A Deployment Framework for Quality-Sensitive Applications in Resource-Constrained Dynamic Environments

A Deployment Framework for Quality-Sensitive Applications in Resource-Constrained Dynamic Environments

Shayan Tabatabaei Nikkhah, Marc Geilen, Dip Goswami, Martijn Koedam, Andrew Nelson, and Kees Goossens
Eindhoven University of Technology, the Netherlands
Email: {s.tabatabaei.nikkhah, m.c.w.geilen, d.goswami, m.l.p.j.koedam, a.t.nelson, k.g.w.goossens}@tue.nl

***Abstract*—Traditional embedded systems and recent platforms used in emerging computing paradigms (e.g., fog computing) have resource limits and require their applications and services to be dynamically added (i.e., deployed) and removed at run-time. These applications often have non-functional (quality) requirements (e.g., end-to-end latency) which are only satisfied when sufficient resources are allocated to them. Hence, a run-time decision-maker is needed to optimize the deployments, in terms of resource budgets that are allocated to applications. Additionally, computing platforms have become heterogeneous in terms of their resources and the applications they execute. However, the existing deployment solutions are limited to specific resources and services. In this paper, we propose a run-time deployment framework that is more flexible in defining constraints and optimization goals and works with more heterogeneous resources and resource models than existing solutions. The framework is implemented on an embedded platform as a proof of concept.**

I. Introduction

In principle, every single computing platform that exists has a limited number of resources—even large server farms in the cloud. However, in practice, a computing platform is considered resource-constrained when it cannot accommodate applications and services it executes at their peak resource usage (e.g., embedded systems, Internet of Things (IoT) devices, fog nodes). These platforms often span a range of heterogeneous resources governed by various resource allocation and arbitration schemes, which complicates their integration in one system. The complexity arises from the fact that first, resources are abstracted in different ways based on their types and arbiters, which complicates model-based operations such as resource optimizations [1]. Second, resource interfaces are varied, which makes their coordination and integration in one system intricate.

The applications these platforms execute often have non-functional requirements besides their functional ones. These non-functional requirements—called *qualities* in this paper—describe how well an application is required to run [2] (e.g., frame rate of processed videos, actuation latency, accuracy of object detection). Application qualities strongly depend on the type and quantity of resources that are allocated to applications [3]. Generally, the larger the *resource budget* an application gets, the higher its quality levels are. Finding the relation between quality levels and resource budgets is not always trivial; however, profiling tools (e.g., [4]) can be used to empirically or analytically extract models and/or discrete application profiles. To ensure desired quality levels, the resource requirements must be compared to provisioned resource budgets—called *budget matching* in this paper. In a heterogeneous environment, both provisioned and required budgets are expressed at various *abstractions levels* [5], each suitable for certain needs (e.g. real-time worst-case, average guarantees, best-effort budgets). More concrete and detailed budgets provide finer guarantees on application behavior but it is harder (i.e., less likely) to find a match for them on the platform. Since applications – IoT services for instance – are not similar in their required guarantees and specificity, support of various budget abstractions is necessary for today's systems.

Given the light-weight nature of embedded systems, as an example of resource-constrained platforms, only a subset of applications can execute on the system at a time and when tasks are no longer needed, they are removed. This *dynamic* presence of tasks is also seen in other contexts such as fog computing and IoT where services are dynamically *deployed* on fog nodes and edge devices [6], [7]. Application deployment includes the entire process required to prepare a software application to run and operate in a specific environment. Dynamic deployments necessitate a run-time *decision-making engine* to plan deployments, in terms of resources on which applications are mapped and budgets that are allocated to them, in such a way that application quality requirements are met and deployment costs are minimized. Adhoc optimization algorithms are suitable for certain settings and possibly require major modifications when new types of applications and resources are added to the environment. Given the high rate of diversification in today's systems, more generic optimization algorithms with high customizability are desired. Additionally, the deployment process requires interacting with heterogeneous resources whose interfaces are often varied. Having a unified interface improves the extensibility of platforms with new resources.

Emerging computing paradigms, such as IoT and fog computing, have led to many studies on application and service deployments. However, most are designed and developed for certain infrastructures and application domains. Since trends point towards heterogeneity of applications and resources, increasing the flexibility, extensibility, and genericity of deployment frameworks is vital. To address such growing needs, we contribute the following developments in this paper:

- A new orchestration and management framework that is built based on a generic component model (which widens the types of applications and resources it handles), runs on top of resource-constrained platforms, and *automates* the deployment of quality-sensitive applications at run-time (Section III).
- A pipelined, distributed deployment flow (which improves the system responsiveness) leveraging a simple unified interface to interact with the infrastructure and perform resource management—whereby new resources can be added to the system by implementing a few functions (Section V).
- A generic Pareto algebraic decision-making engine, to plan deployments, whose optimization goals and constraints are easily *extensible* and *customizable*—which alleviates the effort needed to adapt the system to a new scenario (Section VI).
- Support of various existing resource models, the ability to easily *extend* the supported resources and abstractions (by only implementing three operators), considering hierarchical budgets, and allowing applications and resources to be abstracted differently enabling a more flexible *budget matching* (Section VI).
- A portable implementation and evaluation of the framework on an embedded platform and a PC (Sections IV and VII).

The rest of the paper is organized as follows. Section II discusses the related work. An overview of the proposed framework is presented in Section III. The case-study that is used to evaluate the framework is introduced in Section IV. The steps taken to deploy applications are outlined in Section V, and the brokering step is elaborated in Section VI. The framework is evaluated in Section VII and, Section VIII concludes the paper.

II. Related Works

While container-based deployment is a promising and proliferating technique [8], a study on the applicability of containers to real-time domains has shown that they are not mature enough for this purpose yet [9], which necessitates the use of other middlewares, such as the one introduced in [10]. This makes the existing container deployment frameworks inadequate for quality-sensitive applications with stringent requirements. A deployment framework for cloud distributed applications is proposed in [11] where abstractions for hardware/software resources and middlewares are employed for both development and deployment of cloud services. The framework addresses the heterogeneity of middlewares by using a unified API to perform the common middleware functionalities such as creation and termination of Virtual Machines (VMs). However, VM hardware is abstracted with only three types (based on the type and number of resources) which are too abstract for some settings such as real-time systems because they cannot capture different arbiters, memory allocation schemes, I/O bandwidth, etc.

