Using Quantile Regression in Neural Networks for Contention Prediction in Multicore Processors

Axel Brando
Barcelona Supercomputing Center (BSC), Spain

Isabel Serra
Barcelona Supercomputing Center (BSC), Spain
Centre de Recerca Matemàtica, Barcelona, Spain

Enrico Mezzetti
Barcelona Supercomputing Center (BSC), Spain
Maspatechnologies S.L, Barcelona, Spain

Jaume Abella
Barcelona Supercomputing Center (BSC), Spain

Francisco J. Cazorla
Barcelona Supercomputing Center (BSC), Spain
Maspatechnologies S.L, Barcelona, Spain

Abstract

The development of multicore-based embedded real-time systems is a complex process that encompasses several phases. During the software design and development phases (DDP), and prior to the validation phase, key decisions are taken that cover several aspects of the system under development, from hardware selection and configuration, to the identification and mapping of software functions to the processing nodes. The timing dimension steers a large fraction of those decisions as the correctness of the final system ultimately depends on the implemented functions being able to execute within the allotted time budgets. Early execution time figures already in the DDP are thus needed to prevent flawed design decisions resulting in timing misbehavior being intercepted at the timing analysis step in the advanced development phases, when rolling back to different design decisions is extremely onerous. Multicore timing interference compounds this situation as it has been shown to largely impact execution time of tasks and, therefore, must be factored in when deriving early timing bounds. To effectively prevent misconfigurations while preserving resource efficiency, early timing estimates, typically derived from previous projects or early versions of the software functions, should conservatively and tightly overestimate the timing requirements of the final system configuration including multicore contention. In this work, we show that multi-linear regression (MLR) models and neural network (NN) models can be used to predict the impact of multicore contention on tasks' execution time and hence, derive *contention-aware* early time budgets, as soon as a release (binary) of the application is available. However, those techniques widely used in the mainstream domain minimize the average/mean case and the predicted impact of contention frequently underestimates the impact that can potentially arise at run time. In order to cover this gap, we propose the use of quantile regression neural networks (QRNN), which are specifically designed to predict the desired high quantile. QRNN reduces the number of underestimations compared to MLR and NN models while containing the overestimation by preserving the high quality prediction. For a set of workloads composed by representative kernels running on a NXP T2080 processor, QRNN reduces the number of underestimations to 8.8% compared to 46.8% and 31.3% for MLR and NN models respectively, while keeping the average over estimation in 1%. QRNN exposes a parameter, the target quantile, that allows controlling the behavior of the predictions so it adapts to user's needs.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture

Keywords and phrases Neural Networks, Quantile Prediction, Multicore Contention

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.4

Funding This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant PID2019-110854RB-I00 / AEI / 10.13039/501100011033 and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773).

1 Introduction

Software applications control an increasing number of complex functionalities in real-time embedded products. For example, in the automotive domain, Advanced Driver Assistance System (ADAS) functionalities, like lane keeping and obstacle detection, in modern cars are implemented in software, which is going to be the central element to reach full (L5) autonomy. This trend towards smarter artificial intelligence based on-board software drives an unprecedented increase in the size of the software component of embedded real-time systems in domains like automotive and avionics. In fact, embedded real-time products already encompass software with millions of lines of code. On the other hand, the use of multicores to provide the required computing performance compounds the complexity to develop and validate multi-million line real-time software products.

During the design and development phases (DDP) engineering process, the integrator selects the configuration of the hardware by choosing values for the control registers (critical configuration settings in CAST-32A [9] jargon). Also, in order to complete the intended final configuration (IFC) [9], the integrator determines the mapping of tasks to each computational node, which in turn determines which tasks will be co-executed in the multicore and hence, compete for its resources. Those decisions are steered by the timing and functional requirements of the software functions the system is meant to support. Based on those requirements and the system schedule, the software providers are assigned a time budget for each of their software function which is meant not to be exceeded at run time. In the DDP, multiple configuration scenarios (e.g, configuration, task mapping, schedules) are assessed in order to converge to the system's IFC.

However, timing budgets are only consolidated against timing requirements in the late validation phases when timing analysis is typically performed to derive reliable worst-case execution time (WCET) bounds for each task. Capturing a timing misconfiguration so late in the development process will result in costly roll-backs in the design and implementation phases. In particular, building on optimistic timing estimates to derive and allocate time budgets to an application will result in timing violations to arise in the verification and validation stages and will require changes to the application itself and/or to the system schedule, which will cause the system to undergo once again through the V&V process. For this reason, while early figures are not meant to be as accurate as late WCET timing bounds, they are still required to conservatively over-estimate tasks' timing requirements as much as possible. Moderate over-estimation can lead to slight over-provisioning and will not jeopardize the overall system timing behavior.

Providing early timing estimates for the software functionalities is a challenging task in many ways as estimates are typically derived from previous experience on past projects or from representative early software implementations. The provision of such early estimates is even more challenging when the system is deployed on multicore platforms because tasks affect each other timing behavior causing variable access latencies when simultaneously accessing shared hardware resources. This translates into variability in their execution time, typically referred to as *multicore timing interference* or *contention impact*, which can cause 20x or more performance degradation [29, 55] and ultimately complicates the determination of trustworthy time bounds.

Exploring and assessing a large set of scenarios in the DDP requires assessing also the impact of contention in each considered configuration, where task mapping and schedule play a critical role. A naive approach to derive contention information during DDP consists in exhaustively executing each scenario on the actual board collecting evidence on how tasks affect each other timing. This approach, however, is quickly defeated by the time it takes to carry each experiment, which can limit the design space (i.e the schedules) that can be explored. Instead, a faster approach consists in developing multicore prediction models that can provide accurate estimates of tasks execution time when co-executed in a multicore.

Analysis. Multi-linear regression (MLR) models and neural network (NN) models, which have been originally developed for the mainstream domain [52, 11, 59, 34], can be adapted to the problem at hand to explore a large fraction of the design space in a short amount of time. In particular, such models could be in principle exploited to produce early timing estimates of a system as soon as a binary release of the applications is available. While these approaches produce reasonably accurate estimates of tasks' execution time, they are inherently designed to predict the average behavior of the phenomena they model, since the most accurate prediction is the one closer to the majority of cases and thus closer to average or median patterns. As we observed, it is crucial for early estimates in DDP to be over-approximating the behavior in the final configuration and we must seek for more conservative models to diminish the risk of being misled into optimistic estimates.

Proposal. On these grounds, we propose a prediction model based on quantile-regression neural networks (QRNN) that can conservatively predict the impact of multicore timing interference. QRNN aim at optimizing the quantile regression loss function which, generically, allows approximating any conditional desired quantile. This enables the user to choose the (high) quantile that best adapts to its needs. Overall, QRNN allows fast evaluation of system configuration by providing conservative, yet accurate, predictions of contention impact.

Evaluation. We show the benefits of QRNN over MLR and NN on a set of representative kernels used in artificial intelligence software for autonomous operation on an avionics representative multicore processor, the NXP T2080. Our results show that QRNN reduces the number of workloads for which time budgets under estimate (i.e. are lower than the actual multicore contention time of the task) to 8.8% compared to 46.8% and 31.3% for MLR and NN, respectively.

The rest of this work is organized as follows. Section 2 narrows down the specific multicore contention problem addressed and introduces MLR and NN. Section 3 introduces QRNN. Section 4 presents our evaluation framework. Section 5 reports on the experimental results. Section 6 covers the most relevant related work. Section 7 presents some lines that can be explored as a follow up of this work. Section 8 presents the main conclusions of this work.

2 Multicore Contention Prediction

Multicore contention modeling is a wide problem that spans several domains and stages in the software development process [43, 11, 42, 3]. We start by narrowing down the particular multicore contention problem we address and a set of properties for the resulting techniques to adhere to the specific requirements of the particular application scenario (those properties were summarized already in Section 1). The main acronyms we use are described in Table 1.

We focus on a deployment scenario in which the target multicore platform is fixed and the set of applications to be integrated in the final embedded product is known. That is, the functionality to be provided for the product is frozen and so is the software to implement it. A first release of the applications has been made so there exists an executable of each

Table 1 Main acronyms used in this work.

Acronym	Definition	Acronym	Definition
ADAS	Advanced Driver Assistance System	DDP	Design and Development Phases
DS	Data Set	EMs	Event Monitors
H	Holdout set	IFC	Intended Final Configuration
M	All the model hyper-parameters	MAE	Mean Absolute Error
MCP	Multicore Processor	MLR	Multi-Linear Regression
MSE	Mean Square Error	NN	Neural Network
QR	Quantile Regression	QRNN	Quantile Regression NN
SoC	System on Chip	TOI	Task Order Invariant
TUA	Task Under Analysis	TVE	(T)rain, (V)alidation, t(E)st set
V&V	Validation and verification		

application. Applications can suffer variations as part of the natural development process across different releases, yet preserving the same functionality. These variations include, for instance, optimizations to its performance or functional behavior. We target a *homogeneous multicore processor* in which the performance of each core is identical and the time to access any off-core resource is the same from every core. This is the case of the T2080 when its cache is not partitioned or when the L2 cache is configured so each core receives an even number of ways, as we do in this work. However, it is not the case if, for instance, L2 partitions across cores differ in size. It is nor the case for other architectures, like some models of the Intel Xeon, in which the cores and slices of the L3 cache are connected via a ring interconnect, so the time it takes a core to access a slice depends on their location in the ring.

Our work contrasts with other works that derive early estimates at the model level (e.g. Matlab) [17] and estimates "as the code is written" [18]. Others focus on scenarios where the hardware platform is not even available and compile the source code for different instruction set architectures on generic and parameterizable processor models to obtain early timing estimates on the impact of the architecture setup [19]. The majority of these approaches focus on the analysis of programs in single-core scenarios and do not address the impact of multicore contention. The works addressing multicore interference, instead, necessarily consider more mature setups where consolidated or even final software products are made available [52, 11, 61, 59, 50].

