2

Load and Resource Models

The efficient allocation of computational resources requires some understanding of the resources themselves and their availability, as well as the load that must be allocated to them. Possibly under different names, the concept of resource and load modelling is commonly used in embedded, high-performance and cloud computing. For example, workflow models in HPC and task graphs in embedded computing are common ways to represent application load, while platform and resource models are used to represent the processing, networking and storage capabilities of the computer systems that run those applications.

With the help of meaningful resource and load models, it is possible to evaluate the impact of different resource allocation techniques on the efficiency of resource usage and on application performance requirements. The more accurate the models, the better they can predict the performance of a computer system under a given load. On the other hand, dynamic and complex systems are harder to model accurately, so there is clearly a trade-off here.

In real-time embedded computing, for example, it is common practice to constrain the execution of software to sporadic and bounded time intervals, and to disable advanced features of microprocessors such as out-of-order execution and caching, aiming to simplify the system's behaviour and enable the creation of accurate load and resource models. At that level of accuracy, system designers can use such models to evaluate different resource allocation alternatives and identify the ones under which the system will never violate any of its performance guarantees, not even in a worst-case scenario.

Such practice, however, requires a complete knowledge of the system resources as well as the load to be allocated to them. In many embedded systems, and in the large majority of high-performance and cloud computing systems, that is not the case. Therefore, recent modeling approaches have ways to represent load and resources under different levels of uncertainty. Stochastic models of the arrival and execution times of application-specific load or of the availability of computational resources, for example, are now commonly used to characterise average-case system performance.

2.1 Related Work

In real-time systems, load models are often variations of the sporadic task model [42] or the time-triggered model [71], focusing explicitly on timing and on the repetitive nature of tasks (e.g., data from a sensor must be processed every 2 ms; a new gene sequencing job will be launched at least every millisecond) rather than the functional dependencies between them.

Dataflow application models are usually untimed, and different tasks are synchronised by the data flowing through the system. Dataflows are usually modelled through graphs that represent the functional dependencies between tasks and some information about the nature of the data transfer. Many different dataflow models exist [134], with different types of constraints on the execution of tasks and communication aiming to allow different types of analysis (e.g., statically schedulable, time predictable, bounded communication buffering). Many HPC workflows are also modelled as dependency graphs, often as directed acyclic graphs (DAGs) [144]. Such graphs can be annotated with estimations of execution time and communication volumes, which can be used to optimise resource allocation or implement resource reservation mechanisms [90]. Similarly, estimations of inter-arrival times, execution times and communication volumes can be modelled stochastically, allowing for a more general understanding of the characteristics of a given workload [51]. That approach can also be used to support the generation of synthetic application models that follow the specific characteristics of a realistic scenario.

Advanced workflow management systems augment HPC and scientific computing workflow models with execution semantics [86], allowing such workflows to be analysed in similar ways as in time-triggered and dataflow models mentioned above. Finer granularity models are also used in HPC [10, 111], where application load is represented as a series of computation and communication bursts (often obtained from execution traces), but such models are too complex to be analysed and therefore are used only to drive abstract simulation.

A number of application modelling approaches try to capture characteristics that are critical to specific domains. Within the automotive domain, a component-based software specification standard is established, called AUTOSAR (more in www.autosar.org). Within this standard, software components covering runnable entities can be defined by specifying interfaces, execution rates and timing constraints. However, AUTOSAR takes a conservative stand and does not allow the dynamic allocation of runnable entities to different computational units.

Advanced approaches in application modelling supports the creation of hybrid models, i.e., models created using different underlying rules.

Ptolemy [47] is a modelling and simulation framework supporting hybrid application modelling for embedded systems using actor-orientation (a flexible model for representing concurrent behaviour). It supports different types of time-triggered and dataflow modelling approaches, among others, and is amenable to extensions to specific domains.

2.2 Requirements

Within the scope of this book, we use a model of load and resources that can cater to both worst-case and average-case system performance. It supports complete and accurate description of a system's load and resources, but is also able to accommodate different levels of uncertainty by allowing stochastic descriptions of load.

We therefore define a load model using the notion of jobs, which should represent the different parts of an application and, more specifically, the load each of those parts imposes on platform resources. This is a general-purpose model, aiming to have constructs that are flexible enough to represent multiple types of application components. For example, a job could represent the execution of a software task over a CPU, the transmission of a stream of data over a network, or the dynamic reconfiguration of an FPGA device.

