Energy Efficient Streaming Applications with Guaranteed Throughput on MPSoCs

Jun Zhu, Ingo Sander and Axel Jantsch
ECS/ICT, Royal Institute of Technology, Stockholm, Sweden
junz@kth.se, ingo@kth.se, axel@kth.se

ABSTRACT

In this paper we present a design space exploration flow to achieve energy efficiency for streaming applications on MPSoCs while meeting the specified throughput constraints. The public domain simulators Sim-Panalyzer and Cacti are used to estimate the energy dissipations of the parameterized architectural components. As the main contributions, we schedule the streaming applications on a multi-clock synchronous modeling framework, guarantee the application timing properties by throughput analysis, and customize both processor *voltage-frequency* levels and memory *size*s in the design space to optimize the application pipeline parallelism for energy efficiency. Two widely used heuristic algorithms (i.e., greedy and taboo search) are used during the design optimization process. Our experiments show an energy reduction of 21% without any loss in application throughput compared with an ad-hoc approach.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling Techniques

General Terms

Performance, Design, Experimentation

Keywords

Energy efficiency, Streaming applications, MPSoCs, Synchronous MoC

1. INTRODUCTION

Multi-processor systems-on-chips (MPSoCs) have been increasingly adopted as the physical architectures for streaming applications [8, 22]. In streaming applications, a process corresponds to the given computation running on an MPSoC processor. Processes are connected with each other through communication channels, and operate on data streams in a pipelined fashion. The processors can be heterogeneous, each customized for specific tasks, and are running simultaneously to obtain high computation power.

However, performance is not the only concern in a streaming environment and MPSoC processors are not required to run as fast as possible. Processors, running faster than demanded by the application requirements, will only produce data streams ahead of time and lead to consequent redundant storage buffers [6]. Instead, processors often run at a potentially slower frequency according to the throughput requirements and avoid otherwise higher energy consumption. Nevertheless, to achieve extreme energy efficiency in MPSoCs is non-trivial. The architectural computation, storage, and communication components, which are fully customizable, all contribute to the system energy dissipation. To take all their impacts into consideration in the energy efficiency design, we need appropriate system models to capture both the parallel nature of streaming applications and the inherent heterogeneous timing properties of MPSoCs as well.

Our main contribution in this paper is a design space exploration flow with the following characteristics:

1. The streaming applications are scheduled on a multi-clock synchronous modeling framework.
2. Timing properties are guaranteed by application throughput analysis.
3. High system energy efficiency on MPSoC architecture is achieved with customized processor and memory components from optimized pipeline parallelism.

The rest of the paper is structured as follows. Section 2 discusses the related work. We motivate our work by illustrating the energy efficiency design on an example application in Section 3, and propose our design space exploration flow in the following Section 4. Section 5 introduces our formal modeling framework. Section 6 introduces our design objective on the MPSoCs architecture. Section 7 shows the experimental results. Finally, Section 8 concludes the paper.

2. RELATED WORK

Various models of computation (MoCs) [13, 17] have been developed for streaming applications. They are distinguished by their individual formalisms on how processes interact and how or whether time is represented. One is the Kahn process network (KPN) [16]. In [7], a design space exploration framework based on KPN investigates the performance and power trade-offs in MPSoC systems. However, limited by the unbounded FIFO channels in the KPN semantics, it cannot dimension the memory modules and analyze the effects

EMSOFT'08, October 19–24, 2008, Atlanta, Georgia, USA.

on the system energy cost that arise from parameterized memories.

In the synchronous dataflow (SDF) MoC, the static data rate allows for the construction of periodic schedules with bounded memory size at compile time [15]. Based on the static memory allocation for DSP applications in the SDF MoC, the energy consumption is exploited on the algorithmic level [3]. However, the SDF MoC is untimed, which means the trade-offs between different timing related properties, such as throughput and energy, cannot be explored within this early system model.

A timed extension has been applied to SDF in [9, 21]. Using the timed SDF MoC, a method to minimize the memory size by scheduling the processes appropriately for maximal application throughput is introduced in [9]. In [21], the trade-offs between any specified throughput constraints and the corresponding minimal memory requirements are further studied. Nevertheless, memory is only one of several factors to impact the total system energy cost, besides computation and communication logics; meanwhile, the single unit-time assumption [9, 21] does not suit to model the heterogeneous computation and communication timing in MPSoCs.

By dividing the physical time axis into slots, the synchronous MoC has an explicit order of time [13]. The synchronous method has met industrial success to program safety-critical reactive systems in Esterel [4], Lustre [11] and Signal [14]. Especially, Lustre and Signal/Polychrony [10] define multi-clock models for heterogeneous embedded systems. In this paper, we advocate our multi-clock synchronous MoC framework to analyze timing related properties at the system level in MPSoCs design space exploration.

In contrast to existing work, we leverage the degrees of customizability of both processor *voltage-frequency* levels and memory *size*s, model the heterogeneous timing on our multi-clock synchronous MoC framework, and investigate the minimal energy consumption of streaming applications. High system energy efficiency is achieved without losing throughput guarantees, as confirmed in our experiments.

3. MOTIVATION

Here we introduce a streaming application model, which will be the tutorial example in this paper, and motivate our work by illustrating the energy efficiency design upon some assumed energy models with this application.

3.1 Streaming application model

A streaming application model example with three-stage pipeline is depicted in Fig. 1, where nodes denote the computation processes and edges associated with FIFOs denote the communication channels with finite storage. Processes read tokens from the input-side FIFOs, operate (compute) on the data within a specified amount of time, and emit the resulting data tokens to the output-side FIFOs at the end of the computation. While a process is computing, the data tokens remain on the input-side FIFOs until the computation is completed.

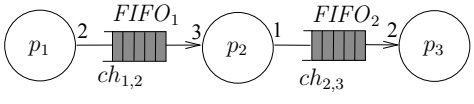


Figure 1: Streaming application model example

A communication channel $ch_{u,v}$ from process p_u to process p_v has the consumption data rate $I_{u,v}$ and emission data rate $O_{u,v}$. We concentrate on static (fixed) data rate applications. In Fig. 1, these static data rates are annotated as numbers beside each channel, such as $ch_{1,2}$ has $I_{1,2} = 3$ and $O_{1,2} = 2$.

We only consider applications with consistent data rate, which can run infinitely with bounded FIFO space [15]. When the data rate is consistent, p_u and p_v can run in a repetitive pattern with non-trivial (non-zero) firing times r_u and r_v, where r_u and r_v are the minimum integer solutions of a set of balance equations $r_u \cdot O_{u,v} = r_v \cdot I_{u,v}$ for all the communication channels. For the example application model, the process firing times vector is $< r_1, r_2, r_3 >=< 3, 2, 1 >$.

The computation constraints are captured in a process computation *latency* list T, which contains the latency t_i in time slots to execute each process p_i once. A time slot equals to an abstract clock cycle and is the elementary time unit, in which time is measured and for which timing related properties (throughput, energy, etc.) are evaluated. The storage constraints are captured in a FIFO *size* list Γ, which contains the storage capacity γ_i in data tokens for each $FIFO_i$. Thus, $T = [t_1, t_2, t_3]$ and $\Gamma = [\gamma_1, \gamma_2]$ in the example application model.

