HOPES: Programming Platform Approach for Embedded Systems Design

29

Soonhoi Ha and Hanwoong Jung

Abstract

Hope Of Parallel Embedded Software (HOPES) is a design environment for embedded systems supporting all design steps from behavior specification to code synthesis, including static performance estimation, design space exploration, and HW/SW cosimulation. Distinguished from other design environments, it introduces a novel concept of "programming platform" called Common Intermediate Code (CIC), which can be understood as a generic execution model of heterogeneous multi-processor architecture. In the CIC model, each application is specified by a multi-mode Synchronous Data Flow (SDF) graph, called MTM-SDF. Each mode of operation is specified by an SDF graph and mode transition is expressed by an Finite-State Machine (FSM) model. It enables a designer to estimate the performance and resource demand by constructing static schedules of the application with varying number of allocated processing elements at each mode. At the top level, a process network model is used to express concurrent execution of multiple applications. A special process, called control task, is introduced to specify the system-level dynamism through an FSM model inside. With a given CIC model and a set of candidate target architectures, HOPES performs design space exploration to choose the best HW/SW platform, assuming that a hybrid mapping policy is used to map the applications to the processing elements. HOPES synthesizes the target code automatically from the CIC model with the mapping information. The overall design flow is verified by the design of two real-life examples.

S. Ha (✉)
Department of Computer Science and Engineering, Seoul National University, Gwanak-gu, Seoul, Korea
e-mail: sha@snu.ac.kr

H. Jung
Seoul National University, Gwanak-gu, Seoul, Korea
e-mail: jhw7884@gmail.com

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_1

Acronyms

API	Application Programming Interface
BDF	Boolean Data Flow
CIC	Common Intermediate Code
FSM	Finite-State Machine
GA	Genetic Algorithm
GUI	Graphical User Interface
KPN	Kahn Process Network
MTM	Mode Transition Machine
NoC	Network-on-Chip
OS	Operating System
PIM	Platform Independent Model
PSDF	Parameterized Synchronous Data Flow
QoS	Quality of Service
SADF	Scenario-Aware Data Flow
SDF	Synchronous Data Flow
SMP	Symmetric Multi-Processing
SysteMoC	SystemC Models of Computation
WCRT	Worst-Case Response Time

Contents

29.1 Introduction 952
29.2 Common Intermediate Code (CIC) Model 957
29.2.1 Extended SDF Model for Application Specification 959
29.2.2 Dynamic Behavior Specification at the Top-Level Specification of the CIC Model 964
29.3 Design Space Exploration in HOPES 966
29.3.1 Static Scheduling Technique of an MTM-SDF Graph 968
29.3.2 Dynamic Mapping 972
29.4 CIC Translator: Automatic Code Synthesis from the CIC Model 973
29.5 Experimental Results 975
29.6 Current Status and Conclusion 978
References 979

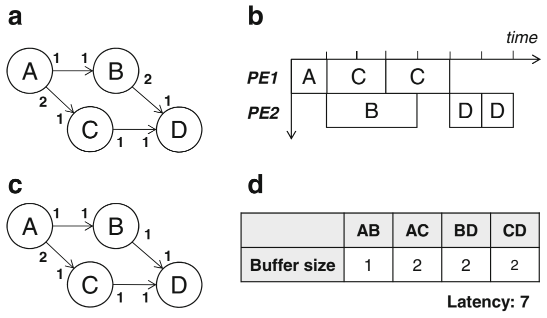
29.1 Introduction

HOPES is under development as a generic design environment to support a wide range of embedded system architectures from system on a chip (SoC) to networked embedded systems. Starting from a target-independent behavior specification and a given set of candidate hardware architectures and available processing elements, we can explore the design space to find an optimal system configuration and mapping of applications, and synthesize the software and hardware components in a unified framework. The abstract target architecture assumed in HOPES consists of heterogeneous processing elements that are connected through a network. HOPES was originally introduced as a parallel programming environment for nontrivial heterogeneous multiprocessors with various design constraints on hardware cost, power, and real-time performance [15]. However, HW/SW codesign can be naturally supported by HOPES, since a hardware IP can be regarded as a processing element.

As the system complexity incessantly grows with more processing elements integrated, design reuse of hardware platforms and IPs becomes the de facto practice to mitigate the difficulty of hardware validation. Then the HW/SW codesign methodology is transformed to an embedded SW development methodology for a given hardware platform. Since the proportion of software components keeps increasing, HOPES puts more emphasis on the implementation of software components unlike our previous HW/SW codesign environment, PeaCE (Ptolemy extension as a Codesign Environment) [7]. While PeaCE focuses on the codesign of hardware and software modules that includes HW/SW partitioning, HW/SW cosynthesis, and HW/SW cosimulation, it takes little account of multi-processor architecture that heavily affects the parallel execution of software.

A systematic design methodology can be understood as a sequence of steps that refine a higher level of abstraction to a lower level from initial specification to final implementation, which is summarized as the following phrase: "design is to represent". Since refinement keeps the properties of the higher abstraction, how to specify the behavior is a key factor to distinguish various HW/SW codesign methods. Actor models that specify an application as a set of concurrent actors are widely adopted in the HW/SW codesign methodology since they express the potential parallelism of an application explicitly and parallelizing an application can be simply realized by mapping actors to the processing elements. Actors are connected to each other through channels that represent the flow of data samples between actors. Among many actor models, Synchronous Data Flow (SDF) is chosen as the baseline actor model of HOPES since it is a formal model that enables us to evaluate each design decision through static analysis. General introduction to dataflow models can be found in ▸ Chap. 3, "SysteMoC: A Data-Flow Programming Language for Codesign".

In the SDF model [17], an application is specified with a data-flow graph where a node represents a function module, or a task, and an arc is a FIFO queue that delivers data samples from an output port of the source node to an input port of the destination node. An input (or an output) port is associated with an integer number that indicates how many samples to consume (or to produce) per task execution; the number is called the sample rate of the port. Figure 29.1a shows a simple SDF graph representation of an application. A node becomes runnable only when all input arcs have no fewer samples queued than the specified sample rate. And the sample rates are fixed at run time in the SDF model. Then we can determine the mapping and scheduling of the SDF graph, which is to determine where and in what order to execute the tasks on a given target architecture, at compile time. For each arc, we can determine the relative execution rates between the source task and the destination task, comparing the output sample rate of the source port and the input sample rate of the destination port. For instance, the execution rate of task C should be twice



	AB	AC	BD	CD
Buffer size	1	2	2	2

Fig. 29.1 (**a**) An example SDF graph with annotated sample rates on the arcs, (**b**) a mapping and scheduling result of the SDF graph onto two processing elements, (**c**) an example SDF graph that has a buffer overflow error, and (**d**) the buffer requirement of each arc and the estimated latency for the static scheduling result of (**b**)

higher than that of task *A* in Fig. 29.1a, in order to make the number of samples produced from the source task the same as the number of samples consumed by the destination task. An *iteration* of an SDF graph is defined by the set of task executions that satisfy the relative execution rates of tasks with minimum number of executions.

An SDF graph is said to be *consistent* if the relative execution rates of tasks are satisfied for all arcs. In case there is any possibility of buffer overflow on any FIFO arc or deadlock, we cannot find a valid schedule of an SDF graph. Figure 29.1b shows an example of a static scheduling result on a target architecture with two processing elements. For a consistent SDF graph, we can repeat the schedule iteratively without buffer overflow. Figure 29.1c shows an SDF graph example that has a buffer overflow error on arc *AC* by giving a wrong sample rate at the output port of node *B*. Such static analyzability is a very desirable feature for embedded system design since it enables us to detect a class of design errors by static analysis [17]. Moreover, once the mapping and scheduling decision is made, we can determine the buffer requirements for all arcs and estimate the real-time performance of the application. We can easily check whether the design constraints on the hardware requirement and real-time performance can be satisfied or not. For instance, the buffer requirement and the estimated latency associated with the static scheduling result of Fig. 29.1b is summarized in Fig. 29.1d.

While the SDF model has the aforementioned benefits from its static analyzability, it has severe limitations to be used as a general model for behavior specification. First, it is not possible to specify the dynamic behavior of an application since the sample rate of a port may not change dynamically. Second, it does not allow the use of shared memory for inter-node communication since the access order to the shared memory may change depending on the execution order of nodes. So the synthesized code may require much larger memory than a manually written code that usually uses shared variables for communication between function modules. To overcome those limitations, we have proposed several extensions to the SDF model in HOPES.

We use the Finite-State Machine (FSM) model in combination with the SDF model to express the dynamic behavior of an application [10]. Furthermore, a special actor, called library actor [20], is introduced to handle shared resources efficiently without side effects.

Following the heritage of heterogeneous modeling of Ptolemy [3] and PeaCE [7], HOPES uses a process network model at the top level to express concurrent execution of multiple applications. An application is modeled as a single process at the top level while the internal behavior of an application is specified by the extended SDF model. The system-level dynamic behavior is specified by a control task whose behavior is specified by the FSM model in the top-level task graph. The overall specification model of HOPES is called the Common Intermediate Code (CIC) model, which will be explained in the next section in detail.

There is a clear distinction between HOPES and the other model-based design environments. As the name implies, the CIC model is not defined as a front-end specification model, but an intermediate specification model, meaning that HOPES design environment can accommodate various front-end specification models as long as the front-end specification model can be translated into the CIC model. In fact, the CIC model can be understood as an *execution* model of tasks at the Operating System (OS) level. At the OS level, the system behavior is represented as a set of tasks no matter what the front-end specification model is. Communication and synchronization between tasks and scheduling of tasks are heavily dependent upon the underlying software platform and hardware platform. In HOPES, we propose to define the execution model of tasks at the OS level and enforce the system to keep the semantics of the execution model. Then, the CIC model will be able to run on any hardware and software platform since the execution model is defined as platform-independent. So, we introduce a new notion of *programming platform* that hides the underlying software and hardware platform from the application programmer. As a program based on the von Neumann execution model can be run on any von Neumann processor, any program based on the CIC execution model can be run on any target architecture that keeps the CIC execution model, we envision. Since the CIC model is based on formal models of computation, we can enjoy the benefits of static analyzability of those models to reduce the design time and cost.

