Declarative Adaptive Optimization of Task-Based Applications on Heterogeneous Architectures

Emanuele De Angelis*, Guglielmo De Angelis*, Romolo Marotta†,
Federica Montesano*, Alessandro Pellegrini†, and Maurizio Proietti*
*IASI-CNR, Rome, Italy
†Tor Vergata University of Rome, Rome, Italy

***Abstract*—This paper presents a knowledge-based technique for mapping task-based applications onto heterogeneous computing resources using Answer Set Programming (i.e., ASP) for dynamic, multi-objective task allocation. Our method models applications through the Actor Model, considering device constraints, task workloads, and performance factors like computational overload and inter-actor communication costs. By formulating these elements as logical rules, our ASP-based method adapts allocations to changing workloads and system dynamics, nearing the theoretical optimum achievable by an oracle with complete knowledge. Simulation experiments show that our approach significantly outperforms (up to 45%) traditional static partitioning techniques by maximizing throughput and preventing unfruitful migrations. These results highlight the effectiveness of declarative optimization for online allocation in heterogeneous architectures, and suggest that a clear syntax for modelling non-functional metrics eases the extrapolation of a broad set of optimization scenarios.**

***Index Terms*—Heterogeneous Architectures, Answer Set Programming, Resource Allocation**

I. Introduction

Task-based applications decompose computational work into discrete units called *tasks*, namely independent operations that can be executed in parallel on multicore or distributed architectures. Each task is modelled as a self-contained entity, and the system dynamically manages the execution order and resource allocation by considering the dependency relations defined between the tasks. This approach is particularly effective in domains such as scientific computing, large-scale data processing, and simulation systems, where workloads can be naturally partitioned into separate operations [1]. The overall performance improves as tasks run concurrently, but ensuring that the workload is balanced across the different computational units can be challenging.

This scenario is depicted in Figure 1. Tasks in the same application can differ in execution characteristics and workload profiles, so imbalances in workload distribution can reduce overall application efficiency: if certain tasks wait excessively for resources while others complete too early, underutilization and extended execution times occur [2]. Differences in processor speed can also exacerbate this problem.

Heterogeneous architectures composed of distinct computing units, such as CPUs, GPUs, or specialized accelerators, can improve performance for task-based applications suffering from workload imbalance. Assigning computationally-intensive tasks to more powerful units and I/O-bound or lighter tasks to general-purpose resources exploits the respective strengths of the heterogeneous components [3]. This strategy can reduce idle times and improve overall efficiency, but it must be handled carefully if the workload profile of the tasks changes over time. In this case, some form of *adaptive resource allocation* is required, which is anyhow complex because it requires continuous decisions based on dynamic conditions and complex constraints imposed by task dependencies.

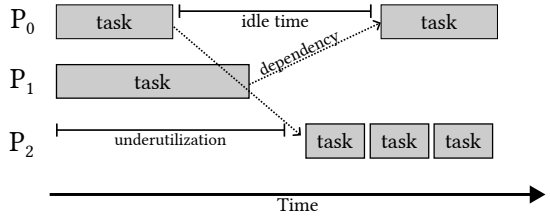


Fig. 1: Three processors execute dependent tasks. P_0 spends most of its CPU time waiting for new tasks to be generated, leading to high idle time. P_2, which is faster at processing, is underutilized even though more tasks are allocated to it. By adjusting the task-to-processor mapping, we can reduce both sources of inefficiency.

In this work, we tackle the problem of providing an adaptive optimal task allocation on heterogeneous systems at runtime, independently of the particular runtime environment used to orchestrate task execution. We consider applications where tasks are processed by stateful jobs that exchange (dependent) tasks. We explicitly consider limiting constraints in the application, such as the impossibility of running a job on a subset of the devices due to technical limitations (e.g., invoking system calls on a GPU) or due to partially missing implementations. The allocation on the heterogeneous devices can be enforced by migrating the jobs (and their state) across the devices, considering performance metrics based on the exchange rate of tasks between the jobs, the workload profile of the tasks, and their number, as it has been already recognized in the literature as a viable approach (see, e.g., [4], [5], [6]).

Task-based applications can vary widely in terms of the number of tasks they involve and their processing profiles. Similarly, the underlying hardware architecture used to run these applications can differ significantly. Our goal is to remain independent of the specific types of applications and

hardware support, so as to concentrate on optimizing the mapping of tasks to processors. For this purpose, we rely on an abstraction based on the Actor Model [7], which has recently been shown to be effective in managing task-based applications on heterogeneous architectures [8]. According to this abstraction, jobs are generically considered *actors*, and tasks are generically mapped to *messages* exchanged between actors.

A task-based application described in terms of the Actor Model is then processed by an adaptive optimization module named **A**ctor **P**lacement **O**ptimizer (APO), and based on Answer Set Programming (ASP). ASP is a declarative problem-solving methodology introduced in the domain of symbolic AI [9], [10] which supports rapid prototyping. APO evaluates sets of logical constraints to generate optimal actor-to-device assignments. Indeed, the ASP language includes optimization statements supporting the declaration of multi-objective strategies for dynamic resource management. In our context, when the workload profile changes over time, APO can adapt the task allocations to the constraints the infrastructural heterogeneity imposes, for example, by balancing execution time or reducing communication overheads.

We report an experimental study that evaluates our proposal in the context of several scenarios for dynamic task allocations. Each scenario refers to a different set of simulated traces replayed through a dedicated evaluation framework, which uses APO to identify an optimal resource allocation under several optimization objectives. We compare the results from APO against solutions obtained using METIS [11], a well-established graph partitioning and load-balancing tool. As we will show, the multiple optimization objectives proper of our allocation problem are difficult to capture with METIS, which can deal with multiple constraints only if managed explicitly using some ad-hoc weighting function. Conversely, APO can easily attack multi-objective optimization problems. Moreover, we compare the results against the theoretical optimum computed over the entire execution traces, and random allocations for a significance control.

The results for this study demonstrate that APO effectively accounts for complex dependencies and constraints, achieving improved performance without requiring the introduction of complex weighting functions, thus also serving as a workbench for the definition of multiple optimization strategies.

Overall, the main contributions of this work are:

1) the formal modeling of task-based applications via the Actor Model, which includes the encoding of device constraints, task workloads, communication costs, and inter-actor dependencies as logical rules, to drive online allocation decisions;
2) the definition and integration of three key performance metrics (i.e., computing-unit overload, inter-device communication cost, and inter-actor annoyance) within prioritized ASP optimization statements to maximize throughput;
3) the use of an ASP-based technique for dynamic, multi-objective task allocation in heterogeneous architectures, capable of near-optimal runtime adaptation to workload and system changes;
4) an extensive simulation-based evaluation demonstrating that the ASP-driven approach can closely match the theoretical optimum.

We release APO as open source software[1]. The remainder of this paper is structured as follows: Section II describes the context and the background information referred to in the work. The APO method is presented in Section III. Section IV presents our comparative experimental evaluation, and the threats to the validity of the study are reported in Section V. Related work is discussed in Section VI.