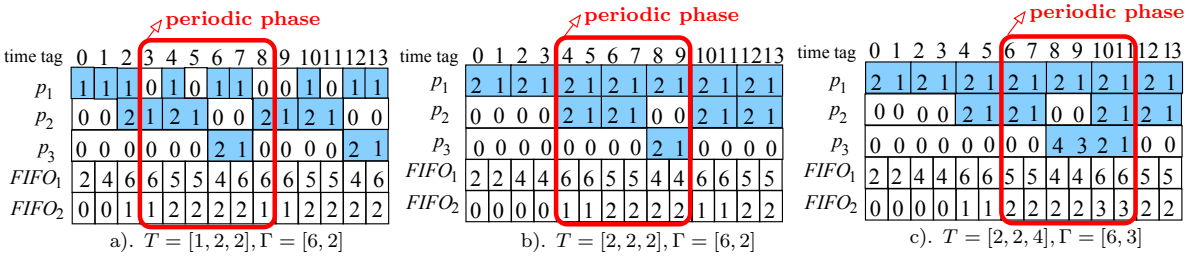
We use self-timed scheduling [20], which means a process *executes* as soon as it is enabled; otherwise, it *stalls*. A process is enabled, only when the input-side FIFOs have sufficient data tokens and the output-side FIFOs have enough vacant space. Obviously, the enable conditions of the corresponding input- or output-side FIFOs are ignored for the signal source or sink processes.

3.2 Design exploration

We explore different design options for the example application of Fig. 1, and analyze the problem only in a single domain synchronous MoC for clarification.

We assume that the computation is constrained by the following latency list $T = [1, 2, 2]$. Using the memory minimization techniques in [21], the self-timed schedule achieved by design option *a)*, with $\Gamma = [6, 2]$, is shown in Fig. 2.a). In the graph, the process and FIFO status are listed in separated rows, and time evolution is depicted in corresponding columns. Using the unit abstract clock, the time tag advances 1 per column. At each time tag, a process p_u in *executing* (shadowed) state has a number t'_u to denote the remaining execution time slots, a *stalling* (non-shadowed) process status is denoted as 0, and a FIFO status γ'_i is denoted as the occupied storage space of $FIFO_i$ in token numbers.

At time tag 0, the process status list is $T'_0 = [t'_1, t'_2, t'_3] = [1, 0, 0]$, in which p_1 is executing with 1 time slot left and p_2 and p_3 are stalled; in the meantime the FIFO status list is $\Gamma'_0 = [\gamma'_1, \gamma'_2] = [2, 0]$, with only 2 tokens space reserved at $FIFO_1$. At time tag 1, p_1 finishes the previous computation, emits 2 tokens into $FIFO_1$, and reserves another 2 tokens space from $FIFO_1$ for a new execution; thus, $T'_1 = [1, 0, 0]$ and $\Gamma'_1 = [4, 0]$. At time tag 2 ($T'_2 = [1, 2, 0]$ and $\Gamma'_2 = [6, 1]$), besides the similar status changes in p_1 and $FIFO_1$, p_2 is enabled and starts to compute. As the schedule advances to time tag 9, the application encounters the same process and FIFO status list as at time tag 3 ($T'_9 = T'_3 = [0, 1, 0]$ and $\Gamma'_9 = \Gamma'_3 = [6, 1]$) and enters a periodic phase. The periodic



process	t_i (v_H, f_H)	t_i (v_L, f_L)	$\hat{\mathcal{P}}_{\mu p_j, p_i}(v_j^2, f_j)$ (v_H, f_H)	$\hat{\mathcal{P}}_{\mu p_j, p_i}(v_j^2, f_j)$ (v_L, f_L)
p_1	1	2	1	0.125
p_2	2	4	1	0.125
p_3	2	4	1	0.125

d). Computation latency and energy

FIFO	$\hat{\mathcal{E}}_{mem_j}(\gamma_i)$
$FIFO_1$	$1 * \gamma_1$
$FIFO_2$	$1 * \gamma_2$

e). Memory energy

design option	energy E_U	energy E_M	energy E_{Sum}	η[%]
a)	9	8	17	50
b)	6.75	8	14.75	67
c)	5.25	9	14.25	78

f). Energy and computation efficiency analysis

Figure 2: Self-timed schedules of the example application with different design options

phase extends from time tag 3 to 8, with the length 6 time slots and process firing times vector $< 3, 2, 1 >$.

Instead of minimizing the memory requirement with respect to a designated computation constraint T, we customize both the process computation *latency* and FIFO *size*, and extend the design space to explore another two design options *b)* and *c)*. Compared with *a)*, *b)* is just to run p_1 at half speed as $T = [2, 2, 2]$ and keeps the same FIFO sizes Γ; *c)* uses $T = [2, 2, 4]$ and $\Gamma = [6, 3]$ to run both p_1 and p_3 at half speed and assign one more storage element to $FIFO_2$. Accordingly, their individual self-timed schedules are shown in Fig. 2.b) and c). Both schedules encounter a periodic phase, with the same length and the same processes firing times vector, as *a)* does. In this sense, all the design options can deliver the same guaranteed application throughput [1].

3.3 Energy efficiency evaluation

In the periodic phase, the energy efficiency of different design options is evaluated, based on the following assumed latency and energy models (only the dynamic energy is considered):

1. Each process p_i is bound to one individual processor μp_j. Each processor operating voltage v_j is set at two voltage levels v_H and v_L (v_L is half of v_H) to explore the design space. The processor operating frequency f_j changes proportionally to v_j ($f_j \propto v_j$); thus, f_L is at half speed of f_H. The computation latency t_i of p_i changes accordingly to $t_i \propto f_j^{-1}$.

2. For each executing time slot of p_i the *dynamic* power consumption on μp_j is $\hat{\mathcal{P}}_{\mu p_j, p_i}(v_j^2, f_j) = \alpha \cdot v_j^2 \cdot f_j$, where α is the average switching capacitance. Since $f_j \propto v_j$ is known, we attain that $\hat{\mathcal{P}}_{\mu p_j, p_i} \propto f_j^3$ is a cubic function of f_j. For n computation time slots the processor energy consumption $E_{\mu p_j}$ is $E_{\mu p_j} = n \cdot \hat{\mathcal{P}}_{\mu p_j, p_i}(v_j^2, f_j)$.

3. Each $FIFO_i$ is bound to one individual memory mem_j, and the memory size equals to the FIFO size γ_i. Within a fixed accessing pattern, the memory energy consumption is $E_{mem_j} = \hat{\mathcal{E}}_{mem_j}(\gamma_i) \propto \gamma_i$, where the memory dynamic energy function $\hat{\mathcal{E}}_{mem_j}$ is proportional to memory size.

