



Universiteit
Leiden

The Netherlands

Towards design space exploration of industrial distributed cyber-physical systems

Saadatmand, F.S.

Citation

Saadatmand, F. S. (2026, September 30). *Towards design space exploration of industrial distributed cyber-physical systems*. Retrieved from <https://hdl.handle.net/1887/4310251>

Version: 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/1887/4310251>

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

Chapter 6

Workload Model Refinement

Publications related to this chapter

- Faezeh Sadat Saadatmand, Todor Stefanov, Andy D. Pimentel, Benny Akesson, and Ignacio González Alonso. Unraveling Parallelism in Automated Workload Modeling for Distributed Cyber-Physical Systems. In *Euromicro Conference on Digital System Design (DSD)*, pages 24–33. IEEE, September 2025.

This chapter introduces the refined application workload model and a method for parallelism analysis, addressing challenge **CHL5**. Section 6.1 identifies the technical limitations of the initial workload model, used in the CompDSE methodology, that prevent the representation of intra-process parallelism. Section 6.2 outlines the key contributions of this chapter. Section 6.3 provides an overview of the related work with respect to the chapter contributions. Section 6.4 explains the motivation for the refinement of the application workload model for large-scale industrial dCPSs. Section 6.5 describes the structure of the refined workload model and its support for parallel execution. Section 6.6 describes which execution traces need to be collected for model refinement. Section 6.7 explains the automated procedure to derive the refined workload model. Section 6.8 introduces the algorithm used to detect parallel execution opportunities based on the refined workload model. Section 6.9 presents experimental results on the ASML TWINSKAN case study. Finally, Section 6.10 concludes the chapter.

6.1 Problem Statement

The CompDSE methodology enables exploration of different platform configurations and process-to-host mappings, as it is common in system-level DSE for dCPSs. While the derived workload models capture accurately inter-process dependencies based on runtime traces and ensure a valid sequential ordering of communication and computation events within each process, the models assume a fixed, static execution order of events for each operation mode of a process. This abstraction limits the exploration space by excluding alternative execution orders that may exist within the same process.

More specifically, the initial application workload model, presented in Chapter 3, does not represent internal task-level dependencies within a process. As a result, potential intra-process parallelism, which can be crucial for improving performance on modern multi-core CPUs in the hosts, remains unexploited during DSE. To overcome this limitation, an enhanced modeling approach is needed: one that captures both inter-process and intra-process dependencies and enables the identification and exploration of different sequential and parallel execution orders of processes and tasks within processes.

6.2 Contributions

In this chapter, we propose significant refinements to the initial application workload model in CompDSE, enabling it to capture both inter-process and intra-process dependencies. Building on this model, we present a method to identify tasks within each process that can be safely reordered or executed in parallel, without violating the system behavior or requiring modifications to the existing software implementation. More specifically, the key novel contributions are as follows:

1. Our refined application workload model incorporates detailed execution traces, enabling automated analysis and identification of parallel execution opportunities between software processes and tasks within individual processes in large-scale dCPS.
2. We propose a method that identifies eligible tasks within each process that can be safely reordered or executed in parallel with other tasks. The method requires no application-specific knowledge and is compatible with the event-driven execution model (Section 2.4) commonly used in industrial dCPSs.

3. We validate our refined application workload model and the proposed method by automatically deriving models from execution traces collected from the ASML TWINSCAN lithography machine.

6.3 Related Work

Workload modeling approaches in CPS can be broadly grouped into two categories: 1) approaches that do not exploit parallelism and rely on abstract models for system-level optimization, and 2) approaches that capture parallelism by representing concurrent behavior or by improving execution order using scheduling, profiling, or simulation.

Workload Modeling Without Parallelism: Several approaches focus on abstract workload modeling for system-level optimizations without considering parallelism. One study proposes a formal modeling framework based on stochastic processes and fractal methods to probabilistically capture task execution and communication patterns [67]. While this study offers analytical insights into system resource utilization and scheduling, it requires manual mathematical definitions and relies on fixed statistical structures, limiting scalability and adaptability needed for complex industrial dCPSs.

Another study presents a power-aware workload model that integrates computational delays and energy constraints into scheduling decisions [8]. It captures complex workload behaviors, such as irregular patterns and long-term dependencies, using advanced mathematical formulations. However, the model is not automatically derived and depends on predefined analytical profiles, making it inflexible across different dCPS configurations.

Another approach combines stochastic Petri nets, fault trees, and Markov models to assess performance, reliability, and resource utilization [71]. Though effective in analyzing system-level trade-offs, it relies on manually constructed workload scenarios and predefined interaction patterns, which limits its adaptability.

While all these approaches support system-wide analysis, they share two key limitations: they depend on manually defined workloads, which are impractical for large-scale industrial dCPSs, and they assume fixed execution orders, preventing the exploration and exploitation of potential parallelism.

Workload Modeling Incorporating Parallelism: Several studies focus on parallelism detection and modeling within CPS workloads. Identifying safe parallel execution opportunities generally relies on analyzing dependencies between computational tasks or execution units. In related domains such as compiler technology and automatic code parallelization, dependency analysis has been extensively studied to determine

whether tasks/processes and operations can be reordered or executed concurrently without violating data or control dependencies [42, 83]. These analyses are typically performed statically at source code or intermediate-representation level and aim to exploit data-level [5, 14], task-level [19, 47], instruction-level [10, 72], or loop-level parallelism [4, 35, 61].

However, such static dependency analysis is not directly applicable to large-scale industrial dCPS. These systems do not consist of a single centralized program, but of multiple distributed and heterogeneous processes. As a result, the dependencies that govern their execution are typically not explicitly represented in the source code or intermediate representations, especially given the scale and distribution of industrial software systems. Instead, execution dependencies emerge from runtime interactions between processes, making dependency analysis based on runtime information a more appropriate approach in this context.