II. Context and Background

In this section, we introduce the formalism and the reference interaction model that we use to build APO. Then we also provide a motivational example of a class of applications that break down complex problems into parallel tasks. This example is used later in Section III to illustrate the proposed multi-objective strategy for actor-to-device allocation.

A. Answer Set Programming

ASP is a rule-based declarative programming language originating from the fields of knowledge representation, nonmonotonic reasoning, logic programming, and constraint solving. Also, ASP has been proven to be a well-suited tool for dealing with knowledge-intensive and combinatorial search problems [9], [12].

By following a declarative problem-solving approach, an *ASP program* specifies a set of rules that describe a computational problem in the ASP language. A model for the ASP program, also called *answer set*, is a solution that satisfies all the rules in the program. Answer sets can then be computed by using an off-the-shelf ASP solver, such as Clingo [10], without requiring to specify an *ad-hoc* algorithm to solve it.

The declarative nature of ASP, combined with the optimization statements provided by the language, makes it a convenient tool for dynamic resource management for heterogeneous architectures. To ease the reading, we will introduce the ASP language along with the presentation of the implementation of the optimization module in both Section III-A, and Section III-B.

B. The Actor Model

The Actor Model [13] is a concurrent software architectural paradigm structured around autonomous entities called actors, each interacting with the environment exclusively through message passing. This design eliminates the need for explicit thread synchronization by ensuring that each actor processes messages sequentially while maintaining an isolated internal state. Actors can independently operate across multiple processing elements, thus effectively capturing parallel or distributed execution.

The Actor Model can be used to abstract task-based applications by mapping tasks to actors, which allows for dynamic

[1] https://github.com/DomainProject/apo

task management and efficient execution across heterogeneous resources. Messages can facilitate task communication, enabling synchronization, data transfer, and dependency resolution. This allows tasks to be retried or reallocated independently without disrupting the overall system. The model's expressiveness can be used for workload distribution, ensuring sustained performance by dynamically routing messages and adapting to fluctuating resource availability, thus optimizing computational resource usage.

APO uses the Actor Model to abstract underlying complexities: it allows designers to focus only on application logic, its constraints, and on the overall hardware configuration. The runtime environment periodically queries APO in order to face evolving conditions or emerging constraints; thus, it handles dynamic allocation based on the decisions of APO.

C. Motivating Example

To simplify the discussion of the optimization methods presented in this work, we introduce a reference example centered on real-time image processing for video surveillance, as an instance of task-based applications that we tackle. In this example, a security camera continuously streams a high-resolution video at 30 frames per second, which then undergoes several processing phases to detect different classes of objects in the stream. The application is organized as in Figure 2. Each part of the application is an actor that exchanges messages. The payload of a message is the output from an actor, which is processed by the recipient actor.

We consider three different classes of tasks [14]. The first class (i.e., crop) deals with the identification of some region of interest to carry out the detection; the second class deals with actual object classification in some region; the last class performs object localization in the original frame. Each job executes tasks belonging to a single class.

When processing a task belonging to the first class, each frame (represented as a matrix of pixels) is cropped into smaller submatrices of different sizes (tiles). A multi-scale sliding window approach is used to cover the whole frame area with each submatrix.

For each identified region of interest, the detection system applies several classifiers to detect whether the submatrix depicts a target object or not; we assume that each classifier is able to recognize one class of objects. This activity is performed by the second class of tasks (i.e., classify).

The last class of tasks (i.e., localize) concerns the generation of (different) bounding boxes for each of the classified objects and their positioning on top of the original frame. A bounding box is labeled with the category the object belongs to; also, it can include other semantic information resulting from the classification task.

A heterogeneous architecture, comprising CPUs and various accelerators, can dynamically allocate tasks based on system load and frame complexity. Computationally intensive tasks, such as handling high-resolution tiles or performing complex classifications, might cause node overload conditions if not managed correctly. For instance, assigning high-complexity classifications exclusively to the very same single processor could lead to its overload, resulting in idle times and underutilization of other processing nodes, as previously illustrated in Figure 1.

Therefore, an efficient strategy involves moving around intensive computational tasks to more powerful nodes (e.g., GPUs) whenever they are idle, while simultaneously assigning smaller or less demanding tiles to CPUs. This balanced allocation reduces overload conditions on any single node, ensuring continuous and efficient utilization across the architecture, thus achieving the desired throughput of 30 frames per second.

Notably, frame-cropping and region-based object classification tasks can generally proceed in parallel as long as tasks remain available for processing. In contrast, bounding box generation inherently depends on completing these preceding tasks. Specifically, accuracy in object localization benefits from multiple processed submatrices, as overlapping sliding windows around the same object produce multiple candidate bounding boxes, from which the optimal fit is selected. Additionally, generating bounding boxes involves accessing the entire frame, and aggregating object locations efficiently reduces redundant processing of the whole frame, mitigating potential overload scenarios. This condition creates a dependency between tasks that, if not managed carefully, could exacerbate resource underutilization.

This simplified scenario serves as a practical backdrop for illustrating how ASP-driven task orchestration optimizes performance by balancing workload distribution, maximizing frame throughput, and minimizing latency in heterogeneous environments.

III. Knowledge-based Optimal Allocation

The goal of APO for dynamic resource management is to find an optimal allocation of actors to computing units, that is, an allocation that aims to enhance the computational efficiency of the application. It makes use of ASP *rules* to specify the logical constraints that the allocation must satisfy.

In the following, Section III-A introduces the actor allocations modeled in terms of ASP rules, Section III-B reports the heuristics that guide the overall allocation strategy, while Section III-C presents in detail the key concept of *annoyance*.

A. Modeling the Actor Allocations

ASP rules are implications of the form `head :- body` stating that if all the (positive or negative) *literals* in the `body` are true, then the *atom* in the `head` is true. Rules with an empty body, called *facts*, represent the knowledge available about the execution environment, such as the computing units, the types of these devices, the actors, and their workloads.

In particular, computing units and actors are encoded[2] in ASP by facts of the form `cu(U)` and `actor(A)`, where `U` and `A` are logic variables ranging over the sets of computing units available in the hardware platform and the identifiers of actors to be executed, respectively. Specifically, with respect

[2]The ASP encoding is presented using the syntax of the ASP solver Clingo [10].

