the *pipeline decomposition tree* (PDT).

The PDT is useful for efficiently representing and exploring various sets of pipe-

lining configurations that provide different trade-offs between latency and throughput. After pipelined schedules are generated through the PDT analysis process, a new technique called *heterogeneous data parallelism earliest start time* (HDEST) maps application tasks onto pipeline stages while considering memory and performance constraints. In the HDEST mapping process, heterogeneous data parallelism is carefully applied to improve both throughput and latency.

Our experimental results on various applications demonstrate the utility of the PDT data structure and HDEST mapping technique for embedded multiprocessor implementation of image processing applications. The applications in our experiments involved image processing because the emphasis in PDT on data parallelism considerations makes the technique especially well-suited for image processing. However, concepts related to the PDT, PDT scheduling, and HDEST can be applied to other domains of signal processing, including speech processing, high fidelity audio processing, and digital communications. Exploration and specialization of our techniques for applications in such domains provide important directions for further study. For example, application to wireless communications will require special attention to integrating power optimization considerations into the PDT analysis framework.

Chapter 4 : Communication optimization of DSP applications implementation

4.1 Introduction

In the previous chapter, we described a novel scheduling technique for mapping a dataflow graph over multiprocessor environment named PDT scheduling. PDT scheduling tackled data parallelism and task parallelism together to improve latency and throughput at the same time. The technique tackled various system constraints such as a memory architecture, system performance and communication cost etc. during scheduling. A preliminary summary of part of this chapter is published in [50][51].

In this chapter, we describe two post-optimization techniques as a communication optimization technique. After modeling and scheduling, more detailed communication optimization technique can be considered, which are sometimes application dependent. Communication optimization technique can be divided into two parts; hardware and software communication optimization. In hardware communication optimization, we perform a FIFO buffer optimization of a dataflow graph in terms of a trade-off of performance and cost among FIFO architectures. In software communication optimization, we perform the case study with a sensor network application. In the sensor network domain, we suggest an application cutting technique for distributing a single dataflow graph over several processing nodes to minimize the overall energy consumption of a sensor network system in consideration of performance change.

4.2 Modeling and optimization of buffering trade-off

4.2.1 Abstract

As modern image and video processing applications handle increasingly higher image resolutions, the buffering requirements between communicating functional modules increase correspondingly. The performance and cost of these applications can change dramatically depending on the implementation methods for FIFO buffers and the data delivery methods between modules. This thesis introduces a new FIFO hardware mapping algorithm based on pointer-based token delivery from dataflow semantics for image and video processing applications. This approach significantly improves the performance of dataflow based implementation of image and video processing systems, and allows effective prediction of changes in performance and buffer memory requirements associated with changes in image resolution. Our pointer-based token delivery method allows indirect token delivery between actors by pointers in conjunction with use of a shared memory. Each pointer references a data block stored in the shared memory. In pointer-based token delivery, a buffer can be configured to be implemented as the combination of a small, fast FIFO and a larger, relatively cheap shared memory while providing an attractive trade-off between performance and hardware cost. We present the complete semantics of our pointer-based modeling method, systematic techniques for mapping representations using these semantics into efficient implementations, and experimental results that demonstrate the performance of the proposed pointer-based techniques.

4.2.2 Related Work

Dataflow [63] is widely used for designing DSP applications. Various research efforts on mapping dataflow graphs into hardware implementations have been undertaken. For example, the approach of [30] exploits loop parallelism to map nested loop kernels onto a coarse-grained reconfigurable architecture. The approach of [33,34] uses direct mapping of each dataflow graph component (actor) onto a corresponding hardware resource. The approach of [38] uses shared resources and looped schedules. The approach of [40] analyzes a given set of applications to extract commonalities across nodes in different applications and uses them to bias the mapping of nodes in the partitioning process. For FPGA implementation, the approach of [92] provides a rapid system prototyping method through a component architecture and an associated set of software tools. The approach of [103] provides a pipelined asynchronous circuit mapping method. For pointer synthesis, the approach of [87] encodes pointer values and generates circuits that can dynamically access different locations with each pointer reference. The approach of [105] points out that pointers can reference indices to RAM, registers or even wires in a hardware mapping. The approach of [8] applies an external memory for mapping FIFO buffers and implements real-time image convolution on an FPGA. The approach of [72] implements image processing applications on FPGAs and points out that such implementations lead to a large on-chip FIFO buffers that prevent flexible usage of FPGAs for image processing applications. The approach of [104] presents an elaborate technique for mapping global, static arrays to distributed communication structures while classifying four types of inter-process communication patterns. The approach of [110] studies memory optimization for embedded software,

particularly the performance of cache-based systems. The approach of [107] presents a novel technique for background memory allocation in multi-dimensional signal processing applications based on dataflow analysis.

The efforts described above make useful contributions to mapping application representations at various levels of abstraction into hardware implementations. However, the simultaneous analysis of both performance and cost implications when mapping image processing applications, which involve especially large volumes of data token delivery, has not been thoroughly investigated in previous work.

This thesis helps to bridge this gap by studying, in the context of mapping dataflow graphs into hardware, the relationship between token delivery methods (indirect, pointer-based token delivery vs. direct-reference, raw token delivery) and FIFO architecture. This thesis exploits pointer-based token delivery to reduce on-chip FIFO sizes, and also provides a range of efficient trade-offs between performance (latency and throughput) and FPGA resource cost through a novel FIFO mapping algorithm. This thesis also shows how overall performance and cost vary in relation to the selected sub-frame size at which block processing is carried out. Finally, this thesis provides a new mapping algorithm for dataflow representations of image processing applications to reduce overall FPGA resource costs without significant performance loss.

4.2.3 FIFO hardware mapping for dataflow graphs

4.2.3.1 Modeling and architecture

In this work, an application is modeled under synchronous dataflow (SDF) [63] semantics and then mapped to an FPGA device. Each vertex (actor) within the given

SDF graph is mapped to a module within the target FPGA. Edges are converted into either pure on-chip *raw data FIFO* architectures or composite FIFO architectures that we call *pointer based FIFOs*. Figure 54 shows a comparison of raw data FIFOs and pointer based FIFOs. In Figure 54b), the raw data FIFO is embedded inside the FPGA chip and holds direct raw data tokens. Here, by *token* we mean the unit of data transfer along an edge in the dataflow graph. The pointer based FIFO involves both an on-chip FIFO, which holds references to token blocks rather than the tokens themselves, and an external (off-chip) RAM-based memory, which may be shared across multiple pointer based FIFOs as well as other storage constructs. In Figure 54a), raw data tokens are located in the external memory, while a relatively small on-chip FIFO buffer holds pointers that provide a stream of indices into the external memory.

The FIFO architectures (raw data vs. pointer based) and FIFO sizes can be configured strategically based on optimization during the synthesis process. This thesis formulates and investigates this optimization problem, and studies various important factors that should be taken into account when configuring dataflow buffers for hardware mapping. This is an important problem because the configurations of the FIFOs in a dataflow graph implementation have significant impact on the overall performance and hardware resource costs. This thesis presents an effective heuristic FIFO mapping algorithm for mapping SDF graphs efficiently into hardware.

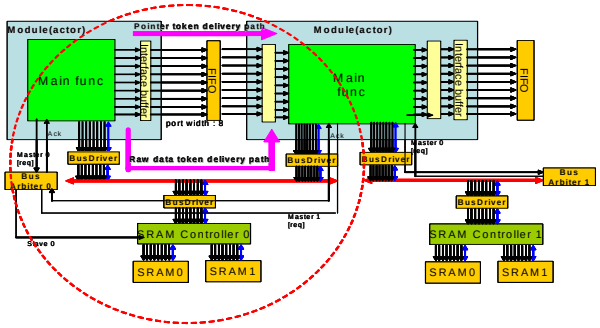
4.2.3.2 Performance and cost impact of token delivery methods

As implied above, we consider two alternative token delivery methods between dataflow actors, pointer based token delivery (indirect token delivery) and raw token delivery (direct token delivery).

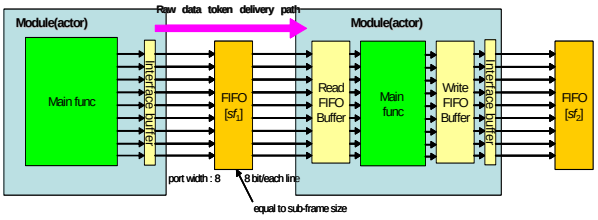
Raw token delivery is the conventional form of data delivery for dataflow graph implementation. Raw token delivery directly transfers data tokens across the FIFOs that connect adjacent pairs of actors in the dataflow graph. Therefore, for applications, such as those found in the image processing domain, that require large volumes of token transfer, very high resource requirements often result from extensive use of raw token delivery. On the other hand, since there is no indirection overhead or external memory access involved, raw token delivery improves performance through faster dataflow communication.

The limited quantities of gates available on FPGAs makes it challenging to implement image processing applications efficiently on these devices. Although FPGA resource density continues to increase from Moore's law, the complexity and resolution requirements of state-of-the-art image processing applications is also increasing at a significant pace.

Pointer based token delivery allows for more efficient use of limited FPGA



a) Pointer based FIFO architecture



b) Raw token FIFO architecture

Figure 54. Comparison of FIFO architectures

resources by dividing inter-actor communication functionality into two parts. These parts consist of a relatively small set of pointers, and blocks of token data that the pointers reference. The pointers are kept in fast but expensive on-chip FIFOs, while the raw token data is located in slow but cost-effective external RAM. Dataflow graph actors send data to other actors by transferring pointers through the on-chip FIFOs. Actors at the receiving end use the transferred pointers to access external memory and retrieve the actual raw tokens. Pointer based token delivery significantly reduces FPGA resource requirements at the expense of some degradation in latency and throughput.

Equation (39) below describes relationships between pointer based token delivery and raw token delivery in terms of performance (execution time) and cost (the required number of gates). Here, g_{F} denotes the number of gates required for the FIFO F; t_{F} denotes the execution time for data token delivery through FIFO F; α_g represents a coefficient for converting the number of gates between two delivery methods; and α_t represents a similar conversion coefficient for execution time. The values of α_g and α_t depend on the sub-frame size sf.

$$g_{\mathrm{raw}} \gg g_{\mathrm{ptr}},\ g_{\mathrm{raw}} = \alpha_g g_{\mathrm{ptr}},\ t_{\mathrm{Fraw}} \ll t_{\mathrm{Fptr}},\ t_{\mathrm{Fptr}} = \alpha_t t_{\mathrm{Fraw}}. \tag{39}$$

The following equation describes the effects of raw token delivery and pointer

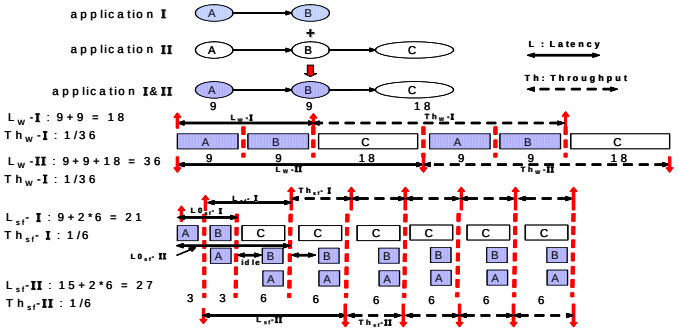


Figure 55. Effect of sub-frame division on latency and throughput.

based token delivery on latency and throughput:

$$\tau = \sum_{i=1}^{n} [t_A(a_i) + t_O(a_i) + \beta_i^{in} t_{Fraw}(a_i) + (1 - \beta_i^{in}) t_{Fptr}(a_i) + \beta_i^{o} t_{Fraw}(a_i) + (1 - \beta_i^{o}) t_{Fptr}(a_i)] \quad . \tag{40}$$

Here, a critical path of the given application must be extracted beforehand for the analysis, and n is the number of actors on this critical path. The symbols β_i^{in} and β_i^{o} are related, respectively, to the input port and output port of a_i in the critical path (i.e., with respect to the edges in the critical path that are incident to a_i). In (40), $\beta_i^{in} = 1$ ($\beta_i^{o} = 1$) if the associated communication is mapped to a raw FIFO architecture, and conversely, $\beta_i^{in} = 0$ ($\beta_i^{o} = 0$) if it is mapped to a pointer based FIFO. The other symbols in (40) are defined below in Section 4.2.3.3.