At the simulation and modeling level, one study explores parallelism in transaction-level modeling (TLM) by analyzing how communication and synchronization mechanisms impact simulation speed in Parallel Discrete Event Simulation (PDES) environments [9]. The authors propose techniques to enhance model-level parallelism in SystemC TLM and evaluate various buffering and communication schemes using a deep neural network case. While effective for simulation optimization, the approach does not model execution dependencies or generate application workload models.

Another approach addresses software inefficiencies by detecting performance antipatterns using queuing network models derived from UML specifications [81]. While the approach effectively quantifies the performance impact of antipatterns and helps identify software architectural bottlenecks, it relies on manually created design artifacts and performance scenarios, which limits its scalability for large-scale industrial dCPS. Additionally, the method focuses on identifying performance antipatterns at the architectural level through resource and deployment modeling, rather than capturing software execution dependencies.

In another study, an AI-driven scheduler, called MCDS, combines Monte Carlo Tree Search with deep surrogate models to optimize workflow execution in edge-cloud systems [106]. It models workflows as Directed Acyclic Graphs (DAGs) and predicts long-term Quality of Service (QoS). However, it assumes that the workflow structure is known in advance and remains fixed, and it also incurs significant computational overhead, making it unsuitable for large-scale dCPS with different configurations.

A more scalable direction is proposed in [87], where application workload models are automatically derived from runtime system traces. While this method supports

DSE of hardware platforms and software-to-hardware mapping in complex dCPS, it assumes fixed task execution orders and does not explore intra-process parallelism.

The aforementioned workload modeling approaches either lack necessary automation and adaptability or do not support parallelism exploration across both inter- and intra-process levels. Specifically, many existing methods rely on manual workload specifications or predefined task execution orders, limiting their use in evolving industrial dCPS configurations.

In contrast, our approach automatically derives refined workload models for each hardware/software configuration using execution traces. These refined models capture both inter- and intra-process dependencies and support the identification of tasks that can be safely reordered or executed in parallel, without modifying the existing software implementation.

6.4 Motivation

In large-scale industrial dCPS, such as the ASML TWINSKAN machine, performance optimization through parallel execution is essential, but achieving it is challenging. These systems typically rely on mature software that has been incrementally developed over many years and involves complex interactions among hundreds of processes. Even minor code changes can disrupt the execution flow and lead to unforeseen downtime or system failures. This raises an important question: How can opportunities for parallel execution be systematically and safely uncovered without modifying the software source code?

To make this challenge more concrete, consider a simplified but representative example of how the forced sequential execution model used in many industrial systems can lead to unnecessary delays. Figure 6.1 illustrates this situation in an event-driven system with single-threaded processes, where requests are queued and handled one at a time. The figure compares two scenarios, each demonstrating a different sequence of request handling across four processes (A, B, C, and D) over time (represented along the vertical axis).

In Scenario 1, process A sends a request `get_state` to process B. To respond, B sends a request `new_state` to process C and waits for its response before replying to A. Meanwhile, process D sends a request `getParam_x` to B to retrieve a parameter needed for executing task `t_x`. However, because B handles requests sequentially, thus this new request is queued until B becomes available, causing the delay shown by the hatched rectangle. Only after completing process A's request, process B can handle process D's

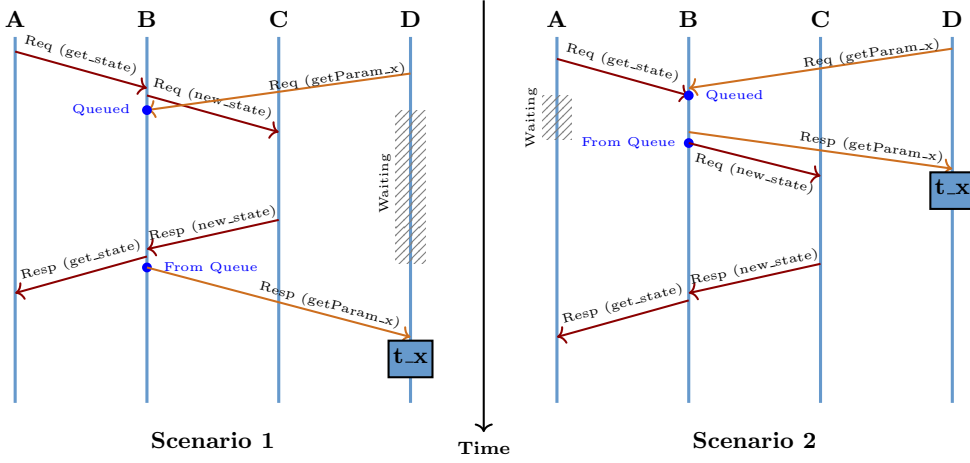


Figure 6.1: The impact of request handling order on execution timing

request, and then send the response.

In Scenario 2, the order of requests arrival is reversed: request `getParam_x` (from D) arrives before `get_state` (from A). By prioritizing process D's request, the waiting time for D is significantly reduced, enabling earlier execution of task `t_x`. Moreover, because `getParam_x` is a relatively lightweight operation, it completes quickly and allows process B to handle process A's request shortly afterward, resulting in less total delay for both. This simple change in execution order demonstrates how intelligent reordering of requests, even without concurrency, can lead to performance gains. The benefit could be even more significant in systems that support multi-threading to concurrently handle independent requests like the independent requests from process A and D to process B.

This example highlights the potential for substantial performance improvements through execution reordering or parallel task execution. However, to support such reasoning, a more detailed and analyzable model of software execution is required, beyond the static, inter-process-only abstractions used in the initial application workload model of CompDSE. To address this limitation, we introduce a refined application workload model in this chapter, one that captures both inter- and intra-process dependencies, enabling the identification of parallelism opportunities without requiring software source code modifications.