Even though the same SDF model is used for behavior specification in HOPES and PeaCE, the granularity of a node is quite different. In HOPES, an SDF node is a unit of mapping and scheduling at the OS level. It implies that the node granularity should be as large as a thread to make the thread switching overhead insignificant. On the other hand, PeaCE assumes mixed granularity of SDF nodes in the initial specification of an application and clusters them to define a thread or a task at the code synthesis step. In our previous work [15], an SDF graph specification of PeaCE has been translated to the CIC model by clustering the nodes to increase the granularity while keeping the potential parallelism as much as possible. It is possible to specify the system behavior with the CIC model manually, regarding the CIC model as the front-end specification. In this case, it is the responsibility of the programmer to define the granularity of the node to trade-off the parallelism and the scheduling overhead.

In the conventional model-driven software development approaches, the system behavior is specified by a Platform Independent Model (PIM) that is translated into a platform-specific model (PSM) manually for a given hardware platform. Then the target code is generated from the translated PSM. Even though the CIC model is a platform-independent model, there is no need to translate it to a PSM since the CIC model can be executed in any hardware platform that supports the proposed execution model. The HOPES framework is distinguished from other model-based design frameworks that use a specific model of computation for behavior specification, which include Daedalus [19], DAL [21], CompSOC [6], and Koski [12]. On the other hand, as mentioned above, HOPES does not assume any specific model for behavior specification as long as it can be translated into the CIC model. Even though the CIC model is based on three different models of computation, its model composition rule is different from that of Ptolemy [3] which allows hierarchical composition of models without limitation on the depth of hierarchy and on the kinds of models.

Figure 29.2 shows the overall design flow in HOPES. The input information consists of the front-end specification of system behavior and the set of candidate platforms and hardware components. As explained above, the CIC model is generated manually or by an automatic translator from a different front-end specification of system behavior. We perform static analysis at the CIC level to detect the buffer overflow and deadlock errors for SDF specifications and to analyze the timing requirements that will be expressed in the proposed FSM model. The next step is to explore the architectural design space by selecting the hardware platform and processing elements and mapping the applications to the target architecture. Note that the profiling information of tasks for all kinds of candidate processing elements is assumed to be given. The DSE step produces a handful of selected target architectures and associated mapping results of applications. Note that if a target

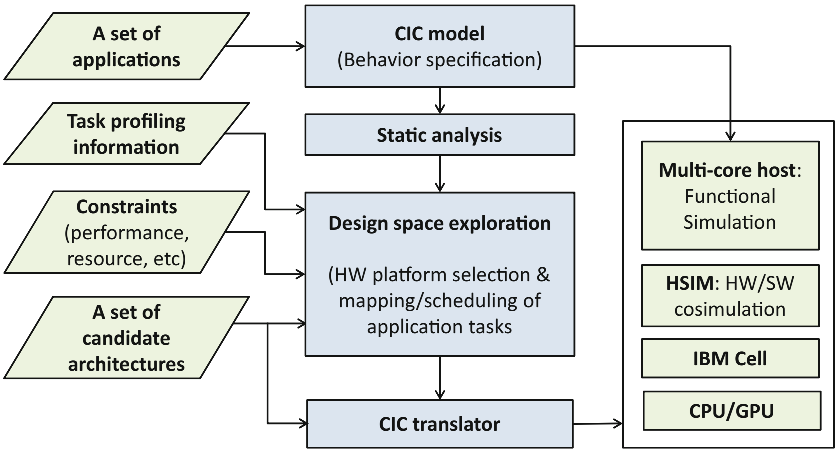

A set of applications
CIC model (Behavior specification)
Task profiling information
Static analysis
Constraints (performance, resource, etc)
Design space exploration
(HW platform selection & mapping/scheduling of application tasks
A set of candidate architectures
CIC translator
Multi-core host: Functional Simulation
HSIM: HW/SW cosimulation
IBM Cell
CPU/GPU

Fig. 29.2 Design flow of HOPES

architecture is given as an input, it just determines the mapping of applications to the target architecture.

For each candidate solution, the CIC translator generates a target C code for each processor. We need to develop a separate CIC translator for each target architecture as we need a different compiler for a different von Neumann processor. A multi-core host processor is a base target platform for functional simulation. The CIC translator generates a multi-threaded C code for functional simulation. Another target platform that HOPES supports is the parallel simulator, called HSIM [26], that has been developed to simulate the target architecture without real hardware platform. A handful of selected architectures will be evaluated more accurately by HSIM simulation. Other target platforms that HOPES supports will be explained later.

The rest of this chapter is organized as follows. The CIC model will be explained in the next section, which will be followed by Sect. 29.3 that explains the mapping and scheduling techniques of the CIC model. Section 29.4 explains the CIC translator. Preliminary experimental results will be discussed in Sect. 29.5. The current status of HOPES development is presented with concluding remarks in Sect. 29.6.

29.2 Common Intermediate Code (CIC) Model

As explained above, the proposed CIC model adopts a hierarchical composition of different models of computation to express the system behavior at two different levels. At the top level, the CIC model expresses the system behavior with a process network. If an application can be specified by the extended SDF graph, the application is encapsulated as a super node that contains an extended SDF graph at the bottom level. Note that the top-level process network and the extended SDF model themselves can be specified in a hierarchical fashion.

The top-level process network consists of CIC tasks and channels as depicted in Fig. 29.3. There are two types of CIC tasks depending on the triggering condition of tasks: *time-driven* and *data-driven*. A time-driven task is triggered by a pre-defined period that is given as a parameter. So it consumes the most recent sample from the input buffer channel. The input channels of a time-driven task are single-entry buffers that store the most recent data samples. An I/O task that interfaces with

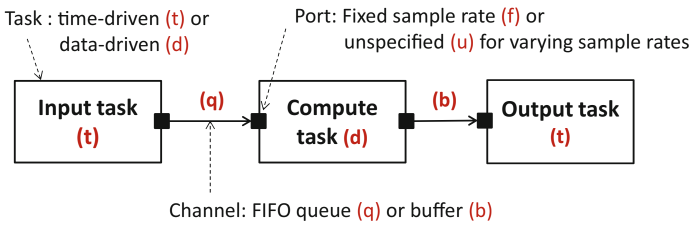

Task : time-driven (t) or
data-driven (d)
Port: Fixed sample rate (f) or
unspecified (u) for varying sample rates
Input task
(t)
(q)
Compute
task (d)
(b)
Output task
(t)
Channel: FIFO queue (q) or buffer (b)

Fig. 29.3 CIC task graph

```
TASK_INIT{ /* task initialization code */ };

TASK_GO {
   /* generic API for data read from an input port */
   MQ_RECEIVE(port_name, data, size);
   ...
   /* generic API for system service request */
   SYS_REQ(command, argument_list);
   ...
   /* generic API for data write to an output port */
   MQ_SEND(port_name, data, size);
}

TASK_WRAPUP { /* task wrapup code */ };
```

Fig. 29.4 CIC task code template

the outside is usually designated as a time-driven task. On the other hand, a data-driven task is triggered by the arrival of data samples on the input ports. The input channels of data-driven tasks are assumed to be FIFO queues. A data-driven task basically follows the semantics of the Kahn Process Network (KPN) model that performs blocking read and non-blocking write access to the channels.

As shown in Fig. 29.4, the code template of a CIC task consists of three sections, enclosed by three keywords, TASK_INIT, TASK_GO, and TASK_WRAPUP. As the name implies, the TASK_INIT section is executed when the task is initialized and the TASK_WRAPUP section is executed just before it is terminated. The TASK_GO function is the main body that will repeat until the task is terminated. A CIC task accesses a channel with target-independent generic APIs, MQ_SEND and MQ_RECEIVE. The MQ_RECEIVE API performs blocking read operation to the associated input port while the MQ_SEND API performs non-blocking write operation to the associated output port. Since the CIC model is defined at the OS level, the CIC task assumes that there is a supervisor that schedules the CIC tasks and provides supervisory services to the CIC tasks. Thus, we define another generic API, SYS_REQ, that requests a service to the supervisor. The first argument of the SYS_REQ API defines the service command whose list will be shown later. In principle, a CIC task does not use platform-specific APIs for portability. The generic APIs will be translated into target-specific APIs at the code generation step. We may define a CIC task that uses platform-specific APIs for efficient implementation at the expense of portability.

The number of data samples consumed or produced per execution of a task can be specified explicitly for each input or output port. The sample rate is specified, if it is fixed and not changing at run time. Otherwise, the sample rate is assumed to be varying at run time. If the input sample rates of all input ports are specified, the data-driven task becomes an SDF task that follows the execution semantics of the SDF model. If all tasks in a CIC subgraph are SDF tasks, the CIC subgraph becomes an SDF subgraph. Since the SDF model has many merits from static analyzability, it is highly recommended to identify SDF subgraphs as much as possible at the top level until no more SDF subgraph can be identified. And each subgraph is replaced by a super node at the top level to make it a two-level hierarchical graph. To alleviate the difficulty of identifying the SDF subgraph automatically, it is recommended to specify an application with the extended SDF model manually inside a super node.

29.2.1 Extended SDF Model for Application Specification

In this subsection, we explain a couple of extensions that are made to overcome the limitations of the SDF model while preserving the benefits of static analyzability. The first extension is to use the FSM model to express the dynamic behavior of the application. The second is to introduce a special actor, called library actor, to allow tasks to share HW or SW resources.

29.2.1.1 Dynamic Behavior Specification

There exist several approaches that have been proposed to increase the expression capability of the SDF model to support intra-application dynamism. One approach is to extend the SDF model itself. Dynamic Data Flow (DDF) and Boolean Data Flow (BDF) are two examples of this approach where they introduce special kinds of nodes that may have varying sample rates [2]. Since BDF was proven to be Turing equivalent and DDF is a super set of BDF, their expression capability is maximal in theory. But they compromise some benefits of static analysis and efficient implementation.

Another approach is to express the dynamism of an application as a set of *mode*s that the application takes and each mode is specified by an SDF graph. This approach assumes that the number of possible dynamic behaviors, or modes, is finite. Then the dynamic behavior can be expressed as dynamic mode change. There are several ways to specify mode change. In Parameterized Synchronous Data Flow (PSDF), the dynamic behavior of a task is modeled by parameters, and the mode change is realized by changing the parameters at run time before starting an iteration of a schedule [1]. An application is specified by a tuple of graphs, init graph and body graph, where the body graph specifies the application behavior and the init graph sets the parameter values to change the mode before a new iteration starts.

