Time-predictable Deep Noise Suppression on an Edge Device

1st Alessandro Cerioli
DTU Compute
Technical University of Denmark
Kongens Lyngby, Denmark
alceri@dtu.dk

2nd Tórur Biskopstø Strøm
DTU Compute
Technical University of Denmark
Kongens Lyngby, Denmark
tbst@dtu.dk

3rd Clément Laroche
Audio Research & Exploration
GN Audio
Ballerup, Denmark
claroche@jabra.com

4th Tobias Piechowiak
Audio Research & Exploration
GN Audio
Ballerup, Denmark
topiechowiak@jabra.com

5th Luca Pezzarossa
DTU Compute
Technical University of Denmark
Kongens Lyngby, Denmark
lpez@dtu.dk

6th Martin Schoeberl
DTU Compute
Technical University of Denmark
Kongens Lyngby, Denmark
masca@dtu.dk

***Abstract*—Hearing aids and remote conference systems benefit from noise reduction. Current noise reduction approaches include machine-learning models that run on edge devices like hearing aids, AirPods, or headsets. Although not a safety-critical application, audio processing is a real-time application.**

We present a real-time enabled solution of speech enhancement with generation of C code for embedded devices, executing on a real-time processor, and analyzing the worst-case execution time for that application.

Using the Patmos processor and the Platin WCET analysis tool, we can guarantee that we process noise canceling within the given deadline.

***Index Terms*—machine learning. real-time systems, edge devices**

I. Introduction

In recent years, various acoustic devices (e.g., hearing aids, AirPods, and headsets) have become part of everyday life in medical, professional, and recreational settings. These devices must operate in noisy environments without degrading sound quality, using deterministic and AI-based noise removal techniques [1].

Although many methods for background noise cancellation exist, executing them in real-time on embedded devices with limited resources remains a challenge. Novel AI and neural network-based methods show promise, but efficient resource management and optimized algorithms are key to ensuring high audio quality and smooth, delay-free operation.

This paper addresses the time-predictable execution of deep learning models for speech enhancement on embedded devices. We demonstrate that neural networks can be executed on resource-constrained devices while meeting critical delays, using WCET analysis. We evaluate NSNet2 for speech enhancement with C code generated via NeuralCasting, showing its deployability on edge devices like headsets, AirPods, and hearing aids.

979-8-3315-9984-3/25/$31.00 ©2025 IEEE

Denoising is not real-time critical, but delays can degrade audio quality. Ensuring processing within time constraints is crucial, and WCET analysis is key in meeting these constraints. An accurate analysis also prevents oversizing the hardware, providing an optimal solution for real-time processing with minimal resource usage.

This paper presents the integration of NeuralCasting [2] with Patmos for real-time, time-predictable deep learning. NeuralCasting converts ONNX neural networks into optimized C code, improving resource management for real-time systems while preserving model accuracy. The code is suitable for edge devices, especially bare-metal systems.

We present a pipeline to convert ONNX models to C code and then compile NeuralCasting for the Patmos processor [3]. Patmos supports time-predictable, real-time systems, allowing the generated C code to meet strict time constraints.

Finally, we use the WCET tool Platin [4] to analyze the maximum execution time, showing that the generated C code is analyzable and meets the necessary time constraints.

The contributions of this paper are:

- The integration of NeuralCasting with Patmos for time-predictable applications, allowing the use of the current framework also for deploying neural networks unrelated to the audio domain.
- The deployment of NSNet2 on Patmos, ensuring predictable execution time in resource-constrained environments.
- Performance analysis of NSNet2 using WCET tools.

Section II provides the background. Section III describes NeuralCasting. Section IV covers NeuralCasting integration on Patmos. Section V reports an experimental evaluation of NSNet2 on Patmos. Section VI discusses related work, and Section VII concludes the paper.

II. Background

A. ONNX

ONNX (Open Neural Network Exchange) [1] is a standard format for neural networks that includes information such as layers, their connections, operators' hyperparameters, and weights. Inference engines like ONNX Runtime [2] leverage ONNX for efficient inference on various hardware architectures. A key advantage of ONNX is its compatibility with Python frameworks like PyTorch and TensorFlow, making it suitable for neural network compilers.