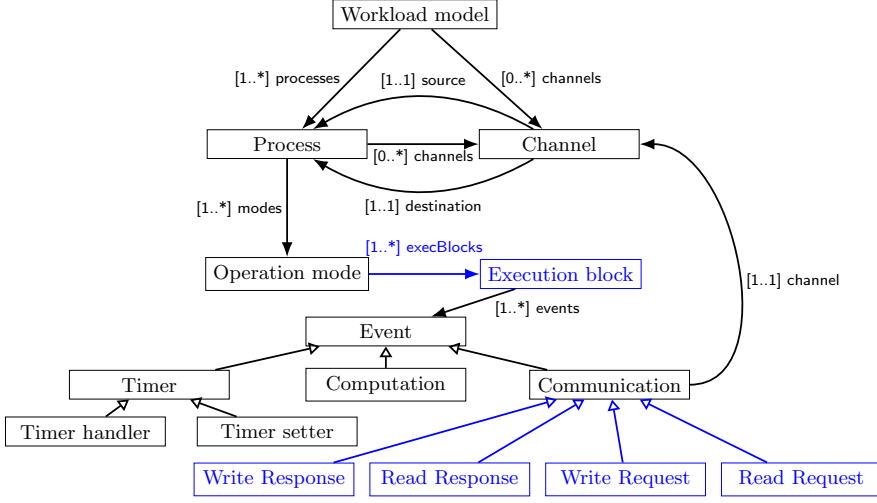


Figure 6.2: The refined application workload model class diagram

6.5 Application Workload Model Refinement

Our refined workload model, illustrated in Figure 6.2, enhances the representation of execution behavior in dCPS. The blue elements in the diagram highlight the key extensions that will be introduced in this section, compared to the initial model described in Section 3.5.1. The workload is structured as a graph of processes and communication channels. Each channel has a source and destination process, and each process may operate in multiple modes to reflect different software/hardware configurations. Our refinement focuses on the internal structure of each operation mode to support intra-process parallelism.

The most significant change in the refined model is the introduction of the *execution block*. Instead of representing each operation mode as a flat sequence of events, we group related events into semantically meaningful segments. Each *execution block* encapsulates a cohesive sequence of communication, computation, or timer events that occur in response to a specific incoming event, such as a received request or response from another process, or the expiration of an internal timer. For example, when a process receives a blocking request, all tasks required to generate the corresponding response – from receiving the request to producing the final reply, including any intermediate computation or further message exchanges – are grouped into a single *execution block*. This level of granularity is chosen not only because it reflects the structure of event-driven execution in dCPS, but also because it captures a contextually

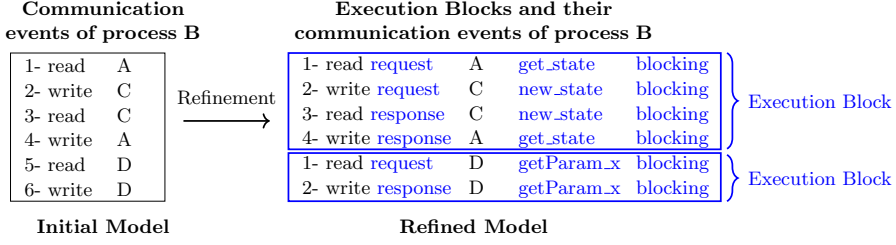


Figure 6.3: Refinement of the application workload model

consistent unit of execution, which is essential for safely identifying dependencies and analyzing parallelism.

Building on the definition of execution blocks, the refined workload model also introduces a more detailed classification of communication events. While the initial model has only two types, **read** and **write**, the refined model distinguishes between four types of communication events: **read request**, **read response**, **write request**, and **write response**. Each communication event also includes a service name, identifying the requested functionality or data, and also a flag indicating whether the interaction is blocking or non-blocking. A blocking request suspends the process until a corresponding response is received, while a non-blocking one allows the process to proceed with other tasks. The other event types, *computation* and *timer*, remain unchanged. Events are assigned to *execution blocks* based on their timestamps and are ordered chronologically within each block. This event order is always preserved.

Figure 6.3 compares the initial and refined workload models for process B, based on Scenario 1 from the example illustrated in Figure 6.1. In this case, process B receives two blocking requests, one from process A (requesting the service `get_state`) and another from process D (requesting `getParam_x`). To respond to the request from process A, process B must send a new request (with service name `new_state`) to process C and wait for a response. As shown, in the initial model (in the left part of Figure 6.3), events form a flat sequence which must be processed in strict order. This structure prevents early handling of new requests, even if they are independent, because they are blocked by preceding events. In contrast, the refined model (in the right part of Figure 6.3) segments the sequence into distinct execution blocks based on request/response boundaries and service names, thereby enabling potential reordering or parallel execution of the blocks under certain conditions that are discussed in Section 6.8.

Table 6.1: Additional trace points for the refined application workload model

Function	Collected Data
handle event loop	timestamp, process name, trace point location, function type (message handler or internal timer handler)
receive request	timestamp, process name, function type, trace point location, command name
receive response	
send request	
send response	

6.6 Collected Execution Traces

Like the initial application workload model, two types of traces, namely application-level execution traces and system-level execution traces are used to derive the refined workload model. While system-level execution traces remain unchanged as used in the initial derivation approach, application-level execution traces require additional information to be collected in order to support automated derivation of the refined application workload model. As discussed in Section 2.4, the processes rely on an event loop to handle both internal and external events. To observe the flow of event processing, we instrument the handle event loop function with trace points at both its start and end. These points record the timestamp, process name, trace point location (start or end), and function type, classified as either a *message handler* or an *internal timer handler*.

In addition to the event loop, we instrument the core message-processing functions. For incoming messages, the event loop receives the message from the appropriate interface (e.g., a file descriptor or socket) and dispatches it to either the *receive request* or *receive response* function, based on header analysis. Similarly, for outgoing messages, the *send request* and *send response* functions are used. All four functions are instrumented at both entry and exit points, capturing the timestamp, process name, function type, trace location, and command name (representing the requested or responded functionality). Since these functions reside in a shared library, the new trace points are automatically inherited by all processes. Table 6.1 summarizes the newly introduced trace points, including the function types, and the data collected.