The other way is to combine the SDF model with another computation model, usually FSM to express the mode change. In the *-chart approach [5], each state of a finite state machine contains an SDF graph inside to make a hierarchical composition of SDF and FSM models. The state change in the FSM can be understood as the mode change of the SDF model. In the SystemC Models of Computation (SysteMoC) approach [9], a task is associated with an FSM that determines the execution behavior of the task. FSM-based SADF, shortly FSM-SADF, is a restricted form of Scenario-Aware Data Flow (SADF) that specifies each mode of operation, called scenario, with an SDF graph [22, 23]. An SDF task may have multiple versions of definition depending on the mode of operation while a special control actor, called detector, that has an FSM inside sends the control information to normal SDF tasks to change the mode of operation.

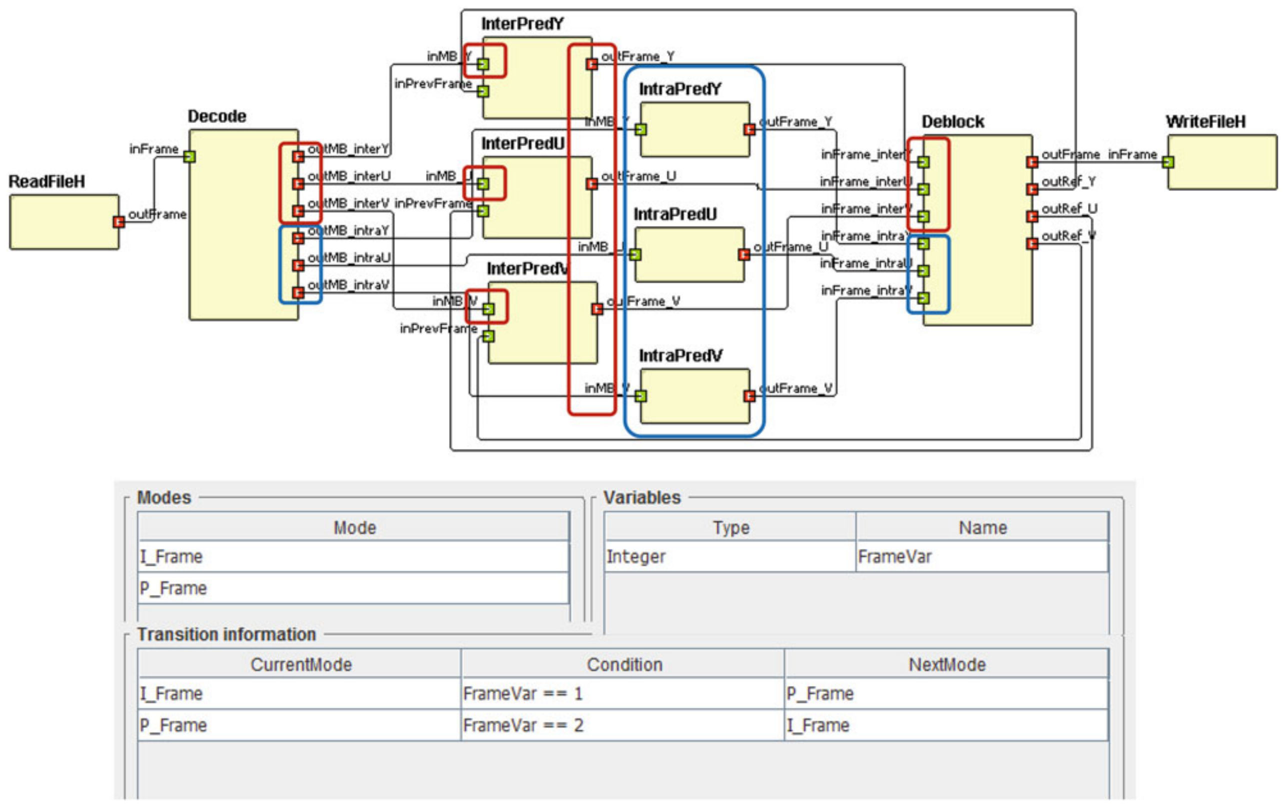
HOPES uses a similar approach as FSM-SADF; an SDF task may have multiple behaviors and a tabular specification of an FSM, called Mode Transition Machine (MTM), describes the mode transition rules for the SDF graph. An MTM is defined as a tuple {*Modes, Variables, Transitions*} where *Modes* and *Variables* represent a set of modes and a set of mode variables respectively, and *Transitions* is a set of transitions that consists of the current mode, a Boolean function of conditions, and the next mode. A Boolean function of transition condition is defined by a simple comparison operation between a mode variable and a value. An MTM-SDF specification of an H.264 decoder is shown in Fig. 29.5. The H.264 decoder has two modes of operation: I-frame and P-frame. In the I-frame mode, the sample rate of each port in red boxes becomes zero while the sample rate of each port in blue boxes becomes zero in the P-frame mode. The MTM is quite simple since it needs to distinguish two modes of operation by a single mode variable. Remind that the granularity of a CIC task is large and the dynamic behavior inside a task is not visible at the CIC level. Thus, an MTM is not complex in general for stream-based applications.

Mode transition is enabled by setting the mode variable so as to satisfy the transition condition. But actual mode transition occurs only at the iteration boundary of the SDF schedule. Since an SDF graph has a well-defined notion of iteration and each task knows how many times it should be executed in each iteration, mode transition can be performed autonomously by individual tasks without global timing synchronization. A mode variable can be set by the hidden supervisor, which will be discussed in the next subsection. Or a designated task may set the mode variable. A stream-based application usually starts with parsing a header information that determines the mode of operation, followed by processing a stream of data. In this case, the SDF task that parses the header information is designated as a special task that may change the mode variable. To satisfy the restriction that the mode transition occurs at the iteration boundary, the designated task should be the first task in the SDF schedule. In the H.264 decoder of Fig. 29.5, *RealFileH* is designated as the special task that determines the mode of operation.

The internal behavior of an SDF task should be defined manually depending on the mode of operation. The code skeleton of an MTM-SDF task is shown in Fig. 29.6; a task first checks the current mode of its MTM before starting the next iteration. If it is designated as a special task, it may change the mode variable. Based on the mode of operation, the sample rates of SDF graph can be changed. For instance, the sample rates for the input and output arcs of *IntraPredY/U/V* tasks become all zero for the P-frame mode and the sample rates for the output arcs of *InterPredY/U/V* tasks become zero for the I-frame mode of operation. Note that the feedback input arcs of *InterPredY/U/V* tasks do not change the sample rates since they need to store the previous frame fed back from the *Deblock* task even in the I-frame mode.

29.2.1.2 Library Task

In addition to dynamic behavior specification, another extension is made to the SDF graph by introducing a special task, called library task, to allow the use of



Modes

Mode
I_Frame
P_Frame

Variables

Type	Name
Integer	FrameVar

Transition information

CurrentMode	Condition	NextMode
I_Frame	FrameVar == 1	P_Frame
P_Frame	FrameVar == 2	I_Frame

Fig. 29.5 An MTM-SDF specification of H.264 decoder: captured screen from the HOPES environment

```
TASK_GO{
    Mode = SYS_REQ(GET_CURRENT_MODE_NAME); // get a current mode
    if Mode == "S1":                       // code for mode S1
      MQ_RECEIVE(port_in, data, size);
      …
      MQ_SEND(port_out, data, size);
    else if Mode == "S2":                  // code for mode S2
      …
    if specific conditions:        // set a variable in an MTM
      SYS_REQ(SET_MTM_PARAM_INT, task_name, var_name, value);
}
```

Fig. 29.6 Code skeleton of an MTM-SDF task

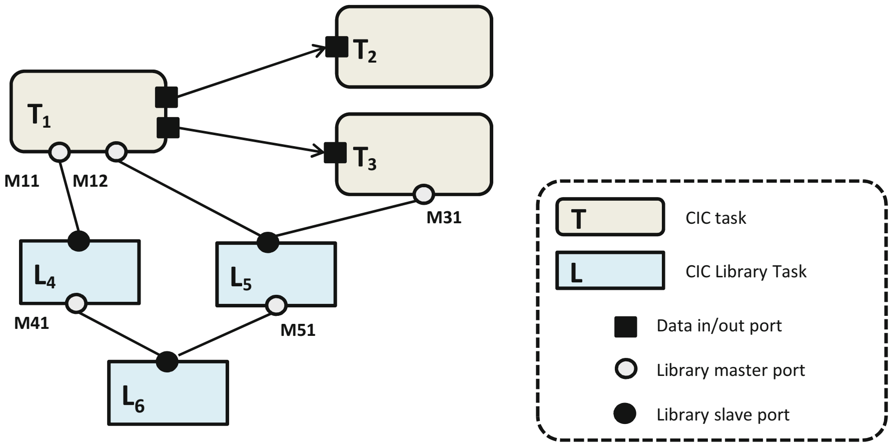
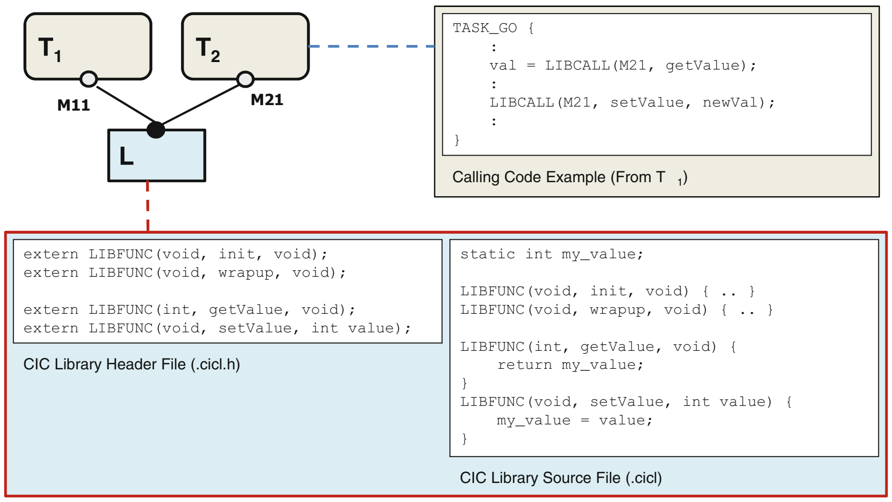


Fig. 29.7 An extended SDF graph that uses library tasks

shared resources in the SDF model. A library task is a sharable and mappable object that defines a set of function interfaces inside. Figure 29.7 shows an SDF graph that consists of three normal SDF tasks (T1–T3) and three library tasks (L4–L6). Connection with a library task is made between a pair of library ports, library master port and a library slave port that are represented by a white circle and a dark circle, respectively. A library task should have a single slave port that can be connected to multiple masters that share the library task. Since each library port has its own type that defines a set of function interfaces, connection between a master port and a slave port can be established only when their types are matched.