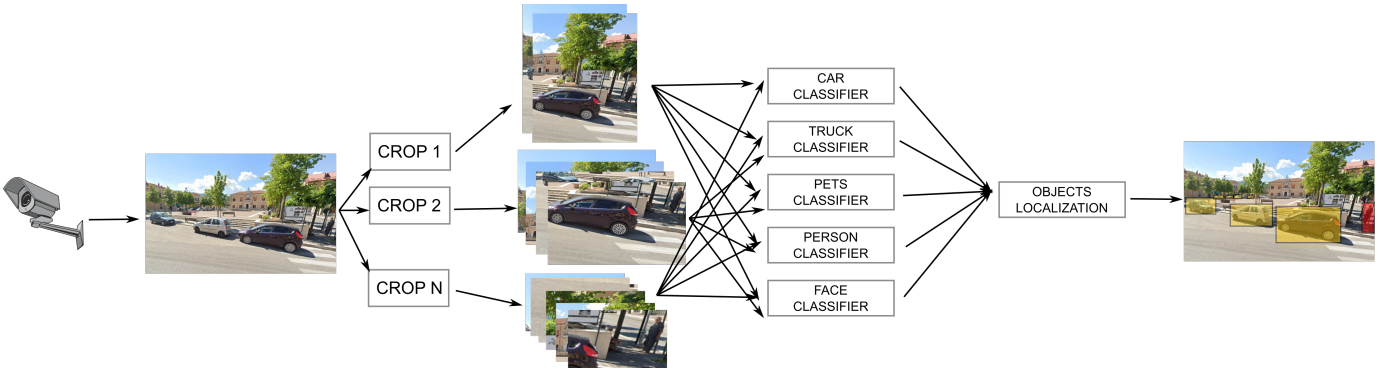


Fig. 2: Pictorial representation of the motivating example.

to the motivating example described in Section II-C: we can use the facts `cu(cpu1) ... cu(cpuN)`, and the facts `cu(gpu1) ... cu(gpuM)` to assert that overall the referred heterogeneous architecture respectively includes N CPUs, and M GPUs. In addition, facts like: `actor(crop1) ... actor(cropK)`, `actor(classify1) ... actor(classifyZ)`, and `actor(localize1)` declare that the real-time image processing is composed of K actor instances for cropping tasks, Z actor instances for classification tasks, and only an actor instance for positioning the classified objects on top of the original frame.

The allocation of an actor `A` to a computing unit `U` is encoded by an atom of the form `run_on(A,U)`. In ASP, any solution of such an atom is given by binding its (free) variables to specific values. Such values are inferred from declared facts, and they have to be consistent with possible constraints the atom specifies.

Given any actor `A` to be executed, the following ASP rule chooses a computing unit `U` on which the actor should run:

```
1 { run_on(A,U) : cu(U), runnable_on(A,U) } 1
    :- actor(A).
```

The aggregate atom in the head (i.e., all the declarations before the syntactic token ':-') specifies that, for each actor `A`, the atom `run_on(A,U)`, appearing on the left side of the colon, must hold for a pair `(A,U)` that satisfies the conjunction on the right side of the colon. The values before and after curly brackets specify the lower and upper bound, respectively, on the number of `run_on(A,U)` atoms that must hold for the given `A`. Thus, the above rule requires that each actor be assigned to exactly one computing unit. The `runnable_on(A,U)` atom maps each actor `A` to all computing units for which there exists a device-specific implementation of `A` enabling its execution on `U`. In this sense, the allocation:

```
run_on(crop1,cpu1), run_on(crop2,cpu1),
    run_on(crop3,gpu1) ...
```

is a valid model for the atom, while the following is not:

```
run_on(crop1,cpu1), run_on(crop1,cpu2),
    run_on(crop3,gpu1) ...
```

because it allows for the allocation of the same actor (i.e. `crop1`) to different computing units (i.e., `cpu1` and `cpu2`). In addition, also the allocation:

```
... run_on(gpu2,crop3) ...
```

is not admissible, as it results from a binding that inverts the roles of the arguments.

The encoding presented so far provides a practical way to allocate computing units to actors. Indeed, from any answer set computed by the ASP solver, we get a feasible allocation (that is, an allocation satisfying the specification on the types of devices on which each actor can run on) by simply extracting from an answer set all the atoms of the form `run_on(A,U)`. However, we are not interested in just computing any allocation, but in finding an optimal allocation aimed at improving the computational efficiency of the execution environment.

In order to achieve this goal, the APO makes use of additional facts to represent metrics, describing the status of the execution environment, which may affect its efficiency. A set of ASP rules is then used to make explicit the relation between an allocation of actors and these metrics. Finally, some ASP optimization statements specify heuristic criteria for computing an optimal allocation.

B. Modeling the Optimization Rules

The allocation strategy we present here focuses on enhancing the computational efficiency, specifically the system throughput, but the strategy can be easily adapted to achieve a different optimization goal, such as reducing energy consumption, by simply extending the module with suitable ASP rules.

In order to define a heuristic for optimizing the throughput, we have considered three metrics: (**M1**) the overload of computing units, (**M2**) the communication cost among actors, and (**M3**) the interference among actors, called *annoyance*, resulting from their interaction when running on different computing units.

In the following, we present: (i) the ASP rules that make explicit the relation between an allocation of actors to computing units and the metrics **M1**–**M3**, and (ii) the related ASP optimization statements looking for an allocation that aims to maximize the system throughput.

The overload O of a computing unit U (i.e. **M1**) is represented by cu_overload(U,O) and it is derived through the following ASP rule as the maximum between the excess of workload W to be processed by U with respect to its capacity C (that is, W−C) and zero (representing a computing unit loaded below its capacity or idle):

```
cu_overload(U,O) :- cu(U),
  O = #max{ 0 ; W-C : cu_workload(U,W),
   cu_capacity(U,C) }.
```

where cu_workload(U,W) represents the total workload W to be processed by U, and cu_capacity(U,C) represents the capacity C of U, which estimates the amount of messages that U process within a given time window[3]. Finally, aiming to maximize the throughput, we need to distribute the workload evenly across computing units by using the following optimization statement:

```
#minimize{ X-Y : max_overload(X),
                 min_overload(Y) }.
```

where max_overload(X) and min_overload(Y) represent the maximum overload X and minimum overload Y, respectively, computed over the set of values O such that cu_overload(U,O) holds. The above optimization statement specifies that an answer set is optimal if X−Y is minimal. The dependency relation between the atoms occurring in the optimization statement and the atoms representing the allocation of an actor to a computing unit introduced by the ASP rules (max_overload(X) and min_overload(Y) depend on cu_overload(U,O), which in turn depends on run_on(A,U) through cu_workload(U,W)), make it possible to evenly distribute actors across computing units according to their workloads.

The communication cost S between two actors A1 and A2 (i.e., **M2**) is represented by a_cc(A1,A2,S), and it is described by the following rule:

```
a_cc(A1,A2,S) :- run_on(A1,U1),
  run_on(A2,U2), U1!=U2,
  msg_exch_rate(A1,A2,R),
  msg_exch_cost(U1,U2,C), S=C*R.
```

where msg_exch_rate(A1,A2,R) represents the message exchange rate R between actors A1 and A2, and msg_exch_cost(U1,U2,C) represents the message exchange cost C between computing units U1 and U2, respectively. Then, if two actors are assigned to different computing units U1 != U2, the communication cost S between A1 and A2 is obtained by multiplying C by R. Similarly to what we have done to distribute the workload evenly, we also need to allocate actors so that the communication cost is minimal:

```
#minimize{ S,A1,A2 : a_cc(A1,A2,S) }.
```

The above statement specifies that an answer set is optimal if the sum of the communication costs S over the set of contributed tuples S,A1,A2 is minimal, thereby requiring the ASP solver to look for allocations that group together actors with higher communication costs on the same computing unit.