4. As the baseline, in design option *a)* all the processor *voltage-frequency* levels are set to (v_H, f_H), and for each time slot the processor dynamic power consumption is normalized to 1, as shown in column 4 of Fig. 2.d); the memory energy consumption in the periodic phase is assigned to 1 per storage element, as shown in Fig. 2.e).

Thus, we derive the computation latency, computation energy and memory energy characteristics of other design options in Fig. 2.d-e). For design option *a)* ($T = [1, 2, 2]$, $\Gamma = [6, 2]$ and $< r_1, r_2, r_3 > = < 3, 2, 1 >$), the total processor energy E_U and total memory energy E_M are

$$\begin{aligned} E_U &= E_{\mu p_1} + E_{\mu p_2} + E_{\mu p_3} = \textstyle\sum_{\forall \mu p_j} t_i \cdot r_i \cdot \hat{\mathcal{P}}_{\mu p_j, p_i}(v_H^2, f_H) \\ &= 1 \cdot 3 \cdot 1 + 2 \cdot 2 \cdot 1 + 2 \cdot 1 \cdot 1 = 9 \\ E_M &= E_{mem_1} + E_{mem_2} = \textstyle\sum_{\forall mem_j} \hat{\mathcal{E}}_{mem_j}(\gamma_j) \\ &= 1 \cdot 6 + 1 \cdot 2 = 8 \end{aligned}$$

The total system energy consumption is $E_{Sum} = E_U + E_M = 17$. Similarly, the energy consumptions for design options *b)* and *c)* are calculated, as shown in Fig. 2.f).

Design option *c)* has the most efficient system energy E_{Sum}. In addition, it achieves the highest average processor utilization η (ratio of the shadowed part in the periodic phase of process schedules), which means the best pipelined parallelism. However, higher pipelined parallelism does not always mean lower system energy dissipation, as demonstrated in Section 7.

4. DESIGN SPACE EXPLORATION FLOW

We propose an energy efficiency design space exploration flow for streaming applications with guaranteed throughput on MPSoCs. As shown in Fig. 3, the inputs of the design exploration flow (shadowed boxes) are the application benchmark C program, the static mapping from the application onto the MPSoC architecture and the architectural design options in the configuration file.

[1]For the formal definition, see Section 5.4

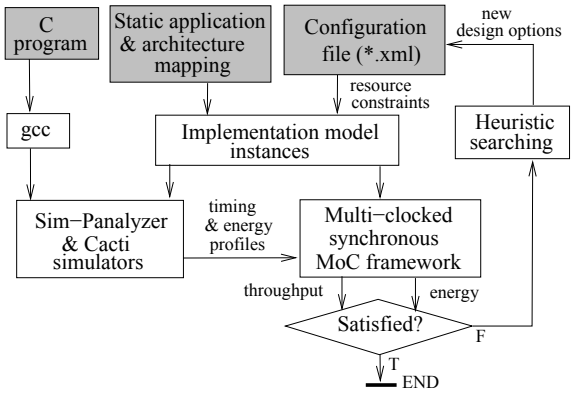


Figure 3: Energy efficiency design exploration flow with guaranteed throughput

The C program for each computation process is cross-compiled into a binary using the ARM gcc toolchain [1]. From the application to architecture mapping and the resource constraints defined in the configuration file, implementation model instances with different architectural parameters are initialized. We explore the design space by customizing both the processor *voltage-frequency* levels and memory *sizes*, as discussed in Section 3.2.

Instead of using the assumed latency and energy models in the previous section, the process binary computation timing and energy dissipations on the architectural component instances are estimated on Sim-Panalyzer [2] and Cacti [23] simulators. Based on the individual process timing and energy profiles, the application throughput and system energy dissipations on MPSoC architecture are analyzed systematically in our multi-clock synchronous MoC framework.

The design objectives for each design option are to meet the application throughput requirement and try to minimize the total system energy dissipation. As the design space increases exponentially with the problem size, heuristic algorithms (greedy and taboo [5] search) are used during the design options searching until the design goals are met.

5. SYNCHRONOUS MOC FRAMEWORK

We adopt the synchronous MoC framework for both application throughput analysis and system energy evaluation. In the synchronous MoC, systems are described as a set of concurrent processes, which communicate through synchronous signals. The static data rate in the SDF MoC are preserved in our synchronous semantics. Furthermore, we exploit multi-clock domains, which suit the heterogeneous timing in MPSoCs well, and relate all the clocks in different system domains to each other (similar to Lustre [11]).

5.1 Signal and process

A signal s with clock clk_s is an indexed [2] set of events, $s = \cup\{e_{(n)}\}^{clk_s} = \{e_{(0)}, e_{(1)}, \cdots, e_{(n)}, \cdots\}^{clk_s}, \forall n \in \mathbb{N}_0$. The signal clock $clk_s \in \mathbb{Q}^+$ is the abstract cycle (time slot) period between two adjacent events. Each event $e_{(n)} = (g_{(n)}, \vec{v}_{(n)})$ has a time tag $g_{(n)}$ and a value $\vec{v}_{(n)}$. Time tags are used to model the global order of events, and are implicitly given by the event indexes in the signal, with $g_{(n)} = n \cdot clk_s$; thus, a signal can be simply denoted as $s = \cup\{\vec{v}_{(n)}\}^{clk_s}$. We visualize the relation of events for signals with different clocks in Fig. 4. Two signals s_1 and s_2 have the clock timing relation $\frac{clk_{s_1}}{clk_{s_2}} = \frac{3}{2}$. The global order of the events in both signals are maintained by time tags.

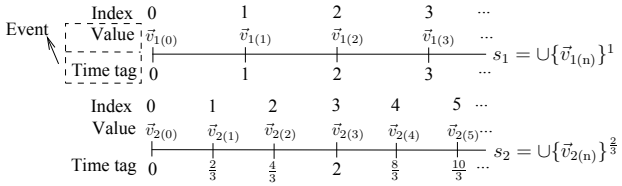


Figure 4: Timing relations of events in signals with different clocks

To capture the data rate consumed in process *executing* cycles and *absent*s in *stalling* cycles, values are a vector $\vec{v}$ of regular tokens, extended with a pseudo value $\perp$. $\vec{v}_{(n)}$ is $\vec{v}$ when the required tokens are *present*, or $\perp$ when *absent*. In static data rate applications, the number of the regular tokens contained in $\vec{v}$ is fixed. A synchronous signal $s_1 = \{\perp, <1,1,2>, <2,3,4>, \cdots\}^{\frac{1}{2}}$ has integer tokens with the data rate 3 and clock period $\frac{1}{2}$.

Processes operate on signals. In each evaluation cycle, processes consume one event from the input signals and output one event to the output signals. In perfect synchrony [13], the computation and communication are executed in zero time and the computation states are maintained in explicit delay process statements. A synchronous model is composed of combinational processes $p_{comb}(f)$ and unit-cycle delay processes $p_\Delta(\vec{v}'_{(0)})$, in which function f specifies the mapping from input events to output events and the given initial state $\vec{v}'_{(0)}$ is the output event at time tag 0 to defer the input events one cycle.