Unlike a normal SDF task, a library task is not invoked by input data but by a function call inside an SDF task; it is a passive object. There are specific rules to specify and use a library task in an SDF graph. Figure 29.8 illustrates code templates associated with a library task. A library task has two separate files associated: a library header file and a library code file. The library header file declares the library functions, while the library code file defines the function bodies. The prototype


T_1
T_2
M11
M21
L
Calling Code Example (From T_1)
CIC Library Header File (.cicl.h)
CIC Library Source File (.cicl)

```
TASK_GO {
    :
    val = LIBCALL(M21, getValue);
    :
    LIBCALL(M21, setValue, newVal);
    :
}
```

```
extern LIBFUNC(void, init, void);
extern LIBFUNC(void, wrapup, void);

extern LIBFUNC(int, getValue, void);
extern LIBFUNC(void, setValue, int value);
```

```
static int my_value;

LIBFUNC(void, init, void) { .. }
LIBFUNC(void, wrapup, void) { .. }

LIBFUNC(int, getValue, void) {
    return my_value;
}
LIBFUNC(void, setValue, int value) {
    my_value = value;
}
```

Fig. 29.8 Code templates associated with a library task

of a library function is defined by a directive, LIBFUNC(), that will be translated into a regular function definition automatically by the CIC translator. A library task should define *init* and *wrapup* functions like a normal SDF task for initialization and finalization of the library task.

A caller task uses LIBCALL directive to call a library function as shown in Fig. 29.8. The first parameter of LIBCALL() is the name of the library master port, the second is the function name, and the others are the arguments. If the function has a return value, it can be taken from the LIBCALL invocation. Note that pointers may not be used for arguments and return values to make the SDF graph portable to a variety of target architectures. For shared address space architectures, however, the developer may use pointers for efficient implementation, giving up portability.

A library task may have a persistent internal state, simply called a state. Then the access to the state should be protected by synchronization primitives, Lock() and Unlock() to avoid data race problems. In case multiple masters access the same library task that has a state, the return value of a library function may depend on the execution order of the master actors, which is anathema to any deterministic model. So, we explicitly specify a property of a library task whether it is deterministic or not. In case the library task has no state or returns the same value to the master tasks regardless of the calling order, the library task is classified as "deterministic." Otherwise, the developer should be aware that the library task does not guarantee deterministic behavior. Even though a library actor is nondeterministic in the sense that the return value to a master task depends on the scheduling order of master tasks, the same behavior can be repeated if the same scheduling order is followed.

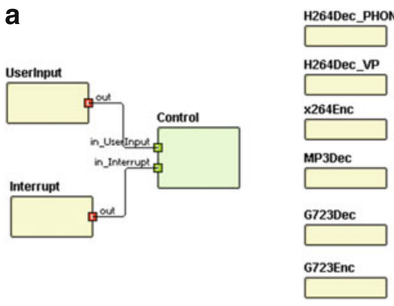
There are several use cases of library task. A library task provides a way to share global variables or HW resources among multiple SDF tasks explicitly in a systematic way. In a server-client application, the server task can be specified by a library task that may be shared by multiple clients. Note that we may change the number of clients arbitrarily since the number of master ports connected to a slave port can vary at run time. Another use case of a library task is to make a vertically layered software structure by providing a set of Application Programming Interfaces (APIs) of the software layer below the application layer.

29.2.2 Dynamic Behavior Specification at the Top-Level Specification of the CIC Model

In this subsection, we explain how to specify the system-level dynamic behavior at the top level of CIC model. At the system level, the set of applications running concurrently may change or applications may change their operation modes according to user requests. Several approaches have been proposed to specify the dynamic behavior of data-flow applications. FunState that was proposed as an internal representation for codesign process [25] uses an FSM to control the activation of data-flow tasks. In STATEMATE [8], an extended FSM model, called statechart, specifies the entire system behavior and determines when to execute each task in the activity chart. Distributed application layer (DAL) [21] uses an FSM to add dynamism to distributed operation layer (DOL) [24] that is based on the KPN model. The FSM model of DAL specifies all use cases and how transitions between use cases occur, where a use case corresponds to a set of applications running concurrently, assuming that the number of use cases of the system is finite.

HOPES inherits the approach of its predecessor, PeaCE [7], where a control task is distinguished from application processes at the top level and plays the role of user-level supervisor that controls the execution status of applications. A control task uses an FSM model inside to specify the system-level dynamic behavior. Consider a simple smartphone example of Fig. 29.9. The system consists of two input processes running in the background, one control task, and six application processes. Each application is specified by an extended SDF graph inside; Fig. 29.5 is the internal specification of H.264 decoder application for instance. Suppose that there are four use cases, modes of operations, for the smartphone system as shown in Fig. 29.9b. In the Menu mode, there is no active application and the system waits for input events to be caught by two input processes, UserInput and Interrupt (phone arrival). Depending on the user input, the system changes the mode of operation and activates the associated applications. When a phone signal is detected, the system suspends the current mode of operation and switches its mode to the VideoPhone mode. After the call is completed, the system goes back to the suspended mode and resumes suspended applications.

The aforementioned description of the dynamic behavior is specified by an FSM inside the control task. Figure 29.10 shows the captured screen for the FSM specification in HOPES and the associated pseudocode automatically generated by the CIC translator. It has four states that correspond to four use cases. The default state is the *Menu* state, denoted by a bold circle. The control task is basically a

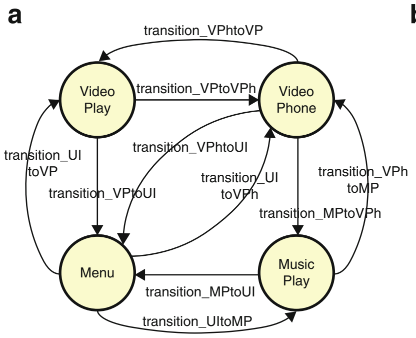


b

Use case	Active applications
Menu	-
VideoPhone	G.723 decoder, G.723 encoder, H.264 decoder (Phone), x.264 encoder
VideoPlay	MP3 decoder, H.264 decoder (VP)
MusicPlay	MP3 decoder

Fig. 29.9 (**a**) A simple smartphone example and (**b**) four use cases of the system

a



b

```
switch (current_state) {
case MENU:
   if(in_UserInput== 1)        next state = VideoPlay;
   else if(in_UserInput ==2)   next state = VideoPhone;
   else if(in_UserInput ==3)   next state = MusicPlay;
case VideoPlay:
   Execute VideoPlay or resume VideoPlay;
   if(in_Interrupt == On)
     Suspend VideoPlay & next state = VideoPhone;
   else if (terminated)
     next state = Menu;
case VideoPhone:
   Execute VideoPhone;
   if (terminated)
     if(previous_state == MusicPlay)
       next state = MusicPlay;
     else if(previous_state == VideoPlay)
       next state = VideoPlay;
…
} // end switch
```

Fig. 29.10 (**a**) FSM specification of the control task in the smartphone of Fig. 29.9 and (**b**) the pseudocode generated by the CIC translator

triggered by an event. There are three kinds of events. The first is an external event that is received from the input port of the control task, which is explicitly drawn at the top level of CIC model. The second kind is generated from the hidden supervisor internally by monitoring the execution status of applications. For instance, the system detects the termination of an application and generates an internal event. The last is a timeout event. The CIC control task can initiate a timer at a certain state. When the specified time is expired, a timeout event is generated by a timer that is another hidden component assumed in HOPES.

At each state, the programmer may use APIs to define the control action, which is similar to action scripts of the statechart in STATEMATE. The control APIs currently defined in HOPES are listed in Table 29.1. The first category is to control the execution status of an application and the second category is to change or monitor a specific parameter of an application. The third category is defined to explicitly specify the timing requirements of the system, and the last category controls the timer modules that are assumed to exist in the system. Since timing correctness is as important as value correctness in system functionality, explicit specification of timing requirement has been recently advocated for real-time

Table 29.1 Control APIs currently defined in HOPES

Category	APIs	Description
Execution status	SYS_REQ({RUN/STOP/SUSPEND/ RESUME} _TASK, task_name)	Run/stop/suspend/resume a task
	status = SYS_REQ(CHECK_TASK_STATE, take_name)	Check the execution status
Parameter control	SYS_REQ(SET_PARAM_{INT/FLOAT}, task_name, param_name, value)	Change the parameter value
	p_value = SYS_REQ(GET_PARAM_{INT /FLOAT}, task_name, param_name)	Get the parameter value
Timing requirement	SYS_REQ(SET_THROUGHPUT, task_name, thr _val)	Set the throughput requirement
	SYS_REQ(SET_DEADLINE, task_name, lat_val , lat_unit)	Set the latency requirement
Timer control	time_base = SYS_REQ(GET_CURRENT_TIME _BASE)	Get the current system time
	timer_id = SYS_REQ(SET_TIMER, time_base, offset)	Set timer to time_base + offset
	ret = SYS_REQ(GET_TIMER_ALARMED, timer_id)	Check if the timer is expired
	SYS_REQ(RESET_TIMER, timer_id)	Reset the timer

embedded system design. While PTIDES [4] uses a discrete event model of computation for timing specification, HOPES uses timing control APIs as a part of control task specification. We may initiate a timer and read the timer. In addition, we may set up the throughput or deadline requirement of an application. Note that the timing requirement of an application may change depending on the use cases. Those timing constraints are referred to in the design space exploration step when constructing the static schedule of an MTM-SDF graph.