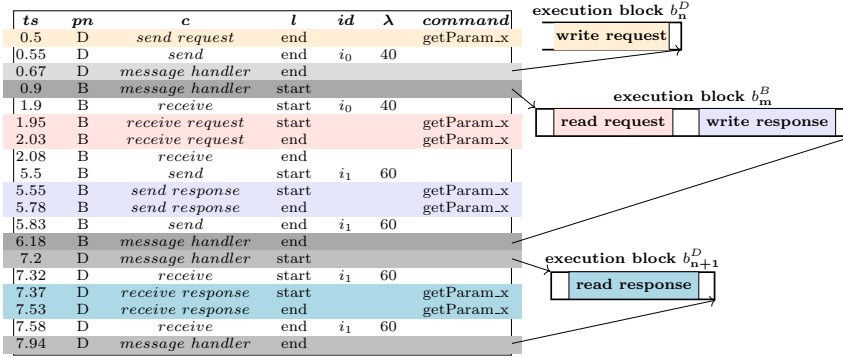


Figure 6.4: Automated derivation of the refined application workload model

6.7 Automated Derivation of Refined Workload Model

To derive the refined application workload model, the aforementioned enhanced execution traces are processed and transformed into the corresponding model components shown in Figure 6.2. This section focuses specifically on the refinement-related aspects, i.e., the blue components in Figure 6.2. The remaining model components are derived by following the initial derivation procedure described in Section 4.5.1. Figure 6.4 illustrates a snippet of collected trace data (left) and its corresponding model components in the refined workload model (right). The trace data includes the timestamp (ts), process name (pn), function type (c), trace point location (l), message identifier (id), message size (λ), and command name. This example is part of the message exchange, illustrated in Figure 6.1, where process D sends a request to process B using the command `getParam.x`. After receiving and processing the request, process B sends a response back, which is later received by process D. This trace segment highlights the key data required for constructing the refined workload model. The derivation process includes three steps: 1) identifying execution blocks, 2) mapping communication events, and 3) determining their blocking behavior. Each step is explained below:

Step 1 - Identify Execution Blocks: Each invocation of the message handler or internal timer handler function in the trace defines the boundaries of an execution block. The start and end trace points determine the temporal scope of each execution block. As shown in Figure 6.4, the gray-highlighted rows indicate the full boundaries of execution blocks b_m^B (for process B) and b_{n+1}^D (for process D). For the earlier block b_n^D , only its end point is visible in the trace snippet.

Step 2 - Map Communication Events: Traces from the four message-passing functions – *receive request*, *receive response*, *send request*, and *send response* – are mapped to the corresponding refined communication event types: **read request**, **read response**, **write request**, and **write response**. Each event is assigned to the execution block whose temporal boundaries contain its timestamp. Every communication event also includes a service name, extracted from the trace (as the command name), indicating the requested functionality. In Figure 6.4, process B execution block b_m^B includes two communication events: **read request** and **write response**. For process D, the earlier block b_n^D may contain multiple communication events, but based on the visible portion of the trace, the last identifiable one is a **write request**. The following block, b_{n+1}^D , contains only a **read response** communication event.

Step 3 - Determine Blocking Behavior: Each communication event is classified as blocking or non-blocking based on whether its corresponding counterpart appears within the same execution block. If the paired event (e.g., a **write response** for a **read request**) is present in the same block and has the same service name with reversed source and destination processes, the event is considered blocking; otherwise, it is non-blocking. For instance, in Figure 6.4, the **read request** in b_m^B is blocking because its paired **write response** occurs in the same block. In contrast, the **read response** in b_{n+1}^D is non-blocking, as its corresponding **write request** occurred earlier in b_n^D .

6.8 Feasibility for Parallel Execution

After deriving the refined application workload model, an analysis is conducted for each process to determine which execution blocks within the process can be safely reordered or executed in parallel. This section describes our analysis approach and is divided into two subsections, i.e., Section 6.8.1 focuses on determining eligibility for parallel execution of blocks, and Section 6.8.2 describes how this information is used during simulation of the modeled dCPS.

6.8.1 Determining Parallel Execution Eligibility of Blocks

To enable parallel execution during simulation, we first determine which execution blocks can safely run in parallel. Algorithm 9 outlines the procedure. It takes two execution blocks from the same process p in operation mode m , and returns a boolean indicating whether they are eligible for parallel execution. The algorithm checks two conditions: 1) whether the blocks are independent (Line 1), and 2) whether they do

Algorithm 9: Check Eligibility for Running in Parallel

Input: $block_1^{p,m}, block_2^{p,m}$
Output: areEligible

```

1  $dep \leftarrow \text{checkDependency}(block_1^{p,m}, block_2^{p,m});$ 
2  $res \leftarrow \text{checkSharedReadResponse}(block_1^{p,m}, block_2^{p,m});$ 
3 if  $dep = \text{false}$  and  $res = \text{false}$  then
4   return true;
5 return false;
```

Algorithm 10: checkDependency

Input: $block_1^{p,m}, block_2^{p,m}$
Output: dependent

```

1  $h_1 \leftarrow \text{getHistory}(block_1^{p,m}); \quad h_2 \leftarrow \text{getHistory}(block_2^{p,m});$ 
2 for  $x_i \in h_1, y_j \in h_2$  do
3   if  $x_i = \text{END}$  or  $y_j = \text{END}$  then
4     return true;
5   if  $x_i.\text{block} = y_j.\text{block}$  then
6     return isBlocking( $x_i.\text{event}$ );
7 return false; /* no shared execution block
```

*/

not share a read response event (Line 2). The blocks are considered eligible for parallel execution only if both conditions are met (Lines 3-4). These checks are described in detail below.