The annoyance C between actors A1 and A2 (i.e., **M3**) is represented by mutual_annoyance(A1,A2,C), which describes how significantly the actor distribution across computing units hampers performance. In other words, internally, annoyance is a dimensionless parameter that guides optimization; externally, it functions as a modeling control parameter for specific hindrances, ultimately penalizing configurations that unnecessarily separate tightly coupled actors. We further detail these aspects in Section III-C. In the motivating example, the annoyance represents the idle time on cropping and classifying actors, caused by the actor responsible for the object localization.

Intuitively, minimizing the annoyance reduces the bottlenecks, which contributes to increasing the system throughput. Therefore, we also ask the ASP solver to minimize the total annoyance:

```
#minimize{ C,A1,A2 :
           mutual_annoyance(A1,A2,C),
           run_on(A1,U1), run_on(A2,U2),
           U1 != U2 }.
```

The optimization statement can be prioritized, allowing the designers of the resource management module to order the criteria according to their relevance. By specifying priorities of the optimization statements, APO associates a tuple of costs to each answer set. To determine whether an answer set is optimal, APO compares cost tuples whose elements are ordered by priority. Criteria with the same priority contribute to the same component of the tuple of costs. In particular, our strategy enforces APO to minimize the difference between the maximum and minimum overload (i.e., it concerns **M1**), to minimize the communication cost (i.e., **M2**), and to minimize the annoyance (i.e., **M3**). Their relative priority is defined according to the relevance evaluated from the facts defining these metrics (that is, the facts defining the predicates msg_exch_rate, msg_exch_cost, and mutual_annoyance).

C. Modelling the Annoyance

In the proposed approach, the concept of annoyance is a unifying abstraction that converts runtime-specific drags on progress (e.g., , excessive message exchanges [16][17], frequent rollbacks [18], or synchronization delays [19]) into a single dimensionless integer fed to mutual_annoyance(A1,A2,C). This abstraction stems from the observation that, whatever task synchronization strategy a runtime adopts, each source of drag manifests as a countable class of micro-events observable in a sliding window Δt. If speculative synchronization is employed, rollbacks are produced; a conservative engine can rely on null messages that occupy the network without advancing simulation time; a fork/join execution pattern leaves processors stalled while waiting on an outstanding dependency.

Typically, runtime environments collect statistics to observe at runtime the occurrence of these classes of events, based on their internal organization: a rollback handler counts every

[3]For simplicity, we assume a periodic evaluation of the optimal allocation of actors to computing units. More advanced decisions regarding when to perform the optimization are beyond the scope of this work. Also, this problem has already been tackled in the literature [15].

restored state, the message layer tags null messages, and the scheduler exposes idle cycles. These classical statistics can be used to instantiate an annoyance value, which thus becomes a runtime-dependent optimization parameter that is used internally by APO to drive the optimization process, as we stated before.

To clarify how a runtime environment developer can instantiate this annoyance value, we provide a simple yet general example on how to model annoyance in a generic runtime environment. Denote by $\rho_{uv}(\Delta t)$ the number of rollbacks that actor u triggers in actor v, by $\eta_{uv}(\Delta t)$ the number of null messages exchanged between the pair, and by $\iota_{uv}(\Delta t)$ the sum of idle cycles spent by either actor because the other has not progressed. These three counters all measure wasted potential progress; they differ only in units. A dimensionless annoyance arises after rescaling each counter by a device-independent baseline latency λ (for instance, the mean CPU time required to validate and enqueue a useful task) and applying positive rational weights γ_ρ, γ_η, γ_ι that express the relative impact of the corresponding phenomenon on long-run throughput. We obtain:

$$\mathcal{A}_{uv}(\Delta t) = \gamma_\rho \frac{\rho_{uv}(\Delta t)}{\lambda} + \gamma_\eta \frac{\eta_{uv}(\Delta t)}{\lambda} + \gamma_\iota \frac{\iota_{uv}(\Delta t)}{\lambda}.$$

Because the numerator and denominator of every term share the same physical unit (seconds or cycles), the sum is dimensionless. Choosing λ once per platform eliminates otherwise cumbersome unit conversions when a heterogeneous node mixes CPU, GPU, and FPGA devices: the cycle counter of each device is already converted into seconds by the local clock frequency. The weights become a policy knob: in a speculative environment prone to frequent causality violations, we may set $\gamma_\rho > \gamma_\eta, \gamma_\iota$ so that the optimizer empties overloaded links once rollbacks increase; in a conservative environment running on a tightly coupled system, we can increase γ_η to reduce the explosion of forward-in-time null messages. Where underutilization hampers the processing throughput, we can raise γ_ι to penalize the idle phases.

Anyhow, these values should be correctly fed into APO. Any ASP solver (e.g., Clingo) expects integer arguments; hence, we can pick a scaling factor σ equal to the least common multiple of the denominators of γ_ρ, γ_η, and γ_ι, and multiply the real-valued annoyance by σ, then compute the nearest-integer function:

```
mutual_annoyance(U,V,N) :-
    rollbacks(U,V,R), nulls(U,V,M),
    idle(U,V,I),
    N = σ*(γρ*R/λ+ γη*M/λ+γι*I/λ).
```

All symbols on the right-hand side are integer atoms. Rollbacks, null messages, and idle cycles are counters accumulated over Δt. When APO subsequently minimizes the sum of N across conflicting actor pairs, it balances every class of runtime hindrances.

If a different runtime (with different sources of hindrances) is considered, it only needs to expose other counters and associated weights: a synchronous barrier engine could add a stalled-barrier counter; a transactional memory subsystem could add an abort counter. Provided that every new term is divided by the same λ factor and incorporated into the weighted sum (if needed), APO logic and the existing ASP encoding remain unchanged, while the designer maintains the freedom to tune weights offline through micro-benchmarks or sensitivity analysis. The annoyance metric thus becomes a portable, dimensionless integer that captures heterogeneous manifestations of inefficiency without privileging any particular synchronization strategy.

IV. Experimental Assessment

We studied the validity of APO relying on traces that describe the behavior of task-based applications. These traces record the logical time at which some actor schedules the execution of a task at a different actor. This notion of logical time captures the dependency among tasks. All tasks must be executed in logical time order, but simultaneous tasks are deemed independent and can be executed in any order, thus capturing concurrency. The considered traces have been generated randomly—both the traces and the generator are available in the online repository.

We have relied on a custom simulation framework, also available in the online repository, that abstractly represents the co-execution of these simulation traces on a heterogeneous architecture composed of CPUs (organized in different NUMA nodes), GPUs, and FPGAs. This setup allows us to explore the behavior of different optimization strategies on different heterogeneous architectures. This approach, which has already been exploited in the literature [15], enables us to plug different optimization strategies and query them using the runtime parameters observed in the simulated heterogeneous architecture to observe how different decisions affect the performance of the task-based application co-executed on the heterogeneous environment.

