Universiteit Leiden
The Netherlands

Estimation and Optimization of the Performance of Polyhedral Process Networks

Haastregt, S. van

Citation
Haastregt, S. van. (2013, December 17). *Estimation and Optimization of the Performance of Polyhedral Process Networks*. Retrieved from https://hdl.handle.net/1887/22911

Version:	Corrected Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/22911

Note: To cite this publication please use the final published version (if applicable).

Cover Page

The handle http://hdl.handle.net/1887/22911 holds various files of this Leiden University dissertation.

Author: Haastregt, Sven Joseph Johannes van
Title: Estimation and optimization of the performance of polyhedral process networks
Issue Date: 2013-12-17

Chapter 4

Performance Estimation

In the previous chapter, we have presented methods to realize an FPGA implementation of a polyhedral process network. Considering a single optimized design point does not necessarily result in the best tradeoff between area and performance aspects. Instead, a designer wants to consider different design points that provide different tradeoffs between for example area and performance aspects. In this chapter, we present four different methods to estimate the performance of design points specified as polyhedral process networks, that differ in accuracy and assessment effort.

4.1 Motivation

Looking at Figure 1.2, which shows the iterative design flow, it becomes clear that the designer gets feedback very late. Only after time-consuming synthesis and place-and-route steps does the designer get feedback about performance. This limits the number of design points that a designer can evaluate in a given amount of time. Since he can evaluate only a limited number of design points, assessing if his design constraints can be met is difficult and frustrating. Prototyping for example a sobel design consisting of five LAURA processors already takes about twenty minutes [HK09]. Also, the forward synthesis flow requires that synthesizable RTL for all components is available, which is often not the case at the early stage of a design process. Instead, the designer should obtain feedback faster, possibly at the expense of reduced accuracy, allowing him to avoid a time-consuming forward synthesis step if he knows a design will not satisfy his constraints. Ideally, a designer wants to know whether a design meets his constraints before entering the time-consuming forward synthesis step.

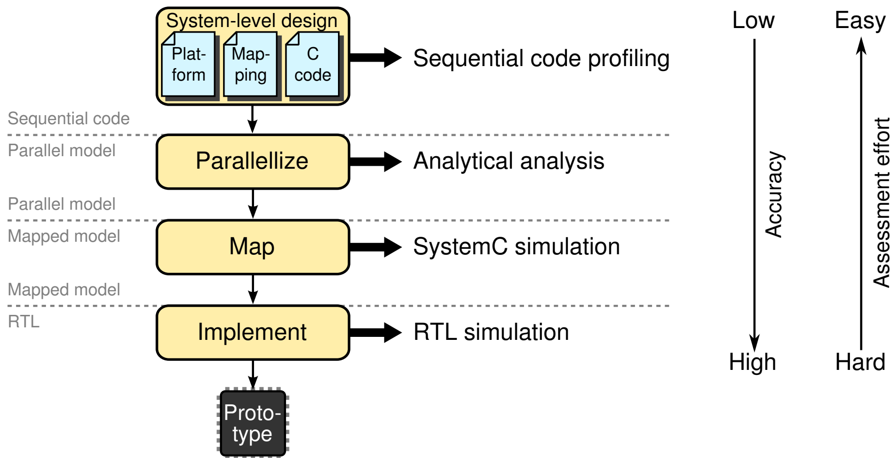


Figure 4.1: Performance assessment at different levels of the Daedalus design flow.

Getting an early performance estimate of a design is not new and has been investigated by for example Meijer et al. [MNS10] and Nikolov [Nik09]. However, these approaches only focus on microprocessor based systems. These approaches are not able to capture the notion of overlap between different iterations of a process and cannot handle cyclic PPNs, rendering them unsuitable for our design flow. Therefore, we investigate in this chapter four different techniques to provide the desired performance estimate.

From the Daedalus design flow, we distill four different levels, as shown in Figure 4.1. For each level, we investigate how to obtain a performance estimate. At the first level, the designer creates a system-level specification consisting of sequential C code and a platform and mapping specification in XML. At the second level, PNGEN parallelizes this C code into a parallel model. At the third level, ESPAM maps the parallel model onto a platform. At the fourth level, commercial synthesis tools implement the low-level RTL model.

We expect that performance assessments at these different levels provide different trade-offs between accuracy and assessment effort [HH96, KDWV02]. A high-level performance assessment can be conducted in a short amount of time, but such high-level performance numbers often deviate from the actual performance of the prototype. On the other hand, low-level performance assessments take a considerable amount of time, but the resulting performance numbers are often very close to the actual performance.

In this chapter, we study the relation between accuracy and assessment effort of performance assessments at the four different levels. We start our investigation from the

RTL at the fourth level and work our way up to the sequential code at the first level. We first present some definitions and concepts that we use to discuss performance estimations in Section 4.2. Performance assessment at the fourth level, RTL simulation, is discussed in Section 4.3. Performance assessment at the third level, SystemC simulation, is discussed in Section 4.4. Performance assessment at the second level, analytical analysis, is discussed in Section 4.5. Performance assessment at the first level, sequential code profiling, is discussed in Section 4.6. After the discussion of performance assessments at the four levels, we compare the four approaches on different aspects such as the ability to incorporate finite buffer sizes in Section 4.7. In Section 4.8, we compare the four approaches by applying each approach on a set of benchmarks. In Section 4.9, we summarize this chapter.

4.2 Definitions

Design constraints on system performance are often expressed as a constraint on throughput [SB00]. To quantify the performance of a process in a PPN, we employ the notion of throughput:

Definition 4.1 (Process Period and Throughput)**.**
The *period* T_p of a PPN process p represents the average time between two subsequent firings of p. The *throughput* $\tau_p = \frac{1}{T_p}$ of a process p represents the average number of firings completed per time unit.

Using the notion of process throughput, we define the throughput of a PPN:

Definition 4.2 (PPN Throughput)**.**
The throughput of a PPN with one sink process equals the throughput of that sink process.

This definition excludes PPNs with more than one sink process. This is not a fundamental limitation because each such PPN can be transformed into a PPN with only one sink process by merging the sink processes. Alternatively, instead of reasoning about the throughput of the entire PPN, one may keep the distinction between different sink processes since they represent different output streams of a system. For example, a video processing system may have a high data rate video output stream and a low data rate control output stream. Combining both data rates is meaningless in practice, and thus it is desirable to keep both throughput rates separated.

The external input and output streams connected to the PPN may affect the throughput achieved by a PPN. For example, if data on the input stream is not delivered fast enough, the throughput of a PPN may drop as the PPN has to wait for data. Similarly,

if data on the output stream is not consumed fast enough, the PPN may be stalled until older data is consumed from the output stream such that storage space for new data becomes available. In this chapter, we are interested in the throughput of a PPN irrespective of environmental factors. Therefore, we employ the notion of isolated throughput:

Definition 4.3 (Isolated Throughput)**.**
The isolated throughput of a PPN is the throughput of the PPN when isolated from external input and output streams.

As such, the isolated throughput represents the theoretical maximum achievable throughput considering only the PPN itself. In the remainder of this chapter we present and review four different techniques to estimate the isolated throughput of PPNs. We want to estimate the throughput of a PPN on a real system, which we refer to as the *absolute throughput*.

Definition 4.4 (Absolute Throughput)**.**
An absolute throughput assessment is used to describe the throughput of an actual FPGA implementation of a PPN.

The goal of each of the four techniques that we present in this chapter is to analyze the performance of a multi-processor system. According to van Gemund, the performance of a parallel system is determined by four key aspects [Gem96]:

- **Conditional synchronization**, which relates to the performance impacts of synchronization due to data dependences. For example, if a PPN process depends on two inputs a and b that become available at times t_a and t_b, then the process should fire no earlier than $\max(t_a, t_b)$.
- **Mutual exclusion**, which relates to contention of processing or communication resources. For example, a processor can only initiate the next PPN process iteration at a valid initiation interval (II) boundary.
- **Basic calibration**, which relates to the performance characteristics of the system constituents. For example, Definition 3.1 provides a systematic way to describe the throughput (II) and latency (Λ) of an IP core that is integrated in a PPN.
- **Conditional control flow**, which relates to non-static control flow inferred by data-dependent control statements. For example, a process function may have a varying latency if the function performs a different computation for different input argument values.

To obtain an accurate assessment of PPN performance, we take the four key aspects into account in the four performance estimation techniques that we present.

4.3 RTL Simulation

At the fourth level in Figure 4.1, we have obtained an FPGA project of the system. This project can be synthesized using vendor-specific low-level synthesis and place-and-route tools to obtain a bitstream. By downloading the bitstream onto an FPGA device, the designer obtains a prototype implementation of the design such that for example functionality and throughput requirements can be verified. However, even for small designs, synthesis of an FPGA project to a bitstream already takes tens of minutes. Obtaining a throughput metric by prototyping is thus a time-consuming approach.

The RTL representation of an FPGA project can be simulated such that low-level synthesis and place-and-route steps are avoided during throughput assessment. The feasibility of such a simulation depends on the types of processors in the platform specification. If one or more programmable processors are involved, an RTL simulation is time-consuming because of the large amount of effort required to simulate a single instruction at the register transfer level, making RTL simulation impractical when using programmable processors. Nevertheless, for platforms consisting entirely of LAURA processors, we found that RTL simulation is a viable approach to obtain a throughput estimate of a design. Therefore, we have extended ESPAM with a backend that produces an RTL simulation project for platforms that consist entirely of LAURA processors. This backend generates a simulation project for the Xilinx ISE simulator.

4.4 SystemC Simulation

At the third level in Figure 4.1, we have obtained a mapped model of the system. As discussed in the previous section, RTL simulation of platforms containing one or more programmable processors is often infeasible in practice. To make simulation of such platforms feasible, we may reduce the amount of simulation details at the expensive of lower accuracy. We achieve this by simulating the mapped model instead of the RTL model, thereby addressing the basic calibration and conditional control flow aspects in less detail. A common solution is to use different simulation techniques for different types of components, known as *co-simulation* [GCD92, Row94]. The different components are then simulated at different levels of detail. Another solution is to use execution traces of an application to simulate different system-level specifications, as done for example by Sesame [PEP06] which is integrated in Daedalus. A widely used standard for simulation of designs with reduced accuracy is the SystemC standard [Sys05]. We have extended ESPAM with two SystemC backends: an untimed SystemC backend and a timed SystemC backend.

The *untimed SystemC* backend generates a functional simulation in the SystemC environment. One of the first backends in the history of ESPAM was the YAPI backend which generates a functional simulation in the YAPI framework [KES+00]. The YAPI backend provides fast functional simulation of a PPN, such that a designer can quickly verify if the functional behavior of a parallelized application is correct. The motivation behind the untimed SystemC backend is to provide similar fast functional simulation, but according to an industry standard. Unlike the YAPI framework, SystemC is an official IEEE standard which implies a more widespread acceptance and better long-term support.

The *timed SystemC* backend generates a functional simulation which includes a notion of time. A designer can use a timed SystemC simulation to obtain throughput metrics in less time than with an RTL simulation. We have explored two different approaches to incorporate programmable processors into timed SystemC simulations. In Section 4.4.1, we present an approach which employs a cycle-accurate instruction set simulator which yields cycle-accurate throughput metrics. In Section 4.4.2, we present an approach which uses fixed execution time estimates, thereby potentially degrading accuracy but further increasing the simulation speed.

4.4.1 Cycle-Accurate Timed SystemC Simulation

To obtain a cycle-accurate simulation environment from a system level specification, we have developed a new backend to ESPAM [HHK10]. This backend generates the C++ code for a SystemC top-level module and the C++ code that has to be run on each processor. The backend currently supports LAURA and MicroBlaze processors. A LAURA processor is simulated using a custom written SystemC module that models the LAURA execution in a cycle-accurate manner. A MicroBlaze processor is simulated using the cycle-accurate GDB-based MicroBlaze *Instruction Set Simulator* (*ISS*) provided by Xilinx. Such an ISS allows for a faster performance assessment, because the ISS simulates only instructions instead of the full RTL implementation of the MicroBlaze processor. However, the MicroBlaze ISS was not designed to operate as a multi-processor simulator. Therefore, ESPAM generates a SystemC top-level module which allows different instances of the ISS to interact.

In Figure 4.2, we show an implementation of the PPN shown in Figure 2.8 using our cycle-accurate timed SystemC simulation model. We map the *source* and *sink* processes onto separate MicroBlaze processors, and the *func1* process onto a LAURA processor. The top-level module contains submodules that implement a simulation model for each processor. The channels of a PPN are implemented using *sc_fifo* primitives from the SystemC standard which interconnect the processor simulation modules.

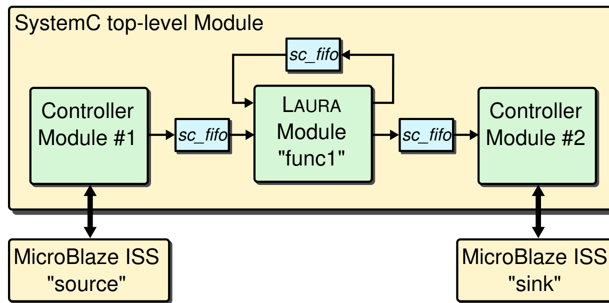


Figure 4.2: A cycle-accurate timed SystemC simulation environment for the PPN of Figure 2.8.