4.2.3.3 Effect of sub-frame size on performance and cost

Sub-frame division reduces FIFO size along with pointer based token delivery since the whole data frame can be processed in smaller units. However, depending on the application, there may be strict constraints on the sub-frame size (sf) that can be employed. Many image processing subsystems have minimum window (or block) sizes for their basic units of operation. Some globally-oriented operations, such as contouring, require the whole image frame as their basic units of input.

Sub-frame division influences both performance and cost. To understand this better, we can decompose the execution time of an actor a_i into three different parts, $t_A(a_i)$, $t_O(a_i)$ and $t_F(a_i)$. Here, $t_A(a_i)$ is the execution time for activation of a_i; $t_O(a_i)$ is the execution time for the main functional logic operation of a_i; and $t_F(a_i)$ is the exe-

cution time required for token delivery of $a_i.t_A(a_i)$ is proportional to the number of sub-frame divisions (δ), whereas the "total summation" of $t_O(a_i)$ and $t_F(a_i)$ are the same regardless of the sub-frame division format. Usually, $t_A(a_i)$ is relatively small compared to $t_O(a_i)$ and $t_F(a_i)$.

Equation 41 shows the relationship among the three different components of execution for an actor, taking into account sub-frame division.

$$L_w(a_i) = t_{A_w}(a_i) + t_{O_w}(a_i) + t_{F_w}(a_i),\ L_{sf}(a_i) = \delta t_{A_{sf}}(a_i) + \delta[t_{O_{sf}}(a_i) + t_{F_{sf}}(a_i)],$$
$$L_w(a_i) \le L(a_i) \le L_{sf}(a_i),\ t_{O_w}(a_i) + t_{F_w}(a_i) \cong \delta[t_{O_{sf}}(a_i) + t_{F_{sf}}(a_i)],\ t_{A_w}(a_i) = t_{A_{sf}}(a_i),$$
$$t_{A_w}(a_i) < \delta t_{A_{sf}}(a_i) \ll t_{O_w}(a_i), t_{F_w}(a_i). \quad (41)$$

Here, w represents the size of the entire image frame; sf is the sub-frame size; δ is the number of sub-frame divisions ($\delta = w/sf$); and $L(a_i)$ is latency of actor a_i. Additionally, $L_w(a_i)$ and $L_{sf}(a_i)$ are latencies of actor a_i under the image frame size w and under the sub-frame size sf, respectively. Unlike the latency and throughput of a single actor, as decomposed in (41), the latency and throughput of the entire application are influenced by the interaction of data dependency, sub-frame size and FIFO architecture. Although sub-frame division generally allows for reduction of FIFO size, and also improves throughput, sub-frame division generally leads to some increase in application latency. For example, in the case where a single dataflow graph represents two or more applications operating concurrently, and those applications share actors in the graph, data dependencies and execution time distributions of paths in the graph influence the performance of each application in the dataflow graph differently.

Figure 55 compares, for an illustrative example, the performance of sub-frame division by $\delta = 3$ to the case where there is no sub-frame division. Here, throughput is improved for both Applications I and II. However, sub-frame division degrades the

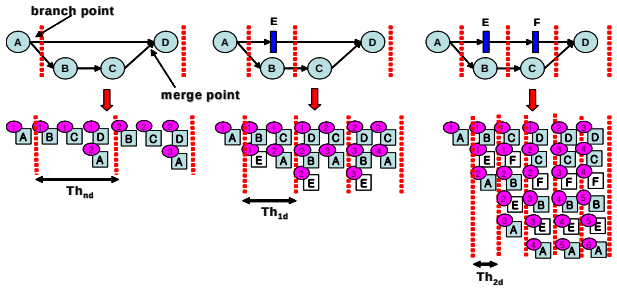
latency of Application I, whereas the latency of Application II is improved. This phenomenon generally arises when two or more applications share actors (e.g., for more compact representation and implementation) in a common dataflow graph and ρ (defined in (42) below) is smaller than 0. This effect becomes prominent especially when the ratio of $idle$ and L_W is large, where $idle$ represents the pipeline idle time. In (42), L_o can be obtained by simply dividing L_W by δ.

$$\rho = \delta L_o - \left[L_o + idle + \frac{(\delta - 1)}{Th} \right] \quad . \tag{42}$$
$$= (\delta - 1)\left(L_o - \frac{1}{Th} \right) - idle$$

$$L_w = \delta L_o,\ Th_w = 1/L_w,\ L_{\mathrm{sf}} = L_1 + \frac{(\delta - 1)}{Th_{\mathrm{sf}}},\ Th_{sf} = \frac{1}{actor_{\max}},\ \text{Always},\ Th_{\mathrm{sf}} < Th_w,$$
$$\rho > 0 \rightarrow L_w \geq L_{\mathrm{sf}},\ \text{otherwise} \rightarrow L_w < L_{\mathrm{sf}}. \tag{43}$$

In (43), $actor_{\max}$ is the execution time of the actor with the largest execution time, and L_1 represents the initial latency for subframe size sf. Here, the number of gates required for each application ($g(Appl)$) in the common graph is reduced by increasing δ. Equation (44) shows the effect of sub-frame division on the number of gates required for an application($Appl$):

$$g_{\mathrm{w}}(Appl) \approx \delta g_{\mathrm{sf}}(Appl). \tag{44}$$



a) 0 delay FIFO b) 1 delay FIFO c) 2 delay FIFO

Figure 56. Effect of data dependency on performance.

4.2.3.4 Effect of data dependency on performance and cost

In case a dataflow graph has a “branch point”, two or more paths following the branch point merge again at some subsequent point, and these paths exhibit a large execution time deviation, the associated data dependency can greatly deteriorate the performance of all the associated applications in the dataflow graph. Here, a “branch point” represents a point where a single actor has two or more output ports or a single output port goes to two or more successor actors.

Figure 56 shows how performance under sub-frame division can be improved through insertion of special FIFOs that we call “delay FIFOs(F_{delay})” (these are the FIFOs labeled E and F in Figure 56). Performance improvement by delay FIFO insertion depends on the execution time distribution of the actors on each critical path following the branch point.

Equation (45) represents the relationship between performance and the added delay FIFOs.

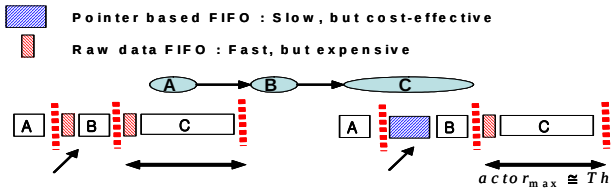
$$L_{\text{nd}} = L_{\text{nd},1} + \frac{(\delta - 1)}{Th_{\text{nd}}},\ L_{1\text{d}} = L_{1\text{d},1} + \frac{(\delta - 1)}{Th_{1\text{d}}},\ L_{2\text{d}} = L_{2\text{d},1} + \frac{(\delta - 1)}{Th_{2\text{d}}},\ L_{\text{nd},1} = L_{1\text{d},1} = L_{2\text{d},1},$$
$$Th_{nd} < Th_{1d} < Th_{2d},\ \therefore L_{nd} > L_{1d} > L_{2d} \quad (45)$$

Here, L_{nd} and Th_{nd} are the latency and throughput, respectively, without F_{delay}. Furthermore, $L_{1\text{d}}$ and $Th_{1\text{d}}$ are the corresponding values with one F_{delay}. And $L_{2\text{d}}$ and $Th_{2\text{d}}$ are those for two F_{delay}s. $L_{\text{nd},1}$, $L_{1\text{d},1}$ and $L_{2\text{d},1}$ are latencies for processing the first subframe in the cases of no F_{delay}, $1F_{\text{delay}}$ and $2F_{\text{delay}}$s, respectively.

Equation (46) represents the increase in the number of gates required for the application as delay FIFOs are added. The overhead of the delay FIFOs can be minimized by using the pointer based FIFO architecture for their implementation.

$$g_{\mathrm{nd}}(Appl) \le g_{1\mathrm{d}}(Appl) \le g_{2\mathrm{d}}(Appl)\,. \tag{46}$$

4.2.3.5 Optimization of FIFO hardware mapping



a)Raw data FIFO Only b) Combined FIFO

Figure 57. Comparison of FIFO mapping.

Idle intervals and uneven execution time distributions exist due to data dependencies and differences in operational complexity across dataflow actors. Performance and cost can be improved by integrating cost-effective, pointer based FIFOs and fast, raw token FIFOs in strategic ways.

Figure 57 provides a simple illustration of how the resource cost $g(G)$ for a data-flow graph G can be reduced significantly while maintaining overall performance through hybrid FIFO architecture selection. Here, the throughputs of both configurations are identical. Furthermore, by using sub-frame division, the difference between latency of Figures 57a and 57b can be made negligible, since the throughput (Th) is ultimately the primary factor for determining latency under sub-frame division, as

Table 11. Comparison of FIFO mapping results.

		No delay			Delay FIFO		
sf(8x8), δ=256		C1	C2	C3	C4	C5	C6
L	L_s	888 us	1157us	888 us	447 us	581 us	447 us
	L_g	884 us	1152us	884 us	443 us	576 us	443 us
	L_t	885 us	1154us	886 us	445 us	578 us	445 us
Th		1/3.5 us	1/4.5us	1/3.5 us	1/1.5 us	1/2 us	1/1.5 us
g(G)		122,915	26,840	101,565	125,516	29,441	104,166
sf(16x16), δ=64		C1	C2	C3	C4	C5	C6
L	L_s	886 us	1158 us	886 us	455 us	594 us	455 us
	L_g	869 us	1136 us	871 us	437 us	572 us	439 us
	L_t	876 us	1144 us	878 us	444 us	581 us	446 us
Th		1/14 us	1/18 us	1/13.5 us	1/6 us	1/8 us	1/6 us
g(G)		562,793	26,969	443,721	565,403	29,579	446,331

L: Latency, Th: Throughput, $g(G)$: the number of gates for graph G.
C1, C4: F_{raw}, C2, C5: F_{ptr}, C3, C6: $F_{\mathrm{raw}} + F_{\mathrm{ptr}}$.

implied by (42) and (43).

Figures 58 and 59 show our FIFO mapping algorithm, which is motivated by the observations and analysis above. It is assumed that the dataflow graph G can involve multiple applications, and moreover, that subsets of applications can share common actors for more compact representation and implementation. The function $initializeGraph()$ sets up information about estimated execution times and execution time distributions of the actors. The function also finds $g_{logic}(G)$ and $g_{\mathrm{Fraw}}(G)$. Here, $g_{logic}(G)$ represents the estimated number of gates for the main functional logic portions the actors, and $g_{\mathrm{Fraw}}(G)$ is the number of gates used for FIFOs under the assumption that only raw token FIFOs are used. The actual $g(G)$ that results from a mapped implementation lies between $g_{\min}$ and $g_{\max}$ as shown in (47).

$$g_{\min}(G) = g_{\mathrm{logic}}(G) + g_{\mathrm{Fptr}}(G),\ g_{\max}(G) = g_{\mathrm{logic}}(G) + g_{\mathrm{Fptr}}(G) + g_{\mathrm{Fdelay}}(G),$$
$$g_{\min}(G) \le g(G) \le g_{\max}(G). \quad (47)$$

For each application($G_{cur}[i]$), a critical path ($G_{\mathrm{curHg}}[i].\mathrm{crPath}$) is selected and an appropriate FIFO type is determined based on the execution time distribution of actors within the path.

For each hierarchical subsystem within the critical path, $detFIFOArch()$ is applied recursively. Finally, delay FIFO (F_{delay}) insertion is performed to improve performance. For F_{delay}, pointer based FIFOs (F_{ptr}) are used, and therefore, the overhead of redundant FIFOs can be minimized while achieving the desired performance improvement.

```
initializeGraph(G) {
   — Analyze the critical path of each application in
 the dataflow graph.
   — Obtain the estimated execution time
   — Obtain the execution time distribution on the path
   — Obtain g_logic(G) and g_Fraw(G)
   return g_logic(G), g_Fraw(G);
}

detSubFrameDivision(G){
   if(L_w(G.appl_highest_priority)
   < L_sf(G.appl_highest_priority))      {
      dataFrame = w;
   }
   else {
      dataFrame = sf;
   }
   — apply all other applications with dataFrame
}
```

Figure 58. FIFO mapping algorithm-PartA.