The goal of our contention modeling exercise is not producing a generic model for the target platform (T2080 in our case) that is application independent. Instead, the model considers the applications provided and contributes to speeding up the selection of the IFC.

▶ **Property 1** (Prediction speed). *DDP multicore contention models for real-time systems must be fast to enable exploring large design spaces.*

Several previous works [29, 55] show that contention may dominate the execution time of tasks running in a multicore with some applications easily suffering an increase above 2x-5x with respect to their solo execution time even for small core counts like 4 cores (corner case programs can suffer much higher slowdown). For DDP, no reference figure has been reported for the accuracy of timing predictions, which is in fact end-user and application dependent. Yet, we regard the pessimism introduced by our QRNN model (1% on average and 1.49% in the worst case) as quite reasonable for DDP. Besides it is key to produce conservative early timing estimates that tend to over-approximate the behavior in the IFC, therefore reducing the risk of producing optimistic estimates.

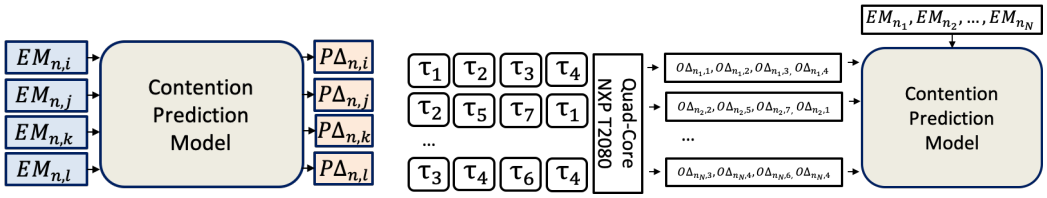


Figure 1 Contention Models Usage. **Figure 2** Contention Modelling Training.

▶ **Property 2** (Tendency to Overestimation). *DDP multicore contention models for real-time systems should tend towards overestimation to reduce the risk of experiencing timing violations too late in the development process, requiring excruciatingly onerous rollbacks and re-design.*

Trustworthy execution time bounds for a task τ_i can be derived when the task executes in isolation, ET_i^{solo}. For multicore processors (MCP), software's timing behavior also depends on the contention factor, often considered as a Δ over its execution time in isolation [1], which is expressed as $ET_i^{mcp} = ET_i^{solo} \times \Delta$.

Deriving time budgets for the multicore execution time requires estimating a bound to Δ. Contention bounds can be derived by experimentation, i.e. by running *all* potential workloads on the target board under the IFC so that the timing budget for τ_i can be expressed as $TB_i^{mcp} = ET_i^{solo} \times O\Delta_i^{max}$, where $O\Delta_i^{max}$ is the maximum observed contention impact suffered by τ_i. However, this approach is inherently time consuming and cannot be exploited for exploring non-negligible design spaces.

In terms of the number of workloads, for a heterogeneous multicore it can be computed as the permutation with repetition of all contenders A^C, where A is the number of applications in the data set (DS) that can repeat in several cores and C is the number of cores. The number of workloads reduces to $CR_A^C = \frac{(A+C-1)!}{C!(A-1)!}$ for homogeneous multicore architectures.

In terms of runs, depending on the experimentation environment in each run of a workload we can obtain the slowdown for one of the tasks in the workload or all C tasks. The former, our case, requires C runs per workload to obtain $O\Delta_{n,i}$ for each task, while the latter needs one per workload. However, for homogeneous multicores, fewer runs are required when several copies of the same task are present in the workload, in particular, $A \cdot CR_A^R = A \cdot \frac{(A+R-1)!}{R!(A-1)!}$ where $R = C - 1$ is the number of contenders.

Overall, in the general case exhaustively covering all configurations on the real board is unaffordable, even if each experiment requires just few milliseconds. With the number of cores in the multicore processors evaluated in the real-time domains increasing (e.g., the NXP Layerscape LX2160 already encompasses 16 cores), the number of workloads increases to millions.

2.1 Contention Modeling

Contention models are generally orders of magnitude faster than experimentation in the target board and can be executed in high-performance computing clusters, which allows many more parallel experiments than making executions on few target boards that can be available for experimentation. In this line, standard fully-fledged timing analysis techniques are not fit for deriving early estimates. Measurement based timing analysis requires running each workload on the target board whereas static timing analysis has known scalability issues.

[1] The main terms used in the mathematical formulation is summarized in Table 2. Instead Table 1 shows the main acronyms used in the main text.

Table 2 Main terms used in the formulation (notation) in this work.

Term	Definition	Size
A	Total number of tasks (applications) in the data set	
C	Number of cores (e.g. $C = 4$)	
DS	Data Set	
H,I	Number of output and input values	
h_i^l	i-th input/output value in layer l	
J	Number of EMs per task/core (e.g. $J = 262$)	
K	Number of EMs per workload, $K = J \cdot C$ (e.g. $K = 1048$)	
N	Number of workloads in the DS	
$EM_{n,i}$	All the EM of task τ_i in workload n when it runs in isolation	J
EM_n	All the EMs of the n-th workload (point) in the DS	K
EM_*	All the EMs for all the workloads in the DS	$N \cdot K$
$em_{n,i}^j$	The j-th EM of τ_i of workload n	1
em_n^k	The k-th EM of the n-th workload (point) in the DS	1
ET_i^{mcp}, ET_i^{solo}	Execution time of task τ_i in multicore processors and in isolation	
$O\Delta_{n,i}$	Observed contention for τ_i in workload n	1
$O\Delta_n$	Observed Contention for all tasks in workload n	C
$O\Delta_*$	Observed Contention for all workloads in the DS	$N \cdot C$
$P\Delta_{n,i}$	Predicted Contention for τ_i in workload n	1
$P\Delta_n$	Predicted Contention for all tasks in workload n	C
ϕ	Neural Network (function)	
R	$C - 1$	
TB^{mcp}	Time Budget in multicore	

Contention models produce an estimate to contention in the form of a *predicted* Δ ($P\Delta$), so that $TB_i^{mcp} = ET_i^{solo} \times P\Delta_i^{max}$, where $P\Delta_i^{max}$ is the maximum predicted contention impact. The process of deriving $P\Delta_i^{max}$ builds on several factors that capture the contention a task can suffer from and generate on co-runner tasks.

In real platforms, event monitors (EMs) provide insightful information about how a task uses shared resources, which in turn are the inherent sources of contention. EMs report metrics like access counts to resources, hit/miss accesses to cache memories, and other activities of the task on the underlying hardware.

In this work, we target the NXP T2080 [22], a quad-core MPSoC which is currently considered for certification for avionics products [48]. The T2080 comprises 262 EMs that provide insightful information on the use of resources of the analyzed application at core, shared L2, and memory levels. For a given workload, the EMs collected for each task while running on the T2080 in isolation are fed as input to the contention model.

As shown in Figure 1 for a quad-core processor, to predict the contention impact, the contention models use the EMs collected (in isolation) for all the tasks in the workload, denoted as n, constituting a function named f. The predicted contention impact for task τ_i when running in workload n, together with τ_j, τ_k, τ_l, is denoted as $P\Delta_{n,i} = f\left(EM_{n,i}, EM_{n,j}, EM_{n,k}, EM_{n,l}\right)$. $EM_{n,i} \in \mathbb{R}^J$ are all the EMs (collected in isolation) of a task τ_i where J is the number of EMs read per core ($J = 262$ in the T2080).

For the training of the model, see Figure 2, we build on the results of executing multicore workloads, generated from a set of A tasks that are executed on the available cores C on the target board (one task per core). The observed (real) contention $O\Delta_{n,i}$ for each task τ_i in each workload n is collected and used in order to compute $P\Delta_{m,i}$ in a different (unseen) workload m.

2.2 Formalization

Several techniques have been proposed in the mainstream domain for multicore contention prediction, from which we identify two families: MLR- and NN-based models [52, 11, 59, 34]. A commonality of the different models is that they create an input data set (DS) for training. Such input DS is composed by the EMs collected for several tasks used to compose the workloads and the observed slowdown when executing a subset of workloads on the target board. Reducing the subset of this input DS used for training contributes to Property 1. The input DS is shown in Equation 1:

$$\mathcal{DS} = (EM_*, O\Delta_*) = \{(EM_n, O\Delta_n)\}_{n=1}^{N} \tag{1}$$

$EM_* \in \mathbb{N}^{N\times K}$ are the EMs of all the N workloads in DS. K is the number of EMs read in total, that for the case of the NXP T2080 is $K = 4 \times 262 = 1048$, since $C = 4$. $EM_n \subset \mathbb{N}^K$ are the EMs all the tasks in the n-th workload when executed in isolation. That is, EM_n is the concatenation of the EMs of each task composing the workload when run in isolation.

$O\Delta_*$ is the observed contention for all the tasks in all workloads in the DS. Likewise, $O\Delta_n = \{O\Delta_{n,1}, \ldots, O\Delta_{n,C}\} \subset \mathbb{R}^C$ is the contention for the C executed tasks in workload n.