For each LAURA processor, the SystemC top-level module implements a SystemC module that simulates the execution of a process on a LAURA processor described in Section 2.4.1. For each MicroBlaze processor, the SystemC top-level module implements a controller module which drives each MicroBlaze ISS instance, shown in the bottom part of Figure 4.2. This controller module communicates with the ISS instance running as a separate heavy-weight process in the operating system. This allows different ISS instances to run in parallel. A process running on a MicroBlaze processor normally communicates data to other processors via its FSL ports, using the `get` and `put` instructions. The original MicroBlaze ISS does not implement these instructions. We have implemented these instructions in the ISS to send and receive data to and from the controller module associated with an ISS instance. The controller module subsequently transfers data between the other simulated processors using *sc_fifo* instances. The `get` instruction stalls the ISS when no data is available on the *sc_fifo* being read, and the `put` instruction stalls the ISS when the maximum capacity of the *sc_fifo* being written is reached. As such, the ISS implements the blocking read and write primitives according to the semantics of the PPN model.

To ensure cycle-accurate simulation, a global execution time should be maintained across all ISS instances. We have modified the ISS such that each ISS instance keeps track of the global execution time. A straightforward way to synchronize the execution times of all ISS instances is to use a *lockstep* approach. With such an approach, each ISS instance waits for a clock signal from the corresponding SystemC controller module before a MicroBlaze instruction is executed. The lockstep approach guarantees that all simulated MicroBlaze processors advance at the same pace, resulting in a cycle-accurate simulation of the system. Unfortunately, the lockstep approach results in extensive synchronization overhead, because synchronization occurs at every simulated clock cycle. Using the lockstep approach, at most 42000 clock cycles can be

simulated per second for a design containing one MicroBlaze processor. When more MicroBlaze processors are simulated simultaneously, this number drops more or less linearly. As an alternative to the lockstep approach, we can synchronize the execution times of two ISS instances only when these ISS instances interact by communicating a data token. Using such global execution time synchronization, we have observed up to 80 times increases in simulation speed.

4.4.2 Light-weight Timed SystemC Simulation

Performing a cycle-accurate timed SystemC simulation using ISSs is a delicate task, because code for all processors has to be compiled for the appropriate instruction set, and because communication channels between the ISS instances need to be established. As a light-weight alternative to cycle-accurate timed SystemC simulation, we propose another timed SystemC simulation technique. Instead of relying on an ISS to obtain cycle-accurate execution times, we require the designer to provide function execution times according to Definition 3.1. Thus, for each function f in the PPN, a value Λ_f represents the number of clock cycles taken by a single invocation of f. By considering only a single value Λ_f, we decrease simulation complexity at the expense of lower accuracy. As a result, we simplify the basic calibration aspect as only one Λ_f value per function is required, instead of a list of latency values per instruction. We ignore the conditional control flow aspect, as functions that may contain data-dependent statements are not executed. As a consequence of the simplification, the accuracy of Λ_f determines the accuracy of the final throughput metric.

For each top-level component in a system, that is, a processor in the platform specification and a channel between two processors, we instantiate a SystemC module. The SystemC module is based on a template for the component type. Each SystemC module runs a thread in which the simulation model of the simulated component is updated at each simulated clock cycle. A top-level module interconnects the SystemC modules and invokes the SystemC simulation kernel. The SystemC kernel schedules all threads according to a discrete-event simulation model that is also employed for RTL simulation. Light-weight timed SystemC simulation thus resembles RTL simulation in which only the conditional synchronization, mutual exclusion, and basic calibration aspects are included in the simulation model. Other aspects that do not affect performance significantly, such as the process functionality and input port multiplexing, are specified as C code without invoking the discrete event scheduler of SystemC. Thus, no simulation primitives are constructed for non-essential aspects, which allows faster simulation compared to RTL simulation.

For an example Sobel design mapped on a platform consisting of five processors, we have measured a simulation speed of about 200000 clock cycles per second. In

terms of simulation speed, this approach roughly compares to the cycle-accurate timed SystemC simulation without lockstep synchronization. Assuming the Λ_f values are accurate, light-weight timed SystemC simulation is a feasible alternative to cycle-accurate timed SystemC simulation.

4.5 Maximum Cycle Mean Analysis

At the second level in Figure 4.1, we have obtained a PPN of the application, which is a particular model of computation. Estimating performance for different models of computation is a well-established field of research [LSV98, SB00]. In this section, we want to leverage existing work to find an analytical performance estimation technique for PPNs. We present a novel analytical technique to estimate the throughput of a PPN based on *Maximum Cycle Mean* (*MCM*) analysis. MCM analysis is an established technique to assess the throughput of an HSDF graph [SB00]. MCM analysis is invariant to the application workload because of the analytical nature. This makes this approach appealing compared to the RTL and SystemC approaches, as the assessment effort of the latter aproaches directly depends on the workload. We present an overview of analytical throughput estimation approaches in Section 4.5.1. We discuss the MCM analysis method for HSDF graphs in Section 4.5.2. We explain how we derive an HSDF graph for throughput estimation of a PPN in Section 4.5.3. We conclude this section by applying MCM analysis to two PPNs in Section 4.5.4.

4.5.1 Related Work

Analytical performance assessment of applications modeled as dataflow graphs is a well-studied research field. An analytical method to compare different instances of an application modeled as a PPN was first presented by Meijer et al.[MNS10]. Their technique had two limitiations. First, the scope was limited to acyclic PPNs. Second, the throughput model was developed to obtain relative throughput assessments between two or more PPNs. In contrast, in this chapter we focus on absolute throughput assessments for both acyclic and cyclic PPNs. Thiele et al. have investigated performance analysis for cyclic SDF graphs [TS09]. Because the approach works only on SDF graphs, it cannot cope with varying production and consumption rates that occur in many embedded applications. Such varying rates can be expressed in the PPN model, but no absolute performance analysis currently exists for PPNs.

The period of an HSDF graph can be analytically obtained by computing the maximum cycle mean [DG98, SB00]. Because a PPN is a special case of a CSDF graph, an equivalent HSDF graph can be derived from a PPN using the *conventional method* with which an HSDF graph can be derived from a CSDF graph [BELP96, Fig. 9].

Moonen et al. use this method to compute a conservative bound on the throughput of a CSDF graph [MBBM07]. Unfortunately, the equivalent HSDF graph often exhibits an exponential increase in the number of nodes compared with the CSDF graph. This increases the running time of the algorithm computing the maximum cycle mean, making analysis of large graphs more time-consuming or even impractical. In Section 4.5.3, we present an alternative approach to enable maximum cycle mean analysis on PPNs which avoids the exponential complexity increase.

Ito and Parhi acknowledge the increases in the number of nodes and edges when deriving the equivalent single-rate data flow ("HSDF") graph for a given multi-rate data flow ("SDF") graph [IP95]. Their solution is to remove edges and nodes through procedures called edge degeneration and node degeneration, in such a way that the iteration bound is not affected. The effectiveness of the approach is not guaranteed, as node degeneration is not applicable for certain graphs, as indicated by the authors.

Instead of working on equivalent HSDF graphs, throughput analysis methods exist that operate directly on SDF [GGS+06] and CSDF graphs [SGB08]. These methods construct the state space of the graph by simulating its execution assuming an unlimited number of processor resources. Once a cycle is detected in the state space, the periodic phase is reached. After identification of the periodic phase, the throughput of the graph can be computed. Instead of performing an explicit state-space exploration, one can also perform the state-space exploration symbolically using max-plus algebra [Gei09]. This allows one to obtain an HSDF graph with identical throughput characteristics that often has fewer nodes than an equivalent HSDF graph obtained using the conventional method. However, for some graphs the method of [Gei09] may produce an HSDF graph that has more nodes than the conventionally obtained HSDF graph.

In summary, existing analytical throughput assessment techniques for PPNs cannot cope with cyclic graphs and are only intended for relative throughput assessment. Various techniques exist for HSDF, SDF, and CSDF graphs that can cope with cyclic graphs and provide absolute throughput assessment. However, techniques for HSDF, SDF, and CSDF graphs cannot be applied directly to PPNs, because the succinct PPN representation has to be converted into a more elaborate HSDF, SDF, or CSDF representation. Such a conversion leads to an HSDF or SDF graph with an exponentially large number of nodes, or a CSDF graph with long phase lengths. The conversion takes an large amount of time, and the size of resulting CSDF graph leads to long running times of the analysis methods. To avoid any potential exponential increase in the number of nodes, we look for an approach in which the number of nodes remains equal to the number of processes in the PPN.

4.5.2 Maximum Cycle Mean Analysis

The *iteration period* of an HSDF graph is defined as the time needed to execute an iteration of the graph [SB00, Chapter 5]. A lower bound on the iteration period, called the *iteration bound*, can be obtained by first computing the computation-to-delay ratios of the cycles in the graph [SB00, Chapter 8]. For each cycle C in an HSDF graph G, we compute the computation-to-delay ratio

$$CM(C) = \frac{\sum_{v \in C} t(v)}{\sum_{e \in C} d(e)}, \tag{4.1}$$

which we refer to as the *cycle mean* of C. Thus, the cycle mean of a cycle C equals the sum of the execution times $t(v)$ of all nodes v involved in C divided by the sum of all initial tokens $d(e)$ on the edges e involved in C. The cycle that yields the maximum $CM()$ value is called the *critical cycle* of an HSDF graph. The iteration bound of an HSDF graph G is determined by the critical cycle. Thus, to obtain the iteration bound we compute the *maximum cycle mean*

$$MCM(G) = \max\{CM(C)\}, \quad C \in G. \tag{4.2}$$

The throughput of an HSDF graph G is the reciprocal of the iteration bound, thus

$$\tau(G) = \frac{1}{MCM(G)}. \tag{4.3}$$

Example

We now illustrate the MCM analysis on the HSDF graph of Figure 2.5 on page 23. This graph contains three cycles:

- $c_1 = (a1 \rightarrow b \rightarrow c \rightarrow a1)$,
- $c_2 = (a2 \rightarrow b \rightarrow c \rightarrow a2)$, and
- $c_3 = (b \rightarrow c \rightarrow b)$.

When auto-concurrency is considered, a node may fire multiple times simultaneously. A node mapped onto a programmable processor executes its firings in sequence, such that no auto-concurrency occurs. To explicitly exclude auto-concurrency of the individual nodes, we assume each node i has a selfloop c_i with one initial token. Then, Equation (4.2) yields

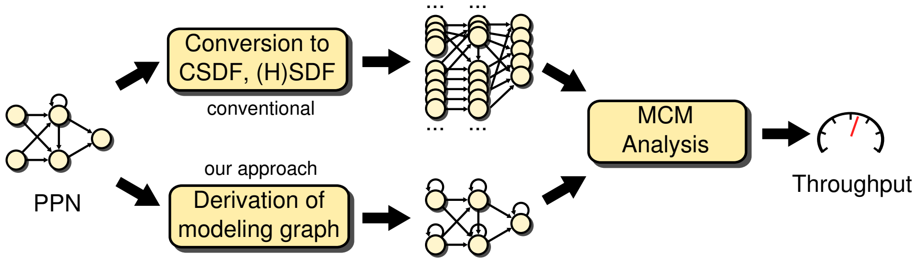


Figure 4.3: Two approaches to apply MCM analysis to PPNs.

$$
\begin{aligned}
MCM(G_{2.5}) &= \max\left\{CM(c_{a1}), CM(c_{a2}), CM(c_b), CM(c_c), CM(c_1), CM(c_2), CM(c_3)\right\} \\
&= \max\left\{\frac{t(a1)}{1}, \frac{t(a2)}{1}, \frac{t(b)}{1}, \frac{t(c)}{1}, \frac{t(a1)+t(b)+t(c)}{d(a1 \to b)+d(b \to c)+d(c \to a1)},\right. \\
&\qquad \left.\frac{t(a2)+t(b)+t(c)}{d(a2 \to b)+d(b \to c)+d(c \to a2)}, \frac{t(b)+t(c)}{d(b \to c)+d(c \to b)}\right\} \\
&= \max\left\{\frac{8}{1}, \frac{8}{1}, \frac{2}{1}, \frac{2}{1}, \frac{8+2+2}{0+0+1}, \frac{8+2+2}{0+0+1}, \frac{2+2}{0+1}\right\} \\
&= 12\,.
\end{aligned}
$$

The first four terms in the max-expressions above correspond to the selfloops of the four nodes. The remaining three terms correspond to the three cycles of the graph. The maximum cycle mean is determined by both $CM(c_1)$ and $CM(c_2)$ which both evaluate to 12. Thus, the iteration bound of the HSDF graph of Figure 2.5 is 12, and consequently the throughput is $\frac{1}{12}$.