In our use case, we use an NSNet2 model with a GRU (Gated Recurrent Unit) [3] re-implemented [4] in PyTorch using elementary operations, as ONNX lacks native support for a quantized GRU. The ONNX model features operators such as *Squeeze*, *Matmul*, *Add*, *Mul*, *Sigmoid*, *Sub*, *Tanh*, and *ReLU*. After quantizing the model with ONNX Quantizer, we generated the C code via NeuralCasting. The quantized ONNX model includes operators like *QuantizeLinear*, *QLinearMatmul*, *QLinearAdd*, *QLinearMul*, *QLinearSigmoid*, and *DequantizeLinear*, which contain meta-information such as zero points and scaling factors. More details on the deployment of quantized ONNX models on edge devices are reported in the work [5].

B. NSNet2

NSNet2 [6] is a neural network proposed by Microsoft for speech enhancement and noise cancellation in audio signals. Therefore, it is a suitable approach for headsets and hearing aids. Contrary to traditional techniques such as adaptive filters, NSNet2 presents a more effective solution based on deep learning that can handle more complex dynamics. Indeed, it demands significant computational power and memory (with approximately 3 million parameters).

NSNet2 consists of fully connected layers, GRUs, and activation functions like Sigmoid, ReLU, and Tanh (within the GRUs). The model takes as input a 257-point Fourier transform frame with 50% overlap and the previous frame's GRU hidden state. It outputs a denoising mask and the new hidden states of the GRUs. The mask is multiplied element-wise to the input frame in the frequency domain, resulting in an output frame with background noise removed.

C. Time-predictable Systems

As audio signal processing is a real-time application, we explore NeuralCasting on the time-predictable platform Patmos [7].

Patmos is a RISC-style, time-predictable processor optimized for real-time tasks and WCET analysis. It features a method cache [8], a stack cache [9], and a write-through data cache for non-stack data. These caches are integrated into the WCET tool Platin [10].

A port of the LLVM compiler and the newlib C standard library enables C programs to be compiled for Patmos. Platin uses the compiler output to generate a WCET report for a chosen function.

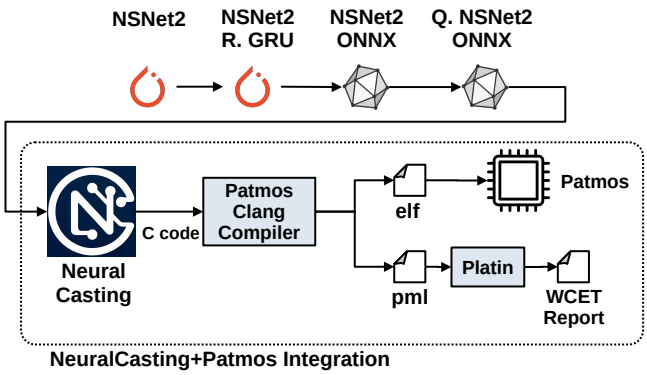

Figure 1. The complete pipeline to deploy the NSNet2 model in PyTorch to Patmos processor. We export the ONNX file from the PyTorch model, then generate the C code via NeuralCasting, and finally compile the C code for the Patmos architecture. Platin is used for WCET analysis.

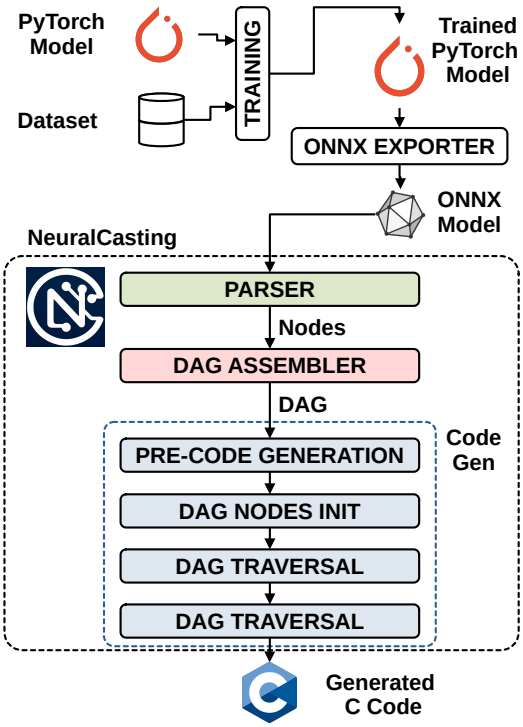

Figure 2. The pipeline of the compiler. In the *Code Generation* stage, *pre_codegen* produces code that includes global variable declarations and a function header with the list of arguments. By contrast, *post_codegen* terminates the generated C function.