4.2.4 Experimental results

Figure 60 shows a complex, composite morphological image processing application used in this thesis for experimentation. Here, the performance and cost of each application under the dataflow representation are influenced by the interaction of to shared actors with the applications that contain them. Figure 60 is implemented by Verilog and is simulated under the modelSim 6.0a environment. Synthesis is performed under Xilinx XST with the Spartan3 (xc3s1500) used as the target device. Input images of size ($w = 128 \times 128$) are consumed and processed by the graph. Experimentation is performed under two different values of sf, corresponding to 8x8 and 16x16 subframes. In Table 11, $C1$ and $C4$ of F_{raw} are lower bounds in performance optimization, and $C2$ and $C5$ of F_{ptr} are lower bounds in cost reduction. Equation (48) shows how, in the following discussion, we compare the performance and costs of two different configurations C_X and C_Y.

$$\sum L_{C_X} / \sum L_{C_Y} + \mathrm{Th}_{C_X} / \mathrm{Th}_{C_Y}] / 4,\ g(G_{C_X}) / g(G_{C_Y}). \quad (48)$$

In comparison of F_{raw} and F_{ptr}, $C1$ and $C4$ provide approximately 23% performance improvement compared with $C2$ and $C5$, while requiring about 81% more gates. In comparison of F_{delay}, $C6$ provides 54% performance improvement compared with $C3$ along with a slight (2%) cost increase. In comparison of sub-frame division effects for $C4$, $C5$ and $C6$, the latency of *Smoothing* is slightly improved, whereas the

```
detFIFOArch(G){
  g_Fraw(G), g_logic(G) =initializeGraph(G);
  g_FG = 0;
  while(G ≠ φ){
    - Select an application(G_curHg) of highest
    priority
    - while(G_curHg[i] ≠ φ){
      g_Fpath =detFIFOType(G_curHg[i].crPath);
      g_Fhier = 0;
      for each hier actor Φ[j] of G_curHg[i].crPath {
        g_Fhier =g_Fhier +detFIFOArch(Φ[j];
      }
      G_curHg = G_curHg – G_curHg[i].crPath;
      g_FG = g_FG + g_Fpath + g_Fhier;
    }
    G = G – G_curHg;
  }
  if(g_Fraw(G) > g_FG) {
    - Perform data dependency analysis of G
    - Insert delay FIFOs(F_delay :F_ptr type)
    and update g_FG
    g_FG = g_FG + g_Fdelay;
  }
  detSubframeDivision(G);
  return  g_FG + g_logic(G)  ;
}

detFIFOType(G){
  g_Fsum =0;
  for each actor on the G {
    if([t_O(G.a[i]) + t_Fptr(G.a[i])]
     < [t_O(actor_max) + t_F(actor_max)])
      g_Fsum = g_Fsum + g_Fptr(a[i]);
    }
    else {
      g_Fsum = g_Fsum + g_Fraw(a[i]);
    }
  }
  return  g_Fsum  ;
}
```

Figure 59. FIFO mapping algorithm-PartB.

latency of *Gradient* is decreased as *sf* is decreased. Here, the latency impact is negligible since *idle* is relatively small compared to the execution time of each actor for processing the entire image frame *w*. On the other hand, the throughput and cost improvements are distinguishable as δ is increased.

Next, we see that $C6$, which involves both performance and cost optimization, provides 54% performance improvement and 16% cost reduction compared with the conventional approach of $C1$. Similarly, $C5$, which leans more toward cost optimization, provides 39% performance improvement and 76% cost reduction compared with the conventional approach of $C1$. Here, delay FIFO insertion in Path 1 (see Figure 60) leads to significant improvement of performance along with 2% increase of $g(G)$. Combined use of F_{ptr} and F_{raw} with F_{delay} significantly improves overall performance along with providing for cost reduction. For cases where cost is the primary issue, it is important to note the significant cost reduction of F_{ptr}.

4.2.5 Conclusions and future work

This thesis studies important issues in the mapping of dataflow representations of image processing applications into hardware implementations. Specifically, we focus on efficient mapping of FIFO buffers, and explore the effects of FIFO architecture, sub-frame division and data dependency on performance and cost. Based on this

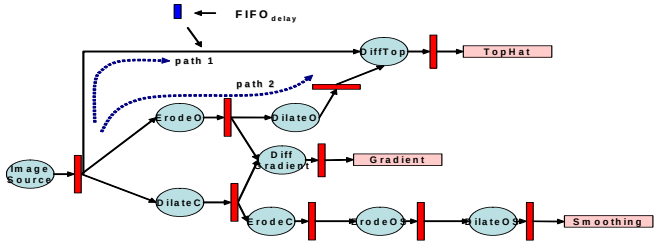


Figure 60. Complex, composite morphological image processing application (TopHat, Gradient and Smoothing).

exploration, we provides heuristic optimization methods that simultaneously improve performance and cost with manageable complexity. A strategic FIFO mapping approach that comprehensively exploits dataflow graph characteristics results in significantly lower FPGA resource requirements with nearly equal performance. Useful directions for future work include extending the methodology developed in this thesis to heterogeneous, embedded multiprocessors that include a variety of processing components, such as conventional FPGAs, platform FPGAs, and programmable digital signal processors.

4.3 Energy-driven partitioning of signal processing algorithms in sensor networks

4.3.1 Abstract

In a sensor network, as we increase the number of nodes, the requirements on network lifetime, and the volume of data traffic across the network, it is often efficient to move towards hierarchical network architectures (e.g., see [31]). In such hierarchical networks, sensor nodes are clustered into groups, and their roles are divided into master and slave nodes for more efficient structuring of network traffic. The operational complexity of each sensor node and the amount of data to be transmitted across sensor nodes strongly influence the energy consumption of the nodes, which ultimately determines the network lifetime. This paper provides a new way of reducing data traffic across nodes by determining and exploiting the lowest data token delivery points within an application graph that is distributed across a network. The technique divides

an application graph into two sub-graphs and then distributes each divided subgraph over a master node and its associated slave nodes. The buffer costs of the graph edges over the cutting line corresponds to the amount of data to be transmitted between nodes after allocating the two partial subgraphs such that one subgraph executes on a master node, and the other subgraph is distributed across the associated slave nodes. Since the energy consumption on each node is dominated by the transceiver, the reduced data traffic allows for reducing the turn-on time of the transceivers, and thereby leads to high energy savings. This technique also distributes the workload of sensor nodes in a systematic manner. The more balanced workload also contributes to efficient battery usage, and also improves the latency for processing the data frames captured by the sensor nodes.

4.3.2 Introduction and Related work

The energy consumption of the nodes in a wireless sensor network must be carefully optimized to increase network lifetime. This paper develops an overall minimization of an energy consumption of a sensor network, and provides an efficient trade-off between latency and network lifetime by balancing the workload of the sensor nodes, and carefully determining the points in the application that must communicate across nodes so that the turn-on time of transceivers is minimized.

Many useful approaches have been suggested previously to reduce the energy consumption of sensor nodes. Shih. have distributed the FFT function over a master node and slave nodes to reduce energy consumption by moving the function from a cluster head node to slave nodes [88]. Kumar, Tsiatsis, and Srivastava [54] explore

energy and latency trade-offs by considering different computational capabilities for master and slave nodes. Other researchers have suggested a hierarchical, physical layer driven sensor network design to reduce data traffic and energy consumption of a sensor node in connection with the physical-layer network functions [66, 91]. In these latter approaches, the node optimization needs to be performed carefully in conjunction with the underlying protocol characteristics.

The technique that we develop in this paper is novel in that it analyzes the pattern of internal data exchange rates within an application to minimize the overall energy consumption of a sensor network, while also taking into account changes in latency due to distributed mapping, and application of a hierarchically clustered sensor network organization. The approach is especially suited for multirate signal processing applications, which exhibit complex and nonuniform patterns of data exchange across functional modules of the application.

Many sensor network applications or important application subsystems can be modeled efficiently with dataflow semantics. By analyzing a well-designed dataflow graph model of an application, operational efficiency can be effectively estimated and optimized at a coarse grain level for various kinds of target architectures (e.g., see [11, 18, 40]). Parameterized dataflow [9] is a form of dataflow that is especially well-suited to sensor network signal processing applications due to its integrated support for adaptation and reconfiguration at various layers of abstraction. Parameterized dataflow allows for dynamic change of variables and configuration settings that can be mapped to module- or subsystem-level parameters of an application.

This paper employs the DGT (dynamic graph topology) [48] method for modeling

applications. DGT is a form of parameterized dataflow that emphasizes support for run-time flexibility by allowing for efficient, dynamic changes in application graph topologies based on run-time requests. In DGT semantics, the connections (dataflow edges) between actors (functional modules), as well as the amount of data produced and consumed by the actors can be changed, with the changes expressed in terms of dynamic parameters of the application. In the context of sensor network optimization, this feature can be used to integrate modeling of master/slave node relationships in a clustered network, and also modeling of dynamically changing application graph topologies that execute on sensor nodes.

4.3.3 Energy consumption optimization by distribution of an application

4.3.3.1 Application cutting in a sensor network

In a clustered sensor network, each sensor node captures data from its set of one or more sensors. The captured data can be sent to the associated master node immediately, or the data can be processed to some degree within the slave node before it is sent to the master node. For the data processing functionality, each edge within the application dataflow graph may have different data transfer characteristics. It is useful to consider these characteristics carefully when dividing a dataflow graph for processing across a master- and slave-node pair.

Dividing an application graph in this manner generally allows us to reduce the amount of data that must be transmitted between the nodes, and it also allows us to balancing the workloads of sensor nodes. The amount of data that must be transmitted

directly influences the turn-on time of the sensor node transceivers, which are major sources of energy consumption. Similarly, distributing the workload of an application for balanced processing increases network lifetime through balanced battery usage across the sensor nodes. Therefore, it is useful to partition dataflow graphs across sensor nodes with joint consideration of data transfer volume and workload balance.

Synchronous dataflow (SDF) is an especially useful model, due to its predictability and formal properties, for representing many signal processing applications [11, 63]. In SDF, the number of data values (tokens) produced and consumed by each actor is constant. As a result of this restriction, graphs can be scheduled statically based on the so-called repetition vector ($\vec{R}$), which is a vector that is indexed by the actors in the graph, and gives the number of times that each actor needs to be invoked in a static schedule for the graph. Such a schedule can be repeated indefinitely with bounded memory requirements to process the indefinite-length data streams that are characteristic in the signal processing domain.

The number of tokens that are transferred across an edge in the dataflow graph in each schedule iteration can be obtained from the repetitions vector $\vec{R}$ and the number of tokens produced by the source actor of the edge. Given a partition of the dataflow graph into two parts, the total number of tokens that must be transferred (buf_{tr}) across the partition can be obtained by summing up the token transfer volumes of the edges that cross the partition.

The repetitions vector can be obtained through (49) and (50) [63]:

$$T(e, v) = \begin{cases} prd(e) & \text{if v = src(e)} \\ -cns(e) & \text{if v = snk(e)} \\ 0 & \text{otherwise} \end{cases} \tag{49}$$

$$T \bullet \vec{R} = 0 \tag{50}$$

In (49), $prd(e)$ is the number of tokens produced onto edge e by each execution of $src(e)$, which denotes the source actor of e. Similarly, $cns(e)$ is the number of tokens consumed from e by each execution of $snk(e)$, which is the sink actor of e.

The total number of tokens buf_{tr} that cross a given partition in a schedule iteration can then be expressed as

$$buf_{tr} = \sum_{i=1}^{N_c} \sum_{j=1}^{Edge_{n_i}} \vec{R}(n_i) \cdot prd(e_j(n_i)) \tag{51}$$

where N_c is the number of actors whose outgoing edges cross the partition; $(n_1, n_2, \ldots, n_{N_c})$ is an ordering of the actors whose outgoing edges cross the partition; $Edge_{n_i}$ is the number of outgoing edges of actor n_i that cross the partition; and $e_j(n_i)$ is the jth outgoing edge of n_i that crosses the partition, based on some ordering of the outgoing edges.