In order to model different types of applications, from embedded to HPC systems, such load models must be powerful enough to cover characteristics, e.g., functional properties, that are commonly found in such systems, as well as non-functional properties that can be used to evaluate the impact of different allocation mechanisms. In the subsections below, we present the requirements for such load modelling approach along four distinct categories: structure, temporal behaviour, resource constraints and load characterisation.

2.2.1 Requirements on Modelling Load Structure

Load models should be able to support multiple levels of abstraction, exposing more or less details of the application architecture according to the level of accuracy that is needed when evaluating the impact of a particular resource allocation mechanism. For instance, it may be useful to assume that all application jobs are completely independent, abstracting away their inter-communication, if the overheads due to data exchange are negligible. Therefore, the application structure denotes how an application can be broken in multiple jobs and how these jobs relate to each other.

Regarding the application structure, we list requirements for a load modelling approach, so that the model is powerful enough to represent the most common types of applications.

2.2.1.1 Singleton

Ability to model applications that are composed of a single job.

2.2.1.2 Independent jobs

Ability to model applications that are composed of an arbitrary number of jobs that do not depend on or communicate with other jobs. It is assumed that jobs constantly have access to all information they need.

2.2.1.3 Single-dependency jobs

Ability to model applications that are composed of an arbitrary number of jobs that can depend on one and only one other job. Therefore, the application model must explicitly have the notion of dependencies between jobs.

2.2.1.4 Communicating jobs

Ability to model applications that are composed of an arbitrary number of job pairs. Intuitively, each pair includes a computing job and a communication job, but the strict definition of a communicating job should be a pair of dependent jobs that cannot be allocated to the same resource type (see requirements on resourcing in Subsection 3.3). This enforces the notion that, in this kind of application, communication can only be performed once the respective computation has completed.

2.2.1.5 Multi-dependency jobs

Ability to model applications that are composed of an arbitrary number of computation jobs, each of them depending on an arbitrary number of communication jobs, and also initiating an arbitrary number of communication jobs. The structure of this type of model constrains the application in such a way that the communication jobs initiated by a given computation job must not depend on computation jobs that depend directly or indirectly on their initiator (no cyclic dependencies).

2.2.2 Requirements on Modelling Load Temporal Behaviour

The temporal behaviour of the load defines the release of application jobs, i.e., when a job can actually be executed over a resource. Behaviours can be generally classified in time-driven (requirements 2.2.2.1, 2.2.2.2 and 2.2.2.3 below) or event-driven (remaining requirements).

Regarding application temporal behaviour, we list the following requirements for a load modelling approach, so that it is powerful enough to represent the following types of application jobs.

2.2.2.1 Single appearance

Ability to model an application job that is not part of a series, and is released at a specific point in time.

2.2.2.2 Strictly periodic

Ability to model an application job that is part of a series of jobs with release times separated by a constant time interval. If the release time of a job and its order within the series is known, the release time of all other jobs can be derived from it.

2.2.2.3 Sporadic

An application job that is part of a series of jobs with release times separated by a time interval that has a known lower bound. For every release of a job, it is therefore known that the release of the subsequent job of the series will not happen before that lower bound.

2.2.2.4 Aperiodic

An application job that can be released at any arbitrary time. It can be used to model event-driven systems where no assumptions can be made about the event sources. If an assumption can be made about the minimum time interval between successive events, such job series can be conservatively (but perhaps not accurately) modelled as sporadic jobs.

2.2.2.5 Fully dependent

An application job that is released immediately after the completion of all jobs that it depends on.

2.2.2.6 N out of M dependent

An application job that is released immediately after the completion of any N jobs out of all M jobs that it depends on, $(M > N)$.

2.2.3 Requirements on Modelling Load Resourcing Constraints

The resourcing of applications defines which kind of resources a given job requires for its execution. This requires a taxonomy of resources over different types. The load model addressed here makes no assumption about such taxonomy, and it may work under different typing systems (e.g., flat type hierarchy, single-parent type hierarchy, multiple-inheritance type systems), as different resource allocation mechanisms might benefit from them. Regarding this classification, we list the following requirements for a load modelling approach,

so that it is powerful enough to represent the following types of application jobs.