Figure 1 shows the integration of the NeuralCasting flow with Patmos' tool flow.

III. NeuralCasting

A. Pipeline

NeuralCasting is a framework that converts neural networks in ONNX format into optimized C code using only the standard libraries. We opted to generate C code as it is a low-level language and allows optimized memory management

more than high-level languages like Python. Additionally, C is widely used for embedded systems and microcontrollers.

Generating C code has significant advantages over inference runtimes like ONNX Runtime, as it removes an abstraction layer that introduces delay and increases compatibility with bare-metal devices.

NeuralCasting consists of two macro-phases (see Figure 2):

- Parser and DAG Assembler break down the ONNX file into units and extract the necessary information to generate an internal graph data structure. This structure, consisting of nodes and memory references, is suitable for traversal-based code generation, unlike the ONNX format.
- Code Generation traverses the graph and generates C code for each node.

B. Parser and Directed Acyclic Graph Assembler

In the parsing phase, the ONNX file is broken down into individual nodes, and it extracts information regarding the type and the nodes connected as input and output. There are four types of nodes: input (input tensors), output (output tensors), operators (e.g., convolution), and initializers (trainable parameters).

The ONNX format lacks a suitable structure for code generation, as nodes reference input and output tensors but do not directly connect to other nodes. To address this, we use the parsed information to initialize a new graph data structure (a directed acyclic graph) within NeuralCasting, where nodes reference connected nodes. This structure is optimal for graph exploration and C code generation.

C. Code Generation

Code generation is performed through a recursive algorithm that explores the directed acyclic graph (DAG) of the neural network and generates the corresponding C code. Nodes have three possible states:

- *not-ready* when they are not ready to generate code.
- *ready* when they are ready to generate code.
- *generated* when they have generated code.

A node is promoted from *not-ready* to *ready* when all its input nodes are in the *generated* state. It is promoted to *generated* when selected for code generation.

Initially, all nodes are *not-ready*, except for the input, the initializer, and constant nodes, which are *ready*. The algorithm iterates until all nodes are in the *generated* state (see Algorithm 1).

Algorithm 1 Pseudocode of DAG traversal

```
0: for node ∈ nodes do
0:     if node ∉ generated_list and node ∉ ready_list then
0:         # get node's input nodes
0:         inputs ← get_inputs(node)
0:
0:         # check if the node is ready to turn to ready state
0:         ready ← is_node_ready_to_turn_ready (
               inputs, ready_list,generated_list)
0:
0:         # if ready, generate the code for each input node
0:         # in ready state
0:         if ready then
0:             code ← turn_to_ready_and_generate_code(
                   node, inputs, ready_list, generated_list)
   =0
```

The *ready_list* contains nodes in the *Ready* state, and the *generated_list* contains nodes in the *Generated* state. The *get_inputs* function returns input nodes for a given node, `is_node_ready_to_turn_ready` checks readiness, and `turn_to_ready_and_generate_code` promotes the node and generates the code.

The selection of *ready* nodes to expand favors cache optimizations with generic LRU policy, prioritizing consecutive operations.

Each node in the graph is associated with a C code template via a macro marked with $, such as $OUTPUT_NAME, $INPUT_NAME, and $KERNEL_SIZE, are extended based on node-specific information to generate the final C code.

D. Memory Management

Memory management is crucial for efficiently running neural networks on devices with limited resources. Optimal memory management minimizes fragmentation, maximizes throughput, and reduces memory accesses, which are the primary performance bottleneck. Therefore, NeuralCasting provides diverse options to allocate memory through a specific *alloc* flag, which can assume the following values:

- *data* allocates memory in the data segment, so the weights are compiled directly into the executable. The static memory allocation is fast and predictable and suitable for deployment on bare metal systems, but inefficient for large models as it significantly slows down the compilation.
- *heap* allocates parameters in the heap, requiring a memory management system with slower memory access. However, it is the optimal option for generating large models. NeuralCasting generates binary files containing parameter information and C functions for dynamic allocation and deallocation of memory.

For embedded systems, generation in the data segment is the best solution. Consequently, we used the code generated with this option for our experiments.

E. Quantization Support

Deploying neural networks on resource-constrained devices enables real-time applications with predictable inference time, and enhance privacy by avoiding third-party cloud systems.

