Unraveling Parallelism in Automated Workload Modeling for Distributed Cyber-Physical Systems

Faezeh Sadat Saadatmand*, Todor Stefanov*, Andy D. Pimentel‡,
Benny Akesson§‡, and Ignacio González Alonso†

*Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands †ASML B.V., The Netherlands
‡Informatics Institute, University of Amsterdam, The Netherlands §TNO-ESI, The Netherlands

***Abstract*—Designing next generation distributed Cyber-Physical Systems (dCPS) requires effective Design Space Exploration (DSE) methods to evaluate system design alternatives and their impact on performance. While existing DSE approaches focus on hardware optimization and software-to-hardware mapping, they often overlook parallel execution opportunities within software tasks. Current application workload models for complex dCPS assume fixed execution orders, limiting the ability to explore and exploit software parallelism. To address this issue, we propose refined workload models derived from execution traces that capture both inter- and intra-process dependencies. Building on these models, we present a method to identify tasks that can be safely reordered or executed in parallel without modifying the existing software implementation. We validate our approach through a case study on the ASML Twinscan lithography machine, demonstrating measurable performance improvements without impacting the system functional correctness.**

***Index Terms*—Distributed Cyber-Physical Systems, Application Workload Model, Parallel Execution, Design Space Exploration**

I. Introduction

Distributed Cyber-Physical Systems (dCPS) are the foundation of modern high-tech industries, integrating computing and physical processes to enable real-time control, automation, and intelligent decision-making. These systems are widely used in various industries, including health, industrial automation, robotics, avionics, and space, where software and hardware must work together seamlessly. As dCPS grow in complexity, designing and optimizing their cyber components, which comprise distributed software processes running on multi-core or many-core processors and communicating over intricate networks, has become increasingly challenging [1]. The shift toward software-intensive dCPS, where more functionality is implemented in software for flexibility and adaptability, further amplifies these challenges [2]. Consequently, system designers must evaluate various system architectures and software execution strategies to ensure that system requirements are met.

To systematically explore system architectures and software execution strategies in dCPS, Design Space Exploration (DSE) provides a structured approach for evaluating different system configurations [3]. DSE systematically assesses various design alternatives to optimize non-functional system properties such as performance, cost, and energy consumption. To achieve this, it commonly follows the Y-chart approach [4], which separates the system into three models: the software application workload, the hardware platform, and the mapping of the workload to the platform. This separation of concerns ensures that the software application and the hardware platform are modeled independently, and both models are explicitly related via the mapping model, thereby allowing for greater flexibility in system optimization.

In addition, application workload modeling for large-scale dCPS requires finding a careful balance between abstraction and accuracy. This balance is essential to ensure that DSE remains computationally efficient while still capturing the key characteristics of the application software workload. However, due to the complexity and variability inherent in dCPS, which typically involve hundreds of interacting processes and their internal computation and communication tasks, manual derivation of workload models is impractical. Specifically, process behaviors can differ significantly based on different hardware/software configurations, making it impossible to manually specify workload behaviors for each scenario. Therefore, it is essential to automatically derive workload models to accurately represent realistic system behaviors under diverse software/hardware configurations.

To address the aforementioned application modeling challenges, the authors of [5] have proposed techniques for automated workload model derivation in dCPS, achieving a good trade-off between model abstraction and accuracy. The derived models leverage system-level traces to capture accurately inter-process dependencies and to represent only a valid sequential execution order of computation and communication tasks within each process. However, the derived models do not capture dependencies among the tasks within the processes, thus preventing exploration and exploitation of potential intra-process parallelism during DSE. To address this limitation, an enhanced modeling approach enables the identification and exploration of different sequential and parallel execution orders of processes and tasks within processes.

In this paper, we propose significant refinements to the application workload model presented in [5], 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 de-

tailed 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 commonly used in industrial dCPS.
3) We validate our refined workload model and the proposed method by automatically deriving models from execution traces collected from a complex industrial dCPS, the ASML Twinscan lithography machine. Our evaluation demonstrates that incorporating intra-process parallelism, in addition to inter-process execution, leads to measurable performance gains.