2.3 Multi-Linear Regression (MLR) Models

For a Tasks Under Analysis (TUA), τ_i, of the n-th workload in the DS, a multi-linear model is a linear transformation from the EM values (EM_n) to the $P\Delta_{n,i} =: \hat{y}_n$, where i is omitted because $\hat{y}_n$ is always referring to the TUA. The MLR can be formulated as follows:

$$\hat{y}_n = W \times EM_n + b \Leftrightarrow \hat{y}_n = w_1 \cdot \underbrace{em_n^1}_{x_1} + w_2 \cdot \underbrace{em_n^2}_{x_2} + \cdots + w_K \cdot \underbrace{em_n^K}_{x_K} + b \tag{2}$$

where each $em_n^k \in \mathbb{N}$ is the k-th EM in the n-th workload. As we can see in Eq. 2, we can also use $\{x_k\}_{k=1}^K$ to refer to them. $EM_n \in \mathbb{N}^K$ refers to the EMs input information for workload n. $w_k \in \mathbb{R}$ and $b \in \mathbb{R}$ are the weights to be learnt that define the linear combination between the EMs values and the predicted $\hat{y}_n = P\Delta_{n,i} \in \mathbb{R}^+$.

The goal of the MLR is to find the weights $\{W, b\}$ that minimize a certain distance function (known as the loss function) between the predicted output and the real response variable value with respect to the training split set. This minimization process can be typically performed in two different ways.

Given that the MLR is a linear combination of coefficients with the input information, the least-square estimate of W can be computed using the DS where we identify the TUA τ_i, $(EM_*, \{O\Delta_{n,i}\}_{n=1}^N)$:

$$\hat{W} = \underbrace{(EM_*{}^T \cdot EM_*)^{-1}}_{\text{Computationally expensive.}} \cdot EM_*{}^T \cdot [O\Delta_{1,i},\ O\Delta_{2,i},\ \cdots,O\Delta_{N,i}]^T, \tag{3}$$

where the T superscript denotes the matrix transpose operation, the -1 superscript refers to the inverse of the matrix[2] and $[O\Delta_{1,i},\ O\Delta_{2,i},\ \cdots,O\Delta_{N,i}]^T$ is the column vector that contains all the contention values for all the N workloads. Importantly, this way of obtaining the optimal weights has a potential drawback in most of the real-world situations, as the inverse of an $N \times N$ matrix must be computed, which has polynomial time complexity.

[2] To simplify the notation, the $\hat{W}$ matrix implicitly contains the bias b column in that case.

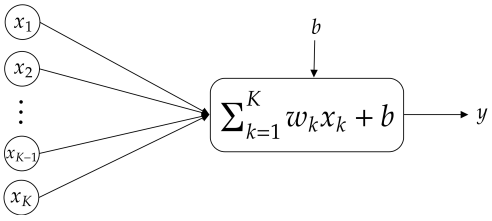


Figure 3 Multi-linear regressor.

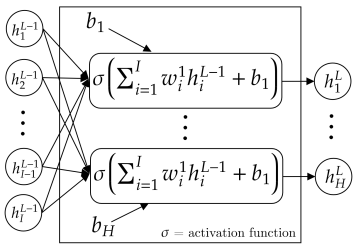


Figure 4 Dense neural layer. Given I layer-input values, $\{h_i^{L-1}\}_{i=1}^{I}$, it provides H layer-output values, $\{h_h^L\}_{h=1}^{H}$.

As an alternative approach to avoid computing the inverse matrix, we can compute this minimization process by slightly modifying the weights in the gradient direction, i.e. applying a gradient descent method. Nowadays, this differentiation process is implemented in most relevant deep learning libraries, which allows native code to be differentiated automatically [1, 45, 12, 7]. This derivative is computed with respect to a loss function, which can be the mean square error (also known as least-square estimate) that approximates the conditional mean, or an alternative function, as we will see in the next section.

2.4 Neural Network (NN) Models

A NN is also a parametric function ϕ that transforms a vector of EM_n to a predicted contention for a task under analysis τ_i in workload n, $\hat{y}_n = P\Delta_{n,i}$, i.e. it is defined as $\phi\colon\ \mathbb{R}^K \to \mathbb{R}$, transforming $EM_n \mapsto \hat{y}_n$. Instead of a single matrix multiplication such as in MLR, the NN considers several internal non-linear transformations from the input, EM_n, to produce the output value $\hat{y}_n$. Each of these transformations is known as a "layer" and combines its input values and weights to produce its output, which for the last layer is the output of the model [36, 16]. Roughly speaking, the NN combines a mixture of weights and its input values to minimize a certain distance loss function (as in the MLR case) between the predicted and the real response compared to the DS used to train the model.

Figure 4 represents one NN layer where $\{h_i^{L-1}\}_{i=1}^{I}$ represent the inputs to the layer and I is the number of neurons in the layer. In the first layer that is $I = K$ and $h_k^0 = em_n^k$ for each $k = 1, \cdots, K$ when a n-th workload is fixed. Each transformation, represented as a rectangle in Figure 4, matches Equation 2 with the addition of the non-linear activation function, denoted as σ, which allows the enhance approximation capabilities of the NN by means of the layer stacking process [13]. Each layer will produce a set of outputs, $\{h_h^L\}_{h=1}^{H}$, where H is the number of neurons in the next layer, then will be either used as inputs to the next layer or as final NN output in case of the last layer.

In probabilistic terms [26, 51], the loss function aims to approximate the conditional probability $p(Y \mid \boldsymbol{X}, M)$, where $\boldsymbol{X}$ represents the theoretically random variable that generate the input values – in our case the EM_n values, Y represents the corresponding random variable that generates the contention values $O\Delta_m$ and, finally, M is the random variable that characterizes all the hyper-parameters in the NN (including the number of layers, the type of layers, the parameters about the learning configuration, etc). Importantly, the conditional probability approximated can be affected by the hyper-parameters selection, which is, therefore, a critical step to consider for the whole process. In this probabilistic context, the common approach [31, 40] is to follow a Maximum Likelihood Estimation (MLE)

or a Maximum A Posteriory (MAP) approach to compute such conditional probability and to assume the sample mean is asymptotically normal, which consequently leads to use the following loss function:

▶ **Definition 1** (Mean Square Error). *Let $\boldsymbol{X} \in \mathbb{R}^K$ be a covariate random variable and $Y \in \mathbb{R}$ be a response random variable (i.e. the random variables that generates the input and output values, respectively). Reducing the conditional* [3] *Mean Square Error (MSE) consists in finding a function $\phi\colon \mathbb{R}^K \to \mathbb{R}$, characterized by M, which approximates the conditional mean of $p(Y \mid \boldsymbol{X}, M)$ by minimizing the loss function defined as*

$$\mathcal{L}_{MSE}(\boldsymbol{X}, Y) = \mathbb{E}\Big[\Big(Y - \phi(\boldsymbol{X})\Big)^2\Big] \approx \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \phi(\boldsymbol{x}_n)\right)^2. \tag{4}$$

For the problem at hand, $\phi(\boldsymbol{x}_n) = \phi(EM_n)$ is the evaluation of the NN over the EMs (1048 for the T2080) for a certain workload n, producing a forecast $\hat{y}_n = P\Delta_{n,i}$. Similarly, the MLR can be used into this equation as the ϕ function, i.e. $\phi_{MLR}(x_n) = W \cdot EM_n + b$.

The conditional mean is a generically good estimator, and an ideal one in scenarios where the Central Limit Theorem is applicable. However, when the approximated $p(Y \mid \boldsymbol{X}, M)$ corresponds to a heavy tailed distribution (or even has some important outliers), computing a conditional mean can lead to unreliable decisions. Then, the median can be a more stable estimator in the presence of certain outlier values. In fact, this is equivalent to repeat the previous MLE reasoning for the normal distribution but using the Laplace distribution. In such case, the conditional loss function is the following:

▶ **Definition 2** (Mean Absolute Error). *Let $X \in \mathbb{R}^K$ be a covariate random variable and $Y \in \mathbb{R}$ be a response random variable. Reducing the conditional* [4] *Mean Absolute Error (MAE) consists in finding a function $\phi\colon \mathbb{R}^K \to \mathbb{R}$ that approximates the conditional median of $p(Y \mid \boldsymbol{X}, M)$ by minimising the loss function defined as*

$$\mathcal{L}_{MSE}(\boldsymbol{X}, Y) = \mathbb{E}\Big[\Big|Y - \phi(\boldsymbol{X})\Big|\Big], \tag{5}$$

which provides results that are more robust to outliers and more interpretable than the commonly used MSE [58, 10]. However, a NN optimized with the MSE or the MAE will predict a central conditional value. Therefore, while being appropriate for deriving predictions that are close to the actual values, by definition it will not be able to compute upper-bounds. In other words, a perfect MAE estimation will have a 50% probability of having real values above and bellow the predicted point. Thus, it should not be used as a proper high-value threshold.

While computing a confidence interval around such central value is technically possible, this brings multiple challenges related to (i) the assumptions on the actual distribution for each value to predict (i.e., whether it can be regarded as Gaussian or not), (ii) computational cost to estimate the confidence interval for each predicted value across the prediction value range, and (iii) variability in the confidence reached (or tightness of the bounds) due to the arbitrary variability in the amount of data that can be available for each predicted value (e.g., for some predicted value ranges we may have very few input observations). Hence, we discard computing confidence intervals for NN based prediction.

[3] The term "conditional" is added to highlight that here the information about X should be provided to compute the error of the f with respect to Y. This also makes this definition consistent with the conditional QR definition introduced afterwards.

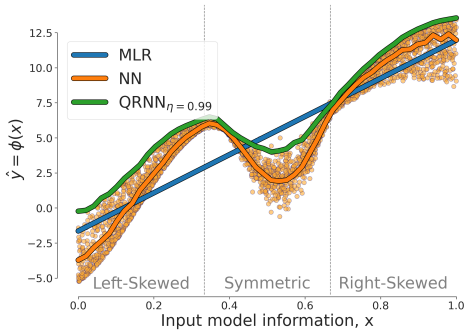


Figure 5 QRNN versus MLR and NN.