Check 1: Independence of execution blocks: Algorithm 10 formalizes the procedure for determining whether two execution blocks are independent. It receives two blocks of process p in the same operation mode m , where $block_1^{p,m}$ occurs before $block_2^{p,m}$, and returns a boolean indicating whether they are dependent. The algorithm begins by retrieving the call history of each block (Line 1). The call history of a given block is defined as the sequence of execution blocks – along with their triggering events – that recursively led to the block’s activation.

To illustrate this concept, Figure 6.5 shows four processes (A, B, C, and D) in mode m_0 , each containing several execution blocks (e.g., A2, B7, B8, etc.). Each execution block includes a sequence of events, represented as vertical slices within the block. Events highlighted in orange indicate write operations (write request or write response) that trigger execution of blocks in other processes. Blue arrows represent inter-process communication. Suppose we want to determine whether execution blocks D5 and D6 in process D are dependent. The call history of D5 is obtained by tracing communication

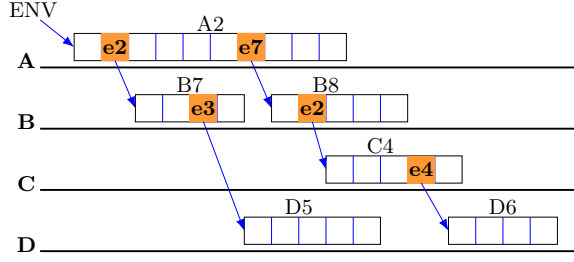


Figure 6.5: The dependency check of execution blocks

backward: from B7 (triggered by event e3), to A2 (triggered by e2), and finally to the environment (ENV), resulting in:

$$B7:e3 \rightarrow A2:e2 \rightarrow \text{END}$$

If an execution block is triggered by a timer event or ENV communication, we mark the end of its call history with **END** because there is no preceding triggering block to trace further. A similar procedure is followed for block D6, resulting in the following call history:

$$C4:e4 \rightarrow B8:e2 \rightarrow A2:e7 \rightarrow \text{END}$$

Once the call histories are obtained, Algorithm 10 compares them to identify the first shared ancestor block (Lines 2-6). If either of the histories reaches **END** without finding a common ancestor, the blocks are conservatively marked as dependent (Lines 3-4), because the causal relationship cannot be fully established. This conservative assumption ensures safe reordering in the absence of traceable triggering dependencies. If both histories terminate at different root blocks without any shared ancestor, the blocks are considered independent (Line 7). However, if a shared ancestor is found (Line 5), the triggering events within that block are compared. If the earlier event is blocking, its execution may impose a constraint on the ordering of the resulting execution blocks. In this case, the algorithm conservatively concludes that the blocks are dependent to preserve the original execution order and prevent unintended behavior (Lines 5-6).

In the example in Figure 6.5, the first shared block is A2, which contains two triggering events e2 and e7. To assess the dependency between D5 and D6, we examine the blocking status of e2. If e2 is a blocking **write request**, the process must wait for a response before executing e7. Similarly, if e2 is a blocking **write response**, it means the associated request occurred earlier in the same block and must be completed first. In both cases, the strict ordering of events implies that not only the events

themselves, but also the execution blocks they trigger, must preserve the original sequence and cannot be reordered. In contrast, if e_2 is a non-blocking **write request**, the response will be handled in a future execution block, so execution of e_7 is not dependent on it. Likewise, if e_2 is a non-blocking **write response**, the corresponding request was received in a previous block, and this response simply completes that interaction without constraining subsequent events. In such cases, the execution blocks are considered independent.

Check 2: Shared read response event detection: After determining the independence of two execution blocks, we further check whether they share a **read response** event. This step ensures that the blocks are not competing to receive the same response, which could otherwise lead to deadlocks or violations of causal message ordering. To perform this check, we inspect all events in both blocks to identify a shared **read response** originating from the same source process and associated with the same service name. If such an event is found, the blocks are not eligible for parallel execution.

It is important to note that it is not necessary to compare every pair of blocks within the same mode of a process. Instead, the analysis is limited to a bounded-size window of neighboring blocks that are likely candidates for reordering. In our implementation, we examine the 30 closest successors of each block. However, runtime profiling shows that, in practice, fewer than 10 neighboring blocks typically run in parallel. This window size is configurable and can be adapted to systems with different workload characteristics or higher potential for parallel execution.

6.8.2 Simulation Strategy for Parallel Execution of Blocks

Once the set of eligible blocks for parallel execution is identified using Algorithm 9, this information is utilized to enable safe concurrent execution during simulations of the modeled dCPS. The refined application workload model is transformed into a format compatible with discrete-event simulators. For our experiments, we use the OMNet++ simulator.

To support parallel execution, we extend the runtime behavior of each process compared to the initial model, and we introduce a shared parallel eligibility file. This file defines, for each process and operation mode, which execution blocks are eligible to run in parallel with one another.

The runtime behavior of each process is illustrated in Listing 6.1, which presents the pseudocode for how execution blocks are managed during simulation. At the start

```

1 void Initialization() {
2     blockList = load_ordered_executionBlocks();
3     readyList = empty; runningList = empty;
4     firstBlk = blockList.front();
5     readyList.add(firstBlk);
6     seqPtr = firstBlk.id;
7     if (allowExecution(firstBlk.initEv)) {
8         runningList.add(firstBlk);
9         readyList.remove(firstBlk);}
10 void onMessageReceived(Message msg) {
11     msgBlk = findRelatedBlock(msg, blockList);
12     eligible = true;
13     foreach (Block bl: runningList) {
14         if (!checkEligibility(msgBlk, bl)) {
15             eligible = false; break; }
16     if (eligible)
17         runningList.add(msgBlk);}
18 void onExecutionFinished(Block* doneBlock) {
19     runningList.remove(doneBlock);
20     if (doneBlock.id == seqPtr) {
21         nextBlock = blockList.front();
22         seqPtr = nextBlock.id;
23         eligible = true;
24         foreach (Block bl: runningList) {
25             if (!checkEligibility(nextBlock, bl)){
26                 eligible = false; break;}}
27         if (eligible) {
28             readyList.add(nextBlock);
29             if (allowExecution(nextBlock.initEv)){
30                 runningList.add(nextBlock);
31                 readyList.remove(nextBlock);}}