Figure 61(a) illustrates how data transmission requirements can change depending the selection of a partition. Figure 61(a) provides four possible candidates for a “cutting line” to determine the partition. The edges that cross the cutting line determine the network data transfer volume that must be incurred on each graph iteration due to the associated application partition. The number shown inside each actor represents the processing complexity in terms of the actor execution time. The number on the left side of an edge represents the number of tokens produced by the source actor, and the number on the right side represents the number of tokens consumed by the sink actor.

In Figure 61, there are four edges, e0,e1,e2 and e3. Figure 61(b) shows the repetition vector for Figure 61(a), and Figure 61(c) shows buf_{tr} for each cutting line candidate $C0$-$C3$.

After a cutting line is determined for a graph, the graph is effectively divided into "left" and "right" subgraphs, where the left subgraph represents preprocessing of sensor signals and the right subgraph represents postprocessing. Accordingly, the left subgraph is allocated to the associated slave node, and the right subgraph is allocated to a master node.

Each cutting line in general leads to different workload distributions of an application graph, as well as different values of buf_{tr}. Intuitively, $C0$ leads to increased workload for the master node, since the master node is in charge of most of the data processing functionality. That value of buf_{tr} for $C0$ is 6 tokens. Similarly, $C3$ increases the workload of the slave node, while alleviating the workload of the master node; however, buf_{tr} for $C3$ increases to 16 tokens. As an alternative to $C0$ and $C3$, $C1$ allows for lower data transmission and more balanced workload distribution.

4.3.3.2 Cutting algorithm

Cutting an application dataflow graph is an NP hard problem. However, in many

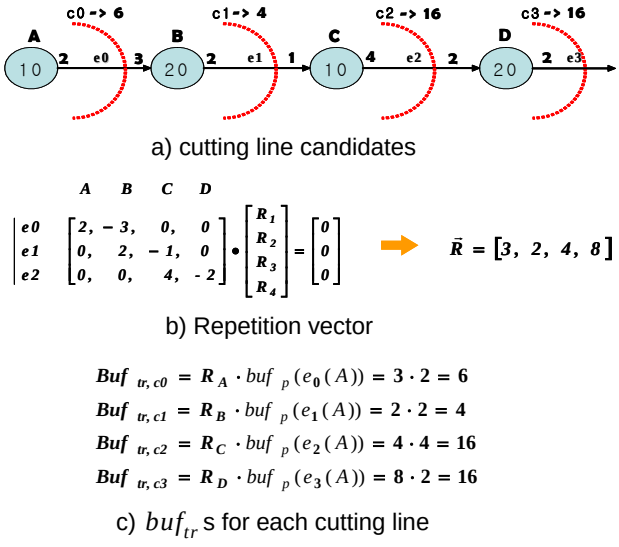


Figure 61. An illustration of partitioning (cutting line) trade-offs.

sensor network applications, particularly those involving very simple, ultra-low cost/ power sensor node processing, the application graphs are of limited size, and are manageable by exact techniques. This paper uses an exhaustive search method for finding the best cutting line to target such applications and to demonstrate the potential of high-level, dataflow graph analysis for coordinating the processing across senor nodes.

More precisely, given an application dataflow graph Φ, our objective is to partition Φ into two subgraphs Φ_1 and Φ_2. In this partitioning, we would like to minimize

$$buf_{tr,\, c_i(\Phi)} \tag{52}$$

subject to

$$\text{if } n \in \text{actors}(\Phi_2), \text{ then } \text{successors}(n) \subset \text{actors}(\Phi_2) \text{ and} \tag{53}$$

$$t(\Phi_1) - \delta t(\Phi_2) \leq \Omega \tag{54}$$

Here, $t(X)$ is the execution time of subgraph X, assuming that the subgraph is assigned to the same sensor node, and processing resources across the nodes are homogeneous. The formulation can easily be extended to handle heterogeneous processing resources, but for clarity and conciseness, we focus here on the homogeneous case. The subgraph execution time is obtained by adding the execution time estimates for the individual actors in the subgraph. Also, $\text{actors}(X)$ represents the set of actors in subgraph X, and given an actor n, $\text{successors}(n)$ represents the set of immediate graph successors of n. The constraint in (22) is necessary to avoid cyclic dependencies (potential deadlock) between the master and slave node.

The parameter δ is a coefficient that affects the load balancing aspect of the optimization. An appropriate choice for δ can be estimated by experimentation, or one can run the optimization multiple times for different values of δ and take the most attractive result. As the value of δ is increased, the workload of the master node is

decreased, and the latency of the application is also generally decreased since the workload of the application is more distributed over slave nodes. The symbol Ω represents a tolerance for workload imbalance in conjunction with δ.

4.3.3.3 Effect on energy consumption

The total energy of a sensor node E can be divided into two parts: E_{radio} and E_{mc}, where E_{radio} represents the energy consumed by the transceiver, and E_{mc} represents the energy consumed by the microcontroller and the associated peripherals, such as the memory, UART, and ADC, apart from the transceiver. Thus,

$$E = E_{radio} + E_{mc} \tag{55}$$

The transceiver energy E_{radio} is usually dominant in the total energy consumption of a sensor node, and in the context of dataflow processing, this energy is proportional to the number of tokens that must be communicated. An optimal cutting of an application graph in terms of token transfer minimization across the cutting line therefore results in optimal streamlining of transceiver turn on time. In other words, by reducing buf_{tr}, E_{radio} can be minimized under the workload balance constraints.

Each partitioned subgraph is mapped to a slave node or a master node. The operations of a subgraph apart from its transceiver-related operations are modeled by E_{mc}. Through a minor abuse of notation, we represent the energy consumption for data processing in an application $appl$ as $E_{\mathrm{mc}}(appl)$. By distributing the application over a master node and a slave node, $E_{\mathrm{mc}}(appl)$ can be divided into two sub energy consumption components: $E_{\mathrm{mc,s}}(appl)$ and $E_{\mathrm{mc,m}}(appl)$, corresponding respectively to the slave and master nodes. Thus, we have

$$E_{\mathrm{mc}}(appl) = E_{\mathrm{mc,s}}(appl) + E_{\mathrm{mc,m}}(appl) \tag{56}$$

In a sensor network cluster that consists of a single master node and η slave nodes, the master node iterates η times to process data frames from all of its slave nodes. Then $E_{\mathrm{mc,m}}$ is the total energy consumption for microcontroller-related functions by the master node during its η iterations of right-side-subgraph processing of data frames received from the slave nodes. The relationships among $E_{\mathrm{mc,m}}$, $E_{\mathrm{mc,s}}(appl)$, and $E_{\mathrm{mc,s}}(appl)$ can be summarized as

$$E_{\mathrm{mc,m}} = \eta E_{\mathrm{mc,m}}(appl) = \eta(E_{\mathrm{mc}}(appl) - E_{\mathrm{mc,s}}(appl)) \quad \text{and} \tag{57}$$

$$E_{\mathrm{mc,s}} = E_{\mathrm{mc,s}}(appl) \tag{58}$$

$E_{\mathrm{mc,s}}$, which is the total energy consumption for microcontroller-related functions of a single slave node, is equal to $E_{\mathrm{mc,s}}(appl)$ since data frames for an application graph are transmitted from a slave node to a master node, and for a single data frame, one iteration of a left-side (slave node) sub-graph is activated. Here, $E_{\mathrm{radio,m}}$ is proportional to η since the transceiver of the master node should be turned on during the entire reception of η data frames from the η slave nodes.

The total energy consumed by the master node can be expressed as

$$E_m = E_{\mathrm{mc,m}} + E_{\mathrm{radio,m}} = E_{\mathrm{mc,m}} + \lambda\eta E_{\mathrm{radio,s}} \tag{59}$$

where λ is a coefficient that relates $\eta E_{\mathrm{radio,s}}$ and $E_{\mathrm{radio,m}}$. Since typically $\lambda\eta \gg 1$, the master node has significantly more energy consumption compared to the slave nodes. To reduce the overall energy consumption, the number of tokens that must be transmitted across the nodes should be minimized under the given workload distribution constraints.

4.3.3.4 Effect on latency

The latency for processing a single data frame of a given application depends on

the number of slaves in the network cluster, the network topology, and the volume of data contained in each data frame. For a cluster that consists of a single master node and η slave nodes, the latency $L(app)$ for processing a single application data frame can be expressed by (60), independent of the underlying transmission protocol.

$$L(app) = \eta L_{m,\,\text{frame}}(app) + L_{s,\,\text{frame}}(app) + \eta L_{tr,\text{frame}}(app) \tag{60}$$

where $L_{m,\,\text{frame}}(app)$ is the latency of master node (right-side subgraph) processing for a single data frame, and $L_{s,\,\text{frame}}(app)$ is the corresponding latency of slave node processing. In total, a latency of $\eta L_{m,\,\text{frame}}(app)$ is induced on the master node to process the data from all of the slave nodes. The slave nodes, however, can operate in parallel, and thus, the latency required for slave node processing is independent of the number of slave nodes within the network cluster.

$L(app)$ also depends on the network delay for transmitting data frames across nodes. $L_{tr,\text{frame}}(app)$ thus denotes the latency for transmitting a single data frame from a slave node to the master node. The total transmission latency for delivering η data frames from the slave nodes becomes $\eta L_{tr,\text{frame}}(app)$.

Clearly, $L_{tr,\text{frame}}(app)$ depends on the data frame size. In particular, $L_{tr,\text{frame}}(app)$ is proportional to buf_{tr}.

Figure 62 shows three different cases of cutting line selection for an application example that involves maximum entropy spectrum computation. This application is based on an example in the Ptolemy II design environment [24]. The application can be divided into two subgraphs, which are allocated to master and slave nodes as illustrated in the figure. The dotted lines represent cutting line candidates. The application is characterized by a parameter n, called the *order* of the spectrum computation.

In Figure 62(a), the slave nodes capture raw data frames and send them directly to the master node, where the maximum entropy spectrum processing is performed. Here, buf_{tr} between a single slave node and the master node is 2^{n+1}. Therefore, the total data transmission for each data frame from the 5 slave nodes is $5 \times 2^{n+1}$.

In Figure 62(b), each slave node fully processes a data frame before sending to the master node. This is a fully distributed approach, which minimizes the workload of the master node. In this approach, each slave node sends 2^n tokens to the master node. Thus, the total data transmission from the 5 slave nodes is 5×2^n.

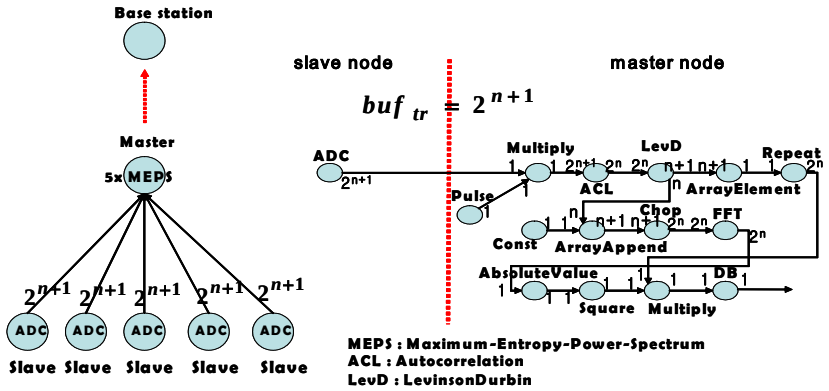
In Figure 62(c), on the other hand, the application graph is divided more evenly into two subgraphs A and B. A copy of subgraph A is assigned to each slave node, and B is allocated to the master node. The carefully-constructed cutting line between A and B reduces buf_{tr} to $(n+1)$, which results in total slave-to-master data transmission of $5 \times (n+1)$.