29.3 Design Space Exploration in HOPES

As overviewed in Fig. 29.2, HOPES uses a Y-chart approach [13] to explore the design space by mapping applications to candidate architectures with a given set of objectives. Since the dynamic behavior of a system is not predictable, it is challenging to make a mapping decision and evaluate the decision. We have developed a novel hybrid mapping technique that combines compile-time static mapping of applications and run-time dynamic mapping of applications to available resources. Remind that each application is specified by an MTM-SDF graph so that each mode of operation can be statically scheduled. When we schedule each mode of

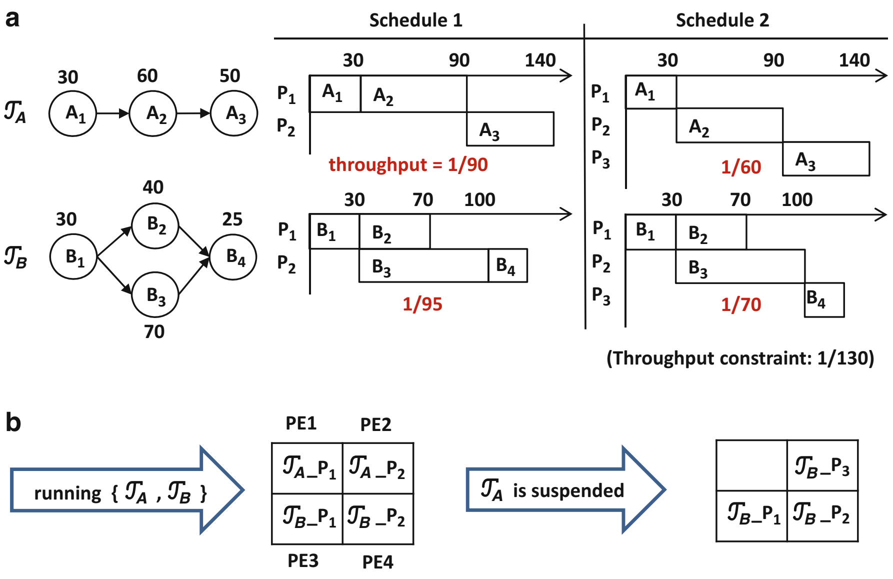

a
Schedule 1
Schedule 2
30 60 50
$\mathcal{T}_A$ A1 A2 A3
40
30 25
$\mathcal{T}_B$ B1 B2 B3 B4
70
30 90 140
P1 P2 P3
throughput = 1/90
1/60
30 70 100
1/95
1/70
(Throughput constraint: 1/130)
b
running { $\mathcal{T}_A$, $\mathcal{T}_B$ }
PE1 PE2 PE3 PE4
$\mathcal{T}_A$_P1 $\mathcal{T}_A$_P2 $\mathcal{T}_B$_P1 $\mathcal{T}_B$_P2
$\mathcal{T}_A$ is suspended
$\mathcal{T}_B$_P3 $\mathcal{T}_B$_P1 $\mathcal{T}_B$_P2

Fig. 29.11 An example of hybrid mapping: (**a**) Pareto-optimal mapping solutions of two applications, and (**b**) dynamic mapping results according to a given scenario of system status change

an application, we find a Pareto-optimal set of mapping decisions for each candidate architecture. Suppose that we have multiple objectives of mapping, minimizing the resource usage and maximizing the throughput performance for instance. Then static scheduling is performed for each application independently to obtain a set of Pareto-optimal solutions for multiple objectives. We assume that no processor sharing is allowed, or a processing element is dedicated to an application, in the current implementation of HOPES.

Figure 29.11 shows a simple example that consists of two applications, each of which has a single mode of operation. For each application, two Pareto-optimal mappings are found with varying number of processing elements. At run time, dynamic mapping is performed by first identifying which applications are running concurrently and next allocating the processing elements to the applications based on their Pareto-optimal mapping solutions. In this example, four processing elements are equally allocated to two applications, two to each. When application *A* is suspended, we reallocate the processing elements to the remaining application, *B*, to improve the throughput performance, which is also a Pareto-optimal mapping of *B*.

Dynamic remapping is triggered at every system status change. Some causes of the system status change are explicitly specified in the CIC model. For instance, the change of execution status or QoS requirement of an application is specified by a state transition defined in a control task. Thus, such a state transition triggers dynamic mapping. The operation mode change of an application is explicitly specified in the MTM-SDF model. There are other causes, however, that are not specified in the CIC model. An example is the failure of a processing element. If a processor failure is detected, remapping of applications is performed [16]. We assume that all system status changes are captured by the hidden supervisor whatever the causes are.

29.3.1 Static Scheduling Technique of an MTM-SDF Graph

Since an application is specified by an MTM-SDF graph, we devised a novel static scheduling technique of an MTM-SDF graph. An important objective is to minimize the mode change overhead that may affect the real-time performance of an application. Figure 29.12 shows a simple MTM-SDF graph that has two modes of operation. When we use a naive technique that schedules each SDF graph independently, we need to migrate three tasks when mode change occurs as illustrated in Fig. 29.12b. It is better to consider the migration overhead when finding a static schedule for each mode. Another extreme approach is to avoid task migration by considering all modes simultaneously; this approach is assumed in the previous work [22]. As shown in Fig. 29.12c, this approach maps a task to the same processor at all modes and so requires more processors. The proposed approach is

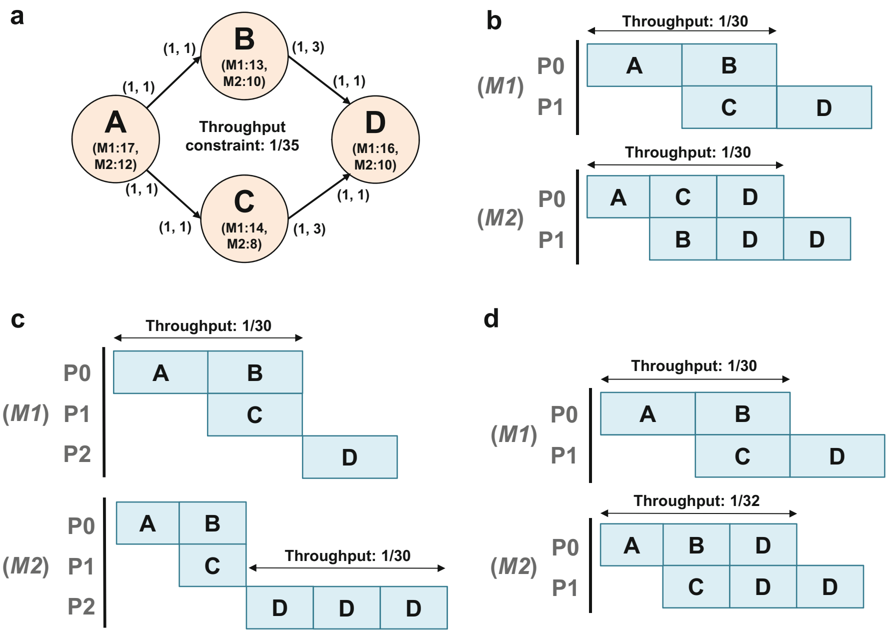


Fig. 29.12 (**a**) An MTM-SDF graph example and scheduling results for three cases: (**b**) scheduling each mode independently, (**c**) scheduling all modes simultaneously disallowing task migration, and (**d**) scheduling all modes simultaneously with task migration to minimize the resource requirement

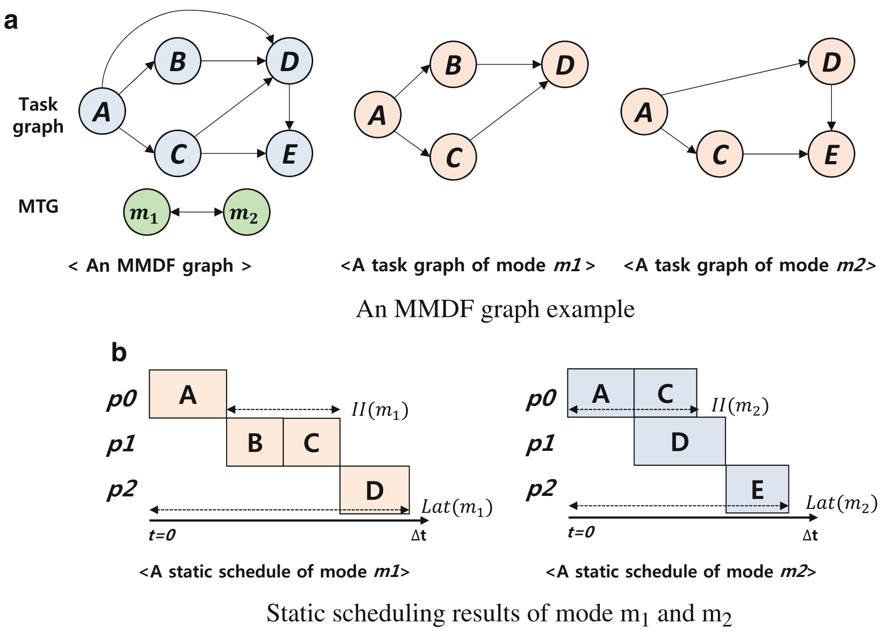

a
Task graph
MTG
A
B
C
D
E
m_1
m_2
< An MMDF graph >
<A task graph of mode m1>
<A task graph of mode m2>
An MMDF graph example
b
p0
p1
p2
$II(m_1)$
$Lat(m_1)$
$II(m_2)$
$Lat(m_2)$
t=0
Δt
<A static schedule of mode m1>
<A static schedule of mode m2>
Static scheduling results of mode m_1 and m_2

Fig. 29.13 An MMDF graph example with two modes of operation and their static scheduling results. (**a**) An MMDF graph example taken from [11]. (**b**) Static scheduling results of mode m_1 and m_2

to consider all modes simultaneously, but allowing task migration to minimize the resource requirement, which results in the schedule of Fig. 29.12d.

How to consider the task migration overhead is the key challenge in the static scheduling of an MTM-SDF graph. Figure 29.13a shows a simple MMDF graph example that consists of two modes of operation. For each mode, a static schedule which satisfies the given throughput constraint is constructed as shown in Fig. 29.13b. If the schedule of each mode is repeated forever, the output samples will be produced periodically. The period is equal to the inverse of the throughput performance, which is denoted as the initiation interval (II) in the figure. Even though the static schedule of each mode satisfies the given throughput constraint, the overall throughput performance of the MMDF graph may not satisfy the throughput constraint because of the mode transition delay if a mode transition occurs.

The mode transition delay between two modes is defined how the time interval between the last output production time of the previous mode and the first output production time of the next mode is larger than the initiation interval of the next mode. Suppose that the last iteration of the previous mode is started at $t = 0$. First we formulate the start offset (χ) of the first iteration of the next mode. The start offset (χ) is determined by the following three factors:

(1) *Scheduling delay* (D_{sched}): To keep the temporal property of the given static schedule, we need to shift the start time of the subsequent mode. The time

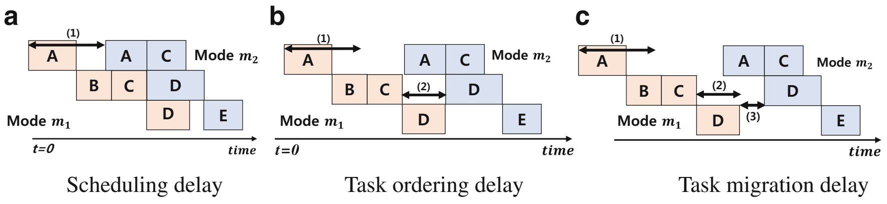


Fig. 29.14 Mode transition delay between static schedules of modes m_1 and m_2 in Fig. 29.13b. (**a**) Scheduling delay. (**b**) Task ordering delay. (**c**) Task migration delay, taken from [11]

interval, denoted by (1) in Fig. 29.14a, illustrates the scheduling delay between modes m_1 and m_2 of Fig. 29.13.