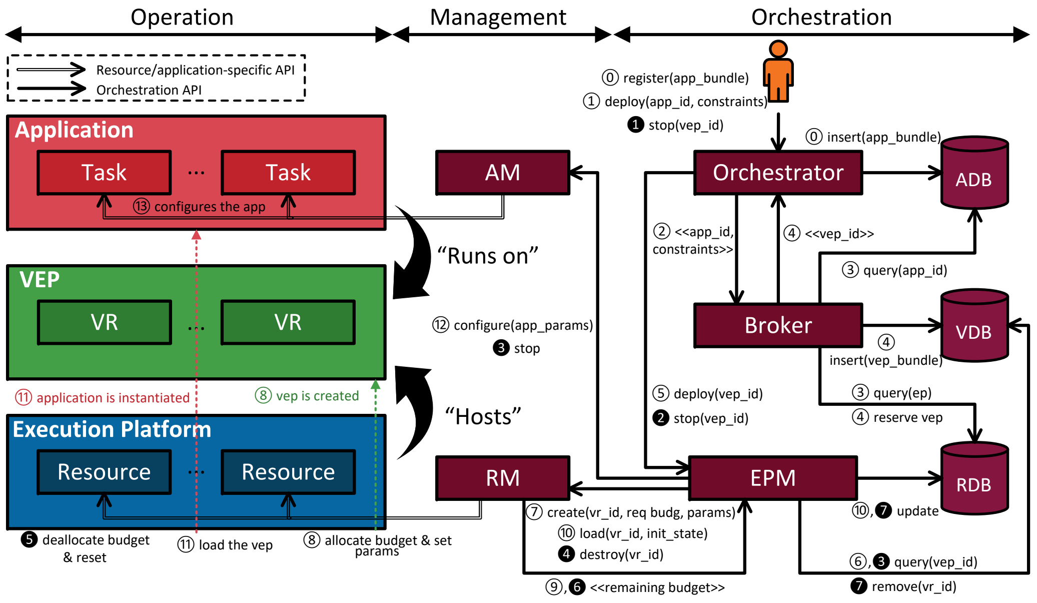
A software architecture for dynamic loading of time-critical applications on multiprocessor platforms is proposed in [10]. While the proposed solution supports a few abstractions for the considered resources, the budgets are specific for the considered platform and resources. Additionally, applications and resources have single configurations and the architecture lacks a run-time optimization engine. The middleware proposed in [12] is built on top of a general-purpose component model and performs dynamic QoS-aware deployment and reconfiguration of multi-mode, periodic applications in a distributed infrastructure. The composition algorithm used to optimize resource allocations performs a schedulability test which is based on Response Time Analysis. However, the analysis only considers the processing budget and it is based on static application priorities and their WCET, which limits the type of applications and resources the middleware can manage.

A resource orchestration framework for a sensor-rich mobile computing platform with scarce resources is proposed in [13]. The framework takes high-level quality requirements and selects the best combination of resources to execute applications. Although the framework deals with dynamic requests in a dynamic environment with scarce resources, it is static in terms of tasks it executes. Also, the employed budget models are relatively simple, making the framework suitable only for applications with soft requirements.

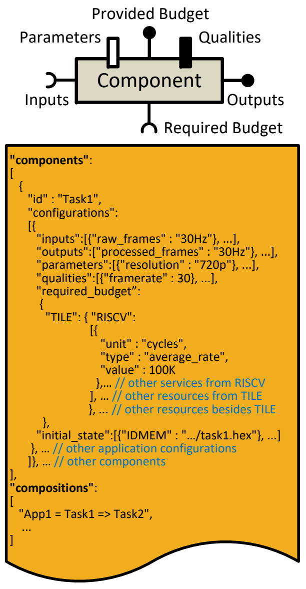
The container deployment algorithm proposed in [14] targets heterogeneous clusters and formulates the deployment problem as a vector bin packing problem where the multi-resource guarantee is the primary requirement. Although the proposed approach does not put any constraints on the types of resources, there is no discussion on budget models and the impacts of abstractions on deployments.

An open-source framework – called TORCH – for the deployment and orchestration of containerized cloud applications is proposed in [15]. Applications are described using the TOSCA standard models. A unified deployment API is exposed to the users and proprietary cloud APIs are hidden from them. Software connectors can be developed to connect any cloud provider with possibly different container-based technologies to the framework. While the framework advocates the need for genericity and flexibility by targeting any containerized application and any container-based platform, it is not suitable for deploying quality-sensitive applications in embedded or fog/edge scenarios where other types of middlewares, deployment technologies, and more detailed budget models are needed.

DIANE [6] is a framework to deploy IoT cloud applications on a cloud infrastructure and IoT gateways. DIANE takes application descriptions and artifacts from users and, upon deployment requests, instantiates them such that deployment constraints – including hardware and software constraints – comply with the mapped infrastructure. While the framework is generic in applications and infrastructure, it lacks an optimization engine, and it only filters out the resources that are not matched with the deployment constraints. Also, it is not discussed how the constraints are checked and to what extent



(a) The architecture of the proposed framework and the deployment flow.



```
"components":
[
  {
    "id" : "Task1",
    "configurations":
    [{
      "inputs":[{"raw_frames" : "30Hz"}, ...],
      "outputs":["processed_frames" : "30Hz"}, ...],
      "parameters":[{"resolution" : "720p"}, ...],
      "qualities":[{"framerate" : 30}, ...],
      "required_budget":
       {
        "TILE": { "RISCV":
                  [{
                     "unit" : "cycles",
                     "type" : "average_rate",
                     "value" : 100K
                   },... // other services from RISCV
                  ], ... // other resources from TILE
                 }, ... // other resources besides TILE
       },
      "initial_state":[{"IDMEM" : ".../task1.hex"}, ...]
    }, ... // other application configurations
  ]}, ... // other components
],
"compositions":
[
  "App1 = Task1 => Task2",
   ...
]
```

(b) Component model and bundle.