Without consideration of $L_{tr,\text{frame}}(app)$, the application latencies ($L(app)$) of the three cases in Figure 62 are related as $(L_{case1} > L_{case3} > L_{case2})$. Case 2 provides the maximal workload distribution by allowing raw data frames to be fully processed in the slave nodes. However, the greatly-reduced $L_{tr,\text{frame}}(app)$ of Case 3 offsets the increase in $L_{m,\text{frame}}(app)$ due to the increased workload of the master, while allowing reduced energy consumption because of reduced transceiver demands.

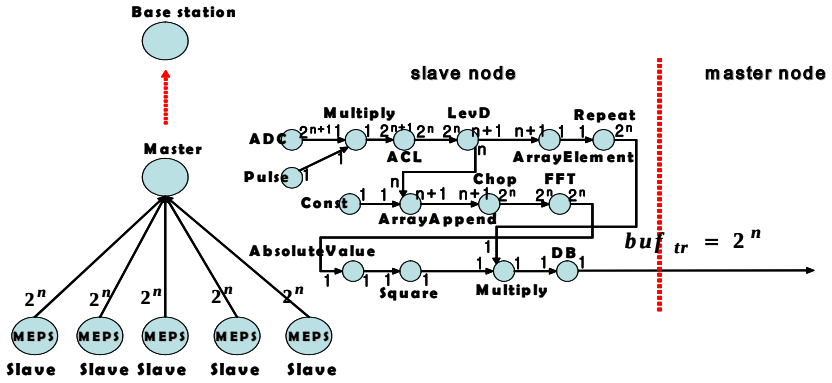
In summary, the example of Figure 62 illustrates the trade-offs that we can explore among processor workload balancing, latency cost, and transceiver requirements when considering different cutting lines for a multirate signal processing application.

4.3.4 Experimental results

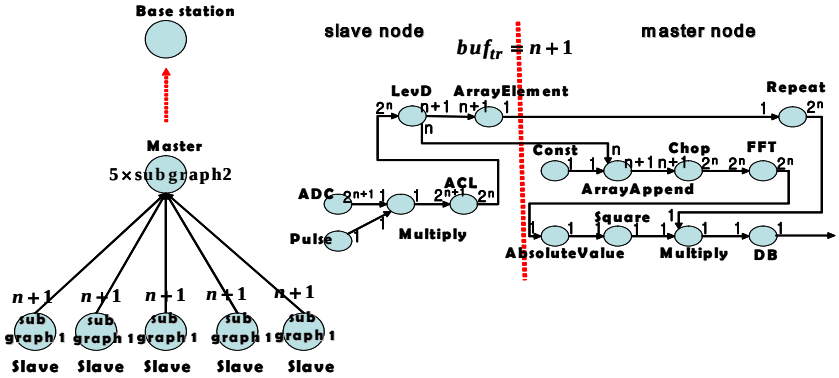
We have developed experimental prototype platforms (Figure 63) for master and slave nodes using reconfigurable off-the-shelf components, including the Texas Instru-



a) case 1



b) case 2



c) case 3

Figure 62. Application mapping over sensor nodes

ments MSP430 microcontroller, the LINX Technologies 916MHz wireless transceiver, and a microphone sensor. The MSP430 provides a 16-bit processor core, along with a 12-bit ADC, 16-bit hardware timer, UART, 48kB program memory, and 10kB data memory.

Figure 64 and Figure 65 show experimental results where we measured the current consumption from our prototype platforms as they were running different partitionings of the maximum entropy spectrum application. In these experiments, we used TDMA operations for wireless communication. For the TDMA operations, we used 10 time slots per frame, and 250ms per time slot to guarantee that transmission and relevant computations can be completed within each slot.

Figure 64 shows experimental results for current consumption comparison in three different application mapping cases involving a single master node and three slave nodes when $n = 8$ is the application order. The amounts of data (in bytes) that must be transmitted and received between nodes in each slot under cases 1, 2, and 3 are, respectively, 512(2^{8+1}), 256(2^8.) and 9(8+1).

Figure 64 shows that sensor node platforms consume much more current when the nodes are transmitting or receiving data compared to when the nodes are in their idle modes. Also, transceiver operation dominates the overall current consumption when data is being transmitted or received.

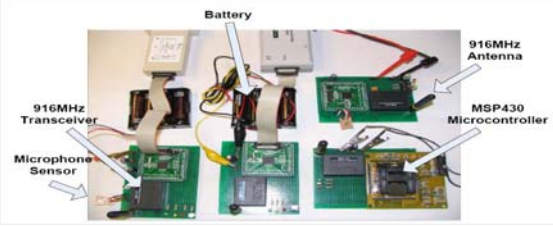


Figure 63. MSP430-based sensor node platforms

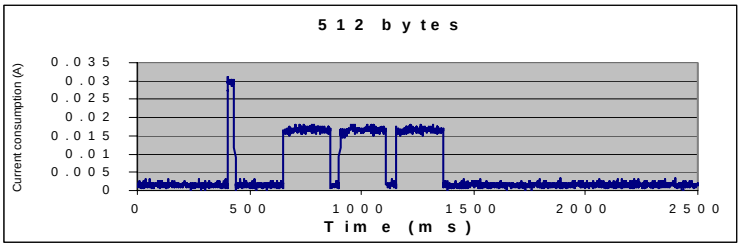
According to the results in Figure 64, we observe that case 3 of the suggested application cutting technique consumes 70.5% less energy than case 1 and 56.5% less than case 2. Here, the current and voltage for each sensor node are obtained by a digital storage oscilloscope. The power consumption for a time frame is obtained according to the sampling points for current and voltage values. The energy consumption within a TDMA time frame is calculated by integrating the power consumption over the time frame. Because the TDMA operations provide a periodic way to generate similar modes of operations for consecutive time frames, we calculate energy consumption results for several time frames and compute average values from these results.

Figure 65 shows how energy comparison varies as the application order parameter n is changed. For each order number, we measured current consumption and voltage on our prototype platforms, and calculated the average energy consumption based on the TDMA time frames. According to the results in Figure 65, we observe that as the order number is increased, the disparities between different application mapping cases become more prominent.

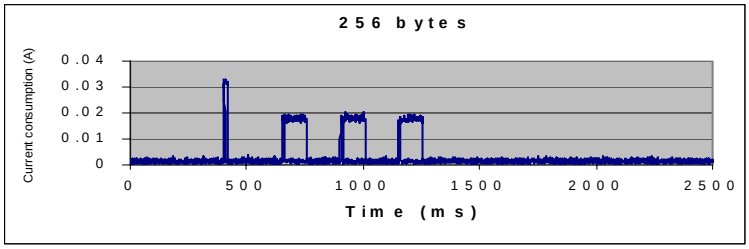
Table 12 shows that as the application order increases, which results in increased data transmission, the relative latency gap between case 2 (best latency) and case 3 (best energy consumption) decreases. For any order, case 1, which is the conventional master-node-centric mapping, generates the worst latency and energy consumption pattern for our benchmark applications.

4.3.5 Summary

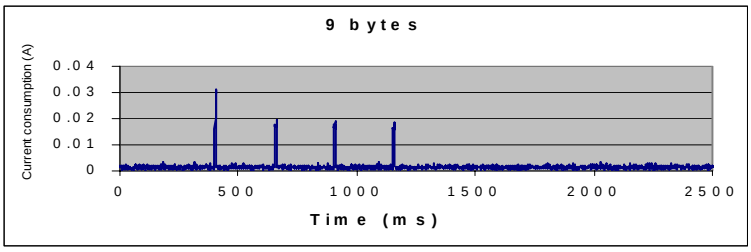
In this paper, we have developed a technique to partition an application graph into



a) case 1(512B)



b) case 2(256B)



c) case 3(9B)

Figure 64. Current consumption comparison of three application mappings.

Table 12. Latency comparison for different values of order.

order	3	4	5	6	7	8
case1	180ms	254ms	404ms	721ms	1364ms	2699ms
case2	64ms	92ms	150ms	270ms	515ms	1021ms
case3	146ms	191ms	280ms	474ms	864ms	1683ms

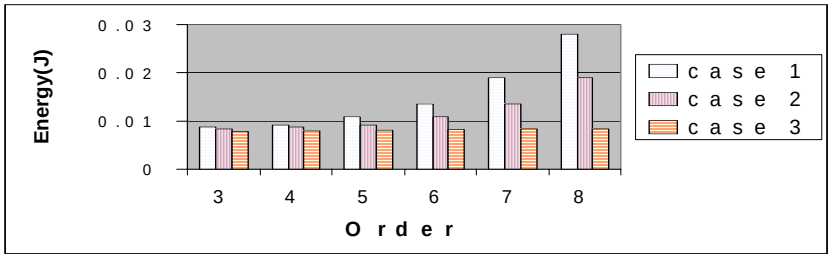


Figure 65. Energy consumption comparison for different order values.

subgraphs to optimize the workload distribution and data transmission when mapping the application onto a hierarchical sensor network. The technique allows the overall energy consumption of a sensor network to be minimized without considerable loss of latency. In our future work, we will explore the integration of error correction into our partitioning framework to provide further savings in energy consumption.

Chapter 5 : Conclusion and Future work

In this thesis, we have proposed novel models and algorithms for streamlining scheduling, memory management, and interprocessor communication in embedded multiprocessor implementations of signal processing applications. We have placed special emphasis on the image processing domain. For application modeling, we have proposed two novel modeling techniques called blocked dataflow (BLDF) and dynamic graph topology (DGT). These modeling approaches capture within formal frameworks the structure of block-based image processing operations and reconfigurable, multi-mode dataflow behaviors, respectively.

For scheduling, we have developed a novel intermediate representation called the pipeline decomposition tree for efficient representation and analysis of alternative multiprocessing configurations for signal processing applications. We have also developed an algorithm, called pipeline decomposition tree scheduling (PDT scheduling), which applies the PDT to systematically derive optimized multiprocessor schedules that employ coarse-grained (task-level) pipelining. To optimize interprocessor communication, we have developed two novel post-optimization techniques for hardware resource mapping and software synthesis.

In the following sections, we provide more detailed summaries of these methods and suggest useful directions for future work.

5.1 Modeling

5.1.1 Blocked DataFlow (BLDF)

This thesis has developed a blocked dataflow (BLDF) modeling semantic for augmenting dataflow-based DSP design tools with integrated capabilities for meta-modeling, block-based processing, multidimensional representation, and dynamic parameter reconfiguration. BLDF builds on parameterized dataflow semantics, and is compatible with decidable dataflow models such as CSDF, MDSDF, SDF, and SSDF.

In BLDF(Blocked Dataflow) model, by exploiting block based operational features most image processing applications commonly have, the BLDF extracts an iteration number of each actor within the associated body subsystem at compile time, which is related to the number of firings of actors within a graph. An iteration number allows for quasi-static scheduling of an application modeled under BLDF semantic. At runtime, by recalculating relative ratios of iteration numbers among actors, the final decision on the number of firings of each actor is made. By the parameterized token delivery method, the BLDF simplifies connections between actors and reduces the buffer size. BLDF intrinsically adapts to a hierarchical design of an application by making an actor in each hierarchical level extracting the corresponding header information and data from nested header and payload data.

5.1.2 Dynamically configured graph topology

In DGT(Dynamic Graph Topology), a new paradigm for the change of control/ data flow beyond the limit of existing dataflows is introduced. The DGT provides a new way of modeling the flexible change of a graph topology and dynamic change of

token consumption and production rates depending on parameters. In addition to providing efficient and flexible support for multiple modes of system operation, DGT allows us to reduce overall memory size by systematically sharing code and applying tailored scheduling methods across the different graph topologies that make up a DGT application. By providing meta-scheduling technique for graph configurations at compile time, the requirements for dynamic change of both control and data path are satisfied.

5.1.3 Future work

Blocked dataflow and DGT(Dynamic Graph Topology) graph model provides a quasi-static and meta scheduling environment. Theses techniques increase the expressivity and the flexibility of a modeling paradigm in a dataflow based modeling approach by providing a way of reconfiguring parameters at runtime while keeping the benefits of major information obtained at compile time. Recently, as the complexity of embedded systems increases, DSP based embedded system integrate several applications with various requirements and conflicting constraints, for example, a fast response time, but loose memory size requirement or a soft-real time, but a small foot print etc.