(2) *Task ordering delay* (D_{order}): Because the proposed technique allows task migration between modes, a task can be mapped onto different processors in each mode. So it needs to be guaranteed that two consecutive executions of the same task are not overlapped or inverted during mode change. In Fig. 29.14a, two executions of task D are overlapped between modes. Thus, the execution of the next mode should be delayed by the task ordering delay denoted by (2) in Fig. 29.14b.

(3) *Task migration delay* (D_{mig}): Tasks which are mapped onto different processors between modes should be migrated during the time interval between the end time in the previous mode and the start time in the next mode. If the time interval is not long enough to migrate the task, additional time delay is required. In Fig. 29.14b, task D should be migrated to another processor after the end of execution in the previous mode, and additional time delay is needed, which is the task migration delay denoted by (3) in Fig. 29.14c. In case of task C, no additional time delay is required.

Summing up all three types of delay mentioned above, we compute the start offset of the next mode as follows:

Definition 1 (Start offset of mode m in the case of mode transition $n \rightarrow m$).

$$\chi^{nm} = D^{nm}_{sched} + D^{nm}_{order} + D^{nm}_{mig}$$

Since the output production time of each mode equals to the latency of the static schedule from the start time, the mode transition delay can be formulated as follows:

Definition 2 (Mode transition delay from mode n to mode m).

$$TransDelay(n,m) = Lat(m) + \chi^{nm} - Lat(n) - II(m)$$

where $Lat(m)$ represents the latency of mode m and $II(m)$ represents the initiation interval of mode m.

Note that, if $Lat(m) + \chi^{nm} - Lat(n) \leq II(m)$, then $TransDelay(n,m)$ will be smaller than zero. It means that the time interval of the output production times during a mode transition can be shorter than the output production time interval of the next mode ($II(m)$).

The mode transition delay will be used to determine the new throughput requirement for each mode of operation to satisfy the throughput constraint. When the number of iterations performed in mode m is N, the average initiation interval becomes $MaxTransDelay(m) + N * II(m)/N$ where $MaxTransDelay(m)$ indicates the maximum value of all possible mode transitions to mode m. Therefore, we need to increase the throughput performance by decreasing $II(m)$, in order to satisfy the given throughput requirement. In other words, the new initiation interval $II_{new}(m)$, whose inverse is the new throughput requirement, should satisfy the following inequality.

$$MaxTransDelay(m) + N * II_{new}(m) \leq N * 1/(throughput\ requirement) \tag{29.1}$$

When we schedule all modes of MTM-SDF graphs, we have to consider the increase of throughput requirement for each possible pair of mode changes. The proposed scheduling technique is based on a Genetic Algorithm (GA) [18], of which the overall procedure is shown in Fig. 29.15. The chromosome for GA represents which processor a task in each execution mode is mapped. Chromosomes of initial population are randomly generated and selected from crossover and mutation. The probabilities of crossover and mutation are given by a user with configuration parameters. In the local optimization step, we shuffle the processor indexes for some

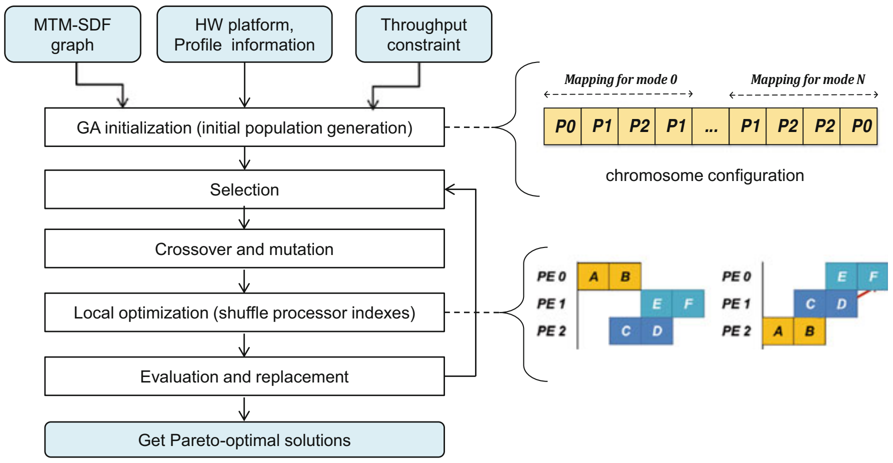

MTM-SDF graph
HW platform, Profile information
Throughput constraint
GA initialization (initial population generation)
Selection
Crossover and mutation
Local optimization (shuffle processor indexes)
Evaluation and replacement
Get Pareto-optimal solutions
Mapping for mode 0
Mapping for mode N
P0 P1 P2 P1 ... P1 P2 P2 P0
chromosome configuration
PE 0
PE 1
PE 2
A B E F C D

Fig. 29.15 GA-based MTM-SDF scheduling framework in HOPES

selected modes in the chromosome in case shuffling reduces the migration cost at mode change. Note that a hardware component is regarded as a special processing element to which a limited set of tasks can be mapped.

In the evaluation step, we apply a list scheduling heuristic to find a static schedule based on the mapping information of the chromosome. Once we construct a static schedule, we evaluate the fitness value of each solution and check whether the throughput constraint is satisfied or not. Chromosomes in the population are sorted by their fitness value and poor chromosomes are eliminated.

For each Pareto-optimal solution, we record the mapping and scheduling result of tasks for a given set of processing elements. And we determine the minimum buffer size for each channel by finding out the maximum number of samples accumulated on the channel during an iteration of the schedule. Note that we may expand the design space by considering the variation of voltage and frequency for power minimization, which has not been implemented in HOPES yet.

29.3.2 Dynamic Mapping

Actual mapping of tasks to processors is performed at run time based on the scheduling information of all applications. When a system status change is detected, the supervisor identifies the set of applications concurrently running and the set of available processors. And it allocates the processors to applications in order to maximize the overall Quality of Service (QoS) metric.

Run-time dynamic mapping is performed in two steps: processor allocation and processor binding. In the processor allocation step, we determine the number of processors allocated to each application. We first allocate the minimum number of processors to each application in order to satisfy the throughput constraint. If the sum of allocated processors is larger than the number of available processors, all applications are not schedulable and we have to discard some applications of low criticality. If there are remaining processors, we repeat the following process until there are no remaining processors or no gain is expected with more processors allocated to any application: find an application that would have the maximum benefit with one more processor and allocate a remaining processor to the application. Suppose that the number of available processors is five in the example of Fig. 29.11. After allocating two processors to both applications initially to satisfy the throughput constraints, one processor is left unallocated. Since the throughput improvement of application *A* with one more processor is larger than that of application *B*, we allocate the remaining processor to application *A*.

After processor allocation is finished, we perform the processor binding step where the physical position of the allocated processors is determined. A popular objective of the binding step is to minimize the average communication overhead over all applications and to minimize the task migration overhead. To minimize the task migration overhead, the same binding is preserved for an application that has no change in the number of allocated processors.

29.4 CIC Translator: Automatic Code Synthesis from the CIC Model

A key benefit of the proposed model-based design methodology is that the target code can be synthesized automatically from the CIC model after the mapping and scheduling decision is made for a given HW/SW platform. The code synthesis step can be understood as model refinement, enjoying the benefit of "correct-by-construction" design paradigm to relieve the designer of heavy burden of verifying the correctness that can be checked by static analysis of the model. To this end, the code should be synthesized in a way to preserve the interface and execution semantics of the model. For an SDF task, for instance, it should be guaranteed that the task starts its execution only after all input ports have as many number of samples as are defined by the sample rates on the associated channels. It implies that we may need to synthesize an interface module in front of the HW IP to synchronize the arrival of input data samples if an SDF task is implemented by a HW IP. Even though the SDF model assumes infinite size of channel buffers, we can determine the buffer sizes at compile time from the static analysis. Then an SDF task should check before starting its execution if there is available space at the output buffers.

In HOPES, we assume that the internal code of an SDF task is given. It is up to the designer to guarantee the correctness of the internal code. Then the CIC translator synthesizes the interface code between tasks and the scheduler code to determine the execution order of the mapped tasks on each processing element. The interface code and the scheduler code depend on the mapping and scheduling policy of the target platform. There are four policies to perform mapping and scheduling of SDF tasks: fully static, self-timed, static assignment, and fully dynamic. If the fully static policy is applied, the run-time scheduler keeps not only the mapping and scheduling decision made at compile time but also the timing information. If a task finishes earlier than the worst-case execution assumed in static scheduling, the run-time scheduler delays the completion of the task until the assumed completion time. By keeping the start and the completion time of tasks, the fully static policy guarantees to produce the same scheduling result as expected at compile time. It means that real-time performance is guaranteed to be correct by construction, which is very desirable for hard real-time systems. The main drawback of this policy is that we should sacrifice the processor utilization in case the worst-case task execution scenario is very different from the average-case scenario. The run-time scheduler simply executes the tasks at the predetermined starting times without checking the buffer status.

Under the self-timed policy, on the other hand, the run-time scheduler does not keep the starting times of tasks while preserving the mapping and scheduling result. Before it starts the next task on the schedule list, it should check the availability of the input data samples. Since the scheduling order is preserved, we may generate a single thread that executes a sequence of function calls in the scheduling order where each SDF task is implemented as a function call. Note that we do not resort to any OS scheduler of the target platform under this policy.

The static assignment policy allows the change of task execution order while the mapping decision is kept. In a self-timed policy, a processor can be idle waiting for the arrival of input samples for the next task to execute in the scheduling order even though there is an executable task. By changing the scheduling order at run time, we may increase the processor utilization, which is the main reason of adopting a static assignment policy particularly when the task execution times vary widely. The static scheduling information can be used to assign the priority of the tasks, giving a higher priority to the task that appears earlier in the scheduling order. If a static assignment policy is used, we synthesize each task as a separate thread and may resort to the thread scheduler that is provided by the SW platform of the target architecture. If there is no built-in thread scheduler, we synthesize a simple run-time scheduler that checks the execution status of all tasks when the processor receives a data sample from the other processors and completes the execution of the current thread. Thus, there is a trade-off between run-time scheduling overhead and processor utilization between self-timed and static assignment policy.