3 Quantile Regression NN

As presented in the previous section, classical NN are usually optimized using the MSE (see Eq 4) or the MAE (see Eq 5), which corresponds to estimate the conditional mean or median, respectively. In this section, we introduce the Quantile Regression (QR) method [32, 8], which allows approximating a desired quantile of the conditional distribution $p(Y \mid \boldsymbol{X}, M)$. This is visually represented in Figure 5 that for a given 1-dimension input (the horizontal axis) the goal is to predict the height of the points (the vertical axis). In particular,

- As it can be seen, MLR assumes a linear correlation between the input and output variable, which induces the prediction to be a conditional line.
- NN introduces the possibility to learn the conditional mean (or median) in a non-linear manner but, still, this cannot be used as an upper threshold.
- QRNN allows to approximate a sky-high conditional quantile in a non-linear way, which avoids strong assumptions such as linearity or symmetry between the predicted distribution, i.e. the conditional predicted distribution $p(Y \mid \boldsymbol{X}, M)$ can be skewed (such as the initial and final part of Figure 5) and the QRNN obtains a proper response.

This is useful since we can capture confidence intervals without making strong assumptions about the distribution function to approximate. The formal definition of QR depending on X is as follows:

▶ **Definition 3** (Quantile Regression). *Let $\boldsymbol{X} \in \mathbb{R}^K$ be a covariate random variable and $Y \in \mathbb{R}$ be a response random variable. Given η in the real interval $[0, 1]$, the conditional quantile regression (QR) consists in finding a function $\phi_\eta \colon \mathbb{R}^K \to \mathbb{R}$ which approximates the η-th quantile of $p(Y \mid \boldsymbol{X}, M)$ by minimizing the η-th QR loss function defined as*

$$\mathcal{L}_{QR}(\boldsymbol{X}, Y, \eta) = \mathbb{E}\Big[\Big(Y - \phi_\eta(\boldsymbol{X})\Big) \cdot \Big(\eta - \mathbb{1}[Y < \phi_\eta(\boldsymbol{X})]\Big)\Big], \tag{6}$$

where $\mathbb{1}[c]$ denotes the indicator function that verifies the condition c.

Unlike MSE Eq 4 or MAE Eq 5, the QR expressed in Eq 6 is not always a symmetric function in the sense that when the predictive system over- or under-estimates it sums to the final loss value in the same manner.

This is illustrated in Figure 6 with the representation of a QR loss function shape centered at zero considering different quantile parameters, ηs, i.e. $\{\mathcal{L}_{QR}(\boldsymbol{X}, Y, \eta_t)\}_{t=1}^{9}$ where $\eta_t = 0.1 \cdot t$, Y is always zero, the $\phi_\eta(\boldsymbol{X})$ in Eq 6 is the horizontal axis value and the vertical axis corresponds to the loss value in such conditions. As we can see in Figure 6, depending on

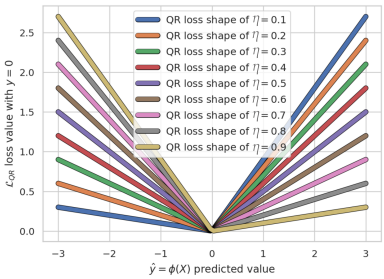


Figure 6 QR loss function shape centered at zero. The $\phi_\eta(\boldsymbol{X})$ in Eq 6 is the horizontal axis value and the vertical axis corresponds to the loss value in such conditions.

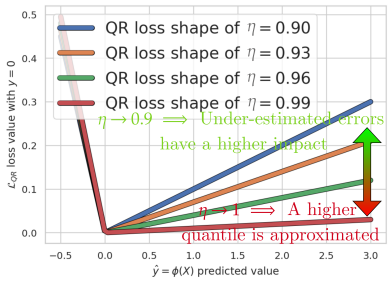


Figure 7 Behaviour of the QR loss value for high η values. When $\eta \to 1$ (red case) the under-estimated predictions are multiplied by an almost zero factor, which produces flat shape in $\eta = 0.99$.

the selected quantile η for the QR, its shape will be different. For instance, when the quantile is the 10th percentile ($\eta = 0.1$) the underestimated errors are multiplied by a lower factor than when the forecaster value $\phi(\boldsymbol{X})$ is overestimating. This is so because the increasing loss value in the positive horizontal axis in such case is clearly higher than the negative part. This effect comes from the second multiplier of the Eq 6, i.e. $\Big(\eta - \mathbb{1}[Y < \phi_\eta(\boldsymbol{X})]\Big)$, which means that when the real value Y is strictly lower than the predicted $\hat{y} = \phi_\eta(\boldsymbol{X})$, then the indicator function takes the value of 1, otherwise it is 0. In the presented case, as $\eta = 0.1$, it means that when $\phi_\eta(\boldsymbol{X})$ is underestimated, the difference between the real value and the predicted value will be multiplied by 0.1, which justifies the lower increasing in the negative part of the blue line of Figure 6 as far as the predicted value is from the (here, always zero) real value. Contrastingly, the positive part will be multiplied by -0.9, which produces a higher increasing as much far as the predicted value is far from the real one but also it ensures that the loss function is always positive.

3.1 Predicting Sky-high Quantiles using QR

The problem at hand requires to have a proper sky-high quantile to ensure most of the predictions are below. However, when we use the QR Eq. 6 to predict a quantile $\eta \to 1$, i.e. when the condition $Y < \phi_\eta(\boldsymbol{X})$ is satisfied, the whole expression of Eq. 6 tend to be zero due to its second factor $\Big(\eta - \mathbb{1}[Y < \phi_\eta(\boldsymbol{X})]\Big)$ approximates to zero. This is an issue because it implies that any overestimated point by the NN ϕ_η for a certain $\eta \approx 1$ almost does not contribute to the expected error. Hence, higher erroneous values that are overestimated are neglected as lower erroneous values. This has a critical effect in the optimization process because high differences, $Y - \phi_\eta(\boldsymbol{X})$, will not be taken into account and it will cause the solution to be unavoidably unstable or wrong.

To solve this issue we propose a solution that considers two edges (represented in two colors of the vertical arrow of Figure 7): First, we want to predict the higher quantile possible to reduce the number of under-estimated cases. And second, we want to avoid the neglecting issue that appears when we are predicting $\eta \to 1$ quantiles. Therefore, (1) the NN model will predict simultaneously several quantiles (including farther and closer quantiles to 1), and (2) all these quantiles will be linearly related with a common previous hidden representation, which means that they will share the last neural network layer.

As we described previously, the closer the quantile value η gets to 1, the worse the effect of avoiding overestimated errors will be. Therefore, (1) considering several (a fixed set of) quantiles that tends to 1 and (2) that preserves a linear relation between a common previous representation (i.e. all the simultaneously predicted quantiles shares the same penultimate

NN layer), we can ensure that lower sky-high quantiles of the set will avoid the neglecting issue when it appears and, at the same time, the higher approximated quantiles will try to push for obtaining a higher extreme upper bound. Combinedly, (1) and (2) allow the model to obtain a balanced solution in both senses. In Figure 8 is described the change we require to perform to predict several quantiles using a single NN model.

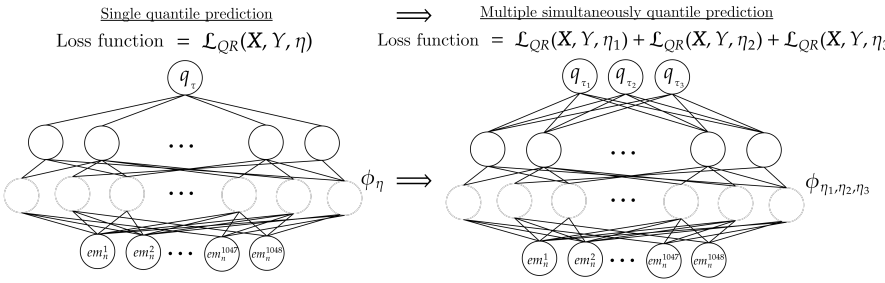


Figure 8 Transformation from a single QR NN model to a multiple QR NN model that simultaneously predicts several quantiles.

As shown in Figure 8, the number of inputs and even the internal number of hidden layers and hidden neurons are preserved (as long as it is enough complex to approximate the desired function). The only we need to change is the number of outputs, represented as the $\{q_{\eta_o}\}_{o=1}^{3}$ last neurons in Figure 8, by changing the number of neurons of the last layer. Each of these neurons will be optimized using a different specific QR-loss function, shown in Eq. 6, for the corresponding quantile η value.

3.2 Task Order Invariance

We set an additional constraint on our multicore contention prediction models that typical ML models do not provide. In particular, the predicted contention for a given task must be the same under any permutation of its contenders.

▶ **Property 3** (Task Order Invariance (TOI)). *For homogeneous multicores, multicore contention models for a given task τ_i in a given workload n must provide the same estimate ($P\Delta_{n,i}$) regardless of the core where τ_i runs and any permutation of its contenders.*

Considering a workload n consisting in tasks τ_i, τ_j, τ_k, and τ_l, the contention suffered by each of these tasks must not be affected by the core in which tasks executes. Therefore, it must not be affected where the task under analysis and the other tasks in the workload are. Specifically, for a given task under analysis (TUA) τ_i in a certain workload n:

$$P\Delta_{n,i} = f\left(EM_{n,i}, EM_{n,j}, EM_{n,k}, EM_{n,l}\right) = f\left(EM_{n,j}, EM_{n,i}, EM_{n,k}, EM_{n,l}\right) = \\ = \cdots = f\left(EM_{n,l}, EM_{n,k}, EM_{n,j}, EM_{n,i}\right)$$

▶ **Definition 4** (Task Order Invariant). *Given a four-core multicore contention forecaster ϕ and four sets of EMs, $\{EM_{n,i}, EM_{n,j}, EM_{n,k}, EM_{n,l}\}$ where τ_i is the TUA, this forecaster can be considered a Task Order Invariant (TOI) predictor if Equation 7 holds regardless of the order in which EM sets appear in the parameter list, which means disregarding the core mapping of both the TUA and the contenders, as long as they run in parallel.*

TUA position can change.

$$\phi(\, EM_{n,i} \,,\, EM_{n,j}, EM_{n,k}, EM_{n,l} \,) = \phi(\, EM_{n,l}, EM_{n,k}, EM_{n,j} \,,\, EM_{n,i} \,) \tag{7}$$

Same contenders with different order.