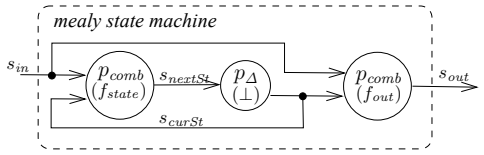


Figure 5: Process skeleton of a mealy state machine

While $p_{comb}(f)$ is suited for describing algorithmic function flow, its combination with $p_\Delta(\vec{v}'_{(0)})$ can be used to construct control logic and more complex components. For instance, the τ-cycle ($\tau \in \mathbb{N}$) delay process is $p_{\Delta,\tau}(\perp) = \underbrace{p_\Delta(\perp) \circ \cdots \circ p_\Delta(\perp)}_{\tau}$ [3]; a combinational process with τ-cycle computation latency is $p_{comb,\tau}(f) = p_{comb}(f) \circ p_{\Delta,\tau}(\perp)$; and a *mealy* state machine, with the state function f_{state} and output function f_{out}, has the process skeleton shown in Fig. 5. The *mealy* state machine is the base to construct the control logic of a finite size FIFO [19] (see Section 3.1).

[2] In this paper, we use the numbers in parentheses especially for indexing purpose.

[3] The process composition operator $\circ$ has the formal definition $p_1 \circ p_2(s_1) = p_1(p_2(s_1))$

5.2 Multi-clock synchronous domains and domain interfaces

In our synchronous model, the abstract clock period corresponds to a given physical time. Processes using the same abstract clock are said to be in the same synchronous domain. To model the heterogeneous timing properties in embedded systems, such as the parallel computation on several processors running at different frequency levels, multi-clock synchronous domains are introduced.

When multi-clock domains exist (i.e., different domain clocks have a rational ratio to each other), asynchronous domain interfaces (DI) are needed to maintain the global timing. In the upper-left part of Fig. 6, two domains D_A and D_B, with different clock periods clk_A and clk_B, have a rational clock ratio $\lambda_{A:B} = \frac{clk_A}{clk_B} = \frac{m_A}{m_B}, \forall m_A, m_B \in \mathbb{N}, m_A \neq m_B$, and m_A and m_B are relatively prime. The domain interface $DI_{A:B}$, which establishes the clock ratio $\lambda_{A:B}$ from A to B is defined as $DI_{A:B} = downDI(m_B) \circ upDI(m_A)$.

As the start point at index 0, all the signal events have the consistent time tag 0. Thus, at tag 0 $upDI(m_A)$ and $downDI(m_B)$ output the value $\vec{v}_{(0)}$ from the input signal. Otherwise, $upDI(m_A)$ is up-sampling the input signal clock m_A times by inserting $m_A - 1$ absent events before each event; and $downDI(m_B)$ is down-sampling the input signal clock m_B times by merging every m_B events. They have the following definitions:

$$upDI\,(m_A)\left(\{\vec{v}_{(0)}, \vec{v}_{(1)}, \vec{v}_{(2)}, \cdots\}^{clk_x}\right) = \{\vec{v}_{(0)}, \underbrace{\perp, \cdots, \perp}_{m_A-1}, \vec{v}_{(1)}, \underbrace{\perp, \cdots, \perp}_{m_A-1}, \vec{v}_{(2)}, \cdots\}^{\frac{clk_x}{m_A}} \quad (1)$$

$$downDI(m_B)(\{\vec{v}_{(0)}, \cdots, \vec{v}_{(m_B)}, \vec{v}_{(m_B+1)}, \cdots\}^{clk_x}) = \{\vec{v}_{(0)}, <\vec{v}_{(1)}, \cdots, \vec{v}_{(m_B)}>, <\vec{v}_{(m_B+1)}, \cdots, \vec{v}_{(2m_B)}>, \cdots\}^{m_B \cdot clk_x} \quad (2)$$

where we reduce $< \underbrace{\perp, \cdots, \perp}_{m_B} >$ *to* $\perp$

Although the domain interface has asynchronous features, its input and output signals do not violate the causality condition [4] of the demand driven simulation.

In Fig. 6.a.1) and b.1-2), the functionalities of several different instances of up- and down-sampling components are illustrated by a unit data rate input signal s_A and two instances of output signal s'_B and s''_B, in which s''_B has the slowest clock with $clk_B = \frac{4}{3}$ (simply denoted as $s''_B @ \frac{4}{3}$). The signal events are listed in the ascending order of time tags. Without losing or gaining data tokens, the specified input events (shadowed) of s_A are mapped to the output events (shadowed) of s'_B or s''_B in a different clock domain. When $\lambda_{A:B} = \frac{3}{2} > 1$ and clk_B is faster than clk_A, the output data token pattern is getting more sparse, but the original data rate (the number of the *non-absent* values in $\vec{v}$) is kept. Otherwise, $\lambda_{A:B} = \frac{3}{4} < 1$ and the denser data tokens can cause increased data rate; especially, when $m_B \neq 1$, the data rate of the output events can vary, as shown in Fig. 6.b.2). However, as the output data stream is always buffered by a FIFO, it does not violate the static data rate assumption for process computation. As shown in the upper-right part of Fig. 6, the output signal s_4 of the FIFO always provide a constant data rate 2 required by p_4. To ensure consistent timing, $DI_{B:A}$ from domain D_B to D_A has the reversed sampling ratio $\lambda_{B:A}$.

Domain interfaces act as the glue processes between different synchronous domains. When a domain clock time changes, only the domain interface sampling ratios need to be reconfigured, which greatly facilitates design space exploration in heterogeneous MPSoCs.

5.3 Scheduling state and cross domain analysis

As discussed in Section 3, in a single synchronous domain D_N, at time tag $g_{(n)}$ ($n \in \mathbb{N}_0$) the process status list $T'_{N(n)}$ associates with each process the remaining number of time slots when it *executes*, or 0 when it *stalls*; meanwhile, the FIFO status list $\Gamma'_{N(n)}$ associates with each channel the amount of FIFO storage space used. The scheduling state in D_N is a tuple $(T'_{N(n)}, \Gamma'_{N(n)})$.

A multi-clock application consists of a set of synchronous domains $\mathbf{D}$, in which domain D_N with the slowest domain clock clk_N is chosen as the logic mold domain. Each component (process or FIFO) status signal in D_K ($\forall D_K \in \mathbf{D}$) can be cast across domain boundary into the single D_N via $DI_{K:N}$, where $\lambda_{K:N} \leq 1$. Such a single component status signal casting ($\lambda_{K:N} = \frac{3}{4}$) can be illustrated by Fig. 6.a.1) and b.2). We look s_A as the status signal in D_K and s''_B the status signal in the mold domain D_N. We call each value at time tag $g_{(n)}$ in s''_B a scheduling pattern.

To be consistent with the scheduling state definition in the single domain, we encode the scheduling patterns into incrementing numbers starting from 0, and use the same number to denote the revisited patterns. The functionality of such a encoding module *patternEnc* is shown in Fig. 7. In $s_{patternSt}$, the scheduling patterns at time tags 1 and 4 are duplicated, and are encoded as the same 1 in s_{encSt}. In this way, we can analyze the timing properties for multi-clock applications by casting the system scheduling states in multi-clock domains into the single mold domain D_N.