The fully dynamic policy ignores the static scheduling information at run time by allowing the change of mapping and scheduling of tasks. It is the same as the global scheduling policy for an Symmetric Multi-Processing (SMP) processor, distinguished from the partitioned scheduling policy where mapping of tasks does not change at run time. Similarly to a static assignment policy, we may use the static scheduling information to assign the priority of the tasks. And we synthesize each task as a separate thread and use the global scheduler that is provided by the SW platform of the target architecture. In the current implementation, the fully dynamic policy can be used for an SMP target only.

Figure 29.16a shows the overall flow of automatic code synthesis by the CIC translator, and Fig. 29.16b shows the structure of the synthesized code. We use colors to show how the synthesized code is matched with the synthesis flow in the figure. We first check which mapping and scheduling policy will be used and

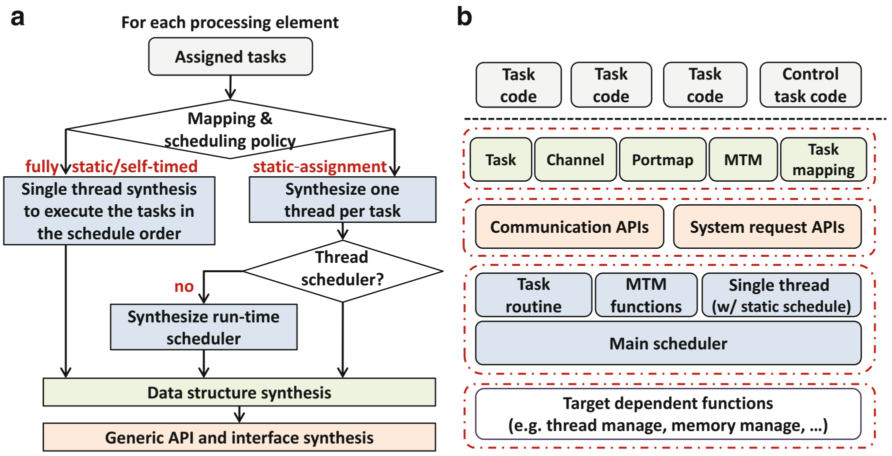

a
For each processing element
Assigned tasks
Mapping & scheduling policy
fully static/self-timed
static-assignment
Single thread synthesis to execute the tasks in the schedule order
Synthesize one thread per task
Thread scheduler?
no
Synthesize run-time scheduler
Data structure synthesis
Generic API and interface synthesis
b
Task code
Task code
Task code
Control task code
Task
Channel
Portmap
MTM
Task mapping
Communication APIs
System request APIs
Task routine
MTM functions
Single thread (w/ static schedule)
Main scheduler
Target dependent functions (e.g. thread manage, memory manage, ...)

Fig. 29.16 (**a**) The overall flow of automatic code synthesis by the CIC translator and (**b**) the structure of the synthesized code partitions the tasks based on the mapping information unless the fully dynamic policy is used. Then the target code for each processing element is synthesized one by one. In case the fully static or the self-timed policy is used for a processor, we synthesize a single thread that executes the mapped tasks by function calls following the scheduling order. In case the static assignment policy is used, a separate thread is created for each task and a run-time scheduler code is synthesized if there is no built-in thread scheduler in the SW platform. Depending on the policy and the SW platform, we translate the generic APIs to the target APIs when the task code is synthesized. Since the interface code with the other processing elements is dependent on the target platform, we assume that the interface code is given as a part of input information to the CIC translator. To this end, HOPES has target-specific library folders that contain target-specific tasks as well.

The CIC translator can be understood as a high-level compiler of the CIC model to generate the target-specific code automatically. As we need a different C compiler to generate the target-specific binary from a target-independent C code, we need to develop a different CIC translator for each target platform. As of now, the following target platforms are supported in the HOPES environment: Linux-based SMP processor, CPU-GPU heterogeneous architecture, IBM Cell processor, Network-on-Chip (NoC)-based many-core virtual prototype, and a multi-robot platform with Bluetooth communication links.

29.5 Experimental Results

In this section, we show two real-life examples to demonstrate the overall design flow to verify the viability of the HOPES methodology. The first example is a smartphone example shown in Fig. 29.9. This example is quite challenging since it consists of multiple applications running concurrently, having inter-application and intra-application dynamism. And each application has real-time constraints. The profiling information of applications is obtained by preparatory experiments in advance using a cycle-accurate ARM processor simulator. The WCET information for each task is reported in Table 29.2 for the H.264 decoder application of Fig. 29.5. Tables 29.3 and 29.4 show the WCET of each task in x264 encoder and MP3 decoder application. Both applications are specified with an SDF graph respectively, having only one mode of operation; the tables show how many tasks each application consists of. For the x264 encoder application, we make a single task for the most time-consuming algorithm, motion estimation (ME), in this experiment. G.723 encoder and G.723 decoder applications are specified by a single task each, and their execution times are profiled to 4×10^3 cycles/iteration and 6×10^3 cycles/iteration, respectively.

With the given profiling information, compile-time analysis is performed to obtain the set of Pareto-optimal mapping and scheduling solutions for varying number of processors for each application. The result of compile-time analysis is summarized in Table 29.5.

To compare the performance of the proposed hybrid mapping technique with a dynamic mapping technique, we tested the following scenario: (1) play a video clip,

Table 29.2 Profiling information of H.264 decoder application (unit: $\times 10^3$ cycles/frame)

Task	Time (WCET/ average)	Task	Time(WCET/average)	Task	Time (WCET/ average)
ReadFile	I: 980/760 P: 590/420	IntraPredY	I: 980/830	InterPredY	I: 80/60 P: 3940/1560
Decode	I: 7500/5010 P: 2990/920	IntraPredU	I: 190/150	InterPredU	I: 20/20 P: 340/270
Deblock	I: 1550/1390 P: 1120/370	IntraPredV	I: 180/150	InterPredV	I: 20/20 P: 340/270
WriteFile	I: 2240/2120 P: 2360/2100				

Table 29.3 Profiling information of x264 encoder application (unit: $\times 10^3$ cycles/frame)

Task	Time (WCET/average)	Task	Time (WCET/average)	Task	Time (WCET/average)
Init	250/170	Deblock	3020/2660	Encoder	4840/4470
ME	15170/14720	VLC	2350/1780		

Table 29.4 Profiling information of MP3 decoder application (unit: $\times 10^3$ cycles/iteraion)

Task	Time (WCET/average)	Task	Time (WCET/average)	Task	Time (WCET/average)
VLD	810/150	Antialias	40/10	Stereo	70/30
DeQ	690/300	Hybrid	1230/160	Reorder	50/10
Subband	630/270	WriteFile	150/20		

Table 29.5 Summary of compile-time analysis

Application	Processors (min, max)	Throughput (min, max)	Throughput constraint
H.264 decoder	(1,2)	(50.9, 52.8) frames/sec	VideoPlay: 30 frames/sec VideoPhone: 15 frames/sec
MP3 decoder	(1,6)	(123.7, 569.1) iterations/sec	150 iterations/sec
x264 encoder	(1,2)	(27.3, 30.1) frames/sec	15 frames/sec

(2) a phone call preempts the video play, (3) resume the video play after the call is finished, and (4) return to the Menu state when the video clip is finished. We assume that the target HW platform has a 3×3 NoC architecture in which there are seven ARM processor tiles (700 Mhz for each) available for executing the applications. Figure 29.17a shows the total sum of throughput excesses over the throughput constraints for all applications. The throughput excess can be used to reduce the power consumption of the system by lowering the voltage and frequency of the processor. Figure 29.17b illustrates the relative latency achieved from the dynamic mapping against the proposed hybrid mapping, varying the communication-to-

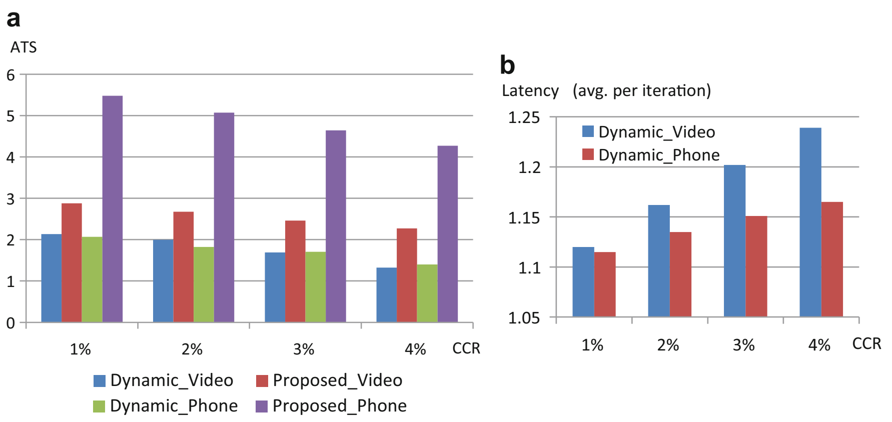


Fig. 29.17 (**a**) The aggregate throughput gain over the throughput constraints for both hybrid and dynamic mapping techniques, and (**b**) the relative latency achieved by dynamic mapping against the hybrid mapping

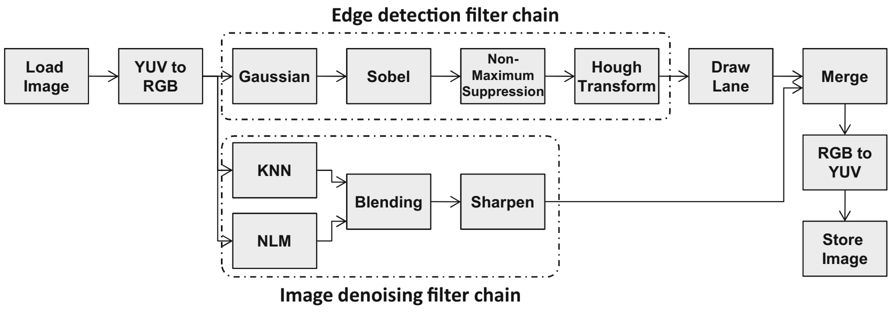


Fig. 29.18 A CIC modeling of lane detection algorithm

computation ratio (*ccr*). These experiments confirm that the hybrid mapping gives significant gain in throughput and latency by utilizing the static scheduling results.