2.2.3.1 Untyped job

Ability to model an application job that can be executed on any type of resource.

2.2.3.2 Single-typed job

An application job that must be executed over a specific type of resource.

2.2.3.3 Multi-typed job

An application job that can be executed over multiple types of resource.

2.2.4 Requirements on Modelling Load Characterisation

The characterisation of the application load defines how long each of its jobs uses the resources they were allocated. Regarding this classification, we list the following requirements for a load modelling approach, so that it is powerful enough to represent the following types of application jobs.

2.2.4.1 Fixed load

An application job that always occupies a resource for a constant amount of time, regardless of the resource. The load of such a job can be characterised by a scalar.

2.2.4.2 Probabilistic load

An application job that occupies a resource for a probabilistic amount of time, regardless of the resource. The load of such a job is a random variable, and can be characterised by a histogram or a probability density function.

2.2.4.3 Typed fixed load

A multi-typed application job that occupies resources of different types by a potentially different, yet constant amount of time. The load of such a job can be characterised by a vector of scalars, and the length of the vector is equal to the number of types of resources that the job can occupy.

2.2.4.4 Typed probabilistic load

A multi-typed application job that occupies resources of different types with a potentially distinct stochastic behaviour on each of them. The load of such a job can be characterised by a vector of probability density functions or histograms, and the length of the vector is equal to the number of types of resources that the job can occupy.

2.3 An Interval Algebra for Load and Resource Modelling

Within this book, we will rely on a novel approach to load and resource modelling based on an interval algebra (IA). It will be used throughout the book to ease our understanding of the impact of different resource allocation mechanisms. But more importantly, it can be used by the resource allocation mechanisms themselves as an internal representation of the resources and the load that they are supposed to manage.

Our IA represents non-functional characteristics of application load using the mathematical concept of intervals. It can be used to analytically derive the impact of using different resource allocation policies on the original application characteristics. The main concerns of this book are performance and time predictability, so most of our examples focus on the representation of time intervals, but the interval algebra can naturally be extended to support other non-functional properties such as energy dissipation.

In a simplistic example, we can consider an application with three jobs A, B and C, and a homogeneous platform composed of two processors with first-come-first-serve scheduling. Each of the jobs can be represented by an interval that denotes the time they need to run: $A = [0, 30[$, $B = [0, 45[$, $C = [0, 20[$ (assuming in this example that they are all independent and ready to run at time $= 0$). By using simple interval algebra operations, a resource allocation heuristic can estimate the response time R of the three tasks under different allocation schemes (e.g., $R_A = 30$, $R_B = 45$ and $R_C = 50$ if A and C are allocated, in that order, to one of the processors and B is allocated to the other), and thus can dynamically decide whether it is likely to meet the applications constraints when using a given allocation.

While trivial, such example can be made arbitrarily complex by allowing different resource scheduling disciplines, a larger number of tasks and processors. For the interval algebra, however, the analysis of the response times under a specific allocation would still involve the application of the same interval manipulation rules.

The advantages of such an approach are numerous, including the following.

- It enables dynamic allocation mechanisms to have an appropriate level of confidence on whether the chosen allocation meets the applications' timing constraints.
- The approach can be used as a fitness function of search-based allocation heuristics, if the algebraic operations are sufficiently lightweight as they have to be applied over a potentially large search space (some examples of integrating IA to genetic algorithms are provided in Chapter 5).
- The solution of algebraic operations can be found in multiple ways, with different levels of performance. Therefore, resource allocation heuristics

can be improved simply by optimising the solution of the employed algebraic operations.

- If absolute predictability is not required (i.e., in soft real-time and best-effort applications), algebraic operations can be solved faster by applying approximations that sacrifice the accuracy of the final result. This enables allocation mechanisms that can be applied to systems with different levels of strictness of their timing requirements.