The remainder of this paper is structured as follows: Section II reviews related work on workload modeling in dCPS. Section III provides background information relevant to our approach. Section IV describes the refinement of the application workload model and its automated derivation process. Section V presents the method for identifying parallel execution opportunities. Section VI reports on the experimental evaluation, and Section VII concludes the paper.

II. Related Work

Workload modeling has been widely studied in cyber-physical and embedded systems. In embedded systems, models such as Kahn Process Networks (KPN), Roofline models, and automatic parallelization tools are commonly used to support performance analysis and optimization. These methods typically rely on static execution graphs and presume predictable task dependencies, which make them suitable for small-scale environments with limited resources. However, large-scale industrial dCPS introduce significantly greater complexity, scalability challenges, and dynamic execution behavior [6]. In these systems, workloads involve hundreds of interacting processes, where execution order, both within and across processes, can vary depending on the hardware/software configuration. As a result, static workload models are insufficient, prompting the need for automated, configuration-specific system-level workload models [5].

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

Workload Modeling Without Parallelism: Several approaches focus on abstract workload modeling for system-level optimization 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 [7]. While this study offers analytical insight into resource utilization and scheduling, it requires manual mathematical definitions and relies on fixed statistical structures, limiting scalability and adaptability needed for complex industrial dCPS.

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 [9]. 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 dCPS, 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. 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 [10]. 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 [11]. 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 [12]. 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 [5], 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.

III. Background

This section outlines the modeling foundations and trace collection methodology used in our work. We begin by describing the event-driven execution model common in distributed systems, followed by a brief overview of the models, the required traces, and how these models are automatically derived for use in discrete-event simulation.

A. Event-Driven Execution in Distributed Systems

High-performance distributed systems, including industrial dCPS, commonly use an event-driven execution model, where software processes respond to incoming events rather than executing in a fixed sequence. At the core of this model is the event loop, which continuously listens for system events, such as messages, timer expirations, or scheduling triggers, and asynchronously invokes the appropriate procedures.

When an event occurs, the corresponding process fires, executes a well-defined set of tasks in response, and then exits. These tasks depend on the triggering event and typically include computation or communication operations, such as sending a message or triggering another event. This reactive approach enables efficient handling of multiple requests by ensuring that processes are activated only in response to relevant events, thereby avoiding unnecessary or periodic execution.

Depending on the system's workload characteristics and available resources, event-driven execution may follow a single-threaded strategy where events are handled sequentially, or a multi-threaded approach where events are processed in parallel by worker threads.

B. Models and Trace-Based Derivation Method

To enable our analysis, we build upon an existing modeling approach that derives abstract application workload models for large-scale dCPS from runtime traces [5]. This approach captures the behavior of software processes through a workload model and accounts for environmental timing effects using an ENV model, which reflects delays introduced by embedded hosts interacting with the physical environment. All models are automatically derived and transformed into executable representations for discrete-event simulation.

Since our contribution focuses on refining this workload model to support parallelism analysis, we briefly describe the original modeling approach by outlining the types of traces it uses, the structure of the workload and ENV models, and how these models are derived automatically from traces. Full modeling details are available in [5] and are not repeated here.

Traces: Model derivation relies on runtime data collected once per hardware/software configuration during correct system operation. Two types of traces are used:

1) Execution Traces (User Space): These traces are collected via a few trace points strategically placed in the software application code to capture the execution flow, function calls, and inter-process communication. They record data such as timestamps, process identifiers, function types, and any attributes specific to each function. For example, traces for send/receive functions include message identifiers and message sizes. In Linux-based systems, LTTng [13] is used to collect this information with minimal overhead.