As mentioned, the tasks in the trace carry dependency information subsumed in their logical time. Considering that heterogeneous architectures are composed of independent hardware components, we have implemented in the simulation framework a *speculative* execution paradigm, similar to [18], in which tasks could be executed out of order, but the causality violation is fixed a posteriori by means of a rollback operation. As mentioned in Section III, this is one of the possible sources of annoyance between actors. We emphasize that this source of annoyance is related to the organization of the trace simulation environment we have used for the experimental assessment: the approach proposed in this paper is general, and its correctness and applicability do not depend on the rollback capability of the task runtime environment. Indeed, the concept of annoyance is general, explicitly accounting for multiple sources of task-processing impediment.

Using this simulation framework, we have compared the APO optimization strategy described in Section III against different approaches. The first is the *ground truth*, namely the optimal actor placement determined by an oracle with perfect foresight of the execution trace. This oracle, knowing the best

placement for each actor at every instant while also accounting for future execution dynamics, serves as a theoretical upper bound for optimization. Since our experimentation is trace-based, we can compute the ground truth by exhaustively exploring all possible actor placements during the execution.

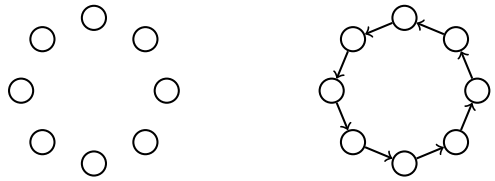
Comparing against the ground truth is relevant because short-lived fluctuations in workload might mislead optimization strategies relying solely on past observations, which could produce transient placements that, while momentarily beneficial, ultimately incur unnecessary migration costs. The oracle, by considering the full execution trace, avoids these inefficiencies by identifying cases where migration would be short-lived and, therefore, suboptimal in the long run.

The second comparison approach uses METIS, a framework for graph partitioning. It creates partitions of an indirect weighted graph to minimize the weights of edges that cross different partitions (the edge-cut). A primary challenge in graph-based partitioning for parallel computing is accurately modeling the optimization problem [20], which often leads METIS to provide only approximate solutions. However, METIS can adapt to heterogeneous architectures by allowing the specification of partition weights, referred to as *metis-std* in the results, which improves its adaptability to different computational resources.

To avoid biasing our evaluation in favor of APO, we have constructed an enhanced configuration of METIS intended to serve as a stronger baseline. This setup, referred to as *metis-apolike* in the results, is not meant as a proposal to improve METIS itself—there are several proposals in the literature in this direction, which we discuss in Section VI. The setup metis-apolike is intended as a benchmark that captures, as faithfully as possible, the optimization objectives pursued by APO. To this end, we translated the rules and facts introduced in Section III into graph-based formulations compatible with METIS. The metrics related to annoyance and overload (i.e., **M1** and **M3** in Section III-B) are modeled through graphs whose vertices represent actors, with edge weights encoding either mutual annoyance or message exchange rates, and vertex weights corresponding to the expected task volume within a fixed time window. In contrast, for communication cost (i.e., **M2**), we construct a graph where each vertex corresponds to a pair consisting of an actor and a candidate computing unit. Here, edges model the cost of communication between actor-device pairs, with weights proportional to the product of message exchange rate and inter-device communication cost. Vertex weights in this case reflect the expected workload and device processing capacity.

To mimic the multi-optimization approach used by APO, we run METIS in sequence on the three graphs. Before each run, we pre-process the current input graph. Specifically, we leverage a set of scale factor functions that update the graph's weights in order to enforce connectivity among the vertices that have been assigned to the same partition during the previous optimization step.

The final approach used for comparison, referred to as *random*, assigns actors to random devices each time the optimizer is invoked. This baseline serves as a control mechanism, ensuring that the solutions produced by the other strategies are meaningfully competitive, providing tangible improvements beyond chance.



(a) Topology A (b) Topology B

Fig. 3: Actor topologies used in the preliminary study.

Topology	random	metis-std	metis-apolike	APO
A	0.82	1.00	1.00	1.00
B	0.60	0.94	0.94	0.95

TABLE I: Performance ratio over the ground truth.

A. Validation on Synthetic Topologies

In this first assessment, we studied a couple of small-scale degenerate configurations, which work as a stress test for our optimization approach. These configurations have 8 actors and 4 computing units; all actors are uniform in terms of both the number of tasks to be processed and the cost for an individual task, independent of the computing unit they are running on. Actors are organized in two topologies that have uncommon message-exchange patterns for task-based applications and do not actually require any dynamic optimization. The topologies are depicted in Figure 3. Topology A is very simple, with no actor exchanging tasks with other actors. Topology B is a ring with actors exchanging tasks only with one neighbour. We have simulated this application for 10^6 simulated time units, with each actor initialized with 1 pending task.

Table I summarizes the results of this experiment, presenting the performance of each strategy as a fraction relative to the ground truth. The data clearly demonstrate that all examined optimization strategies significantly outperform a random allocation baseline. Furthermore, both variants of METIS as well as APO successfully determine optimal allocations in the trivial scenario represented by the disconnected topology A. This outcome highlights the effectiveness and robustness of the proposed optimization strategy, illustrating its ability to consistently derive optimal or near-optimal solutions even in scenarios where explicit optimization might seem unnecessary. This observation is confirmed for the more challenging topology B, where APO achieves the highest performance (although with a minimal gap). Interestingly, the more sophisticated metis-apolike configuration does not yield any measurable improvement compared to the conventional metis-std approach, suggesting that increased complexity does not necessarily translate into enhanced effectiveness in this context.

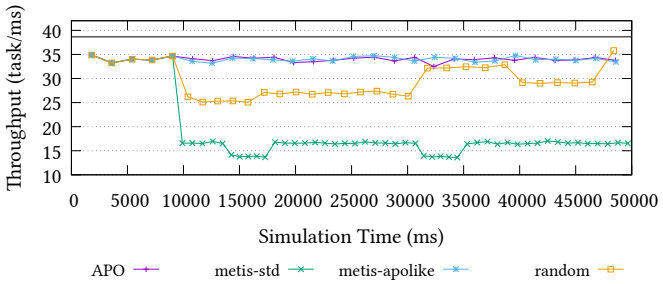


Fig. 4: Throughput of the Surveillance Example.

B. Motivating Example Trace

In a second assessment, we studied the behavior of APO when dealing with a scenario similar to the motivating example presented in Section II-C, when using 8 actors. Among them, one performs the final object localization (i.e., localize), 5 execute the classifications illustrated in Figure 2 (i.e., classify), and 2 perform cropping activities (i.e., crop). In this experiment, we have used the same configuration as in Section IV-A.

The results of this experiment are shown in Figure 4, where the black horizontal line represents the ground truth. As can be seen, after an initial transient period in which no differences are observed since none of the optimization approaches has yet made any choices, we can see that the ASP-based approach and its metis-apolike counterpart show very similar performance results. This result confirms what we observed in the previous validation example: an approach that considers the nature of the tasks and the runtime interdependencies between them is able to come very close to the theoretical optimum. Moreover, we again emphasize that APO can deliver a performance competitive compared to metis-apolike without the complexity required to implement weighting functions in METIS.