4.5.3 Derivation of PPN Modeling Graphs

In Section 4.5.1, we mentioned the possibility of applying MCM analysis on PPNs by considering the equivalent CSDF graph and converting the CSDF graph into HSDF. This approach is depicted in the upper part of Figure 4.3. Unfortunately, this results in an exponential increase in the number of nodes. To keep the time needed for the MCM computation within reasonable bounds, we must avoid the exponential increase in the number of nodes, which leads us to a new approach.

We have found a way to derive a more compact HSDF graph from a cyclic PPN, which we depict in the lower part of Figure 4.3. Our approach works by deriving an HSDF graph that models the throughput behavior of a PPN, and then applying conventional MCM analysis to this graph. The number of nodes in our HSDF graph

equals the number of processes in the PPN. The number of edges in our HSDF graph is linearly bounded, as we show in Proposition 4.3. As such, no exponential increase of the graph size occurs, making the approach a suitable alternative for fast performance estimation of PPNs. We divide the derivation in two main steps. First, PPN processes are converted to HSDF nodes. Second, PPN channels are converted to HSDF edges.

Step 1: Constructing Nodes from Processes

The first step in deriving the PPN modeling HSDF graph is to convert PPN processes to HSDF nodes. One possible approach is to interpret the PPN as a CSDF graph [HZ+10, adg2csdf] and then derive an HSDF graph from this CSDF graph using the conventional approach [BELP96, Fig. 9]. This approach causes $\mathbf{q}(p)$ nodes to be instantiated for each process p. For consistent PPNs, $\mathbf{q}(p)$ always equals the number of points in the process domain D_p:

Proposition 4.1 (PPN Repetition Vector)**.** *For each process p of a consistent PPN, the corresponding element of the repetition vector of an equivalent CSDF graph equals the number of points in its process domain, that is,* $\forall p \in \mathcal{P} : \mathbf{q}(p) = |D_p|$.

Proof. In a consistent PPN, for every channel c, the number of points in the corresponding $OPD^j_{\sigma_c}$ is equal to the number of points in the corresponding $IPD^k_{\delta_c}$, thus $|\mathrm{OPD}^j_{\sigma_c}| = |IPD^k_{\delta_c}|$. Therefore, the solution of the balance equation $\Gamma \cdot \mathbf{r} = \mathbf{0}$ is a vector $\mathbf{r}$ which contains a ‘1’ for every process. As a result, the elements of the repetition vector $\mathbf{q} = S \cdot \mathbf{r}$ are equal to the phase lengths of each node, which equals the number of points in the process domain. □

As a result, a separate HSDF node would be instantiated for each iteration of the domain, resulting in large graphs even for small applications. This makes the conventional CSDF-to-HSDF approach infeasible for practical purposes. We can avoid an increase in the number of nodes based on the following observation. In an HSDF graph, all $\mathbf{q}(p)$ nodes originating from a process p may execute in parallel. However, by definition the iterations of a PPN always execute sequentially. This allows us to represent each process p by a single HSDF node h, where node h represents sequential execution of all $\mathbf{q}(p)$ nodes of the conventional equivalent HSDF graph. We multiply the execution time Λ_p of a single firing of process p by $\mathbf{q}(p)$ to model sequential execution of all $\mathbf{q}(p)$ nodes in the equivalent HSDF graph. As a result, the number of nodes in the resulting HSDF graph equals the number of processes in the original PPN.

The execution time $t(h)$ of an HSDF node is set to the total time needed to fire all iterations of the process consecutively without overlapped execution. Included in this execution time are the read and write latencies and the time needed to fire the function. Time spent on a blocking read or write operation is not included, which means our approach does not address the conditional synchronization aspect introduced in Section 4.2. Our approach cannot accurately assess throughput of applications in which read or write operations block on empty or full channels. To exclude auto-concurrency, we add to each HSDF node a selfloop with one initial token. This avoids multiple simultaneous executions of the entire PPN, which is undesirable when determining throughput.

Step 2: Constructing Edges from Channels

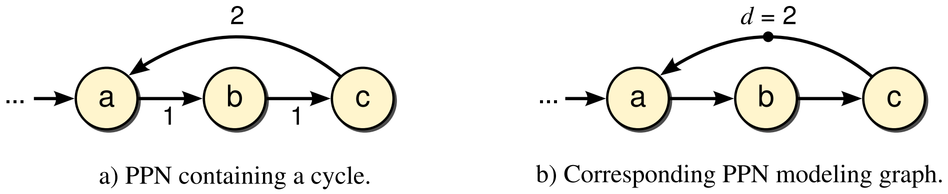
The second step in deriving the PPN modeling HSDF graph is to interconnect the HSDF nodes using edges in such a way that the PPN's throughput characteristics are preserved. This is not trivial, because of the different semantics of HSDF edges and PPN channels: HSDF edges have an unbounded capacity and may contain initial tokens, whereas PPN channels have a bounded capacity and do not have a notion of initial tokens. We now discuss how to represent edges in a PPN modeling HSDF graph such that the PPN's throughput characteristics are preserved.

The PPN modeling graph may contain more than one edge between two nodes a and b, if for example the PPN contains multiple channels between two processes. It is sufficient to represent such a collection of channels by a single edge:

Proposition 4.2 (Pruning Multi-Edges in PPN Modeling Graphs)**.** *A collection of PPN channels from process a to process b can be represented by a single edge $(a \to b)$ in the PPN modeling graph.*

Proof. If an edge $(a \to b)$ is part of a cycle, then another cycle also exists for each additional edge connecting a to b. The only difference among the cycle means of those cycles is the number of initial tokens that occurs in the denominator of Equation (4.1). A cycle with a larger denominator results in a smaller cycle mean, which implies the cycle mean will not be selected by Equation (4.2). Thus, we only need to consider the cycle with the smallest number of initial tokens, which is the cycle containing the edge with the smallest number of initial tokens. □

We distinguish between three classes of channels: selfloop channels, feedback channels, and feedforward channels.



a) PPN containing a cycle. b) Corresponding PPN modeling graph.

Figure 4.4: Handling feedback edges in a PPN.

Selfloop channels

For a selfloop channel, which connects a process to itself, no edge is added to the HSDF graph. We omit such selfloops because in step 1 we have already added a selfloop with one initial token to each node. To see why selfloops can be omitted, suppose that the critical cycle of a PPN modeling graph would be a selfloop s of process p with buffer size $S_s \geq 1$. This selfloop could be modeled by adding an edge $(p \rightarrow p)$ with S_s initial tokens. For this newly added cycle C_s, Equation (4.1) yields

$$CM(C_s) = \frac{\Lambda_p \cdot |D_p|}{S_s}.$$

However, for selfloop e added in line 3 of Algorithm 4.1, we already have

$$CM(e) = \frac{\Lambda_p \cdot |D_p|}{1}.$$

Because $CM(e) \geq CM(C_s)$ for all $S_s \geq 1$, we can ignore $CM(C_s)$ in Equation (4.2). Thus, a selfloop of a PPN never forms the critical cycle, and therefore such selfloops can be omitted from the PPN modeling graph without affecting the MCM value.

Feedback channels

Feedback channels are part of a strongly connected component, and are thus the constituents of a cycle. The cycle mean of a cycle in the PPN modeling graph is computed using the sum of all initial tokens on the edges constituting the cycle, as we have shown in Equation (4.1). Hence, the amount of initial tokens on an HSDF edge representing a feedback channel may affect the MCM value of an HSDF graph. Therefore, we should determine the amount of initial tokens for each feedback edge such that an accurate MCM value is obtained.

Each cycle of a PPN contains one process that is the *first process* from that cycle to be fired. The channel of that cycle from which the first process reads is the *last*

channel of that cycle. Initially, we construct for each PPN channel part of a cycle an HSDF edge and assign zero initial tokens to each edge. Only to the edge corresponding to the last channel of the cycle we assign a nonzero number of initial tokens d. For example, suppose process a in Figure 4.4 first reads from a channel outside of the cycle and in the next firing reads from channel $(c \rightarrow a)$ that is part of the cycle $(a \rightarrow b \rightarrow c \rightarrow a)$. As such, process a is the first process of the cycle that can fire. Therefore, edge $(c \rightarrow a)$ is the last edge of the cycle. Selection between edges is not possible in the HSDF model which requires all incoming edges of a node to be read during every firing. Without assigning initial tokens to the last edge $(c \rightarrow a)$ of the cycle, the HSDF graph would be in a deadlock state, preventing meaningful analysis. To avoid this deadlock state, we assign initial tokens to the last edge.

Initial tokens on an edge of an HSDF graph are also referred to as the *delay* of an edge. Here, "delay" refers to the temporal distance between the nodes in terms of iterations of the graph. For example, if an edge $(a \rightarrow b)$ has 2 initial tokens, then the firing of node b at iteration i depends on the token produced by node a at iteration $i-2$. The PPN model does not have a notion of initial tokens, which means we need to relate the delay between two HSDF nodes that are part of a cycle to the distance between processes in the PPN model.

A notion of dependence distances is available for SANLPs from which we derive PPNs. A dependence distance vector gives the difference between a target iteration vector and the source iteration vector of a dependence [Pug92, definition *d*]. For a PPN channel $(a \rightarrow b)$, the distance vector gives the difference between an iteration of process b that consumes a token and the iteration of process a that produced the token. In general, this difference may not be defined when the process iteration domains are different, which for example happens when the original statements are not located in the same loop nest. However, the PNGEN tool flow puts all processes in a common iteration space to compute buffer sizes. In this common iteration space, the dependence distance vector is defined for any pair of processes that are connected by a channel. We therefore employ the dependence distance in the common iteration space to assign initial tokens to feedback edges in the PPN modeling graph.

We cannot use the dependence distance directly, because of the following two reasons. First, the dependence distance is a vector for common iteration spaces consisting of more than one dimension. In contrast, the number of initial tokens of an HSDF graph should always be a scalar value. Second, a dependence distance may be *non-uniform*, that is, the dependence distances may vary for different pairs of iterations. In such cases, the dependence distance of a single dimension cannot be expressed using a constant integer only, but is expressed using iterators. In contrast, the number of initial tokens of an HSDF graph should always be a constant integer value.

To overcome both problems, we use a constant integral scalar approximation of a

dependence distance. The way in which PNGEN computes the buffer size (cf. Section 2.3.2) gives us a suitable approximation of the maximum dependence distance of a non-uniform dependence. For uniform dependence distances, the buffer size is an accurate measure of the dependence distance. For non-uniform dependence distances, the use of the buffer size introduces a source of inaccuracy in the PPN modeling graph.

The number of initial tokens d_c assigned to the last edge of a cycle is determined as follows. If the cycle is tight, that is, if in every iteration each process depends on the output of the previous iteration of its predecessor process, the processes execute sequentially without overlap between firings of different processes. In such a case, the dependence distance vector contains zeroes for all dimensions except the last for which it contains a one. That is, the dependence distance vector is of the form $[0, 0, \ldots, 1]$. The corresponding buffer size S_c is one, and we assign one initial token to the last edge of the cycle.

If the cycle is not tight, then overlapped execution between firings of different processes may occur. In such a case the dependence vector is different from the form described above. We assign $S_c + 1$ initial tokens to the last edge of a cycle which corresponds to the buffer size plus one additional initial token to accomodate overlapped execution. Currently, this is a known source of inaccuracy in the MCM modeling HSDF graphs derived from PPNs. Determining the number of initial tokens to assign to the last edge of a non-tight cycle is therefore subject of future investigation.

Feedforward channels

Feedforward channels connect a strongly connected component of a PPN to another strongly connected component. As such, the corresponding feedforward edges in an HSDF graph are not part of any cycle and thus would not affect the MCM value. In the HSDF model, edges have infinite capacity which implies that a feedforward edge indeed does not affect the MCM value of an HSDF graph. That is, a feedforward edge cannot reach a "full" state that would cause blocking writes decreasing throughput. In contrast to HSDF edges, PPN channels have a finite capacity which may cause blocking write conditions that decrease throughput.

To take the finite capacity of a channel into account, we add for each feedforward channel $(a \to b)$ a forward edge $e = (a \to b)$ and a *backedge* $(b \to a)$ [SB00, Section 10.4]. We assign zero initial tokens to the corresponding feedforward edge in the HSDF graph. The number of initial tokens $m_e \in \mathbb{N}$ on the backedge represents a particular buffer capacity. Empirically, we found that a value m_e corresponds to a buffer capacity

$$S_c = m_e + d_c - 2, \tag{4.4}$$

where d_c is the dependence distance approximation used in the discussion above on feedback channels. That is, the MCM computed using m_e matches the PPN period achieved with a buffer capacity of S_c tokens.

Bounding feedforward channel delays

According to the HSDF model, any positive number of initial tokens m on a backedge is allowed. This leads to an infinite number of possible buffer configurations. However, when m is below a certain value, a corresponding PPN buffer size may not exist due to the operational semantics of a PPN process. This gives a lower bound on m. Also, when m exceeds a certain value, the MCM is not affected anymore, which means that increasing the buffer size does not lead to a higher throughput. This gives an upper bound on m. Therefore, we can bound the design space by only considering the values that lie between the lower and upper bounds.