2) System Status Traces (Kernel Space): These traces provide system-level information such as CPU usage, process states, and clock frequency changes. A key parameter in the proposed model is the `OnCPU` interval, which indicates when a process is actively running and is derived using context switch events. In Linux-based systems, Trace Compass [14] is used to analyze these kernel traces and extract precise `OnCPU` intervals using predefined event models.

Application Workload Model: is defined as a process graph consisting of software processes and the communication channels between them. It captures each process execution behavior, including both computation and communication tasks. Each process is modeled as a set of *operation modes*, allowing different workload scenarios that correspond to specific software/hardware configurations. Within each mode, execution is expressed as a sequence of abstract events, categorized as follows:

- Computation Event: Represents computation tasks with an abstract workload signature that approximates the computational demand of the task. As an approximation, this signature is calculated by multiplying the `OnCPU` duration by the operating frequency of the core on which the process was running.
- Communication Event: Models message exchanges, categorized as *Write* (sending a message) or *Read* (receiving a message). Each event specifies the communication channel between the source and destination processes, the message size, and a workload signature that reflects the computational cost associated with either sending or receiving the message, depending on the event type.
- Timer Event: Represents internal triggers within a process that activate computation or communication events after a specified delay. Each timer event is either a *timer setter*, which sets a timer with an absolute time value, or a *timer handler*, which is triggered once that time elapses. Timer events include a unique identifier and duration.

Environment Influence (ENV) Model: In dCPS, embedded hosts interact directly with the physical environment through sensors and actuators. These hosts may not be fully traceable due to system constraints, such as limited instrumentation support or the overhead that tracing would impose on resource-constrained platforms. Although these hosts are untraced, their interactions with traced processes can introduce significant timing variations that affect overall system behavior.

To address this, the ENV model estimates the timing influence of untraced hosts on the traced system using a delay-based approach. Specifically, it introduces a parameter Δ to account for unknown delays in message delivery. For each traced process (`tp`), a Δ value is assigned to every *Read* event originating from an untraced process (`utp`). This delay is estimated as the time difference between the timestamp of the *Read* event (from `utp` to `tp`) and the most recent *Write* previously sent by `tp` to `utp`.

Automated Derivation and Model Transformation: To derive the application workload model, matching message identifiers in the send and receive functions are tracked from execution traces. This enables the identification of *Write* and *Read* communication events for each process. The trace also enables the identification of individual software processes, and based on observed message exchanges, communication channels are derived between process pairs. Similarly, *Timer* events are derived from traces of timer-related functions using their unique timer identifiers. Between every communication or timer event, a *Computation* event is inserted, and its workload signature is estimated based on the amount of `OnCPU` time and the core clock frequency during its interval, as derived from system status traces. Although the traces are collected across multiple cores, the transformation focuses on communication and computation events per process, without modeling detailed core-level migrations.

IV. Workload Model Refinement

In this section, we introduce our refined workload modeling approach that enhances the representation of software execution behavior in dCPS. We begin with the motivation for the refinement, then describe our structural changes to the model introduced in Section III-B, outline the additional trace requirements, and explain our automated derivation process used to generate the refined model from these traces.

A. Motivation

In large-scale industrial dCPS, such as the ASML Twinscan machine, performance optimization through parallel execution is essential but challenging. These systems typically rely on mature software that has been incrementally developed over many years and involves complex interactions among hundreds of processes. As a result, even minor 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 source code?

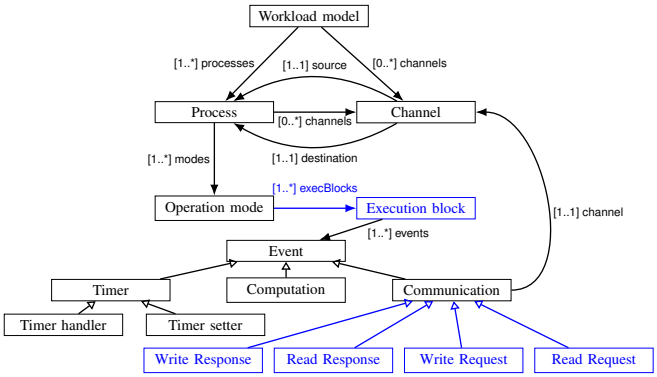

Fig. 1: Workload model class diagram

To address this question, a more detailed and analyzable representation of software execution is needed, beyond what is provided by existing workload models that capture only inter-process dependencies. Such a model must support reasoning about dependencies and opportunities for parallelism. Our refined application workload model, introduced in this paper, addresses this need.