Combining separately modeled multiple dataflow graphs in a single system in terms of performance maximization and resource usage minimization is non trivial problem. This may lead to concurrent running of individually modeled dataflow graphs, which in turn may lead to the use of the context switch of dataflow graphs. Runtime use of scheduling information obtained at compile time for the context switch

of dataflow graphs is a new paradigm of dataflow based modeling of DSP applications.

Figure 66 shows three different cases of scheduling two dataflow graphs at runtime in a single system. Table 13 shows a probable comparison for three methods in figure 66 in terms of a code size, a buffer size and a response time. In table 13, a code and buffer size, and an execution time of actors in graph $G1$ and $G2$ are assumed to be identical for simplicity in comparison. In table 13, C represents a code size of an actor and B represents a buffer size between two actors. T represents an execution time of any single actor within graphs $G1$ and $G2$. In method 1, two graphs;$G1$ and $G2$ run concurrently by sharing tasks in common. Method 1 produces a balanced outcome in terms of code/buffer size and response time as shown table 13. In method 1, FIFO buffer size from actor B through actor C are doubled due to a combined running of shared actors in two separate graphs. It's because while graph $G1$ or $G2$ is running, FIFO buffers for shared actors (actor B and actor C) in the suspended graph must be held. In method 2, two graphs run sequentially. Method 2 is efficient at reducing buffer size since G1 and G2 are processed sequentially. In method 3, two dataflow graphs run simultaneously. Method 3 may require dedicated processing resources for each separate graph. Method 3 may lead to increased code size compared to method 1 and method 2, but allows for the fastest response time for each graph.

For more specific model for method 1, delayed graph context switch model can be used. Unlike a fully dynamic scheduling of tasks by operating systems, delayed context switch model can handle multiple dataflow graphs by a polling mode based graph context switch. Checking points for polling mode named SC (Switch Checker) or Tim-

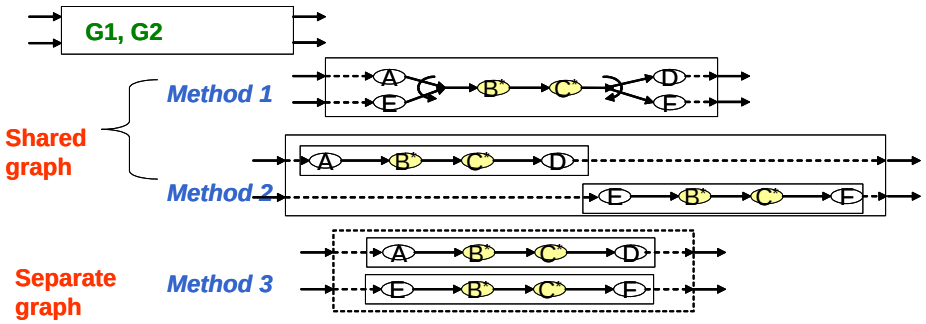


Figure 66. An example of simultaneous running of multiple dataflow graphs

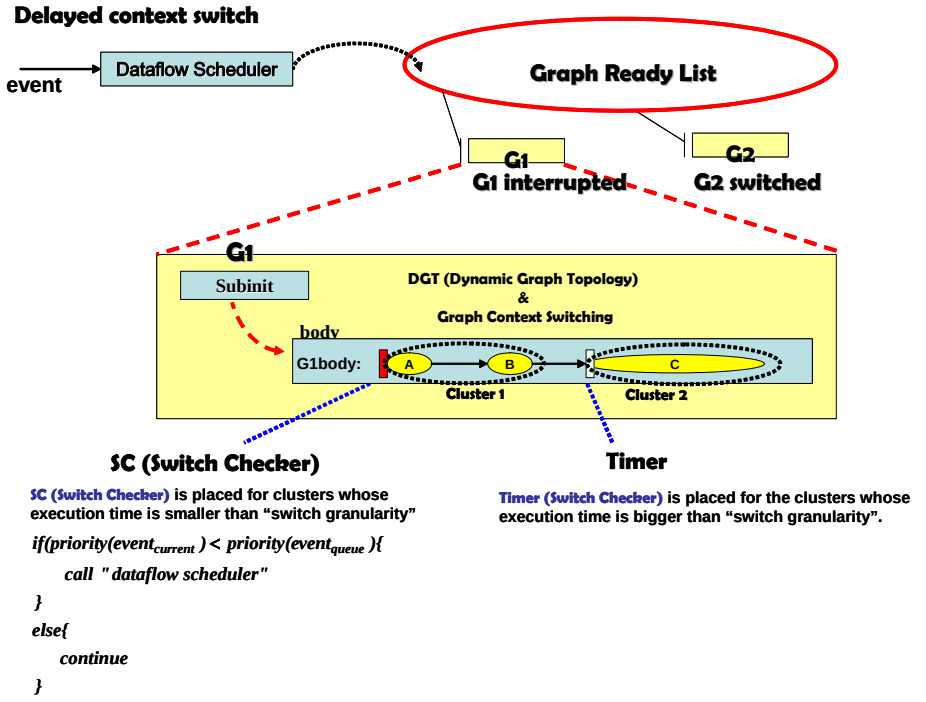


Figure 67. Delayed context switch model of dataflow graphs

Table 13. A comparison of runtime manipulation methods of multiple dataflow graphs

	schedule	Response time	code size	buffer size
method 1	AE**BBCC**DF	G1: 6*T G2: 6*T	6*C	6*B
method 2	A**BC**DE**BC**F	G1: 4*T G2: 8*T	6*C	3*B
method 3	A**BC**D E**BC**F	G1: 4*T G2: 4*T	8*C	6*B

ers can be inserted into between clusters. A cluster may consists of several sequential

invocation of actors. The size of clusters depends on the granularity of graph context switch. SC could be a synchronous graph context switch mechanism which is useful for memory management of loaded graphs since the context switch occurs between clusters. This provides an efficient way of managing runtime memory usage by keeping track of memory usage by loaded graphs. This information can be used for picking up the appropriate graph for the graph context switch in a graph ready list, which holds a list of loaded or interrupted graphs. Figure 68 shows how memory usage of graphs can be used to determine the appropriate graph from a graph ready list. In case 1, the context switch request for $G3$ is allowed since a probable memory usage of $G3$ can be fit within an available memory. In case 2, $G3$ context switch is delayed until $G2$ completes due to a probable memory shortage $G3$ can cause at runtime. Memory usage information can be obtained within each dataflow model semantic at compile time. The graph context switch technique guarantees a bounded memory usage among graphs while allowing the priority based graph context switch at runtime. This technique can be further explored with various parameters and constraints. One of con-

straints could be latency requirement of each gaph.

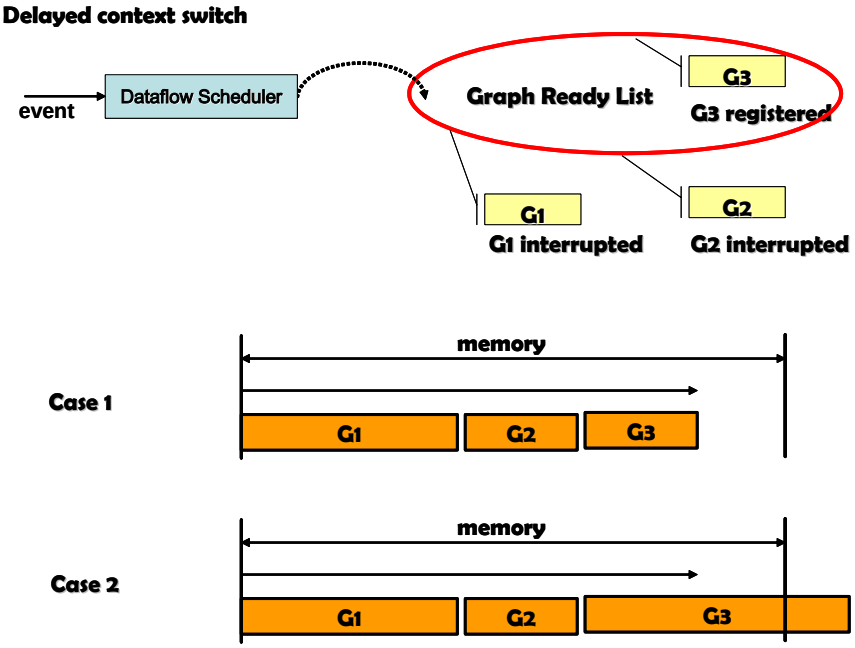


Figure 68. Relationship between memory usage of graphs and the graph context switch

5.2 Scheduling

In this thesis, a novel scheduling technique named PDT scheduling is suggested. PDT scheduling is a deterministic scheduling technique considering various realistic problems occurring during the integration of a DSP embedded system. PDT scheduling exploits a pipelined processor architecture to allocate actors onto processors. PDT scheduling considers two different memory architectures; a shared memory and a separate memory architecture. Each memory architecture entails the associated communication costs; IPC (Inter Processor Communication) and a bus contention. IPC cost is modeled under a separate memory architecture. A bus contention is modeled under a shared memory architecture. The technique studies how a memory architecture influ-

ences performance each under memory constrained condition and unlimited memory usage condition. Two different hierarchical memory models; an on-chip memory and an external memory are considered during scheduling. Only a shared memory architecture is considered for an on-chip memory due to the limited size requirement of DSP on-chip areas. PDT scheduling exploits data parallelism and task parallelism together. Pipelined architecture which relates to a task parallelism improves the throughput, but degrades latency in general. For data parallelism model, this thesis provides a heterogeneous data parallelism model to improve latency and throughput together. PDT generates various sets of pipelines which provide different combinations of latency and throughput. Pipelines are generated by PDT exploration process. And then the suggested technique named HDEST (Heterogeneous Data Parallelism EST) allocates a dataflow graph onto stages of pipelines while considering given constraints such as memory constraints and performance requirements. This technique can result in providing more improved way of prototyping embedded systems integrating image processing applications and more accurate estimation of the system performance depending on constraints at an early stage of system development stages.

5.2.1 Future work

Future work may include hardware or software synthesis using scheduling information obtained during PDT scheduling process such as a minimum memory size for either *on-chip* memory or *external* memory, a memory architecture, or a bus architecture related to the specific memory architecture. Some directions for future work may include how the variation of data frame size influences both execution-times of actors

within a dataflow graph and the final scheduling results. Figure 69 shows how memory architecture models and the associated bus architecture models can be synthesized based on data dependency among actors within a dataflow graph. In figure 69, after scheduling, three stages of a pipelined processor architecture is generated. Data dependency among actors can be considered for configuring hierarchical bus architecture. Processors with close data dependencies are placed to a common bus. For processors for some actors sharing data tokens, a shared memory bus architecture can be considered.

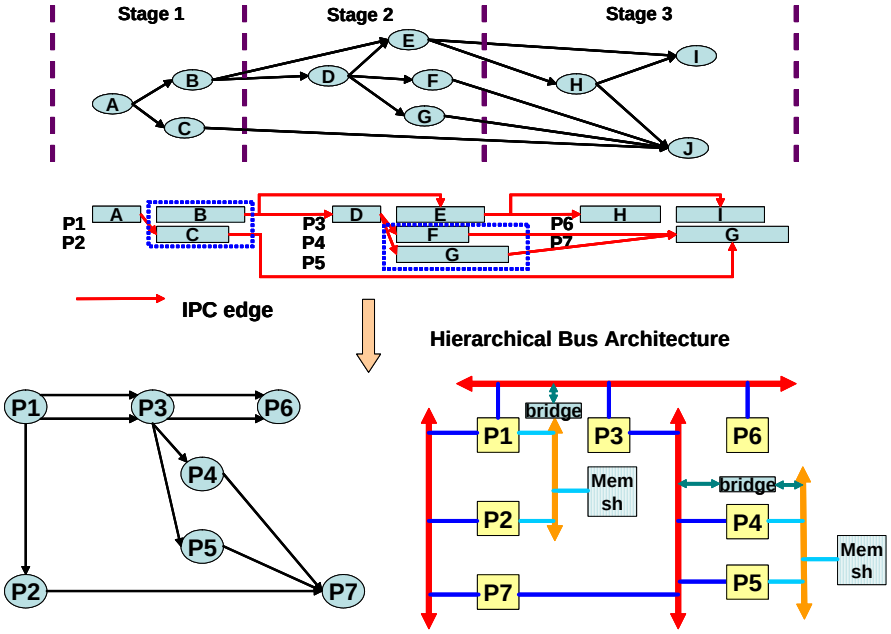


Figure 69. Hierarchical bus architecture synthesis based on data dependency of actors within a graph

5.3 Communication optimization

5.3.1 Hardware communication optimization

This thesis studies an efficient mapping of dataflow representations of image processing applications into hardware implementations. Specifically, we focus on cost-effective mapping of FIFO buffers, and explore the effects of FIFO architecture, sub-frame division and data dependency on performance and cost. Based on this exploration, we provide a heuristic optimization method in consideration of performance and resource cost. A strategic FIFO mapping approach that comprehensively exploits dataflow graph characteristics results in significantly lower FPGA resource requirements with nearly equal performance.