Let us now introduce the main principles behind this interval algebra. Our goal in this book is not to be overly formal, so we will favour intuitive descriptions over mathematical formalism whenever possible (i.e., without sacrificing precision). In general terms, an algebra is a definition of symbols and the rules for manipulating those symbols. Our interval algebra, therefore, establishes rules for the manipulation of intervals. It defines different types of intervals, which represent the amount of time a particular piece of application load requires from a notional resource. For example, a single job can be represented by a time interval using the notation below:

$$\#A\#0\#40 \tag{2.1}$$

where the first element of the tuple is a unique job identifier, the second is a non-negative real number representing the release time of the job and the third is a positive real number representing the job's load, i.e., the actual length of the time interval. In the example above, job A is released at time 0 and requires 40 time units of a resource. The same concept can also be represented using the mathematical notation for a left-closed right-open bounded interval $[0, 40[$.

Following the definition above, our IA must also define rules for manipulations of such intervals: what happens when an interval is allocated to a specific type of resource, what if two intervals are allocated to the same resource, etc. Widely used algebras define a small number of basic operations (e.g., addition, multiplication) and then define more complex operations as composites of those basic operations (e.g., matrix multiplication). Our IA defines two basic algebraic operations: *time displacement* and *partition*. Time displacement changes the endpoints of an interval by an arbitrary value t, and denotes that the job has to wait for its allocated resource (i.e., its starting and ending times were moved t time units to the future). Partition simply breaks one interval in two, and denotes that a job was preempted from a resource (and the second interval produced by the partition is likely to be time-displaced). All other interval-algebraic operations of IA, which can represent an arbitrarily large set of allocation and scheduling mechanisms, can be expressed as compositions of these two. By applying these operations, it is possible to investigate the impact of different resource allocation and scheduling mechanisms on the endpoints of the intervals, which in turn denote the completion times of each application component.

In the following subsections, we show how our IA addresses the requirements described in Section 2.2.

2.3.1 Modelling Load Structure

The interval-based representation of a job presented above is sufficient to express a singleton. By using a set of such intervals, independent jobs can be also represented. To denote a dependency between two tasks A and B, the notation can be extended to include a job identifier instead of the release time of a job:

$$\#B\#A\#50 \tag{2.2}$$

This notation is capable of denoting single dependency jobs, and conveys that interval B's start-point depends on interval A. Multiple dependencies can also be specified as a dependency set, and thus multi-dependency jobs can be covered:

$$\#C\#\{A, B\}\#260 \tag{2.3}$$

This notation assumes that whenever an interval has dependencies, its start-point lies exactly at the highest endpoint among all the intervals it depends on. In this example, assuming that jobs A and B are defined as in examples (2.1) and (2.2), this leads to: $A = [0, 40), B = [40, 90), C = [90, 350)$.

2.3.2 Modelling Load Temporal Behaviour

The intervals described in the previous subsection are single-appearance and have a fixed release time, therefore express singleton jobs. A strictly periodic series of jobs can be characterised by its release time, the period after which a new job is released, and the time interval each job requires from a notional resource. We denote such job series with the notation exemplified below, which is exactly the same as the notation of a singleton task followed by the period:

$$\#D\#0\#40\#100 \tag{2.4}$$

Mathematically, it represents an infinite series of intervals, such as: $D = [0, 40), [100, 140), [200, 240), \ldots$. This extension is expressive enough to represent strictly periodic tasks.

The release time of sporadic tasks is not deterministic but has well defined bounds. In case of aperiodic tasks, those bounds do not exist. To model those cases, IA represents release times with *aleatory variables*. Those variables are associated with probability distributions that can constrain assumed values. We will cover that approach in Section 2.3.5 when we discuss intervals with stochastic representations of time.

2.3.3 Modelling Load Resourcing Constraints

IA represents a resource as the dimension over which jobs are operated upon. Jobs, each represented by its respective interval, are allocated onto a resource; algebraic operations determine how the resource is shared between all of them, and how the resource sharing affects their timings. We denote a resource with the notation exemplified below:

$$+Z_1(\#A\#0\#40) \tag{2.5}$$

where the algebraic operation $+Z_1$ is applied to the set of intervals surrounded by brackets (only A in the example above). The example below shows the same resource, but this time with two distinct jobs mapped to it:

$$\begin{aligned} &+ Z_1(\#A\#0\#40, \#B\#0\#50) = \\ &+ Z_1(\#A\&40, \#B\&90) = \\ &+ Z_1([0, 90]) \end{aligned} \tag{2.6}$$

In this example, we introduce two different ways to evaluate the operator $+Z_1$ (which we can intuitively understand as a resource serving jobs under a FIFO schedule). The first evaluation of the operator preserves the identities of the mapped jobs, and it indicates the completion times of each one of them after the symbol "&". We will refer to this type of evaluation as *information-preserving* (or simply *preserving*). The second way to evaluate the operator is equivalent to the first, but it does not preserve any information about the individual operands. It simply determines the busy period(s) of the resource with one or more intervals. We refer to this type of evaluation as *information-collapsing* (or simply *collapsing*).