B. Refined Application Workload Model

Our refined workload model, illustrated in Fig. 1, enhances the representation of execution behavior in dCPS. The blue elements in the diagram highlight the key extensions, we introduce, compared to the original model introduced by the authors of [5]. 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 such systems, but also because it captures a contextually 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 original model in [5] has only two types, *Read* and *Write*, the refined model distinguishes between four types of communication events: *Read*

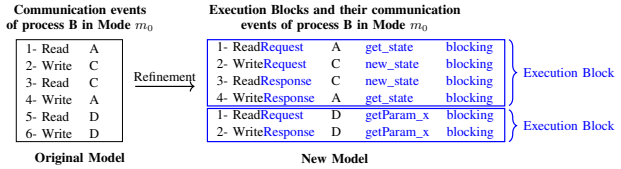

Fig. 2: Refinement of workload model

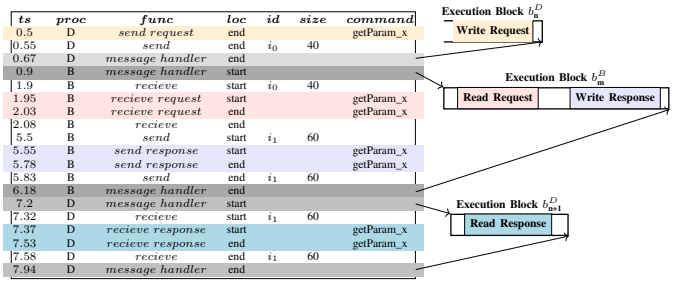

Fig. 3: Automated derivation of refined workload model

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.

Fig. 2 illustrates an example comparison between the original and refined workload models for process B in mode m_0. In this example, 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 original model (left), events form a flat sequence that 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, our refined model (right) segments the sequence into distinct execution blocks based on request/response boundaries and service names.

C. Collected Traces

As mentioned in Section III-B, two types of traces are used to derive the models in [5]: execution traces and system status traces. While system status traces remain unchanged from the original approach, execution traces require additional detail to support the derivation of our refined workload model. As discussed in Section III-A, 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 trace points are automatically inherited by all processes.

D. Automated Derivation of Refined Workload Model

To derive our refined workload model, we process the aforementioned enhanced execution traces and transform them into the corresponding model components shown in Fig. 1. This section focuses specifically on the refinement-related aspects, i.e., the blue components in Fig. 1. The remaining model components follow the original derivation procedure described in [5]. Fig. 3 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, function type, trace point location, message identifier, message size, and command name. In this example, 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: identifying execution blocks, mapping communication events, and 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 Fig. 3, 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 Fig. 3, 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

Algorithm 1: Check Eligibility for Running in Parallel

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

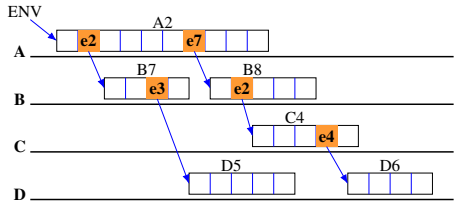

Fig. 4: Dependency check of execution blocks

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 Fig. 3, 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.

V. Feasibility of Parallel Execution

After constructing the refined workload model, we analyze, for each process, which execution blocks can be safely reordered or executed in parallel. This section describes our analysis approach and is divided into two parts: the first part focuses on determining eligibility for parallel execution of blocks, and the second describes how this information is used during a simulation of the modeled dCPS.