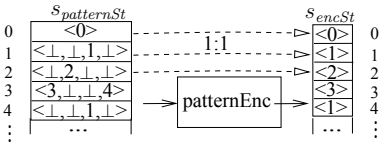


Figure 7: Scheduling patterns encoding

5.4 Throughput analysis

Caused by the bounded process computation latencies and FIFO sizes, the state space of the application scheduling state is always finite. However, as mentioned in Section 3.1, our streaming applications can run infinitely, which means some scheduling states will be revisited in its possibly non-terminating schedule. When this happens, the application schedule enters a periodic phase, such as the schedules in Fig. 2.a-c), caused by the self-timed determinism [20].

In the logic mold domain D_N, when the application scheduling state $(T'_{N(n_2)}, \Gamma'_{N(n_2)})$ at time tag $g_{(n_2)}$ meets a visited one as at $g_{(n_1)}(n_1 < n_2,\ T'_{N(n_2)} = T'_{N(n_1)}\ and\ \Gamma'_{N(n_2)} = \Gamma'_{N(n_1)})$, the application schedule enters a periodic phase with length $g_\Delta = g_{(n_2)} - g_{(n_1)}$. In this periodic phase, the

[4]For the Lemma and Proof, see Appendix A

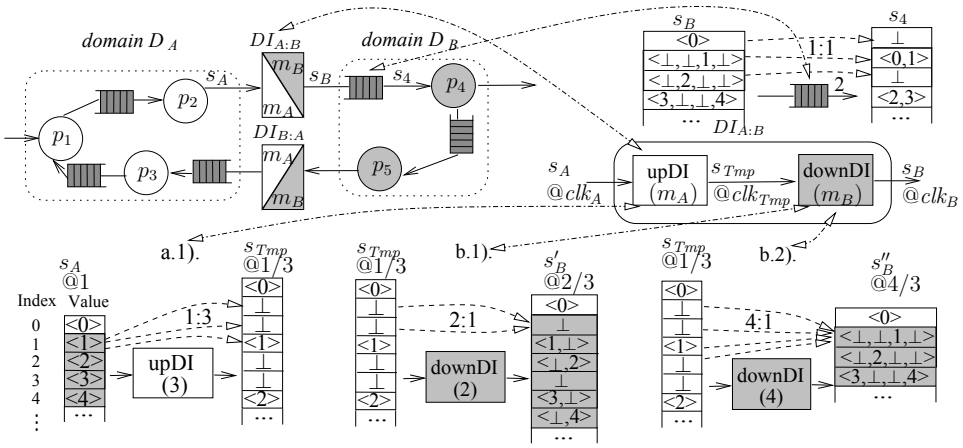


Figure 6: Multi-clock synchronous domains and domain interfaces

process throughput of p_n is defined as

$$Thru(p_n) = \frac{r_{p_n} \cdot I_{p_n} \cdot Sz_{token}}{g_\Delta} \quad (3)$$

in which r_{p_n} is the execution times of p_n, I_{p_n} the output data rate and Sz_{token} the data size per token. When Sz_{token} in bit and the corresponding physical time to g_Δ are known, equation (3) defines the process throughput in bit/s. We simply use the throughput of the sink process as the application throughput, which is the speed the application delivers outputs.

6. ENERGY EFFICIENCY DESIGN ON MP-SOCS

Our design objective is to minimize the system energy dissipation while meeting the required application throughput.

6.1 Implementation model

The architecture model is an MPSoC template, which consists of on-tile components and a packet switched communication network-on-chip (NoC), as shown in the lower part of Fig. 8. Each tile contains one processor (μp) and one local SRAM memory (mem), and is decoupled from the communication network through the network interface (NI). Each processor has a single-cycle access to the local SRAM memory, and no cache is needed.

Given an architecture model with a set of processors **U** and a set of memories **M**, the implementation model (a resource-aware refinement of the application model) has the mappings from a set of computation processes **P** onto **U** and a set of FIFOs **F** onto **M**. For simplicity, we assume FIFOs are mapped to disjoint memory regions as did in [21], and do not consider the techniques in [9] allowing buffer sharing among the FIFOs mapped to the same memory module.

As the mapping optimization on the NoC communication has been studied in [12] and is out of the scope of our paper, we simply use one empirical mapping strategy as the start of our work. Furthermore, we do not investigate the design flexibility in the communication backbone and simply consider the design option in the NoC communication logic to be fixed; instead, we concentrate on exploring the design alternatives of on-tile components.

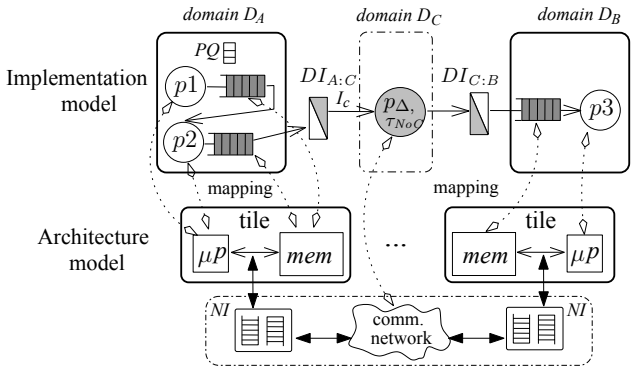


Figure 8: Mapping of the implementation model of the example application (Fig. 1) onto a NoC based MPSoC architecture model

The mapping of the implementation model of the example application (in Fig. 1) onto a two-tile NoC based MPSoC architecture is shown in Fig. 8. We capture the MPSoC architectural characteristics in the implementation model, with the following strategies:

1. Each process p_i is mapped to one on-tile processor μp_j, each FIFO $FIFO_i$ to one memory mem_j, and inter-tile channels to NoC communication.

2. Each tile and the NoC communication are modeled in individual domains, and they interact via domain interfaces. In Fig. 8, the sampling ratios of $DI_{A:C}$ and $DI_{C:B}$ correspond to the heterogeneous timing between different architecture modules.

3. When multiple processes are mapped onto one processor (see domain D_A), their executions are scheduled sequentially according to the non-preemptive assignments in a priority queue (PQ).

4. When multiple FIFOs are mapped onto one memory (see domain D_A), they use independent logic addresses without space sharing, and the memory *size* is the summation of the FIFO *size*s.

5. From the work in [18], we assume that the NoC logic provides the communication between different tiles with a guaranteed bandwidth. In Fig. 8, the inter-tile data token transmission delay quantified by process $p_{\Delta,\tau_{NoC}}$ is predictable, with

$$\tau_{NoC} = \lceil \frac{\frac{I_c \cdot Sz_{token}}{w_{NoC}} + t_{hop}}{clk_C} \rceil \quad (4)$$

in which I_c is the input data rate of $p_{\Delta,\tau_{NoC}}$, w_{NoC} the reserved NoC bandwidth and t_{hop} the network hop delay.