```

Listing 6.1: Runtime behaviour of a process in the simulation

of the simulation (`Initialization` procedure, Line 1), each process loads its list of execution blocks (referred to as the `blockList`) while preserving their original order (Line 2). This ordering is particularly important for blocks triggered by the ENV or by internal timer expirations, as they are not eligible to run in parallel with any other blocks and must therefore be executed sequentially, following the original execution order. To track this order, a dedicated pointer called `seqPtr` is maintained. It always points to the head block of the `blockList`, even after that block has been moved to the `readyList` or `runningList`. The pointer is updated only after the block it references has completed execution.

In addition to the `blockList`, the simulation maintains two other structures: A `runningList`, which stores information about currently executing blocks, such as their event counters and status flags (e.g., whether they are waiting for a message or for platform-side execution), as well as a `readyList`, which holds blocks that are eligible to execute as soon as their triggering conditions are satisfied. Both lists are initialized as empty at the beginning of the simulation (Listing 6.1, Line 3).

The simulation begins by moving the first block from the `blockList` to the `readyList` and updating the `seqPtr` accordingly (Lines 4-6). Each execution block includes an

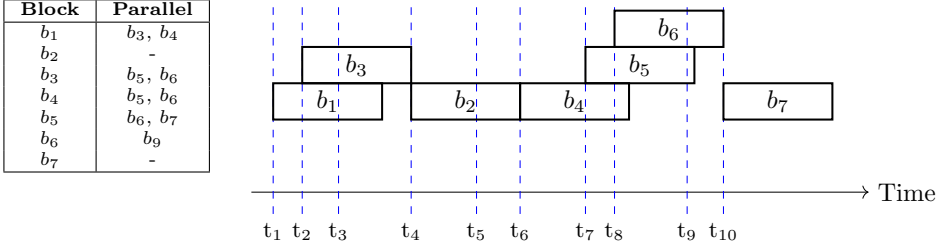


Figure 6.6: An example of parallel execution of blocks

initial communication or timer handler event, depending on what triggered the block, denoted as `initEv`. This event defines the condition under which the block becomes eligible for execution. If `initEv` is a timer handler, the block must wait until the corresponding timer goes off before it becomes eligible for execution. If it is a `read request` or `read response`, the block must wait until the corresponding message is received. If it is a `write request`, the block can be moved directly from the `readyList` to the `runningList` and begin execution (Lines 7-9).

During simulation, when a process receives a message (`onMessageReceived` procedure, Line 10), it searches for the first execution block in its `blockList` whose `initEv` matches the received message (Line 11). The simulator then checks whether this block can be added to the `runningList`, which is determined by verifying that it is eligible to run in parallel with all currently executing blocks (Lines 12-15). If so, it is added to the `runningList` and begins execution (Lines 16-17).

Once a block completes its execution (`onExecution Finished` procedure, Line 18), it is removed from the `runningList` (Line 19). If it is the block currently pointed to by `seqPtr`, the simulator advances `seqPtr` and retrieves the next head block in the original sequence (Lines 20-22). If the new block is eligible to run in parallel with all currently executing blocks, it is added to the `readyList` (Lines 23-28). If its `initEv` allows immediate execution – either because it is a `write request`, or a `read request` or `read response` for which the corresponding message has already been received, or a timer event whose time has already passed – it is transferred directly to the `runningList` (Lines 29-31). If the block is not yet eligible to be added to the `readyList`, it must wait until one or more active blocks complete before it can proceed. This mechanism ensures that execution blocks are scheduled as early as possible to maximize parallelism, while still preserving the original block order when blocks are not eligible to run in parallel.

Figure 6.6 illustrates a scenario involving parallel execution. The table on the left indicates, for each execution block, which subsequent blocks are eligible to run in

parallel. As noted earlier, only blocks that appear later in the `blockList` are considered for reordering, as earlier blocks are assumed to have already completed. The simulation begins at time t_1 with the execution of block b_1 , assuming its `initEv` is a write request, and `seqPtr` is initialized to point to b_1 . At t_2 , the message associated with block b_3 's `initEv` arrives. Since b_1 and b_3 are eligible to run in parallel based on the table, b_3 is immediately scheduled for execution. At t_3 , the message corresponding to block b_2 's `initEv` is also received, making it a candidate for execution. However, due to dependency constraints, b_2 cannot run concurrently with the currently executing blocks (b_1 and b_3), and is therefore not scheduled at this point. Once b_3 completes its execution, the simulator retrieves the next block at the head of the `blockList` and updates `seqPtr` to b_2 . Because the message for b_2 has already been received, it is immediately moved to the `runningList` and begins execution.

At time t_5 , the message corresponding to the `initEv` of block b_4 is received, making it ready for potential execution. However, b_4 cannot run concurrently with the currently executing block b_2 based on the table in the left part of Figure 6.6, so the message is queued. Once b_2 completes its execution, the next block at the head of the `blockList` is retrieved, and `seqPtr` is updated to point to b_4 . At time t_7 and t_8 , the messages for b_5 and b_6 are received. Since both blocks are eligible to run in parallel with the currently executing block b_4 , they are scheduled and executed concurrently. At time t_9 , the message related to `initEv` of block b_7 is received. However, b_7 cannot start yet because it is not eligible to run in parallel with either b_5 or b_6 . Finally, at time t_{10} , after both b_5 and b_6 complete their execution, b_7 becomes the next block at the head of the `blockList` and is scheduled for execution.