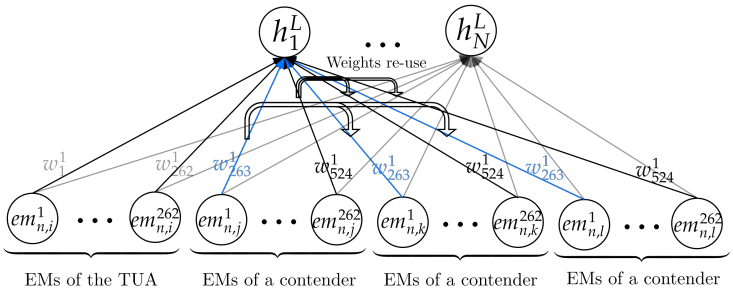


Figure 9 Example definition of the first layer of the NN or the whole MLR model for the T2080 to ensure they provide the same output regardless the order of the contenders. In short for all the $l = 1, \cdots, O$ output neurons of the L-layer, $w^l_{j+1*262} = w^l_{j+2*262} = w^l_{j+3*262}$. Note that $w^l_j, j = 1, \cdots, 262$ are the EMs of the TUA in this case τ_0.

3.2.1 Existing Models

For MLR, in Eq. 2 the set of K weights are divided in C groups/cores of J EM each and the j-th position for each group is the same EM (e.g. data cache misses). Each of these EM has associated a different weight $w_i, w_{i+J}, \cdots, w_{i+(C-1)\cdot J}$, which can get a different value as part of the training. It then follows that the order in which the contenders tasks are passed to the model affect its results. Note that the j-th counter of task τ_i in workload n (i.e. $em^j_{i,n}$) is the $j \cdot (i-1)$-th counter in the workload ($em^{j\cdot(i-1)}_n$) for $j = 1, \cdots, J$ and $i = 1, \cdots, C$.

For NN, as it can be in in Figure 4, the same logic applies. In the first layer, $h^0_k = em^k_n$ and there is a different weight associated to each h^0_k with each of then potentially taking a different value. As a result, the order of the contender tasks matters.

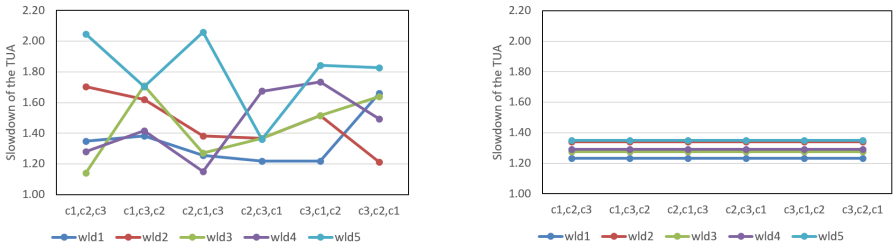
As an illustrative example, Figure 10a shows the contention estimates obtained with the NN model for different permutations of the 3 contenders (C_1, C_2, and C_3) for 5 arbitrary workloads. We can see that depending on the of the contenders (shown in the x-axis) the produced estimate varies showing that NN does not fulfill Property 3 (nor does MLR). Also, across permutations the predictions can vary significantly.

3.2.2 Achieving TOI

In order to achieve TOI, we propose a method that can be commonly applied to any presented NN model as well as the MLR. The method consists in sharing the weights across the same EM in all cores where contenders run, as it is shown in Figure 9. Particularly, the TOI dense layer will satisfy the following expression (τ_i is the TUA in workload n):

$$h^L_l = \sigma\Big(w_1 \cdot em^1_{n,i} + w_2 \cdot em^2_{n,i} + \cdots + w_J \cdot em^J_{n,i} + \sum_{k=1}^{3} \underbrace{(w_{J+1} \cdot em^1_{n,k} + w_{J+2} \cdot em^2_{n,k} + \ldots + w_{2\cdot J} \cdot em^J_{n,k})}_{\text{The new weight-sharing part to be TOI.}}\Big), \tag{8}$$

where the shaded area is the new weight-sharing part and σ is the non-linear function or activation function introduced in Section 2, such as the REctified Linear Unit (RELU). Importantly, when we want to produce a TOI MLR, this activation function does not appear and, therefore, the Eq. 8 without the σ constitutes the overall model instead of a single layer like in the NN case.



(a) Standard model prediction.

(b) TOI model prediction.

Figure 10 Contention predictions of a standard model with and without the TOI for a given workload under different contender permutations.

For the same workloads and contender permutations showed in Figure 10a, we produce predictions with a NN in which the weight of the same EM in the different cores is shared. The net result is that contention predictions are invariant to the permutations of the contenders as shown in Figure 10b.

4 Experimental Setup

Our experimental setup includes the target platform and its configuration (Section 4.1), the kernels we use to compose workloads (Section 4.2), the particular experiments we carried out (Section 4.3), and the configuration used for the specific contention models, such as the number and type of layers in NN and QRNN (Section 4.4).

4.1 Hardware Platform

We perform our experiments on a NXP T2080 Reference Design Board [22]. The T2080RDB includes a NXP T2080 System on Chip (SoC) for which an avionics multi-core certification case has been started [48]. The T2080 SoC includes a CPU cluster with 4 e6500 cores [21], see Figure 11. Each core has its own private 32KB 8-way instruction and data caches. In each core a core-cluster interface (CCI) serves as the bridge for data and instruction cache requests from and to the L2. The L2 cache is shared between all the cores. The core cluster, the DDR memory controller, the DMA and other I/O controllers are connected via the CoreNet coherence fabric (CCF). In this work, we focus on the main path from cores to main memory, and do not address the potential contention arising in the I/O. In order to favor time predictability, we configure the T2080 as follows:

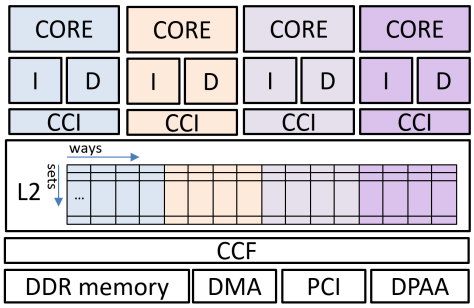


Figure 11 Simplified block diagram of the NXP T2080 SoC.

- *Shared L2 cache*: shared caches are one of the main sources of contention in modern SoC. In order to favor predictability and simplify the work of timing analysis tools in the validation phase, shared last level caches are typically partitioned via software (set partitioning) or via hardware support (way partitioning). The T2080 allows each core to be assigned a subset of the L2 cache ways. In our experiments, we assign each core a disjoint set of 4 ways by properly configuring the L2 cache control registers.
- *Hyper-threading support*: the hyper-threading capability in each e6500 core has been deactivated. From a multicore contention perspective, tasks running in hyper-threading mode in a core share not only first level instruction and data caches but also some core resources, potentially affecting each others performance significantly.

4.2 Workloads

We use several kernels (basic operators) that are commonly used in machine learning libraries, which in turn, are used for many operations of autonomous driving and ADAS software, from perception and detection to planning and control. For instance, matrix multiplication is a central element of YOLOv3 machine learning library [53] and radar applications [23, 54], and has been shown to account in some scenarios for 67% of YOLO's execution time [20]. The kernels we use in this work are:

- Matrix Multiplication is one of the most common kernels for many functionalities like object detection and path planning in autonomous navigation, and covariance matrix computation in radar applications. We experimented with two versions: (1) basic (MMB), and (2) optimized (MMO), which "tiles" input matrices to improve data locality.
- (3) Matrix Transpose is another quite common matrix operator
- (4) Matrix Transpose Multiply combines matrix transpose of the second matrix and multiplication of both of them. It is used, for instance, for certain internal operations in NN [24] and for covariance matrix computation in radar applications;
- (5) Rectifier is an activation function in neural networks taking the positive value of its argument or zero when it is negative;
- (6) Image-to-Columns function is used for transforming raw RGB images into matrices in the format needed by neural networks;
- (7) Vector-multiply-add is a type of linear algebra operator.

We also used a set of basic operators with different data types and precisions. In particular, we use (8-9) vector addition with integer long and with floating-point double precision. For (10-11) vector multiplication and (12-13) vector division we also use integer long and fp double types. We also use (14) quicksort sorting algorithm on a randomly ordered array. All these operators are the building blocks for other basic functionalities in machine learning libraries and radar applications.

Also autonomous driving frameworks like Apollo use deep and recurrent neural networks in several stages like object detection, object tracker, etc [47]. Each of those stages works with different input sizes. In the same vein, radar applications typically operate relatively small matrices in comparison to camera-based and LiDAR-based object detection applications. In order to capture this scenario in which input data may or may not fit in the different cache levels, we have developed 3 variants of our kernels: one fitting in DL1, one fitting in 4-ways of the L2, and one going frequently to memory. Overall, we use $A = 42$ kernel variants.