6.2 Design objectives: application throughput and energy efficiency

The architectural design options on the processor *voltage-frequency* levels are customized by the domain interface configurations, and memory *size*s by the parameterized FIFOs. For each instance of the implementation model, the processes computation timing and energy properties are profiled by the cycle-accurate processor energy simulator Sim-Panalyzer [2] and the memory simulator Cacti [23]. As shown in Fig. 9, Sim-Panalyzer is configured with certain *voltage-frequency* levels, and provides the computation timing and dynamic energy dissipation for the gcc cross-compiled binary of each process. We assume that the static power of the processor during customization is constant and is ignored in calculation. Meanwhile, the memory accessing patterns are profiled in an execution trace file, based on which the static and dynamic energy dissipations of the memory with specified *size* are estimated with Cacti.

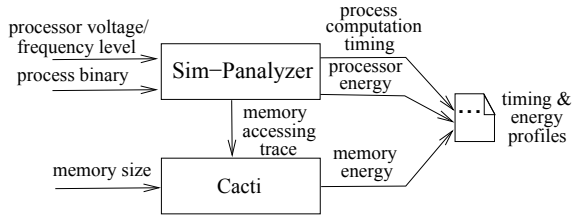


Figure 9: Process timing and energy profiling

With each process computation latency resolved from the computation timing profiles and each communication delay from the guaranteed bandwidth NoC communication logic, the timing of each process on the implementation model is specified. While the processes bounded on the same tile are scheduled sequentially, the application self-timed schedule is determined by its scheduling state. We can use the same techniques as introduced in Section 5.4 for the resource-aware application throughput analysis.

Within a given throughput requirement, the amount of stream data transmitted via the NoC communication is fixed during some period of time. In [12], the *bit energy* model reveals that the average energy consumption to send one bit on NoC is proportional to the Manhattan distance between two tiles. From this metric, the NoC energy dissipation for streaming applications with guaranteed throughput is static and will not be counted in our system energy analysis.

The process and memory energy dissipations are estimated for each time slot in a state based way. When all the processes mapped onto a processor μp_j are stalled, the processor is in *idle* mode and only consumes the static energy $\dot{E}_{\mu p_j}$ (to be ignored as we mentioned), so does the local memory mem_j consumes $\dot{E}_{mem_j}$; otherwise, the processor and memory are in *active* mode, they also consume the dynamic energy ($\hat{E}_{\mu p_j}$ and $\hat{E}_{mem_j}$) of the *executing* process profiled by Sim-Panalyzer and Cacti.

While satisfying a given application throughput requirement $Thru_0$ upon the sink process p_{sink}, we aim to minimize the overall system energy E_{Sum}, which is the energy summation of all the processor and memory modules. The design objectives and constraints are formalized as the following:

$$\begin{aligned}
\min \quad E_{Sum} = & \sum_{\forall \mu p_j \in \mathbf{U}} \hat{E}_{\mu p_j} \\
& + \sum_{\forall mem_j \in \mathbf{M}} (\hat{E}_{mem_j} + \dot{E}_{mem_j}) \quad (5) \\
\textit{where} \quad & \hat{E}_{\mu p_j} = \sum_{\forall p_j \in \mathbf{P}} (t_i \cdot r_i \cdot \hat{\mathcal{P}}_{\mu p_j, p_i}(v_j^2, f_j) \cdot \varphi_{p_i,\mu p_j}) \\
& \hat{E}_{mem_j} = \hat{\mathcal{E}}_{mem_j}(\sum_{\forall FIFO_i \in \mathbf{F}} (\gamma_i \cdot \varphi_{FIFO_i,mem_j})) \\
& \dot{E}_{mem_j} = g_{(\Delta)} \cdot \dot{\mathcal{P}}_{mem_j}(\sum_{\forall FIFO_i \in \mathbf{F}} (\gamma_i \cdot \varphi_{FIFO_i,mem_j})) \\
\text{subject to} \quad & Thru(p_{sink}) \geq Thru_0 \\
& \sum_{\forall \mu p_j \in \mathbf{U}} \varphi_{p_i,\mu p_j} = 1, \qquad \forall \varphi_{p_i,\mu p_j} \in \{0,1\} \\
& \sum_{\forall mem_j \in \mathbf{M}} \varphi_{FIFO_i,mem_j} = 1, \ \forall \varphi_{FIFO_i,mem_j} \in \{0,1\}
\end{aligned}$$

in which $\varphi_{p_i,\mu p_j}$ and φ_{FIFO_i,mem_j} are the decision variables (equals to 0 or 1) to determine the mapping from $\mathbf{P}$ onto $\mathbf{U}$ and $\mathbf{F}$ onto $\mathbf{M}$, $\dot{\mathcal{P}}_{mem_j}$ the *static* power function of the specified mem_j, and $g_{(\Delta)}$ the length of the periodic phase.

The design space exploration has the complexity $\mathcal{O}(n^{|U|+|F|})$, which is NP-hard regarding the problem size. Thus, we use the widely used heuristic algorithms (i.e., greedy and taboo [5] search) for the design options optimization.

7. CASE STUDY

To evaluate the potential of our design flow in energy efficiency design, we apply it on the software FM radio application (presented in [8]) on a NoC based 4×4 mesh tiles MPSoC.

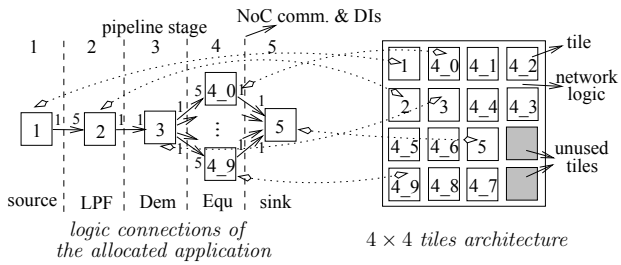


Figure 10: FM radio application mapped onto 4×4 tiles

The application model has 44 concurrent processes and are clustered into 14 partitions in a five-stage pipeline, as shown on the left side of Fig. 10. The pipeline consists of a

Table 1: Voltage-Frequency levels

Voltage(V)	2.5	1.8	1.4	1.2
Frequency(MHZ)	233	180	140	100

signal source, a low-pass filter (LPF), a demodulator (Dem), an equalizer (Equ) with 10 children modules over a range of frequencies, and a signal sink. The arrows marked with numbers show the logic connections and data rates between different partitions.

With each partition allocated to one tile, an empirical mapping from the clustered application to the 14 tiles (the other two are left unused) is shown in Fig. 10. Each pipeline stage of the application is modeled in one individual synchronous clock domain, so is the network logic. The dashed-lines between each pipeline stage stand for the implicit NoC communication and domain interfaces. With the static clock abstraction in the network communication domain, we take it as the logic mold domain in the multi-clock synchronous model. We use symmetrical on-tile resources and NoC communication configurations for the 10 children modules in stage 4, which helps to decrease the problem size.