6.9 Experimental Evaluation

To evaluate the effectiveness of our refined application workload model and the automated parallelism detection method, we conduct an experimental study using trace data collected from the ASML TWINSCAN lithography machine.

6.9.1 Experimental Setup

Our evaluation focuses on the server subsystem of the TWINSCAN machine, which runs on a Linux-based platform and communicates with multiple embedded real-time subsystems. We concentrate on the server subsystem because it executes the majority of software processes and is therefore the most critical location for uncovering

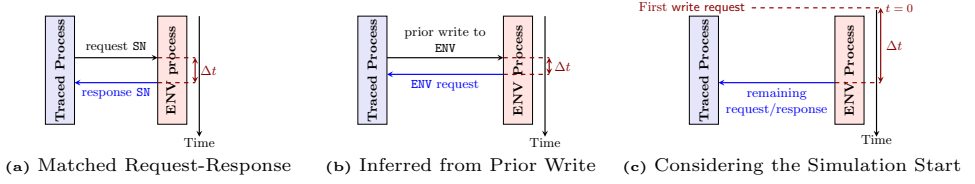
parallelism opportunities. The embedded subsystems, in contrast, operate under strict real-time constraints, where modifying execution behavior is undesirable, and collecting their traces would require additional engineering effort that is not necessary to demonstrate the capabilities of the proposed refined model. From the collected traces of a wafer lot operation involving five wafers, we derive an application workload model containing **309** software processes and **1521** inter-process communication channels. The workload model includes **489459** execution blocks and **597210** messages, representing communications in which the traced host acted as sender, receiver, or both.

To evaluate the wafer processing time, we identified service calls in the traces that correspond to wafer-handling events. These calls, observed in specific request and response messages, are used to mark the start and end points of each wafer's operation. The PDW is then computed as the time difference between the corresponding start and end events.

To model the effects of the environment on the system's timing behavior, we use the ENV model with associated delay parameters, denoted as Δ , as explained in Section 3.5.4. The refinement of the workload model, particularly through the introduction of more detailed communication event types and the inclusion of service names, enables a more fine-grained estimation of the values of these delay parameters. Figure 6.7 illustrates the three scenarios considered in our ENV delay modeling approach. In most cases, since each communication event includes a service name (**SN**), we directly determine the delay parameter value Δ by calculating the time difference between a traced process's **write request** to an ENV process and the corresponding **read response** from that ENV process using the same service name, as shown in Figure 6.7(a).

For situations where the ENV initiates a request toward the traced process (**ENV request**), we estimate the delay Δ based on the most recent **write request** or **write response** previously sent by the same traced process to that ENV process. We assume that the ENV request occurs shortly after this preceding write event, as illustrated in Figure 6.7(b).

Finally, in cases where no matching service name (**SN**) is available or no prior write event can be found, we conservatively assign the delay based on the time difference from the first root-level **write request** in the model. This event marks the simulation's starting point and represents the earliest available interaction, as shown in Figure 6.7(c).

**Figure 6.7:** Three ENV delay modeling scenarios**Table 6.2:** Comparison of PDW between real system execution and simulation considering inter-process parallelism only

Wafer	Reality (s)	Simulation (s)
1	47.638	47.651
2	56.764	56.782
3	50.766	50.782
4	47.715	47.735
5	47.770	47.788
Average	50.130	50.148

6.9.2 Results

First, we validate the accuracy of our simulation model by comparing the simulated PDW with real execution data from the TWINSKAN machine. This comparison is performed under a baseline configuration that includes only inter-process communication and does not leverage intra-process parallelism. This setup ensures that the simulation reflects the actual system behavior and provides a reliable baseline for further performance evaluation.

Table 6.2 shows the PDW for five wafers from both real and simulated executions. The average PDW in the real system is 50.1309 s, compared to 50.1481 s in simulation, a negligible difference of 17.2 ms, corresponding to an error of only 0.034 %. This confirms that the simulation model provides sufficient accuracy for performance analysis.

Next, we examine the impact of enabling intra-process parallelism alongside inter-process parallelism. We vary two key parameters: the number of CPU cores in the platform model, and the value of Δ in the ENV delay model, which controls how quickly new execution blocks triggered by environment interactions become eligible for execution and thus increases the opportunity for parallelism. Five levels of ENV delay reduction are tested: 0 % (baseline), 25 %, 50 %, 75 %, and 100 % (representing immediate ENV response upon sending a request from the traced server). As a reminder, execution blocks triggered by ENV interactions must be executed sequentially in their original order and are not eligible for parallel execution.