Fig. 1: Overview of the proposed framework.

they can be heterogeneous. In addition, it is not discussed how much effort is needed to integrate new heterogeneous platforms into the framework.

The run-time resource management framework proposed in [16] deploys applications with mixed and time-varying requirements on multi-core Linux systems. The framework, which is called BarbequeRTRM, finds the optimum application configurations and their mappings at run-time by considering non-functional aspects of both applications and resources. While the framework does not put any constraints on the type of applications and resources, application budget models and the API to communicate with the platform are not discussed, making the platform suitable only for the considered infrastructure. BarbequeRTRM is employed in [17] to perform resource management in heterogeneous computing platforms. Even though more complex computing settings can be managed by this framework, it lacks the support of various budget abstractions for one resource, reducing the flexibility of application profiling.

An extensive study on the service placement problem in fog and edge computing is done in [18] which shows that a great number of solutions are proposed to solve this problem; however, they are too specific in terms of optimization objectives/constraints and considered resources. Additionally, their budget models are relatively simple and not suitable for scenarios where more detailed and complicated models are needed (e.g., hard real-time systems). Such models must be integrated into the model-driven operations (e.g., run-time service placement) which is not possible in some of the proposed optimization engines such as ILP-based solutions [19], [20]. Also, consolidating various resource abstractions in one framework has been not studied. Similarly, a survey on run-time task mapping on multi/many-core systems is done in [21]. However, the proposed solutions only deal with fixed budget models and optimization criteria. They also enforce the application resource requirements to be expressed in the form that resource arbiters operate with—reducing the mapping flexibility.

In sum, the heterogeneity of resources and applications in the emerging computing paradigms necessitates more generic and flexible middlewares to automate the deployment process. Additionally, quality-sensitive applications – such as real-time applications – require certain guarantee levels on resource budgets to fulfill their non-functional requirements. However, given the heterogeneity of applications, the required guarantee levels are not identical for all the applications targeting a platform. Similarly, depending on the types of resources and their arbitration schemes, platform resources do not offer similar guarantee levels on the budgets they provide. The existing deployment frameworks are mostly designed for specific applications and resources and also do not consider multiple budget abstractions for one resource type. Leveraging a generic Pareto optimization engine and component model, we propose a deployment framework that supports any type of resource budget and application quality as long as their ordering and addition/subtraction are defined. Additionally, it allows applications and resources to be abstracted differently, leading to a more flexible application-to-resource binding. Moreover, the framework exploits a simple unified API to manage the infrastructure, strengthening its portability.

III. Overview: Proposed Framework

The architecture of the proposed framework and the deployment flow are depicted in Fig. 1a. The framework contains three layers (from left to right), namely i) operation, ii) management, and iii) orchestration, which are introduced in the following.

Operation: This layer contains software and hardware components of three types, namely i) *Applications*, which are made up of *Tasks*, ii) *Resources* that are hardware/software components on top of which applications execute, and iii) *Virtual Resources (VRs)* that are spatial and/or temporal resource partitions to share resources. VRs are often implemented by schedulers, microkernels, virtualization, or (RT)OS on top of a hardware resource. Each application runs on a dedicated set of VRs called a *Virtual Execution Platform (VEP)* [22]. A VEP is a hierarchical component containing VR components. In this framework, we use the component interface model proposed in [2], shown in Fig. 1b, due to its flexibility in modeling applications and resources as well as its suitability for quality and resource management. In this modeling framework, each component has one or more *configurations* which are determined by component *parameters*. Component *inputs* and *outputs* model functional aspects of components while their *qualities* model the non-functional ones. While components execute, they require (for their correct functionality and satisfying certain quality levels) and/or provide (as a result of their operation) certain services—called *required* and *provided budget* respectively. Component configurations are modeled with a set of points defined on $Q_{inputs} \times Q_{outputs} \times Q_{reqBudget} \times Q_{prvBudget} \times Q_{qualities} \times Q_{params}$ space where each dimension is a *partially-ordered set (poset)* and corresponds to one of the component interfaces. A partial order represents how quantities of a component interface are better than, worse than, or incomparable to other quantities of the same interface. For example, higher application quality levels, smaller required budgets, and larger provided budgets are considered better. Deployment based on pre-profiled models, especially for applications, is necessary to guarantee application qualities. While adaptive resource allocation reduces the design-time effort, it diminishes the hardness of guarantees. Three types of component composition are considered in [2] where the *free* composition aggregates all the component interfaces without connecting any interface to another (e.g., composing multiple processing cores to model a multi-core platform), the *horizontal* composition models data dependency between components by connecting outputs of one component to inputs of another one (e.g., composing an image decoder task to an image processing task), and the *vertical* composition models application-to-resource bindings by connecting a required budget of one component to a provided budget of another component (e.g., mapping a task onto a CPU).

Management: The management layer prepares the operation layer for execution and consists of *Application Managers (AMs)* and *Resource Managers (RMs)*. AMs are responsible for configuring, booting, starting, and stopping applications. This can be as simple as setting the image resolution or as complex as booting a virtualized OS. RMs, on the other hand, monitor resources and perform lifecycle management of VRs including creating/destroying (by allocating/deallocating budgets), configuring (by setting parameters), initializing/resetting VRs (by programming resources) [22].

Orchestration: The objective of the operation and management layers is to realize application deployments, which includes creating VEPs by partitioning resources, configuring the VEPs by setting resource parameters (e.g., processor frequency), initializing the VEPs (e.g., loading applications), configuring applications (e.g., setting parameters of an image filter), and executing them. Taking these steps requires knowing the size of resource partitions, application-to-resource bindings, resource and application parameters, and initial data of VEPs. All these are determined by the orchestration layer besides its other objective that is automating the deployment process. The orchestration part is comprised of several functional blocks briefly discussed in the following.