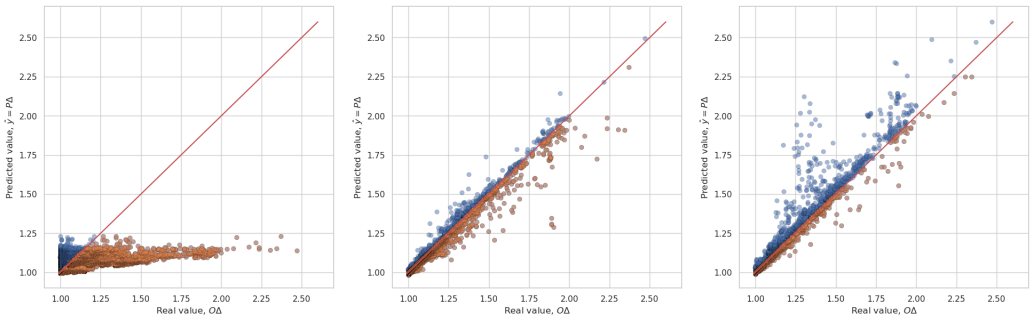
4.3 Experiments

We start by running each kernel in isolation and collect all EMs. EMs are read via performance monitoring counters. While the number of EMs can easily be over hundred, the number of performance monitoring counters is usually below 8 (it is 5 for the T2080). Hence, in order to read all 262 EM, we carried out 53 runs in each of which we read 5 different EMs.

As a second step, we generated a data set with 4-kernel workloads randomly selected from the set of kernels described in the previous section. We run each workload on the target board. For the T2080, it has been reported that the same multicore run is subject to execution time variation [57]. This happens despite exercising a tight control on the experimental setup ensuring that in every run the state of the caches and TLBs is reset. However, other non-resettable resources retain some state that changes across runs. To capture this variability, we repeated each experiment run 50 times and take the high watermark execution time. Note that our experiments show that this variability occurs for multicore executions. For single-core executions, the variability of a repeated measured EM is below its 1% value.

In each modeling experiment, we randomly split the DS into different (sub)sets used to train and validate the models, as commonly done in machine learning literature [15, 44]. The Training set (T) includes the subset of data that will be used to optimize the supervised model. The Validation set (V) includes the subset of data that will be used to decide when to stop optimizing the supervised model. The tEst set (E) includes the subset of data that will be used to verify the quality of the performance or accuracy of the supervised model to generalize.

In all experiments, the percentage of workloads of the overall DS for T and V is 17% and 2%, respectively. The remaining 81% is used as E. Note that only T and V are used to determine the models, while the remaining E of the DS is used in this work to show the accuracy of each model and that hence will not be needed in reality to generate the model. It is also worth noting that when generating the T and V sets, we make sure that the number of times each kernel is used in as TUA is the same. The contenders are generated randomly.



(a) (MAE) MLR prediction. **(b)** (MAE) NN prediction. **(c)** QRNN prediction.

Figure 12 $P\Delta$ vs $O\Delta$ for MLR, NN, and QRNN.

4.4 Model Configuration and Libraries

The current forecasting context is a single value regression. In the presented problem, no time-based input information exists and, therefore, it is not required to encode it into the model by using recurrent NN layers [27, 60]. Similarly, the EMs used as inputs are mostly counters with their own meaning, hence not having *spatial proximity information*.

Consequently, convolutional NN layers [35, 2], which are specially designed for images or text information, are not appropriate in our context. Hence, the considered NN models used are based on fully-connected or dense layers [26], which is a Multi-Layer Perceptron (MLP) model with the ReLU non-linear activation function [33] in the hidden layers. Additionally to the common MLP, all the models presented in this work have the first layer customized by following Section 3.2 to ensure TOI.

Regarding implementation, all the models were developed in TensorFlow [1] using the Keras sub-library [12]. We used a greedy search algorithm [37, 30] to select a proper NN architecture including between 1 and 6 hidden layers considering 100, 200, 300, 400 and 500 neurons. Each model is trained with early stopping, hence requiring different time to train, but none exceeds 10 minutes of training. The final selected architecture includes 3 layers of 300 neurons each (including the TOI-layer) for the standard NN (with a single output), and the same architecture for the QRNN model. Therefore, given that both hidden architectures are analogous for those NN models (except in the case of the last layer), they have similar general function approximation capabilities [26].

5 Experimental Results

We start by comparing the accuracy of the estimates provided by each technique. In Figure 12 we see three charts corresponding to the accuracy results of MLR, NN, and QRNN with a target quantile[4] $\eta = 0.9$. Each point represents a particular workload n with TUA τ_i. The x-axis shows the slowdown observed for the TUA $O\Delta_{n,i}$ and the y-axis the predicted contention $P\Delta_{n,i}$. The bottom-left top-right diagonal highlighted with a red line shows the ideal scenario in which the predicted value matches the observed one. As we can see MLR (Figure 12a) underestimates (i.e. $P\Delta_{n,i} < O\Delta_{n,i}$) for many workloads, see points below the red line. NN (Figure 12b) produces much tighter estimates, yet many of them underestimated. QRNN (Figure 12c) corrects this situation significantly reducing the number of underestimated cases while maintaining high-accuracy.

This is quantified in Figure 13 where we see that MLR underestimates in 46.8% of the cases, NN 31.3% and QRNN reduces it down to 8.8%. In terms of amount of over- and underestimation, Figure 14 shows x-th largest overestimated and underestimated values (referred to as the x-th LOE and LUE value, respectively) of each model. For instance, the (1st) largest overestimated (LOE) value is largest value of $P\Delta_{n,i}/O\Delta_{n,i}$ when $P\Delta_{n,i} > O\Delta_{n,i}$. Likewise the (1st) largest underestimated (LUE) value is the one with the (1st) largest distance to value 1.0 when $P\Delta_{n,i} < O\Delta_{n,i}$, i.e. the (1st) smallest value.

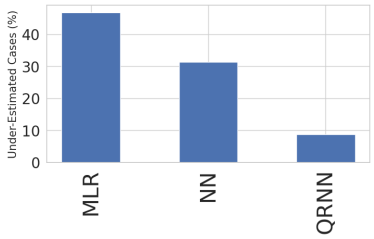


Figure 13 Breakdown.

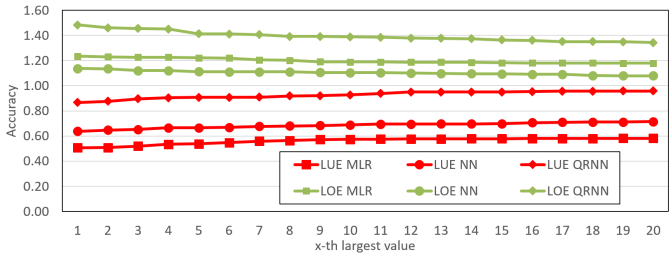


Figure 14 Largest Over- and Underestimation values.

[4] As it is described in Section 3.1, the proposed QRNN model predicts three different quantiles, $\{0.7, 0.9, 0.99\}$, to support the $\eta = 0.9$ prediction and avoid negligence issues of sky-high quantiles.

In terms of LUE, MLR (red squares in Figure 14) produces the worst results with underestimates below 0.6 even for the 20th largest value. NN (red circles in Figure 14) produces similar underestimates that remain below 0.7 for the 20th LUE. Instead, QRNN (re diamonds in Figure 14) reduces this significantly with underestimation very close to 1.0: at most 0.83 and rapidly going to 0.96 for the 20th LUE.

In terms of LOE, QRNN produces worse results than MLR and NN. Yet, the LOE values are moderate going from 1.48 (largest) to 1.34 (20th largest). It follows that, QRNN fulfills Property 2 by tending to overestimation while keeping estimates tight.

5.1 Impact of η

In order to assess the impact of η in the results besides the value used so far $\eta = 0.9$, we evaluate other values of η. In particular, inspired in the analysis of the distributional tails, we select an exponential decay as follows $\{\eta_{it} = 1. - 0.01 * 2^{it}\}, it \in [1, 2, 3, 4]$, which takes values from 1 to 0.8, i.e. $\eta \in [0.99, 0.98, 0.96, 0.92, 0.84]$. Figure 15 and Figure 16 evaluate underestimated cases and $x = 15$-th LUE and LOE for different values of η (similar trends are obtained for other values of x like 10 and 20). We can see that, as η increases, the number of under underestimated cases tends to decrease from over 16% for $\eta = 0.8$ to less than 1% for $\eta = 0.99$. In terms of LUE and LOE, both increase. In the LUE case, this means reducing the underestimation and in the LOE case increasing overestimation. Overall, changing the quantile η, $QRNN_\eta$ provides to the end user a mechanism to control over- and underestimation in the way it better adapts to his/her needs.

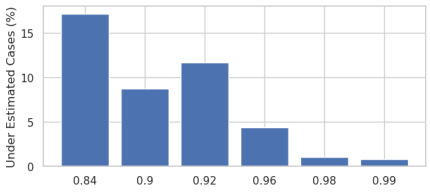


Figure 15 Underestimated cases.

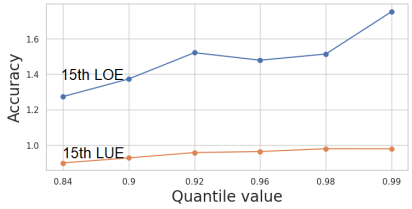


Figure 16 15th LOE and LUE values.

5.2 Different random partitions of the DS

Results so far have been shown for a particular breakdown of the workload space into TVE sets. For $\eta = 0.99$ Figure 17 shows the results for 100 experiments in terms of the number of underestimated cases (UEC) and the 15th LOE and LUE values (15LOE and 15LUE, respectively). In each of the 100 experiments we randomly selected the workloads in TVE sets as described in Section 4.3 and re-trained all models.

As shown, there is some variability due to the fact that, in the DS across the different workloads, the randomly generated contenders do not properly represent the tasks in the DS. This is illustrated in test number 2, for which we see large variations for NN results in high 15LOE and 15LUE. Despite these variations, more notable in 15LOE for QRNN and UEC for NN, the main conclusions remain the same with QRNN tending to overestimation and almost no underestimation (15LUE is very close to 1 and UEC to 0 for QRNN).