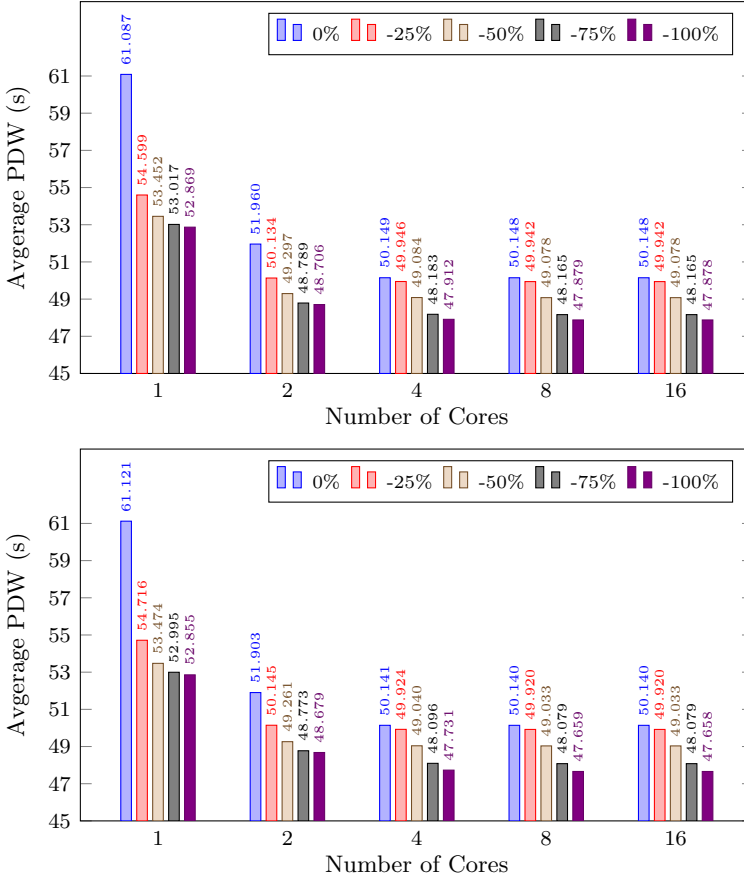


Figure 6.8: The effect of ENV delay reduction and number of cores on PDW. Top: inter-process only. Bottom: inter- and intra-process

Figure 6.8 shows the effect of the aforementioned parameters on the average PDW. The top chart reflects inter-process parallelism only, while the bottom chart includes both inter- and intra-process parallelism. Each bar corresponds to a specific ENV delay level. The horizontal axis indicates the number of CPU cores (the default server configuration has 16 cores), and the vertical axis shows the average PDW in seconds. As shown in both charts, decreasing ENV delays and increasing the number of cores consistently reduce the average PDW.

The results demonstrate that intra-process parallelism yields additional performance gains, particularly when ENV delays are reduced. At the baseline level (0% delay reduction), the improvement on the default 16-core platform is marginal, at around 8 ms,

indicating that the environment latency remains the dominant bottleneck. Although intra-process execution is accelerated, the system must still wait for ENV interactions, especially those modeled with conservative assumptions anchored to the root-level **write request**, as previously shown in Figure 6.7(c), which limits the overall benefit. In contrast, when ENV delays are fully eliminated (100%), the gain increases to approximately 220 ms, highlighting that the effectiveness of intra-process parallelism depends heavily on the environment responsiveness in this case study.

Despite this improvement, the overall gain remains modest, which can be attributed to two main factors. First, the software under study was not originally designed with parallel execution in mind, likely to avoid unintended side effects such as race conditions or inconsistent execution behavior. As a result, the structure of the system may inherently limit the amount of parallelism that can be safely exploited. Second, our method for identifying parallelizable blocks is deliberately conservative. As discussed in Check 1 (Section 6.8.1), conservative assumptions are necessary when trace information is insufficient to guarantee safe block independence. These assumptions prevent potential deadlocks in the system but may also restrict opportunities for concurrent execution.

Finally, we evaluate the impact of CPU frequency scaling using a fixed configuration of 8 cores under 100 % ENV delay reduction. This choice is made because, as shown in Figure 6.8, the difference between using 8 and 16 cores is not significant and falls within the simulation error margin.

Figure 6.9 shows the effect of frequency scaling on PDW under both inter-process-only and combined inter- and intra-process parallelism exploitation cases. The base frequency setting is denoted as 0 on the horizontal axis, while the average PDW is depicted on the vertical axis. The horizontal axis depicts the clock frequency changes as a percentage of the base clock frequency. As CPU frequency increases, performance improves in both cases due to shorter execution times. However, the performance difference between the two cases gradually decreases, because frequency scaling accelerates all execution blocks, including those that must run sequentially. As sequential execution becomes less time-consuming, the relative advantage of parallelism exploitation diminishes. Consequently, the two cases converge in performance at higher frequencies.

All our results above indicate that multi-threaded execution of code blocks within event-driven processes (i.e., exploiting the available intra-process parallelism) can yield measurable performance improvements, particularly when the responsiveness of other embedded hosts (ENV) increases. However, they also reveal a saturation point,

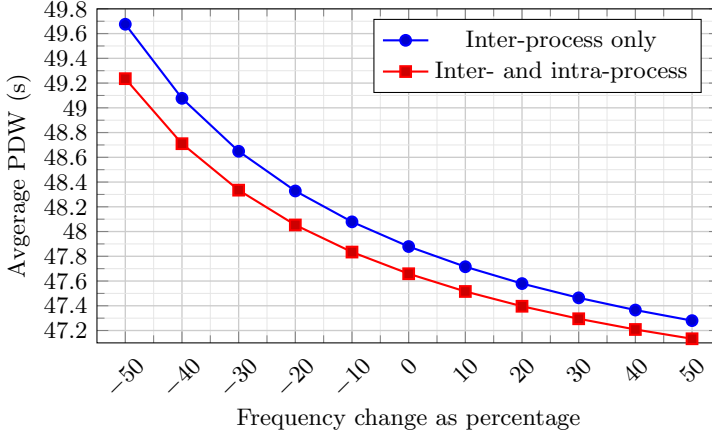


Figure 6.9: The effect of frequency scaling on PDW for 8-core platform with 100% ENV delay reduction

beyond which such multi-threaded execution no longer leads to performance gains unless changes are made to the software structure of dCPS in order to create (more) blocks and processes that can really execute in parallel. This underscores the value of our refined application workload model, not only in enabling realistic performance evaluation of an existing software structure of dCPS but also in supporting exploratory what-if analyses with respect to potential changes in the software structure.

6.10 Conclusion

This chapter presents a refined application workload model that captures both inter-process and intra-process execution dependencies in complex dCPS. The model is derived from execution traces and supports a method for identifying tasks (code blocks) within software processes that can be safely reordered or executed in parallel without modifying the existing software structure. We validate our approach on the ASML TWINSCAN lithography machine and the results reveal measurable performance gains in wafer processing time. The results also highlight the impact of environmental responsiveness and hardware capacity on the effectiveness of parallel execution. These findings underscore the value of our refined workload model not only for realistic performance evaluation, but also for enabling what-if analyses of application workloads that complements platform and mapping exploration in large-scale industrial dCPS.