The lower bound on m for any edge in the PPN modeling graph is two, which is a consequence of the operational semantics of a PPN process. This lower bound of two can be explained as follows. In an HSDF graph, a token is kept on the edge until the consuming node has finished its firing. In a PPN graph, a token is transfered to a buffer internal to the process during the read stage. This effectively increases the buffer size by one. As such, a buffer size of one corresponds to a number of initial tokens $m = 2$.

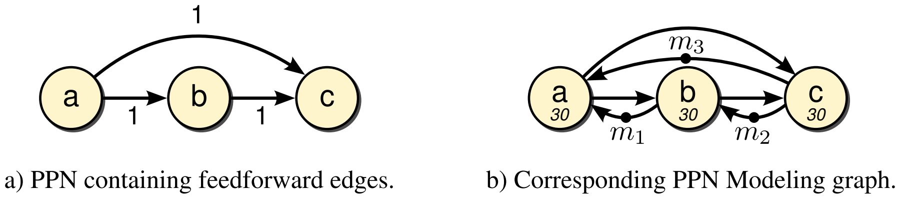
The upper bound on m represents the point where increasing the buffer size does not yield a higher throughput. This corresponds to a value m for which the maximum cycle mean of the graph is determined by cycles of the original graph or selfloops, but not by a cycle introduced by the modeling of a feedforward edge. For an arbitrary feedforward edge $(a \rightarrow b)$, we choose m such that the resulting cycle mean value is less than or equal to the cycle mean of the selfloops of the nodes involved in the cycle:

$$\frac{t(a) + t(b)}{m} \leq \max\{t(a), t(b)\}.$$

For positive execution times t, this inequation holds if m equals the number of nodes in the cycle, which is two. However, other paths between a and b may exist which must be considered as well to avoid that they determine the maximum cycle mean. To ensure that none of the other paths between a and b determine the maximum cycle mean, we generalize the above inequation to a path consisting of n nodes:

$$\frac{\sum_{i=1}^{n} t(i)}{n} \leq \max{}_{i=1}^{n}\{t(i)\}. \tag{4.5}$$

Thus, the upper bound on m originating from a channel c equals the number of pro-



a) PPN containing feedforward edges. b) Corresponding PPN Modeling graph.

Figure 4.5: Handling feedforward edges in a PPN.

m_1	m_2	m_3	$MCM(G_{4.5b})$
1	1	1	90
1	1	2	60
1	1	3	60
1	2	1	90
1	2	2	60
1	2	3	60

m_1	m_2	m_3	$MCM(G_{4.5b})$
2	1	1	90
2	1	2	60
2	1	3	60
2	2	1	90
2	**2**	**2**	45
2	**2**	**3**	30

Table 4.1: MCM values for different numbers of initial tokens. Only for the configurations in boldface a valid PPN buffer size configuration exists.

cesses on the longest path connecting σ_c to δ_c.

In Figure 4.5a, we show a PPN containing three feedforward channels. In Figure 4.5b, we show the corresponding modeling graph. For each feedforward channel in the PPN, we have added a forward edge and a backedge in the modeling graph. The values m_1, m_2, and m_3 specify the amount of initial tokens assigned to the backedges. According to equation (4.5), the upper bound of m_1 and m_2 is two and the upper bound of m_3 is three. In Table 4.1, we show twelve possible combinations of m-values, deliberately assuming a lower bound of 1 for each m-value. This yields twelve different design points trading off buffer size against throughput. If we take the lower bound on m-values for PPN modeling graphs into account, any combination of m-values containing an m-value below two does not have a corresponding PPN buffer configuration. Hence, only for $(m_1, m_2, m_3) = (2, 2, 2)$ and $(m_1, m_2, m_3) = (2, 2, 3)$ an actual PPN buffer configuration exists. As such, for this example the buffer size design space is reduced to only two points.

Summary

The derivation of the more compact HSDF graph is summarized in Algorithm 4.1. The input is a PPN and a number Λ_p representing the execution time of a single firing

of each process p. The output is an HSDF graph that is intended only for throughput analysis by an MCM algorithm. Lines 1–4 in Algorithm 4.1 perform the conversion of PPN processes to HSDF nodes. Lines 5 and onwards in Algorithm 4.1 perform the conversion of PPN channels to HSDF edges, considering the three classes of channels.

To leverage Proposition 4.2, we assume that the "append edge" operations at lines 12 to 16 of Algorithm 4.1 prune any edges $e' = (\sigma'(c) \rightarrow \delta'(c))$ already present for which $d(e')$ is larger than $d()$ of the new edge being added. Here, $\sigma'(c)$ and $\delta'(c)$ give the HSDF node that corresponds to the PPN process given by σ_c and δ_c (cf. Definition 2.13).

Because there is a one-to-one correspondence between PPN processes and HSDF nodes, no exponential increase in the number of nodes occurs. An exponential increase of the number of edges in the HSDF graph is also avoided:

Proposition 4.3 (Number of Edges in PPN Modeling Graphs)**.** *The number of edges in a PPN modeling graph for a PPN* $(\mathcal{P}, \mathcal{E})$ *is at most* $|\mathcal{P}| + 2 \cdot |\mathcal{E}|$.

Proof. For each process $p \in \mathcal{P}$, a selfloop is added, resulting in $|\mathcal{P}|$ selfloops in the HSDF graph. For each channel $c \in \mathcal{E}$, no edge is added if c is a selfloop; at most one edge is added if c is a feedback edge; and at most two edges are added if c is a feedforward edge. Thus, if all channels in a PPN are feedforward edges, then at most $2 \cdot |\mathcal{E}|$ edges are added. □

Hence, any exponential increase in the number of nodes or edges is avoided in our approach.

4.5.4 Case Studies

Acyclic Example from Literature

We first examine an acyclic PPN that was also studied by Meijer et al. [MNS10, Fig. 7]. We show the sequential code and corresponding PPN in Figure 4.6a and 4.6b. The PPN consists of four processes and three channels. We use latency values $\{\Lambda_{P1} = \Lambda_{P2} = 61, \Lambda_{P3} = 126, \Lambda_{C} = 121\}$ which correspond to the process workloads used by Meijer et al. The authors found a system throughput of $\frac{1}{126}$ using their throughput estimation method.

The PPN modeling graph derived using Algorithm 4.1 is shown in Figure 4.6c. Each process has an iteration domain consisting of 1000 points. Therefore, the execution time of each HSDF node is set to 1000 times the corresponding latency Λ. We add a selfloop with one initial token to each node. All of the three channels of the PPN are feedforward channels. Therefore, we add for each channel both the forward edge

Algorithm 4.1 Derive a modeling graph from a PPN.

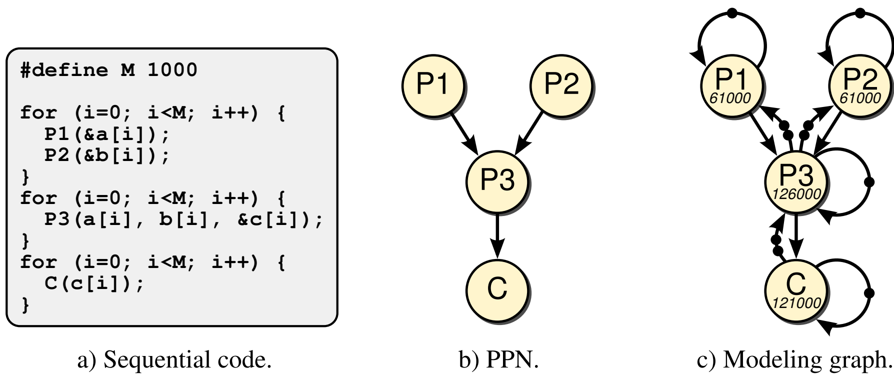
Input: PPN $G = (P, C)$, delays $\{\Lambda_p \mid p \in P\}$
Output: HSDF $H = (V, E, t, d)$

```
 1: for all processes p in P do
 2:     append node h to V with t(h) = |D_p| · Λ_p
 3:     append edge e = (p → p) to E with d(e) = 1
 4: end for
 5: for all channels c in C do
 6:     if c is not a selfloop then
 7:         if c is a feedback edge (i.e., part of an SCC) then
 8:             s = 0
 9:             if δ_c fires before σ_c then
10:                 s = S_c + {1 if a non-tight cycle containing c exists}
11:             end if
12:             append edge e = (σ'(c) → δ'(c)) to E with d(e) = s
13:         else if c is a feedforward edge then
14:             append edge e = (σ'(c) → δ'(c)) to E with d(e) = 0
15:             s = max{|p_i| + 1 | p_i is a path from σ_c to δ_c}
16:             append edge b = (δ'(c) → σ'(c)) to E with d(b) = s
17:         end if
18:     end if
19: end for
20: return H
```

and a backedge in the modeling graph. According to Equation (4.5), the number of initial tokens on each backedge equals two. The maximum cycle mean computation of the resulting modeling graph yields the following:

$$
\begin{aligned}
MCM(G_{4.6c}) = \max \Bigg\{ & \frac{t(P1)}{1}, \frac{t(P2)}{1}, \frac{t(P3)}{1}, \frac{t(C)}{1}, \frac{t(P1) + t(P3)}{d(P1 \to P3) + d(P3 \to P1)}, \\
& \frac{t(P2) + t(P3)}{d(P2 \to P3) + d(P3 \to P2)}, \frac{t(P3) + t(C)}{d(P3 \to C) + d(C \to P3)}, \Bigg\} \\
= \max & \left\{ \frac{61000}{1}, \frac{61000}{1}, \frac{126000}{1}, \frac{121000}{1}, \frac{187000}{2}, \frac{187000}{2}, \frac{247000}{2} \right\} \\
= 126000 & \; .
\end{aligned}
$$

Thus, a single iteration of the graph takes 126000 time units. In an iteration of the graph, sink process *C* fires 1000 times. Therefore, the period $T_C = \frac{126000}{1000} = 126$,



a) Sequential code. b) PPN. c) Modeling graph.

Figure 4.6: Throughput analysis on example from [MNS10].

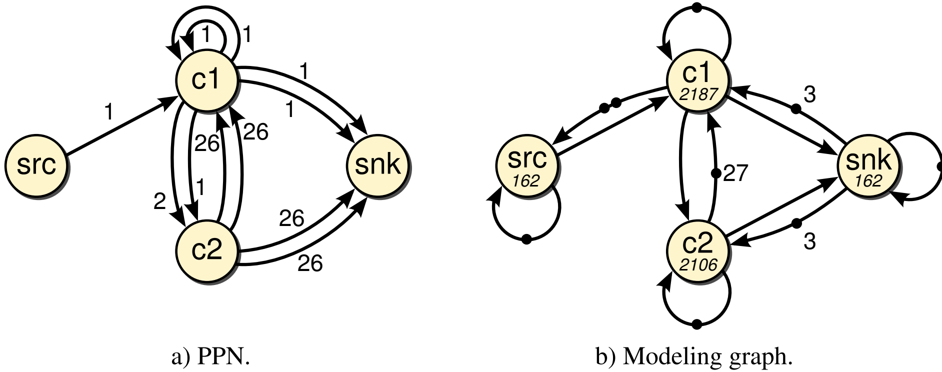
and the PPN's throughput equals $\frac{1}{126}$. This throughput value exactly matches the value found by Meijer et al.

Odd-even Transposition Sorting

With this example we illustrate the MCM analysis applied to a cyclic PPN. The odd-even transposition sorting is a parallel sorting algorithm which sorts an array of n elements. The algorithm consists of n comparator stages. In each odd-numbered stage, all even-indexed elements are compared with their odd-indexed neighbours and swapped if they are not in the correct order. In each even-numbered stage, all odd-indexed elements are compared with their even-indexed neighbours and swapped if necessary. In each stage, all $n/2$ pairs can be compared in parallel.

In Figure 4.7a, we show a PPN for the odd-even transposition sorting algorithm. The PPN consists of four processes. Source process *src* provides the data to be sorted. Processes *c1* and *c2* perform the compare-and-swap operations. Sink process *snk* consumes the sorted data.

In Figure 4.7b, we show the throughput modeling graph derived from the PPN. All four channels between *c1* and *c2* in Figure 4.7a are part of a strongly connected component. From dependence analysis we find that *c2* cannot fire before *c1* has fired. Thus, we put zero initial tokens on the forward edge connecting *c1* to *c2*. The dependence distance vectors for both edges are non-uniform. The upper bound on the dependence distance is 26. According to line 10 of Algorithm 4.1, we put 27 initial tokens on the feedback edge from *c2* to *c1*. All of the remaining edges are feedforward edges to which we assign two or three initial tokens according to Equation (4.5).



a) PPN.

b) Modeling graph.

Figure 4.7: Throughput analysis of odd-even transposition sorting.