As another real example, a lane detection algorithm for driver assistance is implemented by a CPU-GPU heterogeneous architecture. In this experiment, we used Intel Core i7-930 CPU (2.80 GHz) and two Tesla M2050 GPUs. To run a task on a GPU, we used a different version of the task that uses CUDA programming in its internal definition. For CUDA programming, NVIDIA GPU Computing SDK 3.1 and CUDA toolkit v3.2 RC2 were used. The CIC model of the lane detection application is displayed in Fig. 29.18 and the associated profiling information is shown in Table 29.6.

The design space explored in this experiment is defined by the number of CPU and GPU processing elements, task mapping, and communication methods between CPU and GPU. Asynchronous communication between CPU and GPU is supported by defining *streams* in CUDA programming. While operations with the same stream

Table 29.6 Profiling information of lane detection application (unit: usec)

Task	CPU	GPU	Task	CPU	GPU
LoadImage	479	–	KNN	2,999,704	7202
YUVtoRGB	53,111	8152	NLM	1,017,401	16,497
Gaussian	78,100	4591	Blending	16,093	5078
Sobel	10,041	5139	Sharpen	110,139	5455
Non-max	164,013	6611	Merge	32,340	5032
Hough	311,966	5653	RGBtoYUV	66,733	4888
Draw lane	1592	–	StoreImage	1068	-

Table 29.7 Design space exploration of lane detection application (unit:sec)

Configuration	Sync	Async (2 streams)	Async (3 streams)	Async (4 streams)
CPU + 0 GPU	2109.5	–	–	–
CPU + 1 GPU	15.0	12.0	12.3	12.1
CPU + 2 GPUs	11.3	10.2	9.8	9.8

Table 29.8 Task mapping onto 1 CPU + 2 GPUs

Processor	Tasks
CPU	LoadImage, Draw lane, StoreImage
GPU 0	YUVtoRGB, Gaussian, Sobel, Non-maximum, Hough, Merge
GPU 1	KNN, NLM, Blending, Sharpen, RGBtoYUV

should be serialized, those between different streams can be executed in parallel. Thus, asynchronous communication promises potential throughput improvement paying the overhead of memory space and stream management overhead. For this experiment, we used a yuv video clip which consists of 300 frames of HD size (1280×720). We explored the design space manually to obtain the result as shown in Table 29.7. It reveals that using two GPUs gives the best performance in which task mapping is made as shown in Table 29.8.

29.6 Current Status and Conclusion

The HOPES design environment consists of various tools that realize individual design steps in the design flow of Fig. 29.2. It has an eclipse-based Graphical User Interface (GUI) to help a designer to follow the design flow conveniently. Interface between design tools is made by *xml* files so that we may change or add a design tool into the environment by accessing the interface files. We expect that the HOPES environment can be improved by third-party tools.

Besides the techniques introduced in this chapter, there are other tools involved in the HOPES design environment such as Worst-Case Response Time (WCRT) analysis tool (STBA and HPA) [14] and a HW/SW cosimulation tool, HSIM [26]. The WCRT analysis tool is to estimate the latency of an application conservatively when a self-timed or a static assignment policy is adopted. Since the scheduling anomaly may happen due to unexpected interference from the other processing elements in the access to the shared resources, the worst-case performance estimated from the static analysis step is not guaranteed if we change the scheduling times of tasks or the execution order of tasks. Therefore, we use a separate tool to estimate the response of an application after mapping decision is made. The HW/SW cosimulation tool is used to run the target software without the real hardware platform.

In this chapter, it is confirmed that the HOPES methodology is viable to design complex real-time embedded systems with two real-life examples. But it is still far from a general system-level design tool to be used in practice and there is much room for improvement. First of all, we need to consider more real-life systems with diverse characteristics in the system behavior and the target architecture, which is not an easy job for academia. We are testing various types of hardware platforms, including Intel Xeon-Phi, IBM cell processor, many-core simulator, and cooperating heterogeneous robot platforms. The most time-consuming is to make a CIC translator for each target platform. It is similar to building a new C compiler for a new processor. Since the quality of design depends on the CIC translator, generating a target code is not sufficient for practical use. We have to synthesize as good quality code as a manually written code. Since the HOPES framework starts with CIC specification of an application, it is necessary to translate the legacy code to the CIC model for the reuse of a legacy code. If the legacy code is small enough to compose a single CIC task, translation could be made easily by modifying the interface code with the outside. Otherwise, it is necessary to restructure the legacy code to partition it to a set of CIC tasks that follow the assumed execution model, which should be done manually.

Even though the CIC model is independent of the target architecture, we may need to define target-dependent tasks. For instance, a task that accesses I/O devices usually needs to use OS-dependent APIs. To run a task on a special processing element, such as GPU and hardware IP, we need to have multiple versions of the same task that are dependent on the target architecture. Since the granularity of a task is large, careful consideration needs to be made to make it target-independent.

References

1. Bhattacharya B, Bhattacharyya SS (2001) Parameterized dataflow modeling for DSP systems. IEEE Trans Signal Process 49(10):2408–2421. doi:10.1109/78.950795
2. Buck JT (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Technical report, Department of EECS, UC Berkeley, Berkeley. Technical report UCB/ERL 93/69, Ph.D dissertation
3. Buck JT, Ha S, Lee EA, Messerschmitt DG (1994) Ptolemy: a framework for simulating and prototyping heterogenous systems. Int J Comput Simul 4(2):155–182
4. Eidson J, Lee EA, Matic Slobodan SSA, Zou J (2012) Distributed real-time software for cyber-physical systems. Proc IEEE 100(1):45-59

5. Girault A, Lee B, Lee E (1999) Hierarchical finite state machines with multiple concurrency models. IEEE Trans Comput Aided Des Integr Circuits Syst 18(6):742–760
6. Goossens S, Akesson B, Koedam M, Nejad AB, Nelson A, Goossens K (2013) The CompSOC design flow for virtual execution platforms. In: Proceedings of the 10th FPGAworld conference. ACM, p 7
7. Ha S, Kim S, Lee C, Yi Y, Kwon S, Joo YP (2008) Peace: a hardware-software codesign environment for multimedia embedded systems. ACM Trans Des Autom Electron Syst 12(3):24:1–24:25. doi:10.1145/1255456.1255461
8. Harel D, Naamad A (1996) The STATEMATE semantics of statecharts. ACM Trans Softw Eng Methodol (TOSEM) 5(4):293–333
9. Haubelt C, Falk J, Keinert J, Schlichter T, Streubühr M, Deyhle A, Hadert A, Teich J (2007) A SystemC-based design methodology for digital signal processing systems. EURASIP J Embed Syst 2007(1):1–22. doi:10.1155/2007/47580
10. Jung H, Lee C, Kang SH, Kim S, Oh H, Ha S (2014) Dynamic behavior specification and dynamic mapping for real-time embedded systems: HOPES approach. ACM Trans Embed Comput Syst (TECS) 13:135:1–135:26
11. Jung H, Oh H, Ha S (2017) Multiprocessor scheduling of a multi-mode dataflow graph considering mode transition delay. ACM Trans. Des. Autom. Electron. Syst. (TODAES) 22, 2, Article 37
12. Kangas T, Kukkala P, Orsila H, Salminen E, Hännikäinen M, Hämäläinen TD, Riihimäki J, Kuusilinna K (2006) Uml-based multiprocessor soc design framework. ACM Trans Embed Comput Syst 5(2):281–320. doi:10.1145/1151074.1151077
13. Kienhuis B, Deprettere E, Vissers K, Wolf PVD (1997) An approach for quantitative analysis of application-specific dataflow architectures. In: Proceedings of the IEEE international conference on application-specific systems, architectures and processors, pp 338–349. doi:10.1109/ASAP.1997.606839
14. Kim J, Oh H, Choi J, Ha H, Ha S (2013) A novel analytical method for worst case response time estimation of distributed embedded systems. In: Proceedings of the design automation conference (DAC), Austin, pp 1–10
15. Kwon S, Kim Y, Jeun WC, Ha S, Paek Y (2008) A retargetable parallel programming framework for MPSoC. ACM Trans Des Autom Electron Syst (TODAES) 13:39:1–39:18
16. Lee C, Kim H, Park H, Kim S, Oh H, Ha S (2010) A task remapping technique for reliable multi-core embedded systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), Scottsdale, pp 307–316
17. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
18. Man KF, Tang KS, Kwong S (1996) Genetic algorithms: concepts and applications [in engineering design]. IEEE Trans Ind Electron 43(5):519–534. doi:10.1109/41.538609
19. Nikolov H, Thompson M, Stefanov T, Pimentel A, Polstra S, Bose R, Zissulescu C, Deprettere E (2008) Daedalus: toward composable multimedia MP-SoC design. In: Proceedings of the design automation conference, pp 574–579
20. Park Hw, Jung H, Oh H, Ha S (2011) Library support in an actor-based parallel programming platform. IEEE Trans Ind Inf 7:340–353
21. Schor L, Bacivarov I, Rai D, Yang H, Kang SH, Thiele L (2012) Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In: Proceedings of the international conference on compilers architecture and synthesis for embedded systems (CASES), pp 71–80
22. Stuijk S, Geilen M, Theelen BD, Basten T (2011) Scenario-Aware dataflow: modeling, analysis and implementation of dynamic applications. In: Proceedings of the international conference on embedded computer systems: architectures, modeling, and simulation, ICSAMOS'11. IEEE Computer Society, pp 404–411. doi:10.1109/SAMOS.2011.6045491
23. Theelen BD, Geilen M, Basten T, Voeten J, Gheorghita SV, Stuijk S (2006) A Scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Proceedings of international conference on formal methods and models for co-design, MEMOCODE'06. IEEE Computer Society, pp 185–194. doi:10.1109/MEMCOD.2006.1695924

24. Thiele L, Bacivarov I, Haid W, Huang K (2007) Mapping applications to tiled multiprocessor embedded systems. In: International conference on application of concurrency to system design, pp 29–40. doi:10.1109/ACSD.2007.53
25. Thiele L, Strehl K, Ziegenbein D, Ernst R, Teich J (1999) FunState–an internal design representation for codesign. In: White JK, Sentovich E (eds) ICCAD. IEEE, pp 558–565
26. Yun D, Kim S, Ha S (2012) A parallel simulation technique for multicore embedded systems and its performance analysis. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 31:121–131