A. Determining Parallel Execution Eligibility

To enable parallel execution during simulation, we first determine which execution blocks can safely run in parallel. Algorithm 1 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 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.

Algorithm 2: checkDependency

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

Check 1: Independence of Execution Blocks: Algorithm 2 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, Fig. 4 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 in other processes. Blue arrows represent inter-process communication. Suppose we want to determine whether execution blocks D5 and D6 are dependent. The call history of D5 is obtained by tracing communication 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 $\rightarrow$ 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 $\rightarrow$ END

Once the call histories are obtained, Algorithm 2 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), since 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 Fig. 4, 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 e2 is a non-blocking *Write Request*, the response will be handled in a future execution block, so execution of e7 is not dependent on it. Likewise, if e2 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.

B. Simulation Strategy for Parallel Execution

Once the set of eligible blocks for parallel execution is identified using Algorithm 1, this information is utilized to enable safe concurrent execution during simulations of the modeled dCPS. The refined workload model is transformed into a format compatible with discrete-event simulators. For our experiments, we use the OMNeT++ [15] simulator. To support parallel execution, we extend the runtime behavior of each process compared to the original model in [5], 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 1, which presents the pseudocode for how execution blocks are managed during simulation. At the start of the

```
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 1: Runtime behaviour of a process in the simulation

simulation (Line 1), each process loads its list of execution blocks (`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 `blockList`, even after that block has been moved to `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). And, secondly, 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 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 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 `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 (`onExecutionFinished` procedure, Line 18), it is removed from the `runningList` (Line 19). If it was 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.

Fig. 5 illustrates a scenario involving parallel execution. The table on the left-hand side 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 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 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, 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 t_5, the message corresponding to the `initEv` of 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 left table, so the message remains in the queue. Once b_2 completes, the next block at the head of the `blockList` is retrieved, and `seqPtr` is updated to point to b_4. At 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 t_9, the message related to `initEv` of 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 t_{10}, after both b_5 and b_6 complete execution, b_7 becomes the next block at the head of the `blockList` and is scheduled for execution.

Block	Parallel
b_1	b_3, b_4
b_2	-
b_3	b_5, b_6
b_4	b_5, b_6
b_5	b_6, b_7
b_6	b_9
b_7	-

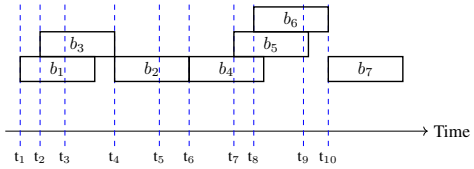

Fig. 5: Example of Parallel Execution of Blocks

VI. Experimental Evaluation