For brevity reasons we omit the full expansion of Equation (4.2) and only summarize the result. The maximum cycle mean is 2187 which originates from the selfloop of process *c1*. Since the *sink* process domain size is 54, the average period of the PPN is $\frac{2187}{54} = 40.5$ time units. This corresponds to the average period observed during simulation of the RTL.

4.6 Sequential Code Profiling

At the first level in Figure 4.1, we have the application specified as sequential C code. The previous performance estimation techniques discussed in this chapter required that a PPN was derived. We now investigate whether we can estimate the performance of a PPN directly from the sequential C code. This has led to a novel profiling-based method that works directly at the sequential source code level. Our novel method was inspired by the work of Kumar on measurement of parallelism in Fortran programs [Kum88].

In Section 4.6.1, we review some existing profiling techniques. In Section 4.6.2, we present the profiling primitives employed by our approach. In Sections 4.6.3 and 4.6.4, we present the two estimation types provided by our approach. In Section 4.6.5, we apply our profiling approach on a case study. In Section 4.6.6, we present how our profiling approach can model process splitting transformations without the need to actually apply the spltting transformations to the application code. In Section 4.6.7, we discuss the performance and memory overhead resulting from the profiling.

4.6.1 Related Work

Profiling is a well-established technique in which the behavior of a program is analyzed by closely monitoring the execution of the program. This monitoring is performed by extending a program with small *instrumentation* code fragments that collect statistics such as function invocation counts during program execution.

A popular free software tool to profile for example C and Fortran programs is *GNU gprof* [GKM82]. To profile a program, the compiler instruments the program with instrumentation code. The instrumentation code collects statistics when the instrumented program is running. After running the instrumented program, gprof processes the statistics into a call graph augmented with the execution time of each call. This allows the developer to determine in which parts of a program most of the execution time is spent. To obtain execution time information, gprof relies on statistical program counter sampling which is an inexact method. The execution times are only valid for the platform on which the profiling is performed, which makes gprof not useful for throughput assessments of programs implemented as PPNs on different, possibly heterogeneous platforms.

The *Valgrind* tool set provides tools for debugging and profiling program binaries [NS07]. Each Valgrind tool translates the individual machine instructions into an intermediate representation, instruments the intermediate representation, and translates the intermediate representation back into machine instructions. This allows for more accurate execution time estimates compared with statistical sampling methods, because each instruction is considered. However, as with gprof, the obtained execution times are only valid for the processor architecture for which the program was compiled. In our design flow, different parts of a program may execute on different processor architectures, such as ARM or MicroBlaze, or may be implemented as a LAURA processor. Such alternative heterogeneous architectures are currently not supported by Valgrind and we believe that adapting Valgrind to support such architectures would require a significant amount of effort.

Support for different processor architectures was a key design goal for the *TotalProf* profiler [GHC+09]. TotalProf processes the intermediate representation of an input program into virtual assembly that is captured in the LLVM intermediate representation. Different architectures are represented using different forms of the virtual assembly. The virtual assembly is instrumented with profiling statements and then taken through the code generator which generates executable code for the host machine. By targeting the host machine, fast execution of the instrumented virtual assembly code is obtained. By providing different architecture descriptions and interconnecting different TotalProf instances, TotalProf supports profiling of heterogeneous MPSoCs. TotalProf currently lacks support for the LAURA architecture, which

prevents us from using TotalProf for performance assessment of PPNs.

Another approach to achieve a high simulation speed when simulating MPSoCs, is by compiling the application code of a system for native execution on a general-purpose computer such as a desktop workstation [SHP12]. Such native simulation techniques eliminate the need for instruction set simulation or binary translation and thus avoid a significant amount of run-time overhead. Native simulation is often intended for debugging and verification of an MPSoC design. In contrast, our interest is mainly in performance assessment to a sufficient level of accuracy to make architectural tradeoffs.

Sackmann et al. presented a profiling-based method to parallelize sequential programs [SEJ11]. To discover the parts that may execute in parallel, the authors analyze call trees obtained using Valgrind. Each node in the call tree represents an invocation of a function. Two nodes are connected by an edge if a data dependence exists between the corresponding function invocations. If a pair of nodes is not connected by an edge, then the corresponding function invocations can execute in parallel. Unfortunately, the authors' approach does not guarantee that all data dependences are added to the graph. They rely on the user to ensure that all dependences are present in the call tree. Ensuring correctness of the call tree manually is tedious and error-prone, because call trees can be large even for small programs, and because a call tree is only valid for one execution of the program. Instead of partially relying on manual effort, we favor discovering the amount of parallelism in a fully automated way.

Kumar presented COMET (COncurrency MEasurement Tool) which measures the total parallelism in Fortran programs [Kum88]. The method assumes a hypothetical ideal parallel machine with unlimited resources and no scheduling, communication, and synchronization overhead. COMET takes a Fortran program and extends it with statements that monitor the execution of the program on the ideal parallel machine. The functionality of the original program is preserved in the extended program. By compiling and executing the extended program, statistics on the absolute amount of parallelism are collected.

At this level two approaches need to be mentioned: the SESAME [PEP06] and SPADE [LSWD01] approaches. They both work at the high level but do not use profiling. Instead they use traces to capture the workload of an application in terms of read, execute, and write events. Both SESAME and SPADE require deriving a PPN, which is what we want to avoid at the first level of Figure 4.1.

4.6.2 Sequential Code Instrumentation of Static Programs

In this section, we present *cprof* which is a novel method for PPN performance estimation that is inspired on COMET. Cprof can measure parallelism in an application

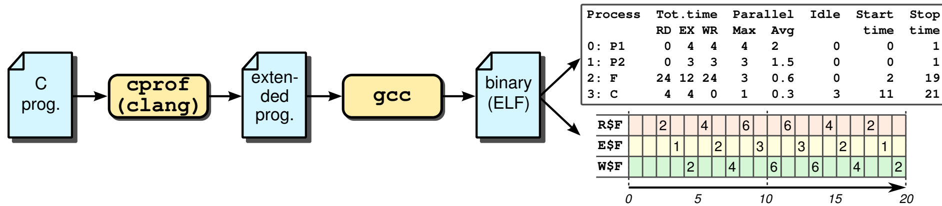


Figure 4.8: The cprof performance estimation technique.

without actually deriving a PPN, but assuming execution as a PPN on one of two machines. First, cprof can evaluate a PPN on an *ideal machine*. We consider an ideal machine to be a platform where each process iteration is mapped on a separate processor unit. The ideal machine lacks shared memory, as separate processor units communicate through separate point-to-point communication channels. Second, cprof can evaluate a PPN on a platform generated by ESPAM.

The ideal machine is used to measure the maximum degree of parallelism in an application, as we discuss in Section 4.6.3. The ESPAM platform is used to obtain an absolute throughput estimate for a PPN, as we discuss in Section 4.6.4.

Like COMET, cprof takes the original sequential program and produces an extended program containing additional profiling instrumentation statements. The cprof flow is depicted in Figure 4.8. We use the Clang/LLVM compiler infrastructure [LA04, Cla07] to automatically generate the extended program from a C program. The extended program is again a sequential program that should be compiled by a conventional C++ compiler and executed to obtain performance statistics. In contrast to COMET, which operates on programs written in the Fortran language, cprof operates on static affine nested loop programs (SANLPs) written in the C language. Another difference between COMET and cprof is the set of statements being instrumented. We only instrument the statements for which PNGEN constructs a process, because our goal is to obtain a performance estimate for a PPN execution. That is, cprof does not instrument control expressions and statements inside for-loop headers and if-conditions, since these are not translated into processes. In contrast, COMET instruments all statements, including loop-statements and if-statements. Instead of targeting only an ideal machine, cprof also targets a machine with a fixed amount of processing resources. This enables performance assessment of programs executing as a PPN on a user-defined platform.

Both COMET and cprof employ a global time scale. In this global time scale, COMET and cprof keep track of the timestamps at which the statements of a sequential program start and finish their executions. We instrument each statement such that the operational semantics of all processes are represented on a single global time

scale. The added instrumentation models the read, execute, and write stages of a PPN process that were described in Section 2.2.4. In the read stage, cprof determines at which time the actual statement can execute, based on the times at which all input data is available. In the execute stage, the statement execution finish time is determined. In the write stage, the times at which the output data of the statement is available are updated.

The main challenge is to determine the starting times of statement executions. We determine such starting times by taking into account the four aspects that were introduced in Section 4.2. To address the conditional synchronization, mutual exclusion, and basic calibration aspects, we employ instrumentation primitives. Since we only consider SANLPs in which control is static by definition and functions have a fixed latency, we do not need to address the conditional control flow aspect. The instrumentation primitives comprise shadow variables, control variables, and execution profiles. In the following paragraphs, we discuss each instrumentation primitive and explain how the first three performance aspects are addressed.

Shadow Variables

For each variable `v` in the original program, we add a *shadow variable* `$v` that holds the timestamp in which variable `v` is written. Similarly, for each array in the original program, we add an array of shadow variables which contains for each array element the timestamp at which a particular array element is written. A shadow variable is updated whenever the corresponding variable is written during the write stage. The new timestamp of a shadow variable `$v` is set to the timestamp at which the statement writing to `v` finishes writing `v`, plus an additional cost Λ_W modeling the write operation latency.

We use shadow variables to address the conditional synchronization aspect. Conditional synchronization in a PPN execution occurs when a process reads from one or more channels. As dictated by the operational semantics, a process function can only fire after all incoming channels have been read. That is, the firing of a process function may be postponed because of a blocking read operation on one of the incoming channels. In contrast to the other performance assessment techniques discussed in this chapter, the cprof technique does not explicitly model a program as a PPN consisting of processes and channels. Despite this, cprof is able to obtain accurate performance assessments that incorporate blocking read operations. To understand why we can analyze the performance of a PPN without actually deriving a PPN, we now explain how blocking read operations are taken into account, thereby addressing the conditional synchronization aspect.

The SANLP class of C programs that we consider are written assuming a sequential

execution model in which statements execute one after another. Only one statement executes at any point in time. The execution order is given by the textual order of statements in the program and control flow statements such as for-loops. For a given program, many alternative execution orders may exist which all yield the same functional meaning of the program as long as all *data dependence* relations are respected [Ban97]. A program translated to a PPN employs such an alternative execution order, in which processes execute concurrently. Each process fires as soon as its input data is available. Thus, the performance of a PPN is determined by the availability times of data. The availability of data is captured by the data dependence relations of a sequential program. Three different types of data dependence relations may exist in a sequential program: flow dependences, anti-dependences, and output dependences [PW86]. We now explain how cprof handles each dependence type using shadow variables, such that an accurate performance assessment of a PPN derived from the sequential program is obtained.

Flow dependences, or read-after-write dependences, occur when a statement reads a variable `v` written by a previous statement. During the write operation, the timestamp at which the write occurs is stored in the shadow variable of `v`. Upon reading `v`, the timestamp in the shadow variable of `v` is taken as the time at which `v` is available. By definition, the read operation occurs after the write operation, meaning the flow dependence is correctly modeled.

Anti-dependences, or write-after-read dependences, occur when a statement writes data to a variable `v` that was read by a previous statement. Upon reading `v`, the timestamp in the shadow variable of `v` is taken as the time at which `v` is available. This shadow variable is then overwritten by the write operation, which by definition occurs after the read operation. However, in both COMET and cprof, the instrumentation for the write operation does not take into account the time at which `v` was actually read. Thus, the anti-dependence is not modeled.

Output dependences, or write-after-write dependences, occur when two statements write data to the same variable `v`. The timestamp of the first write operation is stored in the shadow variable of `v`. The timestamp of the second write operation overwrites the previous timestamp in the shadow variable, without taking into account the timestamp of the first write operation. Only the timestamp of the last write is kept, meaning the output dependence is not modeled.

In summary, COMET and cprof only model flow dependences, and ignore anti-dependences and output dependences.[1] This means that cprof cannot model the performance of a SANLP on a general purpose processor accurately if the program con-

[1] An extension to incorporate anti-dependences and output dependences in COMET was described by Kumar as well [Kum88, Section VI], but this extension has not been incorporated in cprof as anti-dependences and output dependences are not relevant in the PPN context [Tur07, Chapter 3].

tains anti-dependences or output dependences. However, the purpose of cprof is to model performance of a SANLP assuming execution as a PPN. A key property of the PPN MoC is that only flow dependences affect its performance. Anti-dependences and output dependences in a sequential program do not affect the performance of a PPN, such that these can be safely ignored by cprof, as we now explain.

Anti-dependences in a sequential program do not affect the performance of a PPN derived from the sequential program, because each produced data token has a private storage location that is preserved until the token is consumed. Multiple data tokens representing different values for the same variable of the sequential program may exist simultaneously. A property of the PPN model is that the storage location of a data token is never written again after consumption [Tur07, Chapter 3]. This means anti-dependences do not occur in a PPN and thus do not affect performance.