However, neural networks are computationally expensive algorithms and require significant memory capacity. Quantization is a compression technique that maps floating-point values to integer data types, usually 8 or 16 bits, reducing memory usage and computational cost, and energy consumption. However, this procedure entails a loss of precision, which can be mitigated by a calibration phase on a representative dataset to estimate the scaling factors and zero points.

Uniform linear quantization represents all the layers within the same bit-width, and it maps linearly from floating points to integers, as described in Equation 1.

$$\mathbf{Q} = \left\lfloor \frac{\mathbf{W}}{s} \right\rfloor + z \tag{1}$$

where $\mathbf{W}$ is the floating point weights matrix, and $\mathbf{Q}$ is the quantized weights matrix. Finally, z and s are the *zero point* and the *scaling factor* of the linear transformation, respectively. The scaling factor calculation follows the Equation 2.

$$s = \frac{r_{max} - r_{min}}{q_{max} - q_{min}} \tag{2}$$

where $r_{max} - r_{min}$ is the range of values of floating point matrix, and $q_{max} - q_{min}$ is the representable range using N-bits integers, i.e., $\left[-2^{N-1}, 2^{N-1} - 1\right]$ for signed integers or $\left[0, 2^{N} - 1\right]$ for unsigned integers.

The zero point can be calculated accordingly from the Equations 1 and 2:

$$z = round\left(q_{min} - \frac{r_{min}}{s}\right) \tag{3}$$

The operations occur between integer data types, and finally, the values are converted back to floating points according to Equation 4.

$$\mathbf{W} = (\mathbf{Q} - z) \cdot s \tag{4}$$

In ONNX, quantization uses the *QFormat*, which are units containing information about the quantization, such as the data type, scaling factors, and zero points. ONNX Runtime provides an API to calibrate these parameters from a representative dataset.

NeuralCasting presents an ONNX-compliant extension. Consequently, it supports quantization with C templates for the quantized operators. It includes new operators to the framework such as *QuantiLinear*, *DequantizeLinear*, *QLinearMatmul*, or *QLinearAdd*. Other non-linear operators, such as activation functions, present an implementation via lookup tables. However, maintaining ONNX compatibility limits the development of mixed-precision quantization, which is currently under development in ONNX Runtime.

IV. Framework Integration

This paper proposes integrating NeuralCasting with Platin and Patmos. NeuralCasting transforms ONNX neural networks into optimized C code using standard libraries, suitable for bare-metal embedded systems. Patmos is a real-time processor designed for time-predictable execution via WCET analysis, making it ideal for real-time deep learning inference. We evaluate NSNet2 for speech enhancement and noise reduction, which, while not critical, requires strict timing constraints to avoid audio degradation.

The NSNet2 model, developed in PyTorch, required a manual reimplementation of the GRU due to ONNX's lack of support for a quantized GRU operator. We then converted the PyTorch model to ONNX using QFormat, which includes the quantized operators.

We used the ONNX files as input for NeuralCasting to generate optimized C code, which was compiled into Patmos Assembly. Platin was used for static WCET analysis based on memory accesses and cache management (see Figure 1).

NeuralCasting supported linear quantization, but the original implementation used float32 scaling factors, introducing floating-point operations during inference. To meet real-time constraints on Patmos, we extended NeuralCasting to support 8-bit quantization with fixed-point scaling factors (Q16.16), problem that has been addressed in literature by [11].

Quantization and de-quantization operators were implemented as per Equations 5 and 6.

$$\mathbf{Q} = \left\lfloor \frac{\mathbf{W}}{S_F} \cdot S_{BASE} \right\rfloor + z \tag{5}$$

$$\mathbf{W} = (\mathbf{Q} - z) \cdot \frac{S_F}{S_{BASE}} \tag{6}$$

Where S_F is the fixed-point scaling factor (Q16.16), and S_{BASE} is the scaling base.

We developed quantized operators for matrix multiplication (Equation 7) and matrix addition (Equation 8) using fixed points.

$$\hat{\mathbf{Y}} = \frac{S_A \cdot S_B}{S_Y} \cdot \left(\hat{\mathbf{A}} - Z_A\right) \cdot \left(\hat{\mathbf{B}} - Z_B\right) \tag{7}$$

$$\hat{\mathbf{Y}} = \frac{S_A}{S_Y} \cdot \left(\hat{\mathbf{A}} - Z_A\right) + \frac{S_B}{S_Y} \cdot \left(\hat{\mathbf{B}} - Z_B\right) \tag{8}$$

Where $\hat{\mathbf{Y}}$ is the quantized output, and S_Y, S_A, S_B are scaling factors for the output and operands. Zero points are denoted by Z_Y, Z_A, and Z_B. These operations are expressed without fixed-point notation for clarity.