Each on-tile processor is a StrongARM SA-1100 with the customizable *voltage-frequency* levels shown in Table 1, and each local memory is SRAM with the *size* given by the FIFO buffers to be implemented on this tile. The C program task of each process is compiled into binary using the ARM gcc toolchain and used as the input of Sim-Panalyzer and Cacti for architectural energy estimations. In addition, as the energy results for each customizable module is independent of each other, they could be simulated off-line in a linear time proportional to the problem size, and saved in a look-up table for design exploration.

At the application sink process, a unit data token is a compound data type containing 512 32-bit values. According to the sink process throughput, design optimizations have been performed based on three different application workloads. They have the required application throughput 640 kb/s (*Thru-1*), 533 kb/s (*Thru-2*) and 400 kb/s (*Thru-3*) respectively.

As the baseline, an *Ad-hoc* design method [5] is served as the reference. The minimal memory requirements (*Min-Memory*) technique in [21] emphasizes the design optimization in memory usage. We implement both heuristic *Greedy* and *Taboo* [5] search as our methods in design optimizations to make full use of the architectural customizability in our design exploration flow. The termination criteria of both methods are a specified number of iterations, e.g. 10 iterations, during which the objective function Eq. 5 has no improvements on its value.

Within all the workloads, the computation schedules on multiple processors achieved by the design options with different methods are shown in Fig. 11. Being aware of the global timing, the schedules modeled in different clock domains are elaborated on a normalized time axis. Each processor μp_i corresponds to the processor on the tile marked with i in Fig. 10. In addition, the computations on all the $\mu p_{4\_n}\,(0 \leqslant n \leqslant 9)$ in the pipeline stage 4 are symmetrical and have the same scheduling behaviors. We take only one sample in this stage to evaluate the system energy and processor computation efficiency. All the design options meet the application throughput requirements. The heavier the workload is, the denser the periodic execution pattern is. The design options, which require the processors to run at a relatively lower frequency, get the higher computation efficiency(the average *executing* ratio in the processor schedules), as shown in the η columns of Table 2.

[5]From 100 randomly selected design choices with satisfied application throughput, the one with median energy consumption is chosen as the *Ad-hoc* design option, similar to the approach Hu et al. [12] adopts.

The results of the experiments, to deliver a specified amount of application output data in the periodic schedules, are summarized in Table 2, in which δ denotes the energy saving ratio achieved by each other design method according to the *Ad-hoc* approach. In all the experiments, the heuristics methods get the optimized design option in 10^3 searches, which is trivial compared to the searching space, i.e., $> 5 \times 10^9$. Analyzing the experiments we can derive the following results:

1. With minimal memory usage, the *Min-Memory* method consumes the smallest memory energy E_M. However, it ignores the customizability of processors, and achieves only a limited energy saving at around 10%.

2. Higher computation efficiency η does not always mean higher system energy efficiency E_{Sum}. Instead, an energy efficient design assigns the minimum energy budget on both processor and memory modules with a properly pipelined parallelism. For example, in workload *Thru-2* of Table 2 *Taboo* gets the η of 40.7%, which is lower than 53.0% of *Greedy*, but its design option is more energy efficient in E_{Sum} (3.3 mJ less).

3. Our method with the *Greedy* search shows the energy savings δ at around 19%. Using the *Taboo* search, which could escape from the local optima in the searching space, even better solutions at 21% can be found.

We conclude that our design flow with the *Taboo* search takes the advantage of the customizable computation and storage modules, and can be used to design energy efficient streaming applications with guaranteed throughput on MPSoCs.

8. CONCLUSION AND FUTURE WORK

We propose an energy efficient design exploration flow for streaming applications with guaranteed throughput on MPSoCs. Both application throughput analysis and system energy calculation have been carried out on a multi-clock synchronous MoC framework. Instead of only analyzing the memory efficiencies or processor utilizations, our intent is to minimize the overall energy cost. Our investigations suggest that focusing on either best memory efficiencies or processor utilizations is more likely to result in less optimized implementations. By using the heuristic Taboo search, we can find better solutions in terms of total energy cost, which is also supported by our experiments.

Although we focus on the customization of the on-tile resources, our method could be extended to include the communication backbones. In the future, we plan to parametrize the NoC communication logic and use the flexibility in data transmission for energy efficiency design.

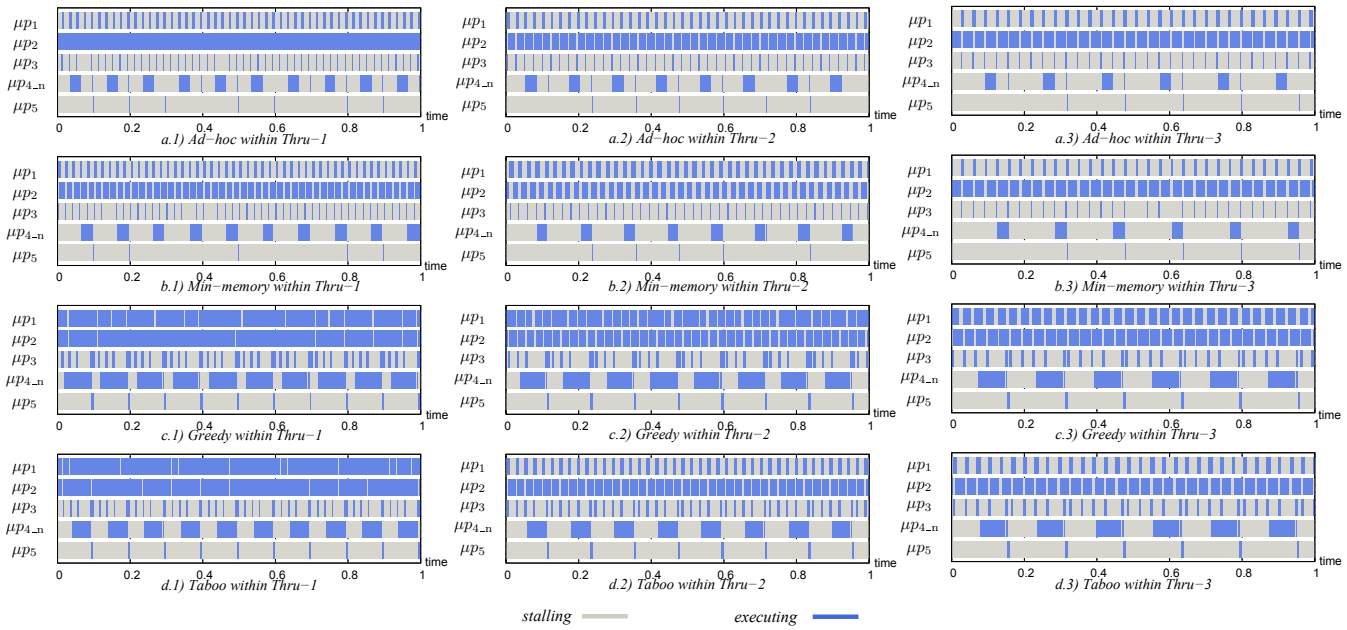


Figure 11: Periodic schedules of the FM radio application on multiple processors. (Three workloads considered are *Thru-1*=640 kb/s, *Thru-2*=533 kb/s and *Thru-3*=400 kb/s. The schedule of μp_{4_n} $(0 \leqslant n \leqslant 9)$ represents the symmetrical computations in the pipeline stage 4.)