Output dependences in a sequential program do not affect the performance of a PPN derived from the sequential program, because a data token is only produced on a channel if the data token is guaranteed to be consumed. This means output dependences do not occur in a PPN, and thus output dependences do not affect the performance.

We have explained that flow dependences are correctly modeled by shadow variables in cprof. We use shadow variables to address the conditional synchronization aspect. The conditional synchronization aspect follows from the operational semantics of a PPN process. These operational semantics dictate that a process is allowed to fire only when all input data is present. In cprof, this means that the maximum value of the shadow variables of the inputs of the statement represents the correct time of firing.

Once a statement starts executing in cprof, no delays or stalls occur until the statement has finished writing its output variables. This implies that cprof uses non-blocking write semantics, assuming that channel sizes are always sufficiently large to store the produced data. Finite channel sizes affecting throughput via a blocking write mechanism are currently not supported by cprof. To consider finite channel sizes, we should explicitly model each channel resource. This requires derivation of a PPN from the sequential program, because the channels are not explicitly present in the sequential program. Modeling finite channels sizes is therefore a subject of future research.

Control Variables

For each statement `s` for which PNGEN constructs a process, we add a *control variable* `C$s` that holds the earliest time at which statement `s` can execute. This earliest time is determined by the availability of the input data to statement `s`. The statement

executes only when all input data is available, resembling the operational semantics of a PPN process. As such, we ensure that data dependence relations are not violated, and thus address the conditional synchronization aspect. Control variables also address the mutual exclusion aspect, which will be detailed further in Sections 4.6.3 and 4.6.4.

A control variable is updated during the read stage by considering the shadow variables of the variables read by statement `s`. A statement `s` can execute after all variables read by statement `s` have been written. Thus, a control variable `C$s` is set to the maximum value of all shadow variables that are read by statement `s`. Taking the maximum value of all shadow variables effectively delays the statement to the timestamp at which all data is available, resembling a blocking read operation in terms of the PPN operational semantics. The statement only executes when all data is available, which is in accordance with the PPN semantics.

Statement Execution Profile

For each statement `s`, we also add three one-dimensional arrays `R$s`, `E$s`, and `W$s` which together constitute the *statement execution profile* of `s`. In the statement execution profile, we collect the read, execute, and write behavior of the statement over time. The statement execution profile collect at a high level the operational behavior of a process. For example, `W$s[23] = 2` means that two write operations are in progress at time 23. All array elements are initialized to zero. Array `R$s` is updated after reading a statement input. Array `E$s` is updated after executing the statement. Array `W$s` is updated after writing a statement output. An update to any of the three arrays involves incrementing the array elements in an interval $[t_s, t_f)$ by one. Here, t_s is the starting time and $t_f = t_s + \Lambda$ is the finish time of an operation with latency Λ.

After executing the instrumented program, we can extract the following information for each process from the statement execution profiles:

- The total time spent on read operations, statement executions, and write operations is obtained by summing all elements in the corresponding `R$s`, `E$s`, and `W$s` arrays.
- The process start time $s(p)$, which is the first time at which a process can fire, equals the index of first non-zero element in `R$s`. For statements that do not consume any input data, this equals zero.
- The process finish time $f(p)$, which is the time at which the process has finished all iterations, equals the index of the last element in `W$s`. For statements that do not produce any output data, we instead take the index of the last element in `E$s`.

- The number of idle cycles, which is the number of time units in the interval $[s(p), f(p))$ in which each of the Rs, Es, and W$s arrays contains a zero.
- The maximum number of statement executions that are in progress simultaneously is obtained by finding the maximum value in E$s.
- The number of process iterations that are in progress simultaneously at a given time t is given by the *flat execution profile*, which we define as

$$\texttt{R\$s[}t\texttt{]} + \texttt{E\$s[}t\texttt{]} + \texttt{W\$s[}t\texttt{]}. \tag{4.6}$$

The average process period can be computed from the process start and finish times as

$$T_p = \frac{f(p) - s(p)}{D_p}. \tag{4.7}$$

We then compute the throughput of a process by taking the reciprocal of T_p. The throughput of the entire PPN is given by the throughput of the sink process, according to Definition 4.2.

Global Execution Profile

From all statement execution profiles, we compute the *global execution profile* G$ as follows:

$$\texttt{G\$[k]} = \sum_{i=0}^{|P|-1} \texttt{R\$}i\texttt{[k]} + \texttt{E\$}i\texttt{[k]} + \texttt{W\$}i\texttt{[k]}, \tag{4.8}$$
$$0 \leq \texttt{k} < \max\{\forall p \in P \mid f(p)\}.$$

That is, we sum for each timestamp k the statement execution profiles of all processes. The global execution profile resembles the *PROFILE* array in COMET. However, the global execution profile includes read and write operations, which are not included in COMET. The global execution profile provides information about the behavior of the application as a whole. We can extract the following information from the global execution profile:

- The *PPN execution time*, which equals the number of elements in G$.
- The *maximum degree of parallelism*, which is the maximum number of simultaneously active processes at any time, equals the maximum element in G$.
- The *average degree of parallelism* is obtained by dividing the sum of all elements in G$ by the number of elements in G$.

Execution Times

To address the basic calibration aspect, we rely on user-provided performance data for the process functions. In particular, cprof uses the Λ and II values from Definition 3.1 to characterize the latency and initiation interval of each process function. This allows cprof to handle both pipelined (e.g., LAURA) processing resources and non-pipelined (e.g., programmable) processing resources. In addition, cprof assumes fixed latencies Λ_R and Λ_W for read and write operations that resemble communication over PPN channels. The particular use of the Λ and II values is discussed in Section 4.6.5.

Currently, cprof assumes that Λ and II are constant for all invocations of a process function. This is the main source of inaccuracy, because this is not always the case. For example, a division function may have a multi-cycle latency in general, but may have a one-cycle latency if the divider equals one. Including such latency characteristics should be possible in cprof, because cprof allows a fully functional execution of the original program. However, our main interest lies in a simple and fast performance estimation approach, and thus detailed dynamic performance models are currently beyond the scope of cprof.

4.6.3 Maximum Degree of Parallelism

To gain insight in the amount of parallelism in a given application specification, we instrument an input program in a way that models execution on a hypothetical *ideal machine*. This gives the *maximum degree of parallelism*, which represents an upper bound on the number of processing resources required to execute the PPN without processing resource contention. Adding more processing resources does not result in a further speedup.

The selection between an ideal machine with an infinite number of processor resources and a real machine with a finite number of processor resources depends on whether or not the mutual exclusion aspect is addressed. Mutual exclusion can be enforced when a control variable `C$s` is updated during the read stage by taking into account the time at which a processor is available. On both the ideal and real machine, the conditional synchronization aspect needs to be addressed, because a statement cannot start until all input data is available. On the ideal machine, the availability of the input data is the only condition for the statement to execute, which means we do not take the mutual exclusion aspect into account. Thus, each control variable `C$s` is set to the maximum value of all shadow variables representing inputs to `s`, which implies `s` executes as soon as all its input data is available. After execution of the instrumented program on the ideal machine, the information from the global execution profile `G$` can be used to draw conclusions about the application performance.

The consequence of using the ideal machine is that the PPN execution time equals the minimum time needed for the PPN execution. By definition, the ideal machine has sufficient processing resources to avoid processing resource contention. A PPN execution time of 1 time unit means that the PPN can fire all iterations of all processes simultaneously. A PPN execution time larger than 1 time unit means that parts of the PPN are inherently sequential.

Related to the maximum degree of parallelism is the *average degree of parallelism*. This represents an upper bound on the speedup that can be obtained with an unbounded number of processors [EZL89] compared to execution of a PPN on a single processor.

4.6.4 Absolute Throughput Estimation

By instrumenting an input program such that the mutual exclusion aspect is taken into account, we can obtain an absolute throughput estimate of the execution of a PPN derived from the input program. This is possible because each statement of a sequential program corresponds to a process in a PPN and because the operational semantics of a process are well-defined (cf. Section 2.2.4). On a realistic execution platform with a finite number of processing resources, an additional condition for the availability of a suitable processing resource needs to be considered before a statement can be executed. This mutual exclusion aspect was not addressed by COMET, because COMET assumes only the ideal machine. If we assume that all executions of a statement are mapped onto the same processing resource, then control variable `C$s` should at least equal the time `C$s` $+ II$ at which the processing resource can initiate a new execution. Instead of taking the maximum only over all shadow variables as described in Section 4.6.3, we now take the maximum over all shadow variables and `C$s` $+ II$.

4.6.5 Case Study

In this case study section, we first apply the absolute throughput estimation (cf. Section 4.6.4) to the program shown in Figure 4.9. Next, we show how to determine the maximum degree of parallelism (cf. Section 4.6.3) of this program.

Case Study: Absolute Throughput

In Figure 4.10, we show the cprof instrumentation for the program code shown in Figure 4.9. We assume that all executions of statement `func1` are mapped onto a single processor. We furthermore assume that a read operation takes Λ_R time units;

```
1  for (i=0; i<9; i++) {
2    source(&x[i], &b[i]);          // Statement 0
3    func1(x[i], &a[i]);            // Statement 1
4    sink(a[i], b[i]);              // Statement 2
5  }
```

Figure 4.9: Example program code on which we illustrate cprof.

that a write operation takes Λ_W time units; that the latency of `func1` is given by $\Lambda_{\texttt{func1}}$; and that the initiation interval of `func1` is given by $II_{\texttt{func1}}$.

At line 17 of Figure 4.10, we determine the starting time of statement 1 using the expression `max(C$1, $x[i])` which takes the maximum from control variable `C$1` and shadow variable `$x[i]`. The control variable in the `max`-expression ensures that the statement executes after the previous execution of the statement has finished, which means the processor resource is available. The shadow variable in the `max`-expression ensures that the statement executes when the variable `$x[i]` has been written. By adding Λ_R, we delay execution of the statement to incorporate a read operation delay of Λ_R time units. At line 18, we add the read operation to the read execution profile. At line 21, the original statement is executed. For the applications that we consider, actual execution of the statements is not required to obtain throughput assessments. By omitting the actual execution of statements, the throughput assessment can be performed in less time. At line 22, we set the finish time of the statement and at line 23 we add the actual execution of the statement to the execution profile. At line 24, we update control variable `C$1` such that the next execution of the statement starts at least after a full initiation interval. At line 27, we set the time at which `a[i]` is written. At line 28, we add the write operation of `a[i]` to the write execution profile.

At line 31, we show the instrumentation for statement 2 which takes two inputs `a[i]` and `b[i]`. Both `a[i]` and `b[i]` have to be available before statement 2 can execute, so we consider the shadow variables of both input variables to determine the starting time of statement 2. By including both in the `max`-expression, we take the conditional synchronization aspect into account.

After executing the instrumented code, the execution profiles are obtained. In Figure 4.11, we show the statement execution profiles for all three statements, assuming $\Lambda_R = \Lambda_W = 1$, $\Lambda_{\texttt{func1}} = 2$, $\Lambda_{\texttt{source}} = \Lambda_{\texttt{sink}} = 1$ and $II_{\texttt{source}} = II_{\texttt{func1}} = II_{\texttt{sink}} = 1$. The first read operation of `source` starts at time 0, because the statement does not depend on any input data. The first read operation of `func1` starts at time 2, which means the process `func1`'s start time equals 2. The last write operation of `func1` finishes at time 38, which means process `func1`'s finish time equals 38. Using Equation (4.7) we find that the average period $T_{\texttt{func1}} = 4$, which can be verified

```
1  for (i=0; i<9; i++) {
2    // Read stage (void, no input arguments to statement 0)
3
4    // Execution stage
5    source(&x[i], &b[i]);           // Original statement 0
6    done = C$0 + Λsource
7    for (t=C$0; t<=done; t++) E$0[t]++;
8    C$0 += IIsource;
9
10   // Write stage
11   $x[i] = done + ΛW;
12   for (t=done; t<=done+ΛW; t++) W$0[t]++;
13   $b[i] = $x[i] + ΛW;
14   for (t=done; t<=done+ΛW; t++) W$0[t]++;
15   ------------------------------------------------------------
16   // Read stage
17   C$1 = max(C$1, $x[i]) + ΛR;
18   for (t=C$1-ΛR; t<=C$1; t++) R$1[t]++;
19
20   // Execution stage
21   func1(x[i], &a[i]);             // Original statement 1
22   done = C$1 + Λfunc1
23   for (t=C$1; t<=done; t++) E$1[t]++;
24   C$1 += IIfunc1;
25
26   // Write stage
27   $a[i] = done + ΛW;
28   for (t=done; t<=done+ΛW; t++) W$1[t]++;
29   ------------------------------------------------------------
30   // Read stage
31   C$2 = max(C$2, $a[i], $b[i]) + 2*ΛR;
32   for (t=C$2-2*ΛR; t<=C$2; t++) R$2[t]++;
33
34   // Execution stage
35   sink(a[i], b[i]);               // Original statement 2
36   done = C$2 + Λsink
37   for (t=C$2; t<=done; t++) E$2[t]++;
38   C$2 += IIsink;
39
40   // Write stage (void, no output arguments to statement 2)
41 }
```

Figure 4.10: Instrumentation by cprof for the program shown in Figure 4.9.