To evaluate the effectiveness of our refined application workload model and the automated parallelism detection method, we conducted an experimental study using trace data collected from the ASML Twinscan lithography machine, a large-scale industrial dCPS. This type of machine is at the core of modern semiconductor manufacturing, utilizing advanced optics and precision mechanics to transfer circuit patterns onto silicon wafers. The machine comprises numerous heterogeneous subsystems, each hosting many distributed software processes. These processes interact through thousands of message exchanges per second, forming a highly complex and performance-critical software infrastructure.

A. Experimental Setup

Our evaluation focused on the server subsystem of the Twinscan machine, which runs on a Linux-based platform and communicates with multiple embedded real-time systems. From the collected traces of a wafer batch operation involving five wafers, we derived 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, were used to mark the start and end points of each wafers operation. The processing duration per wafer (**PDW**) was then computed as the time difference between the corresponding start and end events.

To model the effects of the environment on timing behavior, we applied the ENV model with associated delay parameters, denoted as Δ, as introduced in Section III-B. 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 these delays. Fig. 6 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 measure the delay by calculating the time difference between a servers *Write Request* to an ENV process and the corresponding *Read Response* from that ENV process using the same service name, as shown in Fig. 6(a).

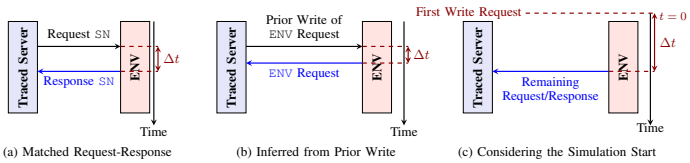

Fig. 6: Three ENV delay modeling scenarios

TABLE I: Comparison of PDW between real system execution and simulation considering inter-process parallelism only

Wafer	Reality (s)	Simulation (s)
1	47.6383	47.6513
2	56.7641	56.7827
3	50.7661	50.7829
4	47.7157	47.7357
5	47.7700	47.7880
Average	**50.1309**	**50.1481**

For situations where the ENV initiates a request toward the server (ENV Request), we estimated the delay based on the most recent *Write Request* or *Write Response* previously sent by the same server process to that ENV process. We assume that the ENV request occurs shortly after this preceding write event, as illustrated in Fig. 6(b).

Finally, in cases where no matching service name 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 simulations starting point and represents the earliest available interaction, as shown in Fig. 6(c).

B. Results

First, we validated the accuracy of our simulation model by comparing the simulated PDW with real execution data from the ASML Twinscan machine. This comparison was 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 I shows the PDW for five wafers from both real and simulated executions. The average PDW in the real system is $50.1309\,\mathrm{s}$, compared to $50.1481\,\mathrm{s}$ in simulation, a negligible difference of $\mathbf{17.2}\,\mathrm{ms}$, corresponding to an error of only $\mathbf{0.034}\,\%$. This confirms that the simulation model provides sufficient accuracy for performance analysis.

Next, we examined the impact of enabling intra-process parallelism alongside inter-process parallelism. This evaluation varied 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 were 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

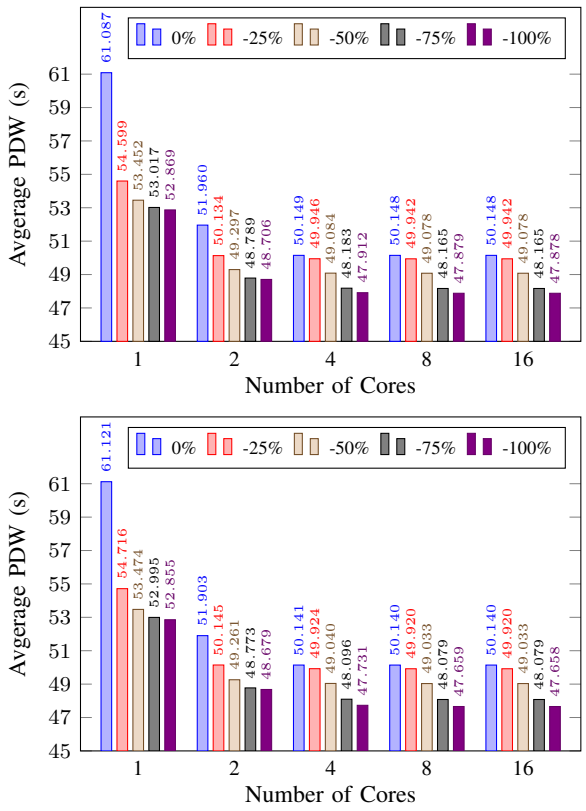

Fig. 7: Effect of ENV delay reduction and number of cores on PDW. Top: inter-process only. Bottom: inter- and intra-process

sequentially in their original order and are not eligible for parallel execution.