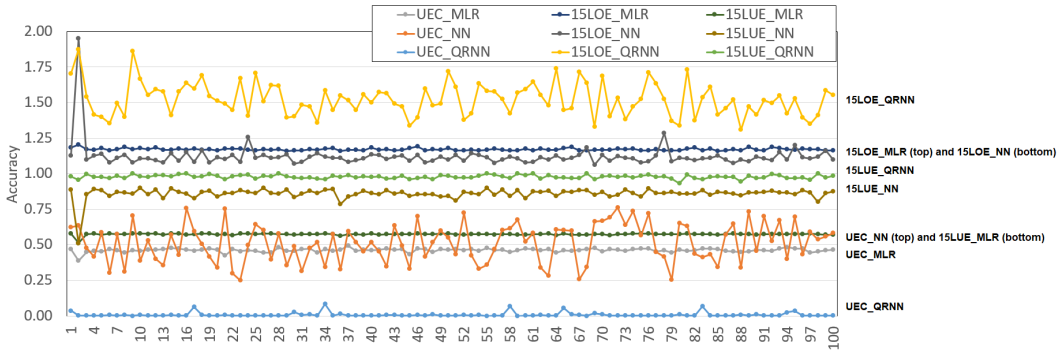


Figure 17 Number of over-estimated cases and the 15th LOE and LUE value for 100 experiments.

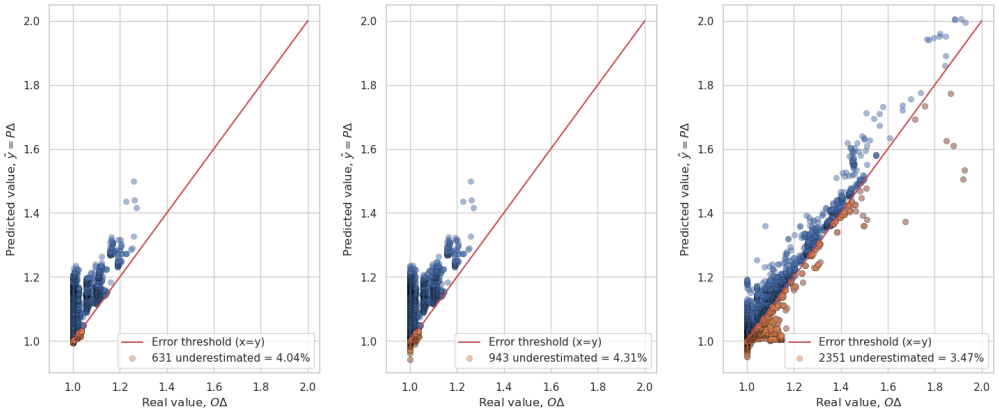
5.3 Changes to the applications

In the application scenario we address, as introduced in Section 2, the applications composing the software component of the embedded real-time product are fixed. However, as part of the usual incremental development process, applications can suffer some updates. This might cause some improvements to the functional behavior of the application, and more importantly to us, it can change the usage of hardware shared resources. Obviously, the more the updated versions differ in their resource usage from previous versions, the more challenging it is to produce tight estimates. In order to capture this situation, we assess the impact of varying some of the applications in the task set. In particular, we consider a scenario in which the QRNN model has been trained with a basic version of matrix multiplication (MMB) that is afterwards optimized resulting in an optimized version of matrix multiplication (MMO). This implies having a holdout set (H) for QRNN, which is an isolated subset of data that will be used to check the capabilities of the model to predict scenarios that could not be observed or are slightly different.

In practical terms, this means that we remove MMO (only use MMB) from TVE so that the weights of the resulting QRNN do not factor in MMO (but MMB). To assess the impact on accuracy, we query QRNN with MMO, so we make contention predictions for MMO, which was not used in TVE. Figure 18a shows the results when we use as contenders of MMO only kernels already used in TVE and in Figure 18b when we also use MMO as contender. In both cases QRNN behaves quite well, keeping both, the number of underestimated cases low and the overall prediction accuracy quite tight. Figure 18c shows the results of the predictions for the kernels already in TVE (that is, all but MMO) when the set of contenders contain at least one copy of MMO. The same trend holds with low number of overestimations (below 3.5%) while accuracy is kept high.

5.4 Execution time requirements

In order to assess the speed of the inference of the models, we perform experiments with TensorFlow software library v2.3.0 on an AMD Ryzen 9 3950X Processor. Our results show that MLR performs $1.25X10^7$ predictions per second while NN and QRNN $1.48X10^5$ and $1.46X10^5$ per second respectively when running the library in a single core. While MLR is faster, we have seen that its accuracy results are rather poor. When we use all 16 cores, performance for NN and QRNN scales perfectly so that we can make more that $2.3X10^6$ predictions per second. Overall, a wide design space can be covered with the presented QRNN model, hence achieving the Property 1.



(a) MMO vs existing kernels. **(b)** MMO vs all kernels. **(c)** Existing vs all kernels.

Figure 18 QRNN predictions when MMO is used instead of MMB.

6 Related Works

The importance of capturing timing requirements already in DDP of embedded real-time systems is widely recognized [17, 5, 42, 19, 18, 49]. Existing approaches mainly focus on deriving and exploit early timing bounds to guide architectural exploration and steer design decisions: early WCET figures are obtained either by exploiting simulation of software designs or partial implementations thereof [17, 18, 3], or by using actual observations to abstract away from certain architectural features [42, 19]. In fact, the main objective of these works is providing quick estimates at the expense of loosening accuracy. However, those works do not address the impact of multicore contention, which is the main focus of our work.

ML techniques are widely used in several fields of computer science [46, 41], for the design, optimization, and simulation of computer systems. In the context of time critical systems, approaches building on statistical and machine learning techniques have been proposed to model uncertainties in deriving timing bounds for tasks running in single cores, both in early and late development stages. In [5, 25] statistical approaches are leveraged to model uncertainties in timing estimation rather than predicting WCET figures, but do not apply to multicore systems. Still on single core systems, a hybrid approach using ML to build the timing model within a standard static WCET analysis framework has been proposed in [4].

Preliminary approaches for deriving early WCET estimates based on machine learning techniques are proposed in [6, 28], where relevant code-level constructs such as arithmetic and memory operations, and conditionals, are used to train simple regression models for WCET computation. Our approach builds on hardware events rather than source code, and focuses on predicting multicore contention instead.

The use of machine learning techniques to model the impact of multicore contention has been mainly investigated from the perspective of high-performance systems in the mainstream domain [52, 11, 61, 59, 50]. These works aim at preventing average performance degradation and implement linear regression models to predict the impact of multicore contention. In the scope of real-time systems, non-linear regression with *random forest* has been recently assessed for predicting multicore interference [14]. In contrast with our work, the proposed approaches are mainly oriented towards average performance estimation and in all cases, the underlying models do not prioritize overestimation as a fundamental requirement to avoid timing misconfiguration to arise in the later development stages. Instead, we introduce the use of QRNN for improved and tunable – conservative – prediction accuracy.

Few works address the need for conservative timing estimates in DDP from the perspective of real-time systems. Early modeling of multicore contention in time-critical systems is addressed in [19, 49], where an empirical approach is presented to capture the worst-case impact of contention on a given platform, which is later used to inflate the execution time of a task in isolation, to obtain a bound guaranteed to hold under any workload. While closer in spirit to our method, the inflated execution time estimates can result in up to 20x bigger than programs' execution time in isolation even for 4-core setups, which makes them barely useful for DDP exploration. The work in [34] proposed an NN approach for deriving early WCET bounds using the program features collected at the source code level by applying static WCET analysis methods. We instead build on hardware events to model multicore contention and use QRNN to force accurate but conservative timing estimates.

7 Future Work

In terms of future work, we identify several research opportunities. First, we have used all the EMs available in the underlying platform. However, while EMs provide insightful information about the activities in the processor, a subset of them could suffice to capture the most relevant factors affecting multicore contention. In this line, techniques like principal component analysis could be used for selecting relevant EMs, allowing a dimensionality reduction of the contention models and therefore faster and more accurate models.

It is also the case that, so far, we have used EM collected during the execution of each application in isolation. In fact, contention models could also build on EMs collected during the execution of a subset of the workloads, as this would provide more accurate information about how a given application reacts to contention. The other side of the coin is that experimentation time would increase and training would be more complex. We are interested in exploring trade-offs between accuracy and complexity.

Focusing on the bigger picture, contention models are to be queried by system-level optimization models to explore, for instance, different task schedules based on the expected contention. System-level optimizers require modeling how tasks overlap in time and how events are distributed within each task execution. The latter aspect may call for collecting EMs within tasks phases rather than end to end. Still at system level, contention models can also be extended to cover other devices beyond memory for which activity descriptors – e.g. in the form of EMs or system-level metrics – are available.

Finally, the present article focuses on NN models, which are just one of the state-of-the-art ML models for regression purposes. NN models are not the only ones designed to learn a conditional quantile using QR. For instance, decision trees [39], random forests [38, 14] or Gradient Boosting [56] methods can be used with analogous purposes. As future work, a comparison between extra QR-based models can be performed to assess the functional approximation capabilities of each model in the current forecasting problem.

8 Conclusions

Early contention estimates in multicore setups tightly upper-bounding real contention reduce the risk to detect timing misconfiguration in late phases of the development process that would result in costly changes to the system design and/or implementation. We use quantile-regression neural networks (QRNN) as an alternative to common NN and multi-linear regression (MLR) models to drastically decrease scenarios with contention underestimation while preserving tightness. Moreover, our approach achieves task order

independence to provide identical contention estimates for equivalent task permutations with identical contention in practice. Both, tendency towards – tight – overestimation and task-order independence are, in our view, fundamental properties for the use of contention models in real-time systems, besides prediction speed. Our results show that QRNN consistently reduces the number of underestimated contention bounds with respect to NN and MLR while its η parameter allows the user to find the tradeoff that fits best his/her needs.