The poor performance of metis-std in the surveillance example stems from its inherently static optimization model, which fails to accommodate the dynamic and phase-shifting nature of task-based workloads. Designed to minimize edge-cut based on a fixed communication graph, metis-std locks in actor-to-device placements that do not adapt as execution patterns evolve—such as the alternation between uniform task distribution and hotspot formation observed in the video processing pipeline. This rigidity leads to excessive inter-device communication, contention on critical paths, and increased synchronization overhead during phases that were not well represented in the original graph abstraction. Moreover, metis-std lacks awareness of heterogeneous constraints, such as actor-specific affinities or resource asymmetries, and cannot reason about time-sensitive dependencies like speculative rollbacks or bottleneck-prone actors. In contrast, the random baseline, despite its simplicity, occasionally benefits from favorable actor co-location by chance, avoiding systematic placement errors, and achieving better throughput under dynamic execution conditions.

TABLE II: Parameters used to configure the simulation.

	CPU	**GPU**	**FPGA**
Task duration (ms)	0.1	0.002	0.01
Causality restoration (ms)	0.01	0.01	0.01
Task Exchange Cost Factors	**CPU**	**GPU**	**FPGA**
CPU	1	5	10
GPU	5	1	20
FPGA	10	20	1

C. Stochastic Traces

In this last assessment, we simulated a stochastic trace on top of a small-scale heterogeneous node composed of an octa-core CPU, two GPUs, and an FPGA. To run our simulations, we have configured the simulator using the parameters reported in Table II, which are representative of off-the-shelf heterogeneous architectures.

The *task processing factor* normalizes device throughput to a CPU baseline of 1: a midrange GPU can sustain ~ 50 tasks/s on parallel kernels, whereas an optimized FPGA handles ~ 10 tasks/s yet often outperforms GPUs on pipeline-oriented algorithms [21], [22], [23]. We set the CPU task time to $100\ \mu$s to reflect moderately complex workloads involving memory accesses, arithmetic operations, and modest system overhead, in line with microbenchmark reports of kernel launches, small matrix multiplications, and memory-copy routines [24], [25]. Rollbacks for causality violations (via checkpointing or transactional memory) revert a confined state in $\sim 10\ \mu$s through in-cache manipulations or journaling buffers, consistent with observed latencies in HPC and concurrency scenarios [26], [27], [28]. Inter-device communication costs take the CPU-to-CPU baseline (i.e. $1\times$) as reference: GPU offloads over PCIe incur $\approx 5\times$ overhead, CPU–FPGA exchanges $\approx 10\times$ (including FPGA reconfiguration), and GPU–FPGA transfers can reach $\approx 20\times$ due to host-mediated staging in the absence of coherence [22], [23], [29], while same-device transfers cost $0.5\times$ the baseline.

We emphasize that all these values are *configuration parameters* to APO. If different heterogeneous architectures are considered, e.g., based on coherent CPU-GPU memory such as NVIDIA Grace Hopper [30] or AMD Instinct MI300A [31], our approach can find optimal task placement by simply adjusting the configuration parameters.

The considered trace comprises 64 actors interacting through two recurring patterns over time. During the first phase, actors exchange tasks with uniformly random destinations. In the second phase, tasks exhibit a hotspot distribution, with only 10% of the actors serving as destinations for newly generated tasks. These two phases alternate repeatedly, with each phase lasting 5,000 simulated time units, constituting a total trace duration of 60,000 simulated time units. The trace has been replayed multiple times, querying the different optimization strategies to determine which one is capable of sustaining the best throughput (measured in terms of committed tasks/millisecond) in spite of the workload variations.

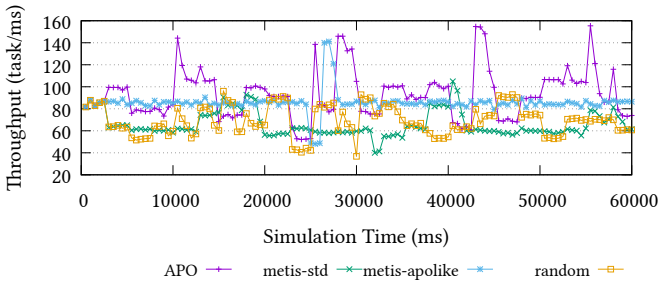


Fig. 5: Throughput of the Stochastic Trace.

For this experiment, we do not report the value of the ground truth because extensively exploring all the possible configurations is unfeasible.

We show the results in Figure 5. Unlike the previous experiment, metis-std is unable to surpass the performance of a random assignment. This result clearly indicates that traditional single-objective partitioning methods fall short of effectively handling the intricate interaction patterns proper of task-based applications. Conversely, both APO and the enhanced metis-apolike variant demonstrate robust performance, achieving considerably higher throughput. In some phases of the execution, APO can outperform metis-apolike by 45%. This result highlights that optimization approaches explicitly accounting for complex interaction dynamics are necessary to effectively manage task-based workloads characterized by sophisticated patterns of communication and dependency.

An interesting observation is that the throughput provided by metis-apolike exhibits less variability over time compared to APO. This difference likely arises from APO 's exclusive reliance on integer arithmetic, causing greater sensitivity to minor fluctuations in task assignment. Nevertheless, ASP generally achieves higher throughput compared to metis-apolike. Additionally, as previously discussed, the declarative nature of APO simplifies prototyping by providing clearer syntax compared to the manually defined weighting functions necessary in metis-apolike. This property may facilitate adapting APO to a broader set of optimization scenarios, including those involving additional non-functional metrics such as energy consumption, a direction planned for future research.

V. Threats to Validity

In the following, we briefly report the threats that may have affected the validity of the results presented in this work.

The first set of threats concerns the constructs we discussed. We found that translating the concepts from the APO module into METIS is not straightforward. METIS uses a graph-based representation for optimization scenarios and applies graph partitioning algorithms. For example, in APO, we explicitly model the overload of a computing unit and optimize by minimizing the difference between maximum and minimum overloads across allocations. In METIS, representing this is complex; we use the rate of task forecast as weights for the graph's vertices. METIS computes the edge-cut to estimate communication costs while load-balancing the sum of the vertices' weights; therefore, it fails to capture the fine-grained characteristics modeled by APO. Similar difficulties arise with multi-objective optimization. APO enables chaining different goals, limiting the search space to sub-optimal solutions of prior goals. In contrast, METIS requires separate graph representations for each optimization problem, necessitating the transformation of the graph based on earlier outcomes. These examples illustrate that while APO and METIS optimization functions are similar, they are not identical, which may affect results. Additionally, modelling multi-objective problems in METIS is inherently challenging, even when a clear graph-based representation exists.

Another class of threats involves the internal aspects of the experiments that could have influenced the observed outcomes. The simulation framework outlined in Section IV manages a defined set of configurable parameters; however, these may not fully capture the complexity of all real-world scenarios. We have tested the simulator's implementation in isolation for correctness. The first study in Section IV-A covers a selection of interesting topologies, but this is not exhaustive. Each topology has the same number of nodes (i.e., actors), corresponding to a multiple of the maximum number of available computing units. In conclusion, while we have validated the simulator carefully, we cannot guarantee it is completely free from bugs or biases that may have impacted the results.