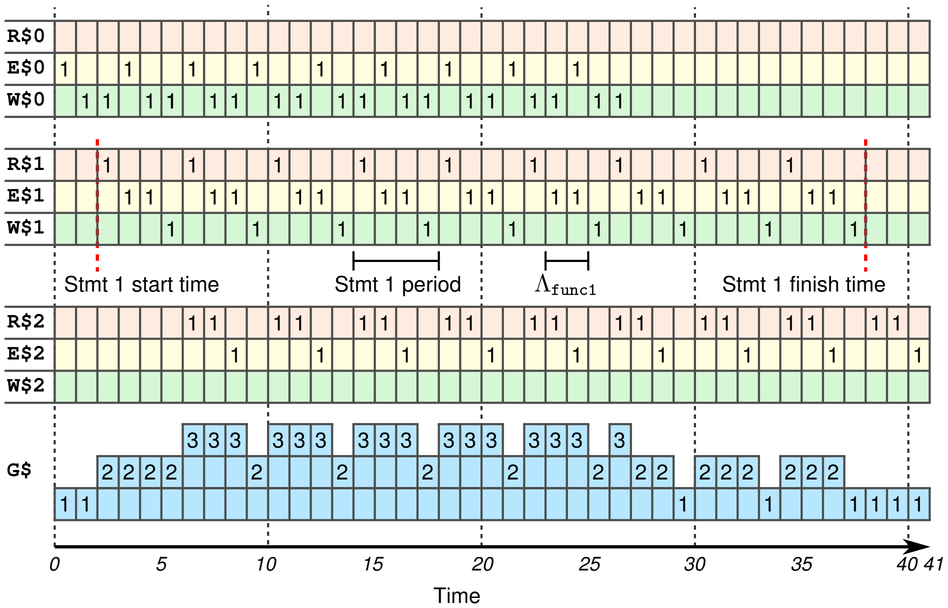


Figure 4.11: Execution profiles obtained by executing the instrumented code of Figure 4.10. Empty cells in the `R$`, `E$`, and `W$` profiles represent '0'.

visually in Figure 4.11.

The first read operation of `sink` starts at time 6, because the first execution of `sink` depends on variables `a[0]` and `b[0]`. Variable `a[0]` is written by `func1` at time 5, and variable `b[0]` is written by `source` at time 2. The first execution of `sink` can thus only start after time 5 at which both `a[0]` and `b[0]` are available. The number of idle cycles for the `sink` statement is eight, since there are eight time units in the interval $[6, 41)$ in which $\mathtt{R\$2} = \mathtt{E\$2} = \mathtt{W\$2} = 0$. This means the `sink` statement does not receive data at a fast enough rate. The number of idle cycles for the other two statements is zero, which means they fully utilize their processing resources.

In the bottom part of Figure 4.11 we show the global execution profile `G$` that is obtained using Equation (4.8). From the global execution profile, we can observe that that a full execution of the PPN takes 41 time units. Furthermore, we can observe that at most three operations are active simultaneously. Thus, the maximum degree of parallelism in this execution equals three. Summing all elements in `G$` gives a total amount of work equal to 90 units. The average degree of parallelism in this execution is $\frac{90}{41} \approx 2.1$. This means that on average, approximately two processes are active.

Case Study: Maximum Degree of Parallelism

To find an upper bound on the throughput of the application, we are interested in the amount of parallelism inherent in the application. To reveal the amount of parallelism in the entire application we instrument the code as described in Section 4.6.3. This requires only a small change to the instrumentation code that updates the control variables. For example, the newly instrumented code for statement 1 only differs in one place from the code shown in Figure 4.10. At line 17, we now assign `$x[i] +` Λ_R to the control variable. That is, we ignore the previous value of `C$1` such that the statement is executed as soon as the input data is available. As a result, each statement execution is performed on its own processing resource, which mimics execution on an ideal machine.

For the input program of Figure 4.9, execution on an ideal machine results in the execution profiles shown in Figure 4.12. All nine iterations of each statement execute in parallel. For example, the process derived from statement 0 executes nine instances of its function at time 0. The two output arguments of each of the nine statement executions are available at time 1, since $\Lambda_{\mathtt{source}} = 1$. This results in eighteen write operations at time 1.

From the global execution profile shown in the bottom part of Figure 4.12, we can observe that execution on the ideal machine takes eight time units. This is a lower bound on the execution time of a PPN derived from the input program under the given latency values. Furthermore, we can observe that at most eighteen operations execute

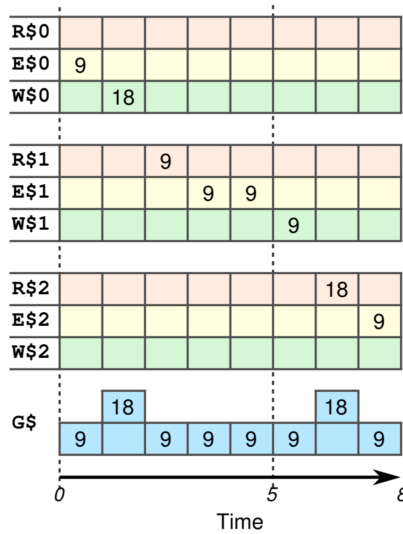


Figure 4.12: Execution profiles obtained by profiling the code of Figure 4.9 on an ideal machine. Empty cells in the `R$`, `E$`, and `W$` profiles represent '0'.

in parallel. Thus, to attain the minimum execution time of eight time units, a system with eighteen processors is required. The average degree of parallelism equals $\frac{90}{8} = 11.25$. This means that on average, 11.25 operations are in progress. Typically, the average degree of parallelism provides a design point which delivers a performance close to the maximum achievable performance, at a substantially reduced number of processing resources [EZL89]. With 11 or 12 processors, the minimum execution time becomes 10 time units, which is 25% above the minimum execution time. The reduction in the number of processing resources is 33–39%.

4.6.6 Transformation Performance Estimation

In the previous sections, we have distinguished two modes of operation of cprof. In Section 4.6.3, we presented how to determine the maximum degree of parallelism, where each iteration of a process is executed on a separate processing resource. In Section 4.6.4, we presented the absolute throughput estimation mode, where all iterations of a process are executed on the same processing resource. These two modes represent the two extremal design points of the possible assignments of iterations to processing resources. Many alternative design points exist between both extremes, which can be obtained by varying the assignment of iterations to processing resources. These two extremal design points are essential information for the

```
1  for (i=0; i<9; i++) {
2    pr$ = i % N;                          // For modulo unfold
3    pr$ = i / ((9-0)/N);                  // For plane cut
4
5    // Read stage
6    C$1[pr$] = max(C$1[pr$], $x[i]) + Λ_R;
7    for (t=C$1[pr$]-Λ_R; t<=C$1[pr$]; t++) R$1[t]++;
8
9    // Execution stage
10   func1(x[i], &a[i]);                   // The original statement
11   done = C$1[pr$] + Λ_func1
12   for (t=C$1[pr$]; t<=done; t++) E$1[t]++;
13   C$1[pr$] += II_func1;
14
15   // Write stage
16   $a[i] = done + Λ_W;
17   for (t=done; t<=done+Λ_W; t++) W$1[t]++;
18 }
```

Figure 4.13: Instrumented code for statement 1 of Figure 4.9 to analyze splitting transformations.

designer. At this stage, the designer knows whether he can satisfy a performance constraint at all from the maximum degree of parallelism. However, this extremal design point has a very high implementation cost, as it assumes execution on an ideal machine. A realistic design point is provided by the absolute throughput estimate. Using splitting transformations (cf. Section 5.1.1, the designer can evaluate intermediate design points with higher performance, eventually satisfying his constraints. Before starting this exploration, a designer already knows if his performance constraint can be satisfied.

A convenient way to obtain the alternative design points is through process splitting transformations that resemble loop unfolding transformations [Muc97, SKD02]. Such splitting transformations are covered in detail in Chapter 5.

In this section, we present how the performance of transformed PPNs can be analyzed using cprof, without the need to actually apply the splitting transformation on the program code. This allows a designer to quickly evaluate different design points, and then select the design point that best matches the design requirements. Then, the designer has to apply only those splitting transformations that result in the selected design point to obtain the desired implementation. To model a splitting transformation with factor N, we generalize the mutual exclusion aspect to N processors.

In Figure 4.13, we show the instrumented code for statement 1 of Figure 4.9. This

code differs from the original instrumented code shown in Figure 4.10, to enable performance estimation of splitting transformations. The differences with the original instrumentation are underlined in Figure 4.13. The main difference with the original instrumentation is that statement 1's control variable `C$1` is changed into an array of N elements, where N is the splitting factor. The control variable array is indexed using a processing resource selection variable `pr$`. At the start of each iteration, this variable is set to the identifier of the processing resource to which the iteration is assigned.

As detailed further in Section 5.1.1, a process iteration domain can be split in different ways. We distinguish between modulo unfolding and plane cutting transformations. The assignment to `pr$` depends on the chosen transformation. At line 2 in Figure 4.13, we assign `i%N` to analyze a modulo unfolding transformation. At line 3 in Figure 4.13, we assign `i / ((9-0)/N` to analyze a plane cutting transformation. Setting the `pr$` variable effectively selects the control variable that is used for the iteration. The instrumentation statements for the read, execute, and write stages then work on this control variable according to the method described in Section 4.6.4.

In Figure 4.14, we show the execution profiles obtained after instrumenting the code of Figure 4.10 such that a modulo unfolding transformation on `func1` is modeled. We assume $N = 3$, which results in three partitions of the `func1` statement. For each partition, we maintain separate statement execution profiles, to which we append an $_n$ suffix, with $0 \leq n \leq 2$ identifying the partition. The execution time is reduced to 33 time units, compared to 41 time units for the untransformed case. The number of idle cycles for the `sink` statement is reduced from 8 to zero, which means it now receives data at a fast enough rate. As a result, the processing resource executing `sink` is now fully utilized. However, the three processing resources executing `func1` are now underutilized, because each exhibits 10 idle cycles.

4.6.7 Instrumentation Overhead

A program instrumented by cprof exhibits two forms of overhead: performance degradation and an increased memory footprint. Performance degradation is caused by the instrumentation statements that update the shadow variables, control variables, and execution profiles. These profiling primitives are updated at each statement execution using a few inexpensive addition instructions. However, these instructions result in significant performance degradation when dealing with large function latencies Λ, because the number of instrumentation instructions depends on the latency.

The memory footprint of the instrumented program may easily be twice the memory footprint of the original uninstrumented program. This large memory footprint is mainly caused by two instrumentation primitives: shadow variables and statement

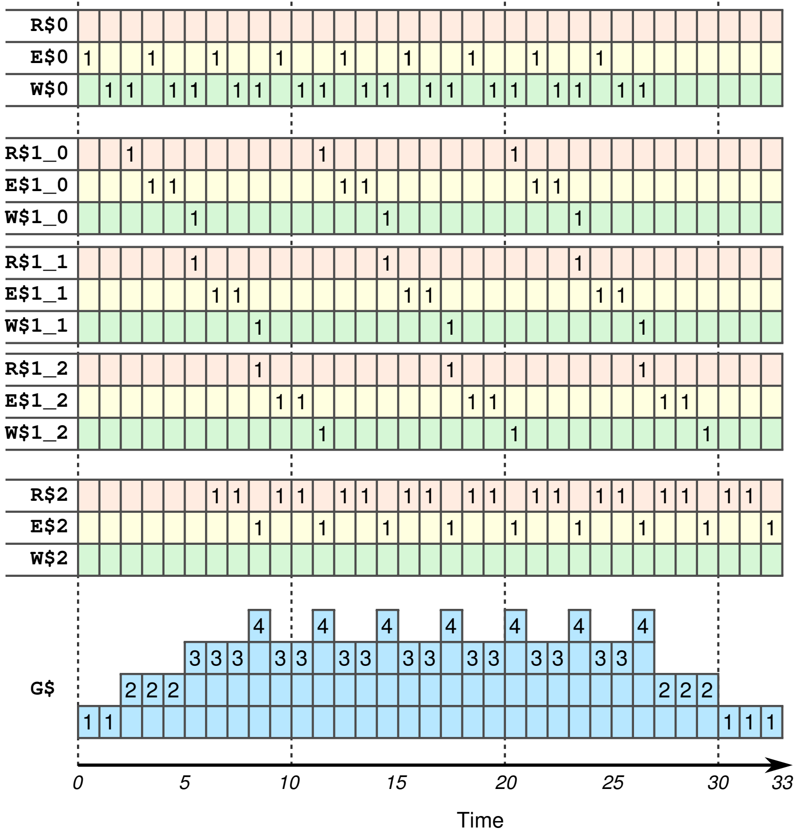


Figure 4.14: Execution profiles obtained by executing the code of Figure 4.10, with statement 1 subject to a modulo unfolding transformation with splitting factor $N = 3$.