References

1 Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. doi:http://10.5281/zenodo.4724123.

2 Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. IEEE, 2017. doi:10.1109/ICEngTechnol.2017.8308186.

3 Peter Altenbernd, Jan Gustafsson, Björn Lisper, and Friedhelm Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52(6):731–760, 2016. doi:10.1007/s11241-016-9250-7.

4 Abderaouf Nassim Amalou, Isabelle Puaut, and Gilles Muller. WE-HML: hybrid WCET estimation using machine learning for architectures with caches. In *RTCSA 2021 - 27th IEEE International Conference on Embedded Real-Time Computing Systems and Applications*, pages 1–10, Online Virtual Conference, France, August 2021. IEEE. URL: https://hal.inria.fr/hal-03280177.

5 Jakob Axelsson. A method for evaluating uncertainties in the early development phases of embedded real-time systems. In *RTCSA*, 2005. doi:10.1109/RTCSA.2005.12.

6 Armelle Bonenfant, Denis Claraz, Marianne de Michiel, and Pascal Sotin. Early WCET Prediction Using Machine Learning. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 5:1–5:9, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2017.5.

7 James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL: http://github.com/google/jax.

8 Axel Brando, Joan Gimeno, Jose A Rodríguez-Serrano, and Jordi Vitrià. Deep non-crossing quantiles through the partial derivative. *International Conference on Artificial Intelligence and Statistics*, 2022. URL: https://proceedings.mlr.press/v151/brando22a.html.

9 Certification Authorities Software Team. *CAST-32A Multi-core Processors*, 2016.

10 Tianfeng Chai and Roland R Draxler. Root mean square error (rmse) or mean absolute error (mae)?–arguments against avoiding rmse in the literature. *Geoscientific model development*, 7(3):1247–1250, 2014. doi:10.5194/gmd-7-1247-2014.

11 Yuxia Cheng, Wenzhi Chen, Zonghui Wang, and Yang Xiang. Precise contention-aware performance prediction on virtualized multicore system. *Journal of Systems Architecture*, 72:42–50, 2017. Design Automation for Embedded Ubiquitous Computing Systems. doi:10.1016/j.sysarc.2016.06.006.

12 François Chollet. Keras, 2015. URL: https://github.com/fchollet/keras.

13 Francois Chollet. *Deep Learning with Python*. Manning Publications Co., USA, 1st edition, 2017. doi:doi/10.5555/3203489.

14 Cédric Courtaud, Julien Sopena, Gilles Muller, and Daniel Gracia Pérez. Improving prediction accuracy of memory interferences for multicore platforms. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 246–259, 2019. doi:10.1109/RTSS46320.2019.00031.

15 Harris Drucker, Christopher J Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. Support vector regression machines. *Advances in neural information processing systems*, 9, 1996. doi:10.5555/2998981.2999003.

16 Oliver Duerr, Beate Sick, and Elvis Murina. *Probabilistic Deep Learning: With Python, Keras and TensorFlow Probability*. Manning Publications, 2020. URL: https://tensorchiefs.github.io/dl_book.

17 Raimund Kirner et al. Fully automatic worst-case execution time analysis for matlab/simulink models. In *ECRTS*, 2002.

18 Trevor Harmon et al. Fast, interactive worst-case execution time analysis with back-annotation. *IEEE Trans. Industrial Informatics*, 8(2), 2012.

19 C Ferdinand, R Heckmann, D Kästner, K Richter, N Feiertag, and M Jersak. Integration of code-level and system-level timing analysis for early architecture exploration and reliable timing verification. In *ERTS2 2010, Embedded Real Time Software & Systems*, 2010.

20 Fernando Fernandes dos Santos, Lucas Draghetti, Lucas Weigel, Luigi Carro, Philippe Navaux, and Paolo Rech. Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 169–176. IEEE, June 2017. doi:10.1109/dsn-w.2017.47.

21 Freescale semicondutor. e6500 Core Reference Manual. https://www.nxp.com/docs/en/reference-manual/E6500RM.pdf, 2014. E6500RM. Rev 0. 06/2014.

22 Freescale semicondutor. QorIQ T2080 Reference Manual, 2016. Also supports T2081. Doc. No.: T2080RM. Rev. 3, 11/2016.

23 Jonah Gamba. *Automotive Radar Applications*, pages 123–142. Springer Singapore, Singapore, 2020. doi:10.1007/978-981-13-9193-4_9.

24 Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016.

25 P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 580–588, 2001. doi:10.1109/DATE.2001.915082.

26 Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

27 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

28 Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx. A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 5:1–5:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2018.5.

29 Javier Jalle, Mikel Fernandez, Jaume Abella, Jan Andersson, Mathieu Patte, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Bounding resource-contention interference in the next-generation multipurpose processor (ngmp). In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS^2)*, 2016.

30 Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956, 2019. doi:10.48550/arXiv.1806.10282.

31 Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? *Neural Information Processing Systems, 2017*, pages 5580–5590, 2017.

32 Roger Koenker and Kevin F Hallock. Quantile regression. *Journal of economic perspectives*, 15(4):143–156, 2001. doi:10.1257/jep.15.4.143.

33 Alex Krizhevsky and Geoff Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7):1–9, 2010.

34 Vikash Kumar. Deep neural network approach to estimate early worst-case execution time. In *Proceedings of Digital Avionics Systems Conference (DASC)*, 2021.

35 Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

36 Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015. URL: https://www.nature.com/articles/nature14539.

37 Guohao Li, Guocheng Qian, Itzel C Delgadillo, Matthias Muller, Ali Thabet, and Bernard Ghanem. Sgas: Sequential greedy architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1620–1630, 2020.

38 Nicolai Meinshausen. Quantile regression forests. *Journal of Machine Learning Research*, 7:983–999, December 2006. doi:10.5555/1248547.1248582.

39 AV Meshcheryakov, VV Glazkova, SV Gerasimov, and IV Mashechkin. Measuring the probabilistic photometric redshifts of x-ray quasars based on the quantile regression of ensembles of decision trees. *Astronomy Letters*, 44(12):735–753, 2018. doi:10.1134/S1063773718120058.

40 Kevin P Murphy. *Probabilistic machine learning: an introduction*. MIT press, 2022.

41 Atul Negi and K Rajesh. A review of ai and ml applications for computing systems. In *2019 9th International Conference on Emerging Trends in Engineering and Technology - Signal and Information Processing (ICETET-SIP-19)*, pages 1–6, 2019. doi:10.1109/ICETET-SIP-1946815.2019.9092299.

42 Stefana Nenova and Daniel Kastner. Worst-case timing estimation and architecture exploration in early design phases. In *In Niklas Holsti, editor, 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik*, 2009.

43 Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conference on Real-Time Systems, ECRTS*, 2014.

44 Yaniv Ovadia, Emily Fertig, Jie Ren, Zachary Nado, David Sculley, Sebastian Nowozin, Joshua Dillon, Balaji Lakshminarayanan, and Jasper Snoek. Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift. *Advances in neural information processing systems*, 32, 2019.

45 Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch, 2017.

46 Drew Penney and Lizhong Chen. A survey of machine learning applied to computer architecture design. *ArXiv*, abs/1909.12373, 2019. arXiv:1909.12373.

47 Roger Pujol, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

48 David Radack, Harold G. Tiedeman, and Paul Parkinson. Civil certification of multi-core processing systems in commercial avionics. Technical report, Rockwell Collins, 2018.

49 Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–25, 2012.

50 Jitendra Kumar Rai, Atul Negi, and Rajeev Wankar. Machine learning based performance prediction for multi-core simulation. In Chattrakul Sombattheera, Arun Agarwal, Siba K. Udgata, and Kittichai Lavangnananda, editors, *Multi-disciplinary Trends in Artificial Intelligence*, pages 236–247, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

51 Russell Reed and Robert J MarksII. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press, 1999. doi:10.7551/mitpress/4937.001.0001.

52 Shenyuan Ren, Ligang He, Junyu Li, Zhiyan Chen, Peng Jiang, and Chang-Tsun Li. Contention-aware prediction for performance impact of task co-running in multicore computers. *Wireless Networks*, February 2019. doi:10.1007/s11276-018-01902-7.

53 Hamid Tabani, Roger Pujol, Jaume Abella, and Francisco J. Cazorla. A cross-layer review of deep learning frameworks to ease their optimization and reuse. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 144–145. IEEE, May 2020. doi:10.1109/isorc49007.2020.00030.

54 Lee Teschler. The basics of automotive radar, 2019. URL: https://www.designworldonline.com/the-basics-of-automotive-radar/.

55 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, 2016.

56 Jasper Velthoen, Clément Dombry, Juan-Juan Cai, and Sebastian Engelke. Gradient boosting for extreme quantile regression. *arXiv preprint*, 2021. arXiv:2103.00808.

57 Sergi Vilardell, Isabel Serra, Roberto Santalla, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. HRM: merging hardware event monitors for improved timing analysis of complex mpsocs. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(11):3662–3673, 2020. doi:10.1109/TCAD.2020.3013051.

58 Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005. URL: https://www.jstor.org/stable/24869236.

59 Felippe Vieira Zacarias, Rajiv Nishtala, and Paul Carpenter. Contention-aware application performance prediction for disaggregated memory systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, CF '20, pages 49–59, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387902.3392625.

60 Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint*, 2014. arXiv:1409.2329.

61 Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. Predicting cross-core performance interference on multicore processors with regression analysis. *IEEE Trans. Parallel Distrib. Syst.*, 27(5):1443–1456, May 2016. doi:10.1109/TPDS.2015.2442983.