We also replaced the floating-point activation functions (Sigmoid and Tanh) with lookup tables. Both tables have a size of 256 samples. However, the domain of Sigmoid and Tanh range between -6.0 and 6.0, and between -4.0 and 4.0, respectively.

The lookup table size, min and max values, and Δ between points are presented, with linear interpolation used for accuracy.

This fixed-point implementation significantly improves Patmos performance, as the execution time was 12,095,825,083 clock cycles on the Patmos emulator *pasim* before, whereas after, we measure the execution time to be 28,584,720 clock cycles, as show in Section V. Interestingly, the previous execution time was dominated by 88.05% of stalls.

V. Evaluation

The evaluation of our solution is twofold: WCET analysis and timing measurement of the generated inference function (*run_inference_int*).

The total WCET of the function, shown in Table I, is reported with the data for the three types of caches: instruction, stack, and data. For each cache type, the minimum number of cache hits and the maximum number of cache misses are shown, highlighting the maximum time required for data access.

Our platform is the Altera DE2-115 board with the Patmos processor configured on the Cyclone EP4CE115 FPGA running at 80 MHz. Patmos is configured with a 2 MiB SRAM main memory by default. However, the inference requires more memory, so we use the two 64 MiB SDRAM chips instead. The compiler is instructed to make use of the larger memory area:

```
patmos-clang -O2 -mserialize-pml=rt-neuralcasting.
    pml -I.  -Wl,--defsym,__heap_end=0x02000000 -Wl
    ,--defsym,_stack_cache_base=0x04000000 -Wl,--
    defsym,_shadow_stack_base=0x03ff8000 *.c -o rt-
    neuralcasting.elf
```

i.e., the heap ends at half the memory size (the heap start is set by the compiler as right after the program), the stack for the first core starts at the end of the memory, and the size of the stack is 0x04000000-0x03ff8000 =32 KB. A second core's memory would start at 0x03ff8000, however, we only configure the platform with a single Patmos core.

For the WCET analysis, we provide platin with a timing/memory model of Patmos (*config_de2_115.pml*) and execute the command:

```
platin wcet --enable-wca --disable-ait --target-call
    -retu
rn-costs -b rt-neuralcasting.elf -i rt-neuralcasting
    .pml -i config_de2_115.pml -e run_inference_int
    --report
```

This results in a reported WCET of 64,397,198 cycles for *run_inference_int*. That platin successfully found a WCET shows that no unbounded loops were found.

To measure the execution time of *run_inference_int* we call *get_cpu_usecs* before and after the function, and calculate the difference. This results in an execution time of $0.357\,309$ s or $0.357\,309\,\mathrm{s} \cdot 80\,\mathrm{MHz}$ = 28,584,720 cycles. The WCET bound is roughly 2.25 times larger than the measured execution time.

This discrepancy between WCET and measured execution time highlights the difference between theoretical worst-case estimates and real-world performance, remarking the importance of WCET analysis for time-predictable execution in real-time systems.

A. Possible Improvements

Integrating NeuralCasting with the Patmos processor shows significant potential for real-time, time-predictable execution of neural networks, but several areas for further research and improvement remain.

An area is the use of time-predictable multicore architectures. Extending NeuralCasting to generate code for efficient multicore utilization on the Patmos processor (i.e., the T-CREST platform [12]) can enhance performance by parallelizing tasks, such as layer-wise computations. NeuralCasting already supports parallelization automatically generating OpenMP directives. However, to maintain time predictability, multicore execution would require careful synchronization. If done correctly, the code running on each core could be independently analyzed with network-on-chip (NoC) like S4NOC [13] and Argo facilitating data movement. This approach can lead to WCET speedups, but increasing the complexity in code generation and WCET analysis.

Another area for exploration is the integration of specialized hardware accelerators for computationally intensive operations, such as matrix multiplication, and activation functions. Hardware accelerators [14] could reduce WCET bounds, improving the system responsiveness and allowing a more accurate sizing. NeuralCasting could be extended to generate code that interfaces with these accelerators.