- The *Orchestrator*, the entry-point of the framework, accepts deployment requests and coordinates the operation of other functional blocks to plan and realize deployments.
- The *Broker* determines the optimal application deployments including the optimal component configurations (i.e., application and resource configurations) and compositions (e.g., VR-to-resource bindings). This is done through the Pareto optimization described in Section VI.
- The *Execution Platform Manager (EPM)* coordinates creation, configuration, and destruction of VEPs by RMs.
- Based on the component types, three *databases* exist that store *component bundles*. A component bundle contains the model of the component as well as initialization data required to load component instances with (e.g., application instructions and data, VM image, hardware parameters). As shown in Fig. 1b, component models are stored in the JSON format due to its readability and simplicity. Each database is composed of two sets, `components` and `compositions`. The former contains bundles of atomic components (e.g., application tasks) and the latter demonstrates how composite components (e.g., applications) are made up of other components (e.g., `A1 = T1 => T2`, which means `A1` is the horizontal composition of `T1` and `T2`). A bundle of an atomic component contains a unique identifier and a set of `configurations` describing component properties.

Distributing the orchestration tasks among multiple functional blocks lets us pipeline deployments, which improves the responsiveness of the system. Additionally, separation of resource management and coordination of RMs enables us to dynamically add resources and their managers to the system by just hooking RMs to the EPM at run-time (using Data Distribution Service [23]). This is highly beneficial for fog/edge settings where extra nodes can be dynamically added to the system. This paper focuses on the automated flow and the optimization done in the Broker.

IV. Case-study

We have implemented our proposed framework on the embedded platform shown in Fig. 2 as a proof of concept. The platform (i.e., PYNQ-Z2 board [24]) is comprised of

TABLE I: Required/provided budget of the components used in the case-study.

Component	Required/Provided Budget
RISCV	RISCV:{(cycles, TDM_Table, (period_ms:2.5, period_SUs:100k, slots:{(5001, 95k)})), (#partitions, int, 3)}
IDMEM	IDMEM:{(memory_blocks, Mem_Table, blocks:{(32768 , 32K),(65536, 32K),(98304, 32K)})}
MEM	MEM:{(mem_size, int, 16k)}
ARM	ARM:{(cycles, rate, (average:600M, max:650M), (os_type, broadcast, Linux), (io_bps, average_rate, 100M)}
DRAM	DRAM:{(mem_size, int, 250M)}
Sense	PL:{TILE:{RISCV:{(cycles, TDM_Table, (period_ms: 2.5, period_SUs: 100K, slots:{(10001, 50K)})), (#partitions, int, 1)},IDMEM{memory_blocks, Mem_Table, blocks:{(32768, 32K)}}}, MEM:{mem_size, int, 16K}}, PS:{ARM:{(cycles, average_rate, 350M), (os_type, broadcast, Linux), (io_bps, average_rate, 50M)}, DRAM:{(mem_size, int, 16M)}}
Compute	TILE:{RISCV:{(cycles, TDM_Slot, (period_ms: 5, slot_length_SUs: 80K),(#partitions, int, 1)}, IDMEM:{(memory_blocks, Block_size, 64k)}}, MEM:{mem_size, int, 2K}}
Actuate	TILE:{RISCV:{(cycles, Periodic, (period_ms: 4, #SUs: 10K), (#partitions, int, 1)}, IDMEM:{(memory_blocks, Block_size, 10k)}, Mem:{mem_size, int, 2K}}
Log	C_{low}: ARM:{(cycles, average_rate, 100M), (os_type, broadcast, Linux), (io_bps, average_rate, 10M)}, DRAM:{(mem_size, int, 5M)} C_{medium}: ARM:{(cycles, average_rate, 200M), (os_type, broadcast, Linux), (io_bps, average_rate, 20M)}, DRAM:{(mem_size, int, 5M)} C_{high}: ARM:{(cycles, average_rate, 300M), (os_type, broadcast, Linux), (io_bps, average_rate, 30M)}, DRAM:{(mem_size, int, 5M)}
Predict	C_{riscv}: TILE:{RISCV:{(cycles, share, 40%), (#partitions, int, 1)}, IDMEM:{(memory_blocks, Block_size, 64K)}}, Mem:{(mem_size, int, 16K)} C_{arm}: ARM:{(cycles, share, 10%)}, DRAM:{(mem_size, int, 10MB)}

two parts, namely i) Processing System (PS) – containing a dual-core Cortex-A9 processor, memories, and IO devices – and ii) Programmable Logic (PL)—containing a FPGA fabric on which three predictable RISC-V processing tiles are synthesized. The PS runs Ubuntu 18.04, making it suitable for soft quality requirements, and it partitions its resources using Linux resource controllers (also known as subsystems, e.g., cpu, memory, blkio). The PL contains a RT Verintec platform [25], following CompSOC concepts [22]. A microkernel (called VKERNEL [26]) runs on the processing tiles to create predictable and composable partitions, making it suitable for applications that require hard guarantees on resources. The platform can be powered by either a power supply or a battery. A synthetic control application is considered where three tasks (i.e., *Sense*, *Compute*, and *Actuate*) constitute the control pipeline, the *Log* task logs the system status, and the *Predict* task is used to predict the system inputs to use in the case of sensing failure. The list of components (i.e., applications and resources) and their required or provided budget are presented in Table I. Note that only two components, namely *Log* and *Predict*, have multiple configurations—denoted by $C_{.}$ in the table. Configurations of the *Log* task are different in the logging rate and the *Predict* task has two different implementations executable on ARM and RISCV. The framework is implemented in C++ and the functional blocks use OpenDDS [23] for their communications—allowing them to be run on the same or a different platform. The case-study demonstrates how the control application is deployed at run-time.