Comparing with elementary algebra, the two evaluations of the operator are akin to solving an expression like $(3 + 5) + (1 + 2)$ using an intermediate step $8 + 3$ before arriving to the final result 11. In both algebras, there is an infinite set of possible operands that could lead to a particular result, and there is no information in the final result that could allow the backtracking of the initial operands.

A slightly different example is shown below, using the same jobs but this time mapped onto resource $+Z_2$ that uses a time-division multiplexing (TDM) scheduler with a quantum of 8 time units:

$$\begin{aligned} &+ Z_2(\#A\#0\#40, \#B\#0\#50) = \\ &+ Z_2(\#A\&72, \#B\&90) = \\ &+ Z_2([0, 90]) \end{aligned} \tag{2.7}$$

It is worth noticing that only the intermediate expression (i.e., after the preserving evaluation) differs, and the final result after the collapsing evaluation

is the same. This is always the case if the operand denotes a work-preserving scheduler (i.e., a resource is never idle if there are jobs ready to be served).

The two following examples show jobs allocated to a resource that is shared under a priority-preemptive scheduler, assigning priorities in the same order the jobs are passed to the operator (higher to lower):

$$\begin{aligned} &+Z_3(\#C\#15\#40, \#D\#10\#50, \#E\#0\#50) = \\ &+Z_3(\#C\&55, \#D\&100, \#E\&140) = \\ &+Z_3([0,140)) \end{aligned} \tag{2.8}$$

$$\begin{aligned} &+Z_4(\#F\#10\#4, \#G\#0\#18, \#H\#26\#5, \#I\#24\#8) = \\ &+Z_4(\#F\&14, \#G\&22, \#H\&31, \#I\&37) = \\ &+Z_4([0,22),[24,37)) \end{aligned} \tag{2.9}$$

In both cases, the algebraic operations abstracts away the specific interleaving patterns of the execution of every job. Each of the evaluation types focusses solely on, respectively, the finish times of each job or the idleness of the resource. For example, formula (2.9) represents the following: job G starts to be executed at time zero, but after 10 time units it is preempted by job F which runs to completion for 10 time units; then G resumes and runs for its remaining execution time until time equals 22 units; resource Z_4 becomes idle until job I is released at 24 time units, which in turn suffers a preemption from H between times 26 and 31 units and then executes until time equals 37 units .

Just like single appearance jobs, periodic jobs can be allocated to resources:

$$\begin{aligned} &+Z_1(\#A\#0\#40\#100, \#B\#0\#50) = \\ &+Z_1(\#A\&40, \#B\&90, \#A\#100\#40\#100) = \\ &+Z_1([0,90), \#A\#100\#40\#100) \end{aligned} \tag{2.10}$$

It is important to notice that a periodic job series always remains as a distinct interval in the result of both preserving and collapsing evaluations of an operator. This reflects the infinite nature of the series.

One of the crucial properties of a job is its affinity, which means that it can be served only by the designated resources. The job that can be executed on any resource available in a system is referred to as *untyped job*. If a job can be executed on a single type of resources only, it is a *single-typed* job. A *multi-typed* job can be executed on a few (enumerated) resource types, possibly with different execution times on each of them. In all examples so far, only untyped jobs have been used. To describe a single-typed or multi-typed job, the notation should support the definition of different types of resources and

different types of resource affinity. This can be expressed as follows, where each scalar in pointy brackets denotes a different type and the absence of type constraints implies untyped jobs or resources (as in examples above):

$$+ Z_5\langle 2\rangle(\#J\langle 2\rangle\#0\#15, \#K\langle 2,3,8\rangle\#0\#20, \#L\#0\#14) \tag{2.11}$$

By allowing the definition of resources types and resource requirements, it is also possible to present communicating jobs by modelling the job as two fully dependent intervals with distinct resource requirements, one for computation and one for communication (i.e., the job can only communicate over resource 2 once it has finished being computed over resource 1):

$$\begin{aligned} &\#L\langle 1\rangle\#0\#14 \\ &\#M\langle 2\rangle\#L\#340 \end{aligned} \tag{2.12}$$

2.3.4 Modelling Load Characterisation

The representation of load as the interval length, denoted by a positive real number (as defined in Subsection 2.3.1), is already capable of representing a fixed load.