Our final consideration relates to the generalizability of the observed outcomes. We acknowledge that the traces we generated randomly in the experiments reported in Section IV are synthetic and do not reflect all the possible characteristics of real-world applications. Although we are confident that similar results could be achieved in other contexts, we cannot yet provide solid arguments to support the general validity of our findings.

VI. Related Work

Several proposals in the literature focus on dynamically orchestrating task-based applications across heterogeneous platforms. Notably, the works in [32], [33] utilize domain-specific knowledge for task assignment optimization. In [33], the authors introduce an offline toolchain for preconfiguring applications alongside a runtime orchestrator that manages computation and data movement, specifically for multiphysics and multicomponent scientific applications. In contrast, our work targets generic task-based applications using the actor model, without explicitly optimizing data transfer between heterogeneous devices, highlighting the distinct nature of our contributions.

Conversely, Delite [32] is a framework aimed at simplifying the creation of performance-oriented DSLs for heterogeneous computing environments. Delite provides reusable components, such as parallel patterns, optimizations, and code generators, enabling DSLs to efficiently target CPUs, GPUs, and other accelerators In our work, we do not consider the generation of code for the different heterogeneous devices but focus only on dynamic allocation. Our proposal could

be therefore exploited in conjunction with code generation capabilities of Delite.

The work in [34] proposes a framework based on machine learning techniques to enhance programmability, flexibility, and efficiency in heterogeneous architectures comprising CPUs, GPUs, and hardware accelerators. At runtime, a distributed reinforcement learning-based scheduler dynamically maps tasks onto processing elements, adapting execution policies based on Q-learning to optimize performance. We do not exploit machine learning techniques that could require extensive exploration phases at runtime to effectively optimize performance. Rather, similarly to [35], we monitor runtime dynamics and use this information within a reasoning system. In this work, such a reasoning system finds an optimal actor placement considering also explicit constraints.

Several works focus on optimizing task execution in heterogeneous architectures by dynamically distributing workloads. In [36], the authors introduce a runtime system that schedules compute kernels across CPUs, GPUs, and FPGAs using performance profiling and adaptive migration to ensure near-optimal load balancing without manual intervention. Similarly, [37] presents a data-parallel execution model that automatically distributes workloads across diverse resources through runtime profiling and asynchronous scheduling. In contrast, our proposal emphasizes logical reasoning over system constraints, specifically targeting task-based applications with complex dependencies, which are essential for achieving high performance in heterogeneous architectures.

Our proposal also shares some goals with proposals related to load balancing in various task-based scenarios (see, e.g., [38], [39], [40]). Indeed, these proposals address the execution optimization and resource management of complex applications, accounting for workload variability and heterogeneity. Our approach is more versatile, as ASP-based optimization can be leveraged to determine an optimal placement reasoning on constraints that do not necessarily relate to performance metrics only.

Our proposal directly relates to other works that exploit ASP to carry out optimizations. In [41], the authors propose an approach to system synthesis based on ASP modulo theories (ASPmT) that improves symbolic synthesis for heterogeneous embedded systems. By integrating Quantifier-Free Integer Difference Logic directly into ASP solving, their method enables early pruning of infeasible solutions through partial assignment evaluation, improving scalability. The work in [42] introduces a design-space exploration framework for heterogeneous multiprocessor systems-on-chip, combining profiling, simulation, and constraint-solving techniques to optimize hardware architectures. This proposal extracts task precedence, communication costs, and computational patterns from applications, using Integer Linear Programming (ILP) and ASP to generate optimal processor mappings and hardware accelerator configurations. To the best of our knowledge, we are exploiting ASP for the first time to provide dynamic optimization of the allocation of task-based applications on heterogeneous architectures.

A notable effort toward extending graph partitioning to multi-objective settings is presented in [43], In this work, the authors introduce a formulation and algorithm that enhance METIS with support for preference-driven tradeoffs among multiple edge-cut objectives. Their method yields partitionings that are tunable, predictable, and capable of balancing both similar and dissimilar metrics across objectives. It is important to emphasize that our work does not aim to solve the graph partitioning problem per se, nor to compete with or improve upon the state-of-the-art in that domain. Our use of METIS—specifically the construction of the metis-apolike baseline—is not proposed as a contribution, but solely as a stronger benchmark to avoid trivial comparisons against simplistic partitioning strategies. At the same time, unlike the multi-objective formulation in [43], which assumes a standard homogeneous graph representation with scalar or vector weights on edges and possibly vertices, our allocation constraints are inherently heterogeneous and not expressible within that model. In particular, some constraints in our system involve relationships between actors (which correspond to graph vertices) and computing units (which, in graph terms, would correspond to partitions themselves). These mixed-dimensional constraints cannot be naturally encoded in METIS's graph abstraction, which assumes that all optimization criteria can be localized on the graph structure itself. Consequently, our advanced METIS baseline must be understood as an approximation—constructed solely to enable a non-trivial empirical comparison with our ASP-based optimization, which, by design, can natively represent and reason about such heterogeneous constraints.

VII. Conclusions and Future Work

In this paper, we present a dynamic optimization technique to determine the optimal placement of task-based applications on heterogeneous architectures. The proposed ASP-based multi-objective strategy for actor-to-device allocation is able to produce allocations that capture the workload dynamics of such applications, also when the workload varies during the execution. Indeed, APO is able to outperform, up to 45%, allocations determined using other state-of-the-art tools and methods, such as those based on graph partitioning.

In the future, we plan to rely on the fast prototyping capabilities of ASP to optimize the allocations and also to improve other non-functional metrics, such as energy efficiency, or to maximize the performance of the application under a power cap. We also plan to apply our method to applications whose actors cannot run uniformly on every available device.

Acknowledgments

This paper has been partially supported by the European Union—Next Generation EU, Mission 4, Component 2, CUP E53D23008200006 (Domain), and partially by the Spoke 1 "FutureHPC & BigData" funded by the European Union—Next Generation EU, Mission 4, Component 2.

REFERENCES

[1] S. Jha, M. Cole, D. S. Katz, M. Parashar, O. Rana, and J. Weissman, "Distributed computing practice for large-scale science and engineering applications," *Concurrency and computation: practice & experience*, vol. 25, no. 11, pp. 1559–1585, Aug. 2013.

[2] D. Bohme, F. Wolf, and M. Geimer, "Characterizing load and communication imbalance in large-scale parallel applications," in *Proc. of the Int. Parallel and Distributed Processing Symposium Workshops*. Piscataway, NJ, USA: IEEE, May 2012, pp. 2538–2541.

[3] H. Sutter, "Welcome to the jungle: Or, a heterogeneous supercomputer in every pocket," Sutter's Mill: Herb Sutter on software development, Tech. Rep., 2014.