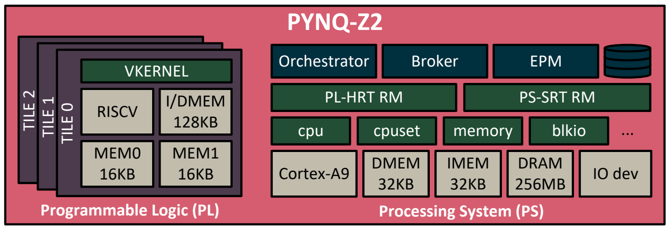


Fig. 2: The embedded platform used for our case-study.

V. Automated Deployment Flow

As shown in Fig. 1a, the flow is composed of ⑬ steps for deploying and ❼ steps for stopping an application. The separation of orchestration and management layers enables us to execute them on different platforms. For example, we can run the orchestration layer, which is more computationally intensive, on more powerful resources to speed up the deployment process. Additionally, in distributed systems, this design lets us have an RM for each subsystem, enabling them to operate in parallel and speeding up the deployment of distributed applications. The orchestration tasks are also distributed among three functional blocks, pipelining the deployment process. For instance, the deployment (done by EPM) and brokering of two consecutive deployment requests can be done in parallel. The deployment steps are explained in the following.

Deploying Applications: ⓪ An application deployment starts with registering the application bundle into the *Application Database (ADB)*). This lets users dynamically introduce new applications to the system. Each application bundle can be instantiated multiple times independently.

① Once the bundle is stored in the database, a deployment request containing the application identifier and deployment constraints including set-points for application parameters (e.g., image resolution), minimum quality levels (e.g., minimum frame rate), and maximum deployment costs (e.g., overall power) is sent to the Orchestrator.

②,③ The deployment request is forwarded to the Broker for deployment planning using the Pareto optimization described in Section VI. The Broker builds the exploration space by fetching application and resource bundles from the ADB and Resource Database (RDB). This ensures that deployments use the latest system state.

④ The optimization result is a VEP configuration (stored as a VEP bundle in the Virtual Resource Database, VDB) containing the optimal application configuration, resources the application is bound to, their optimal configurations, and the budget that must be allocated to the VEP. Updating the VDB is followed by budget reservations done by updating the remaining budget of resources in the RDB. Since remaining steps may take time, the status of resources are updated first to make sure that the subsequent deployment plannings are done based on the latest state of the system so that they can be done in parallel (pipelined). Additionally, the Broker sends the identifier of the reserved VEP (`null` in case of no feasible solution) to the Orchestrator. This lets the Orchestrator know the feasibility of deployment and it can inform the user.

⑤ In case of a feasible solution, a deployment request containing the VEP identifier is sent to the EPM.

⑥ The EPM fetches the VEP bundle from the VDB.

⑦ It sends VR creation requests to RMs. Requests are sent per VR and contain an identifier (used for future reference), the budget that the VR is supposed to provide, and possibly existing VR parameters to set (e.g., vCPU frequency). Note that the VR-to-resource bindings are included in the budget.

⑧ Upon receiving the creation requests, RMs create VRs using resource-specific southbound APIs (concurrently).

⑨ Once VRs are created, RMs respond with the remaining budget of resources expressed in possibly more detailed abstractions to retrieve the possibly lost budget details (explained in Section VI).

⑩ The EPM sends loading requests for VRs that require to be initialized for their operation. Since loading VRs (⑪) is often slower than creating them (⑧), this step is done in parallel with updating the state of the system (i.e., remaining budget of resources).

⑪ Once the VRs are initialized, the application is instantiated and ready to start.

⑫,⑬ If the application has parameters to set, the EPM sends them to the AM and it configures the application using application-specific interfaces.

Stopping Applications: Stopping entails first asking the AM to stop the application. Then all steps are reversed, omitting brokering.

VI. Brokering: Pareto Optimization

As explained in Section V (Steps ③,④), the goal of brokering is to determine a VEP configuration such that deployment constraints are satisfied and a cost function is optimized. To do so, the component configurations in the bundles are retrieved. Note that using component configurations that are profiled at design-time improves the predictability of application qualities at run-time. Adaptive resource management techniques can be employed to adjust resource allocations at run-time instead of the design-time profiling; however, they are only suitable for guaranteeing QoS in long time windows rather than any arbitrary moment in time, which is necessary for quality-sensitive applications. The Pareto optimization [27] is a poset-algebraic expression shown in Eq. 1, where all the operations are performed on the component interfaces explained in Section III (e.g., required and provided budgets). First, parameter constraints (D_P) are applied to the application configuration points (*app*). Next, a set of *VEP candidates* are built where each candidate (vep_i) is a component containing all the resources that the application requires and corresponding to an application-to-platform binding. All the candidates are vertically composed to the application (to perform budget matching) to build a configuration space containing feasible bindings, deployment costs, and application quality levels. Note that resource configurations, such as different CPU operating points realized by dynamic voltage and frequency scaling, are also considered in this configuration space. Next, the possibly existing quality (D_Q) and cost constraints (D_C) are applied (e.g., desired quality), resulting in a set of Pareto points. Finally, the Pareto frontier is minimized to one configuration (*vep**) by considering a certain policy (e.g., workload balancing) or by arbitrarily picking a point.

$$vep* = Min(D_C \cap D_Q \cap (\cup_i(vep_i \Uparrow (app \cap D_P)))) \quad (1)$$

Note that, as shown in Fig. 3, the vertical composition operator ($\Uparrow$) requires addition and subtraction operations to be defined on posets [2] (i.e., component interfaces such as provided and required budgets). For example, the provided budget of a vertical composition $C_2 \Uparrow C_1$ is the summation of the provided budget of C_1 and the residual of their composition, which is the provided budget of C_2 minus the required budget of C_1. Hence, to find the optimal deployment, we need to know how to i) compare ($\preceq$), ii) add ($+$), and iii) subtract ($-$) two component interfaces (e.g., required/provided budget), which makes the framework generic and suitable for heterogeneous components. To achieve this, all the component interfaces are modelled with simple *(name, value)* pairs (e.g., *(resolution, 720p)*). A partial order is defined on the values of each quantity type as well as any Cartesian product of component interfaces. Budgets have an additional hierarchy (encoded as nested JSON objects). The hierarchical structure lets us model how the resources are coupled together without adding adhoc optimization constraints, which is not the case in other works (e.g., [7], [19]). For example, if we ask for a processing tile comprised of a processor and a memory, we want them to be mapped on the same tile; however, asking for a processor and a memory without the notion of tile does not enforce this constraint.