Table 2: Experimental results

	Thru-1					*Thru-2*					*Thru-3*				
	E_U	E_M	E_{Sum}	δ[%]	η[%]	E_U	E_M	E_{Sum}	δ[%]	η[%]	E_U	E_M	E_{Sum}	δ[%]	η[%]
Ad-hoc	28.1 [a]	181.2	209.2	-	37.5	31.0	174.4	205.4	-	32.0	34.1	174.5	208.6	-	28.5
Min-memory	33.9	152.3	186.2	11.0	33.0	32.9	152.2	185.2	9.8	29.7	34.1	152.4	186.5	10.6	27.0
Greedy	9.3	159.0	168.3	19.6	61.5	9.8	158.8	168.6	17.9	53.0	8.2	155.8	163.9	21.4	47.5
Taboo	10.3	155.6	165.8	20.7	55.3	12.8	152.4	165.3	19.5	40.7	10.9	152.6	163.5	21.6	38.2

[a]The unit of energy is mJ.

9. ACKNOWLEDGMENTS

Thanks to Mladen Nikitovic, Sandro Penolazzi, Roshan Weerasekara, Kaiyu Chen, Dr. Sander Stuijk, and Dr. Juan Chen for the useful discussions, and four anonymous reviewers for helpful comments on the techniques and content of this paper.

10. REFERENCES

[1] ARM Ltd. `http://www.arm.com`.

[2] The SimpleScalar-ARM power modeling project. `http://www.eecs.umich.edu/~panalyzer/`.

[3] L. Benini, M. Ferrero, A. Macii, E. Macii, and M. Poncino. Supporting system-level power exploration for DSP applications. In *GLSVLSI '00*, pages 17–22, New York, NY, USA, 2000. ACM.

[4] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[5] D. Cvijovic and J. Klinowski. Taboo Search: An Approach to the Multiple Minima Problem. *Science*, 267:664–666, Feb. 1995.

[6] M. Duranton. The challenges for high performance embedded systems. In *DSD '06*, pages 3–7, Washington, DC, USA, 2006. IEEE Computer Society.

[7] C. Erbas. *System-Level Modeling and Design Space Exploration for Multiprocessor Embedded System-on-Chip Architectures*. PhD thesis, 2006.

[8] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *Source ACM SIGOPS Operating Systems Review archive*, 40(5):151–162, 2006.

[9] R. Govindarajan, G. R. Gao, and P. Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing*, 31(3):207–229, July 2002.

[10] P. L. Guernic, J. Talpin, and J. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design*, 2002.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] J. Hu and R. Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. In *ASPDAC '03*, pages 233–239, New York, NY, USA, 2003. ACM.

[13] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. In *IEE Proceedings on Computers and Digital Techniques*, pages 114–129, 2005.

[14] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Marie. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1335, September 1991.

[15] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[16] E. A. Lee and T. M. Parks. Dataflow process networks. *IEEE Proceedings*, 83(5):773–799, May 1995.

[17] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[18] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *DATE '04*, page 20890, Washington, DC, USA, 2004. IEEE Computer Society.

[19] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(1):17–32, 2004.

[20] S. Sriram and S. S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC Press, 2000.

[21] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *DAC '06*, pages 899–904, CA, USA, July 2006.

[22] S. wei Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for Brook streaming applications on multiprocessors. In *CGO '06*, pages 196–207, Washington, DC, USA, 2006. IEEE Computer Society.

[23] S. J. Wilton and N. P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

APPENDIX

A. DOMAIN INTERFACE CAUSALITY

LEMMA 1. *Domain interface $DI_{A:B}$ has input signal s_A at domain D_A and output signal s_B at domain D_B, as shown in Fig. 12.* $\forall a_1 \in \mathbb{N}_0$, $\exists b_1 \in \mathbb{N}_0$,

$$\begin{aligned} DI_{A:B}(\{\cdots, \vec{v}_{A(a_1)}, \cdots\}^{clk_A}) &= \{\cdots, \vec{v}_{B(b_1)}, \cdots\}^{clk_B}, \\ where \quad \{\vec{v}_{B(b_1)}\}^{clk_B} &= \{<\cdots, \vec{v}_{A(a_1)}, \cdots>\}^{clk_B}. \end{aligned}$$

s.t. the timing relation $Tag(\vec{v}_{\mathrm{B}(\mathrm{b}_1)}) \geqslant Tag(\vec{v}_{\mathrm{A}(\mathrm{a}_1)})$ exists, in which the operator Tag is to get the time tag $g_{(n)}$ for a specified signal value $\vec{v}_{(n)}$. Hence, $DI_{A:B}$ has causality between its input and output signals.

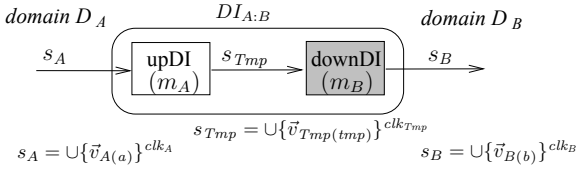


Figure 12: Domain interfaces $DI_{A:B}$

PROOF. In case of $a_1 = b_1 = 0$, $Tag(\vec{v}_{B(0)}) \equiv Tag(\vec{v}_{A(0)}) \equiv 0$ meets the timing relation. Otherwise, $a_1 \geqslant 1$ and $b_1 \geqslant 1$. From the definition of $upDI(m_A)$ in (1), we know that $clk_{Tmp} = \frac{clk_A}{m_A}$. From the definition of $downDI(m_B)$ in (2), we know that $\forall b_1 \in \mathbb{N}_0$

$$\begin{aligned} \{\vec{v}_{B(b_1)}\}^{clk_B} &= \{<\cdots, \vec{v}_{A(a_1)}, \cdots>\}^{clk_B} \\ &= downDI(m_B) \\ &\qquad (\{\vec{v}_{Tmp(m_B\cdot(b_1-1)+1)}, \cdots, \vec{v}_{Tmp(m_B\cdot b_1)}\}^{clk_{Tmp}}) \end{aligned}$$

Tag is monotonically increasing, thus

$$Tag(\vec{v}_{A(a_1)}) \leqslant Tag(\vec{v}_{Tmp(m_B\cdot b_1)})$$

From the definition of *Tag*, we know that

$$\begin{aligned} Tag(\vec{v}_{B(b_1)}) &= b_1 \cdot clk_B \\ Tag(\vec{v}_{Tmp(m_B\cdot b_1)}) &= m_B \cdot b_1 \cdot clk_{Tmp} = b_1 \cdot (m_B \cdot \frac{clk_A}{m_A}) \\ &= b_1 \cdot clk_B. \end{aligned}$$

We get the conclusion $Tag(\vec{v}_{B(b_1)}) \geqslant Tag(\vec{v}_{A(a_1)})$. □