5.3.2 Software communication optimization

As a post optimization technique for satisfying application specific requirements, this thesis provides an application cutting technique in consideration of power consumption minimization and performance improvement. The technique is applied to a sensor network application which has a high priority on power management of sensor nodes.

The suggested technique divides an application graph into two sub-graphs in terms of the workload distribution and data transmission between sub-graphs. The technique allows the overall energy consumption of a sensor network to be minimized by energy aware mapping of an application onto a sensor network.

5.3.3 Future work

For hardware resource mapping technique, useful directions for future work may include extending the methodology developed in this thesis to heterogeneous, embedded multiprocessors that include a variety of processing components, such as conventional FPGAs, platform FPGAs, and programmable digital signal processors.

For software communication optimization technique, the future work may include how application dependent optimization technique can further improve application specific requirements such as power consumption, latency or memory usage.

For example, for a sensor network application, the future work will include how an additional error correction routine can reduce further energy consumption by reducing output power of a transceiver in conjunction with an effect of an increased functionality on an energy consumption of a microcontroller and the increase of the latency in relation with the suggested technique.

BIBLIOGRAPHY

[1] I. Ahmed and Y. K. Kwok, “A new approach to scheduling parallel programs using task duplication”, International Conference on Parallel Processing, August 1994, Vol. 2, pp 47-51.

[2] T. Back, U. Hammel, and H.-P. Schwefel, “Evolutionary computation: Comments on the history and current state”, IEEE Transactions on Evolutionary Computation, vol. 1, pp. 3.17, Apr. 1997.

[3] J. L. Baer, Tien-Fu Chen, An effective on-chip preloading scheme to reduce data access penalty, Proceedings of the 1991 ACM/IEEE conference on Supercomputing, p.176-186, November 18-22, 1991, Albuquerque, New Mexico, United States.

[4] S. Bakshi, Daniel D. Gajski, Partitioning and pipelining for performance-constrained hardware/software systems, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, v.7 n.4, p.419-432, Dec. 1999.

[5] N. Bambha, S. S. Bhattacharyya, J. Teich, and E. Zitzler. Hybrid search strategies for dynamic voltage scaling in embedded multiprocessors. In Proceedings of the International Workshop on Hardware/Software Co-Design, pages 243-248, Copenhagen, Denmark, April 2001.

[6] N. K. Bambha, S. S. Bhattacharyya, J. Teich, and E. Zitzler. Systematic integration of parameterized local search in evolutionary algorithms. IEEE Transactions on Evolutionary Computation, 8(2):137-155, April 2004.

[7] S. Banerjee, T. Hamada, P. M. Chau, and R. D. Fellman, Macropipelining based scheduling on high performance heterogeneous multiprocessor systems. *IEEE Trans. Signal Processing*, vol. 43, pp.1468-1484, June 1995.

[8] A. Benedetti, A. Prati, N. Scarabottolo, “Image Convolution on FPGAs: The Implementation of a Multi-FPGA FIFO Structure,” 24 th. EUROMICRO Conference Volume 1 (EUROMICRO'98), August 25 - 27, 1998.

[9] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408-2421, October 2001.

[10] S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for DSP. *IEEE Transactions on Circuits and Systems -- II: Analog and Digital Signal Processing*, 47(9):849-875, September 2000.

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

[12] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, “Resynchronization of Multiprocessor Schedules:Part 1-Fundamental Concepts and Unbounded-Latency Analysis,” emorandum No. UCB/ERL M96/55, 15 October 1996.

[13] S. S. Bhattacharyya, “Exploiting Free Space Optical Interconnects for System-on-Chip DSP Applications,” presented at DARPA opto-center kickoff meeting, December 11, 2000.

[14] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, “Optimizing Synchronization in Multiprocessor DSP Systems,” IEEE Transactions on Signal Processing, June, 1997.

[15] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397-408, February 1996.

[16] E.R. Bonsma, “Multiprocessor scheduling of fine-grain iterative data-flow graphs using genetic algorithms,” M.S. thesis, University of Twente, Department of Electrical Engineering, June 1997, EL-BSC-018N97.

[17] J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems. In Proceedings of the *IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 508-513, October 1994.

[18] J. T. Buck, E. A. Lee, “Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model”, *Proc. ICASSP*, April, 1993.

[19] B. P. Buckles and F. E. Petry. Genetic Algorithms. The IEEE Computer Society Press, Los Alamitos, 1992.

[20] A. Choudhary, W. K. Liao, P. Varshney, D. Weiner, R. Linderman and M. Linderman ``Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers," 12th International Parallel Processing Symposium, to be appeared, 1998.

[21] J. Y. Colin and P. Chritienne, “C.P.M. Scheduling with small Communicational delays and task duplication”, Operational research, Vol. 39, No. 4, July 1991, pp. 680-684.

[22] S. Darbha and D. P. Agrawal, “A task duplication based scalable scheduling algorithm for distributed memory systems”, Journal of parallel and Distributed Computing, Vol. 46, No. 1, October 1997, pp. 15-27.

[23] M.K. Dhodhi, Imtiaz Ahmad, and Ishfaq Ahmad. “A multiprocessor scheduling scheme using problem-space genetic algorithms”. In IEEE Conf. on Evolutionary Computation, pages 214{219, 1994.

[24] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity. the Ptolemy approach. Proceedings of the IEEE, January 2003.

[25] M. Engels, G. Bilsen, R. Lauwreins, J. Peperstraete, °×Cyclo-Static dataflow, IEEE Transactions on Signal Processing, 44(2), pp 397-408, February 1996.

[26] H. Forsberg, M. Jonsson, and B. Svensson, “A scalable and pipelined embedded signal processing system using optical hypercube interconnects,” Proc. IASTED 12th International Conference on Parallel and Distributed Computing and Systems (PDCS 2000), Las Vegas, NV, USA, Nov. 6-9, 2000, pp. 265-272.

[27] A. Gerasoulis and T. Yang, “A comparison of clustering heuristics for scheduling directed acyclic graphs onto multiprocessors”, Journal of Parallel and Distributed Computing, Vol. 16, No. 4, December 1992, pp. 276-291.

[28] D.E. Goldberg. “Genetic Algorithms in Search, Optimization and Machine Learning”. Addison-Wesley, 1989.

[29] S. Ha, “Compile-time scheduling of dataflow program graphs with dynamic constructs”, University of California at Berkeley, Berkeley, CA, 1992.

[30] F. Hannig, H. Dutta and J. Teich, “Regular Mapping for Coarse-grained Reconfigurable Architectures”, In Proceedings of the 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004), Vol. V, pp. 57-60, Montreal, Quebec, Canada, May 17-21, 2004.

[31] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In Proceedings of the Hawaii International Conference on System Sciences, 2000.

[32] P. Hoang and J. Rabaey, Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput. In I*EEE Transactions on Signal Processing*, vol. 41, no.6, June 1993.

[33] J. Horstmannshoff, T. Grotker, H. Meyr “Mapping multirate dataflow to complex RT level hardware models”, IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '97), July 14 - 16, 1997 Zurich, SWITZERLAND.

[34] J. Horstmannshoff, H. Meyr, “Efficient building block based RTL code generation from synchronous data flow graphs”, Annual ACM IEEE Design Automation Conference Proceedings of the 37th conference on Design automation, Los Angeles, California, United States, Pages: 552 - 555, 2000.

[35] C. C. Hui and S. T. Chanson, “Allocating task interaction graphs to processors in heterogeneous networks”, IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 9, September 1997, pp. 908-926.

[36] J. Jonsson, J. Vasell, “Real-Time Scheduling for Pipelined Execution of Data Flow Graphs on a Realistic Multiprocessor Architecture”, In Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), May 7-10, 1996, Atlanta, Georgia, USA, pp. 3314-3317.

[37] J. Jonsson, J. Vasell, “On Objective Function Selection in List Scheduling Algorithms for Digital Signal Processing Applications”, In Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), April 21-24, 1997, Munich, Germany, pp. 667-670.

[38] H. Jung, S. Ha, “Hardware Synthesis from Coarse-Grained Dataflow Specification for Fast HW/SW Cosynthesis”, International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'04), September 08 - 10, 2004.

[39] M. Kafil and I. Ahmed, “Optimal Task Assignment in Heterogeneous Distributed Computing Systems”, IEEE Concurrency, Vol. 6, No. 3, July-September 1998, pp. 42-51.

[40] A. Kalavade and P. A. Suhrahmanyam, “Hardware / Software Partitioning for Multi-function Systems”, *Proc. International Conference on Computer Aided Design*, pp. 516-521, Nov. 1997.

[41] A. Kahn, C. McCreary, J. Thompson, and M. McArdle, “A Comparison of Multiprocessor Scheduling Heuristics”, Proceedings of 1994 International Conference on Parallel Processing, vol. II, pages 243-250, 1994.

[42] M. Katevenis, P. Vatsolaki, and A. Efthymiou. Pipelined memory shared buffer for VLSI switches. In Proc. Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 1995.

[43] J. Keinert, C. Haubelt, and J. Teich. Windowed Synchronous Data Flow. Department of Computer Science 12, Hardware-Software-Co-Design, University of Erlangen-Nuremberg, Am Weichselgarten 3, D-91058 Erlangen, Germany Co-Design-Report 02-2005.

[44] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. The Bell System Technical Journal, pages 291--307, February 1970.

[45] M. Khandelia and S. S. Bhattacharyya. Contention-conscious transaction ordering in embedded multiprocessors. In Proceedings of the International Conference on

Application Specific Systems, Architectures, and Processors, pages 276-285, Boston, Massachusetts, July 2000.

[46] B. Kienhuis and E. F. Deprettere. Modeling stream-based applications using the SBF model of computation. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 385-394, September 2001.

[47] S. J. Kim and J. C. Brown, “A general approach to mapping of parallel computations upon multiprocessor architectures”, proceedings of International Conference on Parallel Processing, August 1988, Vol. 3, pp. 1-8.

[48] D. Ko and S. S. Bhattacharyya. Modeling of block-based DSP systems. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 381-386, Seoul, Korea, August 2003.

[49] D. Ko and S. S. Bhattacharyya. Dynamic configuration of dataflow graph topology for DSP system design. In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, pages V-69-V-72, Philadelphia, Pennsylvania, March 2005.

[50] D. Ko, S. S. Bhattacharyya. Modelling and Optimization of Buffering Trade-offs for Hardware Implementation of Image Processing Applications, In Proceedings of the IEEE Workshop on Signal Processing Systems, pages 591-596, Athens, Greece, November 2005.

[51] D. Ko and C. Shen and S. S. Bhattacharyya and N. Goldsman. Energy-driven partitioning of signal processing algorithms in sensor networks, In Proceedings of the SAMOS Workshop on Wireless Sensor Networks, Samos, Greece, July 2006[To appear].

[52] M. Ko, P. K. Murthy, and S. S. Bhattacharyya. Compact procedural implementation in DSP software synthesis through recursive graph decomposition. In *Proceedings of the International Workshop on Software and Compilers for Embedded Processors*, Amsterdam, The Netherlands, September 2004.

[53] K. Konstantinides, R. T. Kaneshiro, and J. R. Tani, "Task Allocation and Scheduling Models for Multiprocessor Digital Signal Processing," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 38, no. 12, pp. 2151{2161, Dec. 1990.