Fig. 7 shows the effect of these 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 $\mathbf{8}\,\mathrm{ms}$, indicating that 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 Fig. 6(c), which limits the overall benefit. In contrast, when ENV delays are fully eliminated ($100\,\%$), the gain increases to approximately $\mathbf{220}\,\mathrm{ms}$, highlighting that the effectiveness of intra-process parallelism depends heavily on environment responsiveness.

Despite this improvement, the overall gain remains modest, which can be attributed to two main factors. First, the software

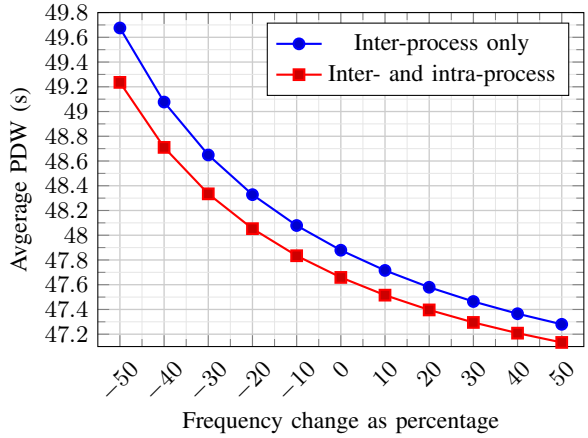

Fig. 8: Effect of frequency scaling on PDW for 8-core platform with 100% ENV delay reduction

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 V), conservative assumptions are necessary when trace information is insufficient to guarantee safe block independence. These assumptions prevent potential deadlocks but may also restrict opportunities for concurrent execution.

Finally, we evaluated 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 Fig. 7, the difference between using 8 and 16 cores is not significant and falls within the simulation error margin.

Fig. 8 shows the effect of frequency scaling on PDW under both inter-process-only and combined inter- and intra-process parallelism exploitation cases. 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, 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.

VII. Conclusions

This paper presented a refined application workload model that captures both inter-process and intra-process execution dependencies in complex distributed Cyber-Physical Systems (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 an 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 complement platform and mapping exploration in large-scale industrial dCPS.

Acknowledgment

The authors would like to thank Kostas Triantafyllidis and Ivo ter Horst from ASML for their valuable support and constructive input throughout the development of this work.

References

[1] R. Alur, *Principles of Cyber-Physical Systems*. The MIT Press, 2015.
[2] S. Acur *et al.*, "Vision and outlook for systems architecting and systems engineering in the high-tech equipment industry," TNO, Tech. Rep. R10542, 2024.
[3] B. Meier *et al.*, "Htsm systems engineering roadmap," *Tech. rep*, 2020.
[4] B. Kienhuis *et al.*, "A methodology to design programmable embedded systems: the y-chart approach," *SAMOS*, pp. 18–37, 2002.
[5] F. S. Saadatmand *et al.*, "Automated derivation of application workload models for design space exploration of industrial distributed cyber-physical systems," *ICPS*, May 2024.
[6] M. Herget *et al.*, "Design space exploration for distributed cyber-physical systems: State-of-the-art, challenges, and directions," in *DSD*, 2022, pp. 632–640.
[7] R. Marculescu *et al.*, "Cyberphysical systems: Workload modeling and design optimization," *IEEE Design Test of Computers*, vol. 28, no. 4, pp. 78–87, 2011.
[8] H.-C. An *et al.*, "A formal approach to power optimization in cpss with delay-workload dependence awareness," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 750–763, 2016.
[9] M. Modaber *et al.*, "A method for building trustworthy hybrid performance models for cyber-physical systems of systems," *IEEE Access*, 2024.
[10] E. M. Arasteh *et al.*, "Improving parallelism in system level models by assessing pdes performance," in *FDL*, 2021, pp. 01–07.
[11] R. Pinciroli *et al.*, "Modeling more software performance antipatterns in cyber-physical systems," *Software and Systems Modeling*, vol. 23, no. 4, pp. 1003–1023, 2024.
[12] S. Tuli *et al.*, "Mcds: Ai augmented workflow scheduling in mobile edge cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2794–2807, 2021.
[13] M. Desnoyers *et al.*, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS*, vol. 2006, 2006, pp. 209–224.
[14] Trace Compass Developers, "Trace Compass [Software]," https://tracecompass.org/, 2024.
[15] A. Varga *et al.*, "An overview of the omnet++ simulation environment," in *SIMUTools*, 2010.