To represent a typed fixed load, we allow the specification of different interval lengths for different resource types using a similar notation as the one introduced at the end of Subsection 2.3.3:

$$\#N\langle 2,4,6\rangle\#0\#\langle 10,20,20\rangle \tag{2.13}$$

To represent a probabilistic load or typed probabilistic load, we have to rely on aleatory variables to represent the load. This can be done for both typed and untyped jobs.

2.3.5 Stochastic Time

In many cases, it may be desirable to represent intervals with non-deterministic temporal behaviour or load characterisation. In these cases, IA allows the use of aleatory variables, which follow a probability distribution, instead of scalars. It does not impose any limitation on the choice of probability distributions, and their parameters should be provided following a well established notation. For example, a normal distribution $\mathcal{N}(\mu, \sigma^2)$ with parameters mean $\mu = 2$ and variance $\sigma^2 = 1$, $\mathcal{N}(2, 1)$ can be used to denote the release time of job P, and similarly $\mathcal{N}(40, 1)$ can denote its execution time:

$$\#P\#normal(2,1)\#normal(40,1) \tag{2.14}$$

The time when job P finishes its execution is described by the convolution of two Gaussians: $\mathcal{N}(2, 1) * \mathcal{N}(40, 1)$.

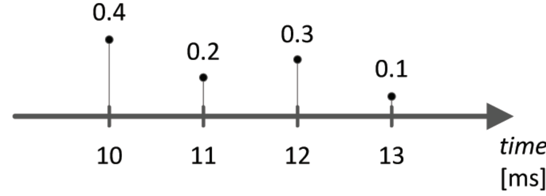

Figure 2.1 Example of a probability mass function of a discrete random variable describing a job's execution time.

It is particularly convenient to represent time as a discrete random variable described by a probability mass function (PMF), i.e., a function giving the probability that a discrete random variable is equal to a provided value. All values of a PMF should be non-negative and sum up to 1. Using this function for describing a job's execution time, the best-case execution time (BCET) and the worst-case execution time (WCET) correspond to the first and the last probability value of the distribution, respectively. Between these two extremes, the probabilities of the remaining possible execution times are described. For example, the PMF of a job execution time whose BCET and WCET equals to 10 ms and 13 ms is shown in Figure 2.1. This job can be described using IA as:

$$\#Q\#0\#pmf(10, 0.4), (11, 0.2), (12, 0.3), (13, 0.1) \tag{2.15}$$

To show how tasks with stochastic timing can be mapped to a resource, we use job T whose release time is described by discrete uniform distribution $\mathcal{U}\{0, 1\}$ and is executed in 40 time units, and job U depending on T executed in $\mathcal{U}\{1, 4\}$ time units by a notional resource $+Z_1$ with FIFO scheduling. Then:

$$\begin{aligned}&+ Z_1(\#T\#pmf(0, 0.5), (1, 0.5)\#40,\\&\#U\#T\#pmf(1, 0.25), (2, 0.25), (3, 0.25), (4, 0.25)) =\\&+ Z_1([pmf(0, 0.5), (1, 0.5), pmf(41, 0.125), (42, 0.25),\\&(43, 0.25), (44, 0.25), (45, 0.125)))\end{aligned} \tag{2.16}$$

2.4 Summary

This chapter presented the requirements for workload and platform models that are suitable to support resource allocation mechanisms in embedded, high-performance and cloud computing. Such models can be used as internal representations, allowing resource allocation mechanisms to evaluate different

allocation alternatives. A specific modelling approach has been introduced, based on an interval algebra, which fulfils the listed requirements and is amenable to compact and efficient implementations. A reference implementation of the presented algebra is available from the DreamCloud project website[1].

[1] http://www.dreamcloud-project.org