An atomic budget is modeled with a *(unit, type, value)* tuple specifying the unit of the budget (e.g., `cycles` for processing power, `bytes` for memory capacity), the model of the budget (e.g., `average_rate`, `TDM_Slot`), and quantity of the budget (e.g., 20% for the CPU share). Similar to *names*, *units* of operands must be identical. Various resource models and abstractions are proposed in the literature (e.g., Latency-Rate [28], Service Curves [29]). Budget abstraction is a trade-off between analysis time and accuracy [29](e.g. overhead). More specific budgets may be less likely to be granted. Our framework allows RMs to use specific *provided budgets* (for high efficiency) and at the same time allows applications to *require abstract budgets* (for mapping flexibility) by automatically converting between abstraction levels of budgets of the same type. For example, we still can do the budget matching if an application is profiled with average resource requirements (e.g., average processing rate) while resources are abstracted with Time-Division Multiplexing (TDM) tables

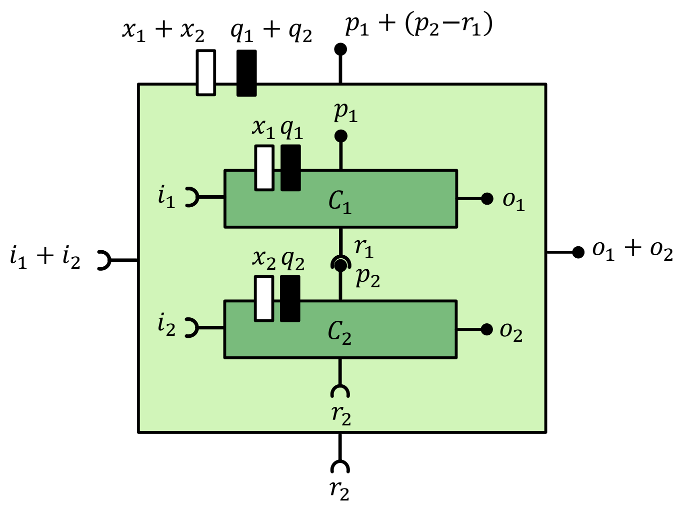


Fig. 3: The vertical composition of two components [2].

[10]. However, this comes at the cost of losing information when we add/subtract two budgets of different abstractions. For instance, if the required and provided budgets are expressed with the number of memory blocks and the address of blocks respectively, we only know the number of remaining blocks after subtracting the two budgets—unless we know how the memory manager allocates blocks to applications. However, including the allocation policies and implementation details slows down the optimization process. On top of that, the allocation algorithms may be proprietary and not be available. Therefore, we only allow budgets with less or the same details to be subtracted from (or compared with) other budgets and the abstraction of the result is similar to the less-detailed budget. In the allocation phase, RMs *refine* the abstracted budgets, and to solve the information loss problem, they report the remaining budget after allocating/deallocating budgets to/from VRs, and the RDB is updated with the retrieved abstractions (i.e., Steps (10) and ❼ in Fig. 1a).

VII. Experiments

A set of deployments, shown in Table II, are considered to do experiments on the case-study discussed in Section IV. The experiments aim to illustrate how the brokering works, evaluate the deployment performance, and demonstrate the framework's generality and flexibility. As discussed before, we use JSON files to store bundles in the databases. The size of the ADB and RDB for this case-study are 6KB and 10.5KB respectively. Note that bundles contain only references to the initialization data (e.g., object files, docker images). The run time of the brokering phase, VEP creations (i.e., budget allocation and loading VRs), and the total deployment are reported in Table II. The reported numbers are an average of 20 samples for each deployment. We have reported two numbers for the brokering and the total deployment time which correspond to our two experimental settings. In the first setting (*embedded setting*), we run the whole framework on a PYNQ board, while in the *distributed setting*, the orchestration layer runs on a PC. The results show that the brokering run time is proportional to the number of feasible mappings. The first deployment, which has the largest brokering time, has requirements on both the PL & PS and 12 feasible mappings exist (2 ARM cores, 3 RISCVs, and 2 MEMs per each tile). On the other hand, brokering the *log* task (*SysStat* in the distributed setting) takes the least time due to its requirement on only the PS (PC in the distributed setting). VR creations are faster on the PS because i) the ARM cores are faster than RISCVs, ii) the RM that manages resources of the PS (*PS-SRT RM*) works directly with cgroups, which is reasonably fast, and iii) the RM that manages resources of the PL (*PL-HRT RM*) runs on the ARM, adding extra latency due to the communication between *PL-HRT RM* and VKERNEL.