[54] R. Kumar, V. Tsiatsis, and M. B. Srivastava, Computation Hierarchy for In-network Processing, the 2nd ACM international conference on Wireless sensor networks and applications, pp. 68.77, 2003.

[55] Y. K. Kwok, Ishfaq Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Computing Surveys (CSUR), v.31 n.4, p.406-471, Dec. 1999.

[56] K. Lahiri, Anand Raghunathan, Sujit Dey, Fast performance analysis of bus-based system-on-chip communication architectures, Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design, p.566-573, November 07-11, 1999, San Jose, California, United States.

[57] G. De Micheli, Synthesis and Optimization of Digital Circuits, McGraw Hill Inc., New Jersey, 1994.

[58] K. N. Lalgudi, M. C. Papaefthymiou, and M. Potkonjak. Optimizing computations for effective block-processing. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):604-630, July 2000.

[59] M. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation, pp. 318-328, June 1988.

[60] E. A. Lee and J. C. Bier, Architectures for Statically Scheduled Dataflow, Journal of Parallel and Distributed Computing, Vol. 10, pp. 333-348, December 1990.

[61] E. A. Lee, A Coupled Hardware and Software Architecture for Programmable DSPs, Ph. D. Thesis, Department of EECS, University of California Berkeley, May 1986.

[62] E. A. Lee. Overview of the Ptolemy project. Technical Report UCB/ERL M01/11, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, March 2001.

[63] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transactions on Computers*, February 1987.

[64] E. A. Lee and S. Ha, Scheduling Strategies for Multiprocessor Real Time DSP, Proceedings of the Global Telecommunications Conference, November 1989.

[65] Y. Li, W. Wolf, “Hardware/Software Co-Synthesis with Memory Hierarchies, “IEEE Transaction Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 10, pp. 1405--1417, 1999.

[66] S. Lindsey, C. Raghavendra, and K. Sivalingam, Data Gathering in Sensor Networks using the Energy Delay Metric. IEEE Transactions on Parallel and Distributive Systems, special issue on Mobile Computing, pp. 924-935, April 2002

[67] J. Madsen, P. Bjorn-Jorgensen. Embedded System Synthesis under Memory Constraints. Proceedings of the seventh international workshop on Hardware/software codesign table of contents. Rome, Italy. Pages: 188 - 192. 1999.

[68] S. Meftali, Ferid Gharsalli, Frederic Rousseau, Ahmed A. Jerraya, An optimal memory allocation for application-specific multiprocessor system-on-chip, Proceedings of the international symposium on Systems synthesis, September 30-October 03, 2001, Montreal, P.Q., Canada.

[69] L. J. Miller, “A heterogeneous multiprocessor design and the distributed scheduling of its task group workload”, Proceedings of the 9th annual symposium on Computer Architecture, p.283-290, April 26-29, 1982, Austin, Texas, United States.

[70] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064-2079, August 2002.

[71] P. K. Murthy and E. A. Lee, “On the Optimal Blocking Factor for Blocked, Non-Overlapped Schedules,” ERL Memo No. UCB/ERL M94/93, Electronics Research Lab, November 1994, UCBerkeley, CA.

[72] A. E. Nelson, “Implementation of Image Processing Algorithms on FPGA hardware”, MS Thesis, Vanderbilt University, May 2000.

[73] S. Neuendorffer and Edward Lee, “Hierarchical Reconfiguration of Dataflow Models”, Conference on Formal Methods and Models for Codesign (MEMOCODE), June 22-25, 2004.

[74] C. Nicolescu, P. Jonker, A data and task parallel image processing environment. Parallel Computing 28(7-8): 945-965 (2002).

[75] M. Pankert, O. Mauss, S. Ritz, H. Meyr, “Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis,” Proceedings of the 1994 IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 2, pp 449-452, Adelaide, Australia, April 19-22, 1994.

[76] C. Park, J. Chung and S. Ha, Efficient Dataflow Representation of MPEG-1 Audio (Layer III) Decoder Algorithm with Controlled Global States, IEEE Workshop on Signal Processing Systems (SiPS): Design and Implementation, Taiwan, ROC, Oct, 1999.

[77] Thomas M. Parks, Jose Luis Pino and Edward A. Lee, “A Comparison of Synchronous and Cyclo-Static Dataflow”, Proc. IEEE Asilomar Conference on Signals, Systems, and Computers, Nov., 1995.

[78] J. Pino, S. S. Bhattacharyya, E. A. Lee, “A Hierarchical Multiprocessor Scheduling System for DSP Applications,” Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers, October 1995.

[79] M. Potkonjak, M. Srivastava, “Design of High Throughput, Low Latency and Low Cost Structures for Linear Systems”, ICASSP-94 International Conference on Acoustic, Speech, and Signal Processing, Minneapolis, MN, Vol. 2, pp. 497-500, April 1994.

[80] S. Ranaweera, D. P. Agrawal, “A Task Duplication Based Scheduling Algorithm for Heterogeneous Systems”, 14th International Parallel and Distributed Processing Symposium (IPDPS'00), p 445-450, May 01 - 05, 2000, Cancun, Mexico.

[81] J. Rehg, Kathleen Knobe, Umakishore Ramachandran, Rishiyur S. Nikhil, Arun Chauhan, Integrated Task and Data Parallel Support for Dynamic Applications,

Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, p.167-180, May 28-30, 1998.

[82] H. E. Rewini, T. G. Lewis, “Scheduling parallel programs onto arbitrary target architectures”, Journal of Parallel and Distributed Computing, Vol. 9, No. 2, June 1990, pp. 138-153.

[83] S. Ritz, M. Pankert, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application Specific Array Processors*, October 1993.

[84] S. Ritz, M. Pankert, and H. Meyr. High level software synthesis for signal processing systems. In *Proceedings of the International Conference on Application Specific Array Processors*, August 1992.

[85] V. Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessors. Cambridge, MA. The MIT Press, 1989.

[86] P. L. Shaffer, “Minimization of Interprocessor Synchronization in Multiprocessors with Shared and Private Memory,” International Conference on Parallel Processing, 1989.

[87] Luc Semeria, K. Sato, G. Micheli, “Synthesis of hardware models in C with pointers and complex data structures”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems archive Volume 9, Issue 6 (December 2001), Pages: 743 - 756.

[88] E. Shih, S. Cho, N. Ickes, R. Min, A. Sinha, A. Wang, and A. Chandrakasan, “Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks”, in Proc. ACM MOBICOM'01, July 2001.

[89] G. C. Sih and E. A. Lee, “A compile time scheduling heuristic for interconnection-constrained heterogeneous processors architectures”, IEEE Transactions on Parallel and Distributed Systems,Vol. 4, No. 2, February 1993, pp. 175-187.

[90] G. C. Sih and E. A. Lee, “Declustering: A New Multiprocessor Scheduling Technique,” IEEE Trans. on Parallel and Distributed Systems, vol. 4, No. 6, pp. 625-637, June 1993.

[91] M. Singh and V. K. Prasanna, System-Level Energy Tradeoffs for Collaborative Computation in Wireless Networks Norwell, MA: Kluwer, 2002.

[92] G. Spivey, S. S. Bhattacharyya, K. Nakajima, “A Component Architecture for FPGA-based, DSP System Design”, In Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors, San Jose, California, July 2002.

[93] A. Srinivasan, J. H. Anderson, Optimal rate-based scheduling on multiprocessors, Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, May 19-21, 2002, Montreal, Quebec, Canada.

[94] S. Sriram, MINIMIZING COMMUNICATION AND SYNCHRONIZATION OVERHEAD IN MULTIPROCESSORS FOR DIGITAL SIGNAL PROCESSING, Ph.D. Dissertation, Dept. of EECS, Technical Report UCB/RL 95/90 University of California, Berkeley, October, 1995.

[95] S. Sriram and S. S. Bhattacharyya, Embedded Multiprocessors: Scheduling and Synchronization, Marcel Dekker Inc., 2000.

[96] S. Sriram and E. A. Lee, “Determining the Order of Processor Transactions in Statically Scheduled Multiprocessors”, Journal of VLSI Signal Processing, pp. 207-220, 1997.

[97] Siram and E. A. Lee, “Statically Scheduling Communication Resources in Multi-processor DSP Architectures,” Proceedings of the Asilomar Conference on Signals, Systems, and Computers, November, 1994.

[98] M.B. Srivastava, M. Potkonjak, “Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pp 2-19, Mar 1995.

[99] J. Subhlok, G. Vondran. Optimal Latency-Throughput Tradeoffs for Data Parallel Pipeline. *SPAA, 1996, Padua, Italy.*

[100] R. Szymanek, K. Kuchcinski, A constructive algorithm for memory-aware task assignment and scheduling, Proceedings of the ninth international symposium on Hardware/software codesign, p.147-152, April 2001, Copenhagen, Denmark.

[101] K. Taura, A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In Heterogeneous Computing Workshop, May 2000.

[102] N. Thepayasuwan, A. Doboli, “OSIRIS: Automated Synthesis of Flat and Hierarchical Bus Architectures for Deep Submicron Systems-on-Chip”, IEEE Proceedings of International Symposium on VLSI (ISVLSI), Lafayette, 2004.

[103] J. Teifel and R. Manohar. An Asynchronous Dataflow FPGA Architecture. IEEE Transactions on Computers, Special issue on Field-Programmable Logic, November 2004.

[104] A. Turjan, B. Kienhuis, Ed F. Deprettere: An Integer Linear Programming Approach to Classify the Communication in Process Networks. SCOPES 2004: 62-76

[105] N. Vanspauwen, E. Barros, S. Cavalcante, C. Valderrama, “On the Importance, Problems and Solutions of Pointer Synthesis”, Proceedings of the 15th symposium on Integrated circuits and systems design, pp: 317, 2002.

[106] G. Varatkar, R. Marculescu, Communication-Aware Task Scheduling and Voltage Selection for Total Systems Energy Minimization. Proc. Intl. Conf. on Computer-Aided Design (ICCAD), November, 2003.

[107] I.Verbauwhede, F.Catthoor, J.Vandewalle, H.De Man, “Background memory management for the synthesis of algebraic algorithms on multi-processor DSP chips”, Proc. VLSI'89, Int. Conf. on VLSI, Munich, Germany, pp.209-218, Aug. 1989.

[108] Z. Wang, M. Kirkpatrick, Edwin Hsing-Mean Sha, “Optimal two level partitioning and loop scheduling for hiding memory latency for DSP applications”, Proceedings of the 37th conference on Design automation, p.540-545, June 05-09, 2000, Los Angeles, California, United States.

[109] P. Wauters, M. Engels, R. Lauwereins, J.A. Peperstraete, “Cyclo-dynamic dataflow,”4th EUROMICRO Workshop on Parallel and Distributed Processing, Braga,Portugal, January, 1996.

[110] W. Wolf and M. Kandemir, “Memory system optimization of embedded software,” Proceedings of the IEEE, 91(1), January 2003, pp. 165-182.

[111] T. Yang and A. Gerasoulis, “A Fast Scheduling Algorithm for DAGs on an Unbounded Number of Processors,” Proceedings of the 5th ACM International Conference on Supercomputing, pages 633-642. ACM 1991.

[112] T. Y. Yen, Wayne Wolf, Communication synthesis for distributed embedded systems, Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design, p.288-294, November 05-09, 1995, San Jose, California, United States.

[113] Q. Zhuge, B. Xiao, E. H.-M. Sha and C. Chantrapornchai Efficient Variable Partitioning and Scheduling for DSP Processors with Multiple Memory Modules, IEEE Transactions on Signal Processing, Vol. 52, No. 4, April 2002, pp. 1090-1099.

[114] X. Zhu, S. Malik. A Hierarchical Modeling Framework for On-Chip Communication Architectures. Proceedings of International Conference on Computer-Aided Design 2002, November, 2002.

[115] E. Zitzler, J. Teich, and S. S. Bhattacharyya. Evolutionary algorithms for the synthesis of embedded software. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 8(4):452-456, August 2000.

[116] V. Zivojnovic, H. Koerner, and H. Meyr, “Multiprocessor Scheduling with A-priori Node Assignment,” VLSI Signal Processing, IEEE Press, 1994.