[4] S. Peluso, D. Didona, and F. Quaglia, "Application transparent migration of simulation objects with generic memory layout," in *Proc. of the Workshop on Principles of Advanced and Distributed Simulation*. Piscataway, NJ, USA: IEEE, Jun. 2011, pp. 169–177.

[5] S. Pickartz, R. Gad, S. Lankes, L. Nagel, T. Süß, A. Brinkmann, and S. Krempel, "Migration techniques in HPC environments," in *Lecture Notes in Computer Science*, ser. Lecture notes in computer science. Cham: Springer International Publishing, 2014, pp. 486–497.

[6] M. Rodríguez-Pascual, J. Cao, J. A. Moríñigo, G. Cooperman, and R. Mayo-García, "Job migration in HPC clusters by means of checkpoint/restart," *The journal of supercomputing*, vol. 75, no. 10, pp. 6517–6541, Oct. 2019.

[7] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," *International Joint Conference on Artificial Intelligence*, pp. 235–245, Aug. 1973.

[8] S. Bauco, G. De Angelis, R. Marotta, and A. Pellegrini, "A model-driven platform for software applications on heterogeneous computing environments," in *Proc. of the Int. Conference on Software Architecture - Companion*, ser. ICSA'25. Piscataway, NJ, USA: IEEE, Mar. 2025.

[9] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *Commun. ACM*, vol. 54, no. 12, p. 92–103, Dec. 2011.

[10] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-shot asp solving with clingo," *Theory and Practice of Logic Programming*, vol. 19, no. 1, p. 27–82, 2019.

[11] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Proceedings of the 1998 IEEE/ACM High Performance Networking and Computing Conference*, ser. SC'98. Piscataway, NJ, USA: IEEE, 1998.

[12] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *Proc. of Int. Logic Programming Conference and Symposium*. Cambridge, MA, USA: MIT Press, 1988, pp. 1070–1080.

[13] G. Agha, "An overview of actor languages," *SIGPLAN notices*, vol. 21, no. 10, pp. 58–67, Oct. 1986.

[14] R. Kaur and S. Singh, "A comprehensive review of object detection with deep learning," *Digital Signal Processing*, vol. 132, p. 103812, 2023.

[15] P. Andelfinger, A. Pellegrini, and R. Marotta, "Sampling policies for near-optimal device choice in parallel simulations on CPU/GPU platforms," in *Proc. of the Int. Symposium on Distributed Simulation and Real Time Applications*. USA: IEEE, Oct. 2024, pp. 101–109.

[16] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, 1985.

[17] K. M. Chandy and R. Sherman, "Space-time and simulation," *Proc. of the SCS Multiconference on Distributed Simulation*, pp. 53–57, 1989.

[18] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[19] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: portable shared memory parallel programming*, ser. Scientific and Engineering Computation. Cambridge, MA, USA: MIT Press, 2007.

[20] B. Hendrickson, "Graph partitioning and parallel solvers: Has the emperor no clothes?" in *Solving Irregularly Structured Problems in Parallel*, A. Ferreira and et al., Eds. Springer, 1998, pp. 218–225.

[21] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *Proc. of the Int. Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'16. Piscataway, NJ, USA: IEEE, Nov. 2016, pp. 409–420.

[22] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, Dec. 2012.

[23] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *Proc. of the Int. Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: ACM, Feb. 2017.

[24] J. D. McCalpin, "STREAM: Sustainable memory bandwidth in high performance computers," 2006.

[25] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2024.

[26] R. M. Fujimoto, "Performance of time warp under synthetic workloads," in *Distributed Simulation*, ser. PADS'90, D. Nicol, Ed. San Diego, CA, USA: Society for Computer Simulation International, 1990, pp. 23–28.

[27] M. P. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, pp. 289–300, 1993.

[28] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proc. of the Int. Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 316–327.

[29] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions," *ACM computing surveys*, vol. 51, no. 3, pp. 1–39, May 2019.

[30] A. C. Elster and T. A. Haugdahl, "Nvidia hopper GPU and grace CPU highlights," *Computing in science & engineering*, vol. 24, no. 2, pp. 95–100, Mar. 2022.

[31] A. Smith, E. Chapman, C. Patel, R. Swaminathan, J. Wuu, T. Huang, W. Jung, A. Kaganov, H. McIntyre, and R. Mangaser, "AMD InstinctTM MI300 series modular chiplet package – HPC and AI accelerator for exa-class systems," in *Proc. of the Int. Solid-State Circuits Conference*, vol. 67. IEEE, Feb. 2024, pp. 490–492.

[32] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4s, pp. 1–25, Apr. 2014.

[33] J. O'Neal, M. Wahib, A. Dubey, K. Weide, T. Klosterman, and J. Rudi, "Domain-specific runtime to orchestrate computation on heterogeneous platforms," in *Euro-Par 2021: Parallel Processing Workshops*, ser. Lecture notes in computer science. Cham: Springer International Publishing, 2022, pp. 154–165.

[34] Y. Xiao, S. Nazarian, and P. Bogdan, "Self-optimizing and self-programming computing systems: A combined compiler, complex networks, and machine learning approach," *IEEE Transactions on VLSI systems*, vol. 27, no. 6, pp. 1416–1427, Jun. 2019.

[35] C. Bartolini, A. Bertolino, G. De Angelis, A. Ciancone, and R. Mirandola, "Apprehensive qos monitoring of service choreographies," in *Proc. of SAC*, Shin and Maldonado, Eds. ACM, 2013, pp. 1893–1899.

[36] G. F. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *Proc. of the Int. Symposium on High performance distributed computing*. NY, USA: ACM, Jun. 2008, pp. 197–200.

[37] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: A compiler and runtime for heterogeneous systems," *SOSP '13*, pp. 49–68, 2013.

[38] R. Vitali, A. Pellegrini, and F. Quaglia, "Load sharing for optimistic parallel simulations on multi core machines," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, pp. 2–11, Dec. 2012.

[39] M. P. Robson, R. Buch, and L. V. Kale, "Runtime coordinated heterogeneous tasks in charm++," in *Proceedings of the 2nd International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM'16. Piscataway, NJ, USA: IEEE, Nov. 2016, pp. 40–43.

[40] A. Semmoud, M. Hakem, B. Benmammar, and J. Charr, "Load balancing in cloud computing environments based on adaptive starvation threshold," *Concurrency and computation: practice & experience*, vol. 32, 2020.

[41] K. Neubauer, P. Wanko, T. Schaub, and C. Haubelt, "Enhancing symbolic system synthesis through ASPmT with partial assignment evaluation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, Mar. 2017, pp. 306–309.

[42] C. Bobda, H. Ishebabi, P. Mahr, J. M. Mbongue, and S. K. Saha, "MeXT: A flow for multiprocessor exploration," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2019, pp. 1–7.

[43] K. Schloegel, G. Karypis, and V. Kumar, "A new algorithm for multi-objective graph partitioning," in *Euro-Par'99 Parallel Processing*, ser. LNCS, A. Patrick and et al., Eds. Berlin, Heidelberg: Springer, 1999, vol. 1685, pp. 322–331.