Embedded setting: Initially, the platform is powered by a battery and the optimization goal is to minimize the power consumption. For simplicity, we only consider the power consumption of ARM and RISCV cores (with an ARM consuming more than a RISCV core) and deployment costs relate to the budgets required from these components. As shown in Table I, the *Sense* application is distributed over PL and PS, and its resource requirement is stated at the same abstraction level as the implementation (e.g., detailed TDM allocation including the period of the TDM wheel in both milliseconds and number of *Service Units*, e.g., cycles, and a set of TDM slots with known lengths and starting points) which is necessary for its correct sensing. Since all the resources provide enough budget to host this task, there are 12 feasible configurations with identical costs; hence, one mapping is picked arbitrarily. The second application is implemented for RISCVs and is profiled at a higher abstraction level compared to the implementation (i.e., abstracting the starting point of TDM slots and memory blocks). Note that in such cases, RMs are responsible to *refine* the budgets using their own allocation strategies. In this experiment, the application asks for 80K cycles in every 5ms and a TDM slot of length 40k starting at the 60001st cycle and repeating every 2.5ms is allocated to it. The third deployment, asks for a *periodic* budget (not necessarily a TDM one) of 10K cycles in every 4ms. Since the asked period is different from the period of the TDM wheels, the RM allocates a slot of length 10K (repeating every 2.5ms) to guarantee the budget. Additionally, the application asks for a memory block of 10KB; however, since the memory is divided into blocks of 32KB, a full block of 32KB is allocated to the application. Note that this information is reflected in the RDB after the RM creates VRs and reports its state (Step (10)). While the first three requests did not enforce any constraints, the fourth request requires the rate of logging to be at least medium, which removes the first configuration (i.e., C_{low}) from the feasible solutions. Given the optimization goal, the configuration C_{medium} is chosen to minimize the power consumption. After the fourth deployment, a power supply is connected to the board and the optimization goal changes to maximizing application qualities. To demonstrate how this change influences the deployments, we stop (Request

TABLE II: Deployment requests, corresponding optimization results, and deployment time for the embedded/distributed settings.

Request	Allocated Budget	Brokering	Alloc&Load	Total
1) deploy(Sense)	RISCV_1:{(cpu_cycles, TDM_Table, (period_ms: 2.5, period_SUs: 100K, slots:{(10001, 50K)})), (#partitions, int, 1)},IDMEM_1{memory_blocks, Mem_Table, blocks:{(32768, 32K)}}}, MEM_01:{mem_size, int, 16K}}, ARM:{(cpu_cycles, average_rate, 200M), (io_bps, average_rate, 50M)}, DRAM_1:{(mem_size, int, 16M)}}	2411ms/69ms	812ms	3.3s/0.96s
2) deploy(Compute)	RISCV_1:{(cpu_cycles, TDM_Slot, (period_ms: 2.5, period_SUs: 100K, slots:{(60001, 40K)})),(#partitions, int, 1)}, IDMEM_1:{memory_blocks, Mem_Table, blocks:{(65536, 32K), (98304, 32K)}}}, MEM_02:{mem_size, int, 2K}}	574ms/24ms	893ms	1.9s/1.4s
3) deploy(Actuate)	RISCV_2:{(cpu_cycles, TDM_Slot, (period_ms: 2.5, period_SUs: 100K, slots:{(5001, 10K)})),(#partitions, int, 1)}, IDMEM_2:{memory_blocks, Mem_Table, blocks:{(32768, 32K)}}}, MEM_02:{mem_size, int, 2K}}	630ms/26ms	740ms	1.9s/1.4s
4) deploy(Log, log_rate $\geq$ medium)	ARM_1:{(cpu_cycles, average_rate, 200M), (io_bps, average_rate, 20M)}, DRAM_1:{(mem_size, int, 5M)}	328ms/13ms	20ms	0.6s/0.04s
5) stop(Log)	-	-	-	0.2s
6) deploy(Log)	ARM_2:{(cpu_cycles, average_rate, 300M), (io_bps, average_rate, 30M)}, DRAM_1:{(mem_size, int, 5M)}	243ms/12ms	20ms	0.6s/0.04s
7) deploy(Predict)	RISCV_3:{(cpu_cycles, TDM_Slot, (period_ms: 2.5, period_SUs: 100K, slots:{(5001, 40K)})),(#partitions, int, 1)}, IDMEM_3:{memory_blocks, Mem_Table, blocks:{(32768, 32K)}}}, MEM_12:{mem_size, int, 16K}}	440ms/28ms	782ms	1.8s/1.4s
8) deploy(SysStat)	best-effort	-/9ms	580ms	-/0.6s

5) and redeploy (Request 6) the *Log* application (dynamic reconfiguration is left for future work). This time C_{high} is chosen given its higher *log_rate* and the binding changes to ARM_2 because ARM_1 cannot accommodate C_{high} (only 250M cycles remain). The last application has two configurations and is executable on both the PL and PS. Since there is no defined quality for this application, the RISCV implementation dominates the other in terms of deployment costs. Hence, C_{riscv} is chosen and a TDM slot of length 40K is allocated to it after refining the required 40% share to the TDM allocation.

Distributed setting: To demonstrate the portability and extensibility of the framework, we consider another setting in which a PYNQ board is connected to a Core i7-based PC (running Ubuntu) where VEPs are realized by Docker containers [30]. In this setting, the orchestration layer as well as another Resource Manager to manage PC resources run on the PC. Thanks to the portability of OpenDDS and our C++ implementation, the framework does not need to change for operating in this setting. Since the PC runs a generic Ubuntu, it is suitable for running non-real-time tasks. For our experiment, we consider a web service packaged in a Docker image to visualize the system status. Initially, when the system boots, the board is not connected and only the PC resources compose our execution platform. A deployment request to run the web service (*SysStat* in the table) is issued. Since we consider this application as a non-real-time one, no required budget is specified for it. The brokering time is relatively small and the total deployment time is dominated by the container creation phase. Note that since we use Docker instead of directly using cgroups, in this setting, the allocation and loading phase takes much longer. We repeat the deployments performed in the first experimental setting and measure the run times. Note that only the VEP creation phase is done by the resources and their arbiters; hence, they remain the same in this experimental setting. However, brokering and total deployment times are changed. Since the broker runs on the PC now, the brokering times are significantly reduced and the VEP creation phase has become the bottleneck for all the performed deployments.

VIII. Conclusion

Application and service deployment on computing platforms with limited resources has been vastly studied. Most of the proposed frameworks, however, are designed to operate on certain infrastructures. To follow the current trend – which is having heterogeneous resources and applications in one system – we have proposed a deployment framework equipped with a run-time optimization engine that is both generic (in terms of application qualities, deployment costs, and resource constraints) and flexible in defining constraints and optimization goals. Supported resources can be easily extended by implementing three operators on resource models (i.e., $+, -, \preceq$). The framework allows platform resources and application resource requirements to be expressed at different abstractions, which improves the flexibility of deployments in terms of supported application types (e.g., hard/soft/non-real-time) as well as application-to-resource bindings. The framework is implemented and evaluated on an embedded platform for a proof of concept. This paper focuses on dynamic adding/removing applications to/from systems, and other dynamic scenarios such as dynamic resource scaling, VEP remapping (migration), and application reconfiguration are considered for our future work.