Aspect	RTL sim.	SystemC	MCM	cprof
Analytical	✘	✘	✔	✘
Functional validation	✔	✔	✘	✔
Runtime	min-hrs	minutes	seconds	seconds
Accuracy	very high	high	medium	high
Effort	medium	high	small	small
Buffer sizes	✔	✔	✔	✘
Reordering	✔	✘	✘	✔
Interconnect type	✔	✔	✘	✘
Intra-process overlap	✔	✔	✘	✔

Table 4.2: Characteristics of different estimation methods.

execution profiles. For each scalar variable or array element a shadow variable is instantiated. Thus, programs that contain many variables or large arrays result in programs with many shadow variables, increasing the memory footprint. The statement execution profiles mainly affect the memory footprint depending on the execution time. Thus, the memory footprint increases as the execution time of a program increases.

4.7 Comparison

We have discussed four performance estimation methods in the previous sections. How do these four methods compare to each other? In this section, we compare nine different aspects for the four estimation methods in Table 4.2. Below, we discuss each aspect in more detail.

Only the MCM method is *analytical*. This means it does not rely on actual execution of (a simulation model of) the PPN, but computes a performance estimate by analytical means. The other three methods rely on execution of the PPN. That is, each firing of each process is simulated during the estimation. As such, a *functional validation* can be performed with little additional effort, which allows a designer to verify functional correctness.

The *running time* of each method varies from minutes or hours for RTL simulation, to seconds for the MCM and cprof methods. The difference in running times is caused by the difference in the level of detail of the estimation method. For example, the RTL simulation method works at the level of logical gates and registers, while the cprof method works at the level of process firings. For most systems, the number of logical

gates and registers is a few orders of magnitude higher than the number of processes. As such, the RTL simulation method needs to update more simulation primitives per time step than the cprof method, which leads to a longer running time. Because the MCM method is analytical, its running time is independent of the process domain sizes and latencies. In contrast, the non-analytical methods are dependent on these factors, since they simulate every time unit of the system execution.

The *accuracy* of the RTL simulation method is very high, since the method uses the same RTL code that is used for implementation of the system. The accuracy of the SystemC and cprof methods is lower than the accuracy of the RTL simulation method, because low-level details of for example communication delays are omitted in the SystemC and cprof methods. Nevertheless, both approaches have a comparable accuracy, because they use the same characterization of process execution times. The MCM method only has high accuracy for PPNs with uniform dependence distances and without reordering communication. For PPNs with non-uniform dependence distances or reordering communication, the MCM method's accuracy decreases.

The *effort* for the designer to obtain a throughput estimate varies significantly between the four methods. The generation of an RTL simulation project is highly automated in ESPAM's ISE backend. Because the RTL simulation uses the same RTL that is also used for synthesis, no custom simulation models have to be developed. Still, some effort is required from the designer, such as integrating custom IP cores into the system. A SystemC simulation requires considerably more effort, in particular if the system contains custom processing or communication components for which no SystemC model exists. In such a case, the designer has to develop a SystemC model for the unsupported components before a SystemC simulation can be performed. The MCM and cprof methods are both fully automated and require no effort from the designer.

The cprof method currently does not take the finite *buffer sizes* of a PPN into account, as explained in Section 4.6.2. The other three methods do take buffer sizes into account, such that a blocking write resulting from a full FIFO buffer may result in a smaller throughput estimate.

Reordering channels are currently only supported by the RTL simulation and cprof methods. Reordering support for the RTL simulation method is provided by the synthesizable reordering buffer presented in Section 3.6. Reordering support for the SystemC simulation method requires development of a SystemC reordering buffer model. Reordering support for the MCM method requires further investigation. Reordering support for the cprof method is provided because tokens are not stored in a channel model, but are stored in shared random access memory instead.

The MCM and cprof methods assume fixed-latency communication between processes and do not take the *interconnect type* into account. In contrast, SystemC and

Name	Type	#processes	#channels (OO)	cyclic
mns10	kernel	4	3 (0)	no
grid	kernel	4	5 (0)	self
oddeven-sort	kernel	4	13 (0)	yes
dv97ex4	kernel	4	7 (2)	self
qr	kernel	8	15 (0)	yes
mmm	kernel	8	10 (0)	self
mvt	kernel	7	11 (1)	self
sobel	kernel	5	15 (0)	no
mp3dec	application	28	58 (0)	yes
mrvd-qrd	application	43	110 (0)	yes
mjpeg-enc	application	6	6 (0)	no
H.264dec	application	11	24 (0)	yes

Table 4.3: Characteristics of benchmarks used in experiments.

RTL simulation may include any type of communication component, such as a FIFO buffer or Network-on-Chip (NoC). However, the designer needs to develop a SystemC model to use a new interconnect type in a SystemC simulation.

The RTL simulation, SystemC simulation, and cprof methods support *intra-process overlapped execution* of subsequent iterations of a process. The MCM analysis method does not support overlapped execution of subsequent iterations, because it assumes sequential execution of all iterations when determining the execution times of the throughput modeling graph nodes.

4.8 Experimental Results

To assess the feasibility and accuracy of the four performance estimation methods presented in this chapter, we have performed experiments on twelve different applications. The first eight applications are small kernels, whereas the remaining six applications perform a larger amount of work per process. In Table 4.3, we list for each application the number of processes; the total number of channels; the number of out-of-order channels; and whether the PPN is cyclic, acyclic with selfloops, or truly acyclic. For example, application *dv97ex4* consists of 4 processes, and 7 channels of which two are out-of-order, and contains selfloops but no cycles involving multiple nodes.

Not all aspects are covered by each of the four throughput estimation methods. To

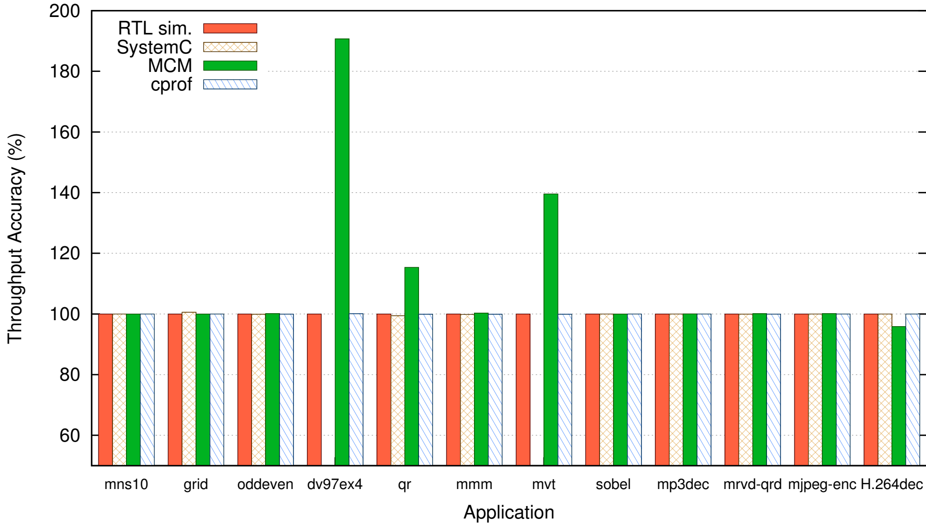


Figure 4.15: Accuracy of throughput estimation methods.

enable a comparison across the four methods, we make the following four assumptions. First, we assume buffer sizes are large enough to avoid performance penalties due to blocking write operations on full channels. That is, increasing any buffer size by any amount does not result in a higher throughput of the PPN. Second, we assume a fixed latency Λ_f for each firing of a function f. Third, we explicitly exclude overlapped execution between iterations of a process by setting each function II_f to $\Lambda_f + c$, where c equals the number of cycles to read and write a single token. As such, no overlap occurs between the read, execute, and write stages of a process. Fourth, we assume that all read operations of an iteration happen in parallel in one cycle according to the LAURA execution model. Similarly, we assume that all write operations of an iteration happen in parallel in one cycle.

4.8.1 Accuracy

We show the accuracy of each of the four methods for our set of twelve applications in Figure 4.15. On the vertical axis, we show the percentual deviation from the actual throughput value. We assume that RTL simulation gives a fully accurate assessment, and thus use the RTL simulation as the baseline for comparing accuracy of the SystemC simulation, MCM, and cprof methods.

As can be seen in Figure 4.15, only the accuracy of the MCM method exhibits significant deviations. The inaccuracy of *qr* and *H.264dec* is caused by incorporating

non-uniform dependences in the MCM modeling graph. The inaccuracy of *dv97ex4* and *mvt* is caused by out-of-order communication in the application. We cannot define tight bounds on the inaccuracy of the MCM method, nor whether the method overestimates or underestimates the actual throughput.

The SystemC and cprof methods deliver highly accurate results for all applications. The difference in reported throughput with RTL simulation is at most on the order of tens of clock cycles, which can be attributed to (re)initialization of components. SystemC simulations are missing for the *dv97ex4* and *mvt* applications, because ESPAM's current SystemC backend does not support reordering communication.

Using any of the four methods described in this chapter, the period of a PPN can be obtained. This period is expressed as a number of clock cycles. However, to obtain the absolute execution time of a PPN period, the number of clock cycles should be multiplied by the clock cycle length. This clock cycle length depends on factors such as combinational path lengths and routing delays. These factors are known only after place-and-route of the PPN's RTL implementation. Thus, none of the four methods allow obtaining throughput assessments expressed in absolute time.

4.8.2 Running Time

We have measured running times of the different estimation methods on an Intel Core 2 Duo system running at a 2400 MHz clock frequency and having 4 GB of RAM available. The running time for RTL simulation includes scripted Xilinx ISE 13.1 project creation, simulation model compilation using Xilinx Fuse 0.40d, and simulation model execution. The running time for SystemC simulation includes compilation and execution of the simulation model. The running time for MCM analysis includes generation of the model and execution of the SDF^3 MCM analysis tool [SGB06]. The running time for cprof includes generation, compilation, and execution of the instrumented code. For all estimation methods, we disable generation of waveforms or traces, to eliminate tracing overhead from the results.

We show the running times of each of the four methods for our set of twelve applications in Table 4.4. The running times for RTL simulation vary from tens of seconds for applications with small function latencies and small domains, to hours or even days for applications with large function latencies or large domains such as *mjpeg-enc*. The running times for SystemC simulation vary from seconds to minutes, making SystemC simulation a few orders of magnitude faster than RTL simulation. The running times for the MCM method are in all but two cases well below one second. Exceptions are *mp3dec* and *mrvd-qrd*, where the large number of edges and cycles in the PPN results in a large number of cycle means to be computed. The running times for cprof are in most cases below one second. Exceptions are *mjpeg-enc*

Application	**RTL sim.** (s)	**SystemC** (s)	**MCM** (s)	**cprof** (s)
mns10	46	2.6	0.1	0.3
grid	34	2.7	0.1	0.3
oddeven-sort	36	2.7	0.1	0.3
dv97ex4	46	n/a (OO)	0.1	0.3
qr	38	3.0	0.1	0.3
mmm	37	2.9	0.1	0.3
mvt	35	n/a (OO)	0.1	0.3
sobel	320	33	0.1	0.3
mp3dec	63	1.2	1.2	0.4
mrvd-qrd	64	4.6	140	0.4
mjpeg-enc	248433	220	0.1	6.4
H.264dec	13771	91	0.1	2.7

Table 4.4: Running times of different estimation methods.

and *H.264dec*, which have large function latencies on the order of thousands of clock cycles. These long latencies result in many updates to the execution profiles, which increases the total running time of cprof. In contrast, the running time of the MCM method does not depend on actual function latency values due to the analytical nature of the MCM method.

4.9 Conclusion and Summary

In this chapter, we have evaluated four different performance estimation methods for PPNs. The first is RTL simulation, which is often not attractive or feasible due the amount of time required to obtain a performance estimate for a given system. The second is SystemC simulation, which yields accurate results in significantly less time compared to RTL simulation. The third is a novel analytical approach for PPNs based on MCM analysis. Our MCM method is able to deliver accurate results for a subset of PPNs. However, we cannot define tight bounds on the inaccuracy of the MCM method, nor whether the method overestimates or underestimates the actual throughput. This model is theoretically attractive and gives insight in the behavior of a PPN, but is impractical because of the lack of accuracy bounds. The fourth is a novel profiling-based approach for PPNs, named cprof. This allows one to obtain accurate results, often in less than one second, without deriving a PPN. Moreover, cprof also allows assessment of the amount of parallelism in the application, and allows early

performance assessment of transformed versions of the applications without the need to actually transform the application.

Each performance estimation method works at a different level of the design phase, providing different tradeoffs between estimation time, effort, and accuracy. In particular the cprof method that works at the sequential code provides a fast, robust, and scalable performance assessment method. Given the characterization using Definition 3.1, cprof can deliver a very accurate performance estimate of a possibly heterogeneous system. As a result, the designer can perform the design iteration depicted in Figure 1.3 in significantly less time.