Acknowledgments

This research has received funding from the Electronic Component Systems for European Leadership (ECSEL) Joint Undertaking under grant agreement No 783162 (FitOptiVis project). This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme.

References

[1] C.-H. Hong and B. Varghese, "Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–37, 2019.
[2] M. Hendriks, M. Geilen, K. Goossens, R. de Jong, and T. Basten, "Interface Modeling for Quality and Resource Management," *Logical Methods in Computer Science*, vol. 17, 2021.

[3] M. García-Valls and R. Baldoni, "Adaptive middleware design for CPS: Considerations on the OS, resource managers, and the network run-time," in *International Workshop on Adaptive and Reflective Middleware*, 2015, pp. 1–6.

[4] B. Lisper, "SWEET – A Tool for WCET Flow Analysis," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, 2014, pp. 482–485.

[5] F. C. Delicato, P. F. Pires, T. Batista *et al.*, *Resource Management for Internet of Things*. Springer, 2017, vol. 16.

[6] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "DIANE – Dynamic IoT Application Deployment," in *IEEE International Conference on Mobile Services*, 2015, pp. 298–305.

[7] H. Sami and A. Mourad, "Dynamic On-Demand Fog Formation Offering On-the-Fly IoT Service Deployment," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 1026–1039, 2020.

[8] W. do Espírito Santo, R. d. S. M. Júnior, A. d. R. L. Ribeiro, D. S. Silva, and R. Santos, "Systematic Mapping on Orchestration of Container-based Applications in Fog Computing," in *International Conference on Network and Service Management (CNSM)*, 2019, pp. 1–7.

[9] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-Time Containers: A Survey," in *Workshop on Fog Computing and the IoT (Fog-IoT)*, 2020.

[10] S. Sinha, M. Koedam, G. Breaban, A. Nelson, A. B. Nejad, M. Geilen, and K. Goossens, "Composable and predictable dynamic loading for time-critical partitioned systems on multiprocessor architectures," *Microprocessors and Microsystems*, vol. 39, no. 8, pp. 1087–1107, 2015.

[11] M. B. Nguyen, V. Tran, and L. Hluchy, "A Generic Development and Deployment Framework for Cloud Computing and Distributed Applications," *Computing and informatics*, vol. 32, no. 3, pp. 461–485, 2013.

[12] A. Agirre, J. Parra, A. Armentia, E. Estévez, and M. Marcos, "QoS Aware Middleware Support for Dynamically Reconfigurable Component Based IoT Applications," *International Journal of Distributed Sensor Networks*, vol. 12, no. 4, 2016.

[13] Y. Lee, C. Min, Y. Ju, S. Kang, Y. Rhee, and J. Song, "An Active Resource Orchestration Framework for PAN-Scale, Sensor-Rich Environments," *IEEE Transactions on Mobile Computing*, vol. 13, no. 3, pp. 596–610, 2013.

[14] Y. Hu, C. De Laat, and Z. Zhao, "Multi-objective Container Deployment on Heterogeneous Clusters," in *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 592–599.

[15] T. Orazio, C. Domenico, M. Pietro *et al.*, "TORCH: a TOSCA-Based Orchestrator of Multi-Cloud Containerised Applications," *Journal of Grid Computing*, vol. 19, no. 1, 2021.

[16] P. Bellasi, G. Massari, and W. Fornaciari, "Effective Runtime Resource Management Using Linux Control Groups with the BarbequeRTRM Framework," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 2, pp. 1–17, 2015.

[17] G. Agosta, W. Fornaciari, G. Massari, A. Pupykina, F. Reghenzani, and M. Zanella, "Managing Heterogeneous Resources in HPC Systems," in *Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM)*, 2018, pp. 7–12.

[18] F. A. Salaht, F. Desprez, and A. Lebre, "An Overview of Service Placement Problem in Fog and Edge Computing," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–35, 2020.

[19] J. Wang, H. Qi, K. Li, and X. Zhou, "PRSFC-IoT: A Performance and Resource Aware Orchestration System of Service Function Chaining for Internet of Things," *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 1400–1410, 2018.

[20] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar, "Towards QoS-aware Fog Service Placement," in *International Conference on Fog and Edge Computing (ICFEC)*, 2017, pp. 89–96.

[21] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends," in *Design Automation Conference (DAC)*, 2013, pp. 1–10.

[22] K. Goossens, M. Koedam, A. Nelson, S. Sinha, S. Goossens, Y. Li, G. Breaban, R. van Kampenhout, R. Tavakoli, J. Valencia *et al.*, "NoC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications." 2017.

[23] "Opendds," http://www.opendds.org.

[24] "Pynq-z2," http://www.tul.com.tw/ProductsPYNQ-Z2.html.

[25] "Verintec solutions," http://www.verintec.com.

[26] A. Nelson, A. B. Nejad, A. Molnos, M. Koedam, and K. Goossens, "CoMik: A Predictable and Cycle-Accurately Composable Real-Time Microkernel," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014, pp. 1–4.

[27] M. Geilen, T. Basten, B. Theelen, and R. Otten, "An Algebra of Pareto Points," *Fundamenta Informaticae*, vol. 78, no. 1, pp. 35–74, 2007.

[28] D. Stiliadis and A. Varma, "Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms," *IEEE/ACM Transactions on networking*, vol. 6, no. 5, pp. 611–624, 1998.

[29] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour, "Influence of different abstractions on the performance analysis of distributed hard real-time systems," *Design Automation for Embedded Systems*, vol. 13, no. 1, pp. 27–49, 2009.

[30] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux journal*, vol. 2014, no. 239, 2014.