TABLE I
WCET Analysis Results for *run_inference_int*

Metric	Value
WCET (cycles)	64,397,009
Cache-Max-Cycles-Instr	6,405
Cache-Min-Hits-Instr	18,705
Cache-Max-Misses-Instr	39
Cache-Max-Cycles-Stack	0
Cache-Max-Misses-Stack	0
Cache-Max-Cycles-Data	36,549,597
Cache-Min-Hits-Data	0
Cache-Max-Misses-Data	1,732,400
Cache-Max-Stores-Data	8,057
Cache-Unknown-Address-Data	1,740,457
Cache-Max-Cycles	36,556,002

VI. Related Work

With this section, we review the state-of-the-art of deep noise suppression on edge devices, and we compare the works in literature to our approach based on NeuralCasting and Patmos architecture.

The paper [15] presents a multi-head self-attention network (MHANet) for speech enhancement, outperforming RNNs and TCNs, but is too computationally expensive for embedded systems. Nevertheless, NSNet2 presents a lower computational cost, and it is more suitable for resource-constrained devices.

SA-TCN model [16] uses self-attention and dilated TCN for denoising, but it requires high resource usage, making it currently unsuitable for deployment on embedded systems.

TinyLSTMs [17] presents an efficient approach for deployment on low-power devices. This approach relies mainly on the compression of RNNs, while our approach relies on deploying a quantized NSNet2 on Patmos.

WaveVoice [18] uses multimodal data (audio and lip movement) for speech enhancement. Unlike WaveVoice, we use only signal processing and ensure predictable inference time on Patmos.

ClearBuds [19] proposes a dual-channel system to improve audio processing on AirPods. Unlike ClearBuds, our approach relies on frequency-domain masking and introduces hardware optimizations via NeuralCasting and Patmos.

The DNN-TF method [20] uses spatial information for speech enhancement on drones. Instead, we focus on NSNet2 for single-channel, time-predictable latency.

DeepLPC [21] estimates Kalman filter coefficients for the speech enhancement. Contrarily, we use NSNet2 directly to denoise the audio signal.

EA-PSE [22] proposes a personalized speech enhancement method but does not focus on time-predictable inference.

The paper [23] proposes an effective method to optimize GEMM-based convolutions, allowing real-time applications on hardware architectures. In contrast, our work emphasizes the use of quantization of recurrent neural networks.

There are other frameworks for generating C code from the ONNX format, such as onnx2c [24], ACETONE [25] (which is also time-predictable in combination with OTAWA WCET Analyzer), and AIDGE [26] (which supports quantization). However, NeuralCasting specifically features compatibility with the Patmos architecture. Additionally, the keras2c [27] framework converts the tensorflow models to C code.

In conclusion, while NeuralCasting and Patmos enable efficient deployment with time-predictable inference, quantization improves performance but introduces errors in the output.

VII. Conclusion

In this paper, we presented the integration of NeuralCasting and Patmos for time-predictable execution of neural networks on edge device. We considered NSNet2 as a case study, demonstrating the possibility to use denoising and speech enhancement applications on resource-constrained devices like headsets, AirPods, and hearing aids. Through WCET analysis, we showed that NSNet2 can be executed on Patmos with low latency for real-time performance, starting from quantized C code generated by NeuralCasting. However, quantization introduces errors in the output, which may affect audio quality, although speech enhancement is not a critical application. Our approach ensures time-predictable latency while addressing this issue.

Data Availability

The repository is at the following link: https://github.com/alecerio/NeuralCasting. The demo for the quantized NSNet2 is in examples/qnsnet2/. The T–CREST platform that we selected for our experiments is available at: https://github.com/t-crest

Acknowledgment

Alessandro Cerioli is funded by the European Union's Horizon research and innovation program under grant agreement No 101070374.

References

[1] ONNX Community, "ONNX: Open Neural Network Exchange," https://github.com/onnx/onnx, 2024.

[2] microsoft, "onnxruntime," https://github.com/microsoft/onnxruntime, 2023.

[3] A. Shewalkar, D. Nyavanandi, and S. A. Ludwig, "Performance evaluation of deep neural networks applied to speech recognition: Rnn, lstm and gru," *Journal of Artificial Intelligence and Soft Computing Research*, vol. 9, no. 4, pp. 235–245, 2019.

[4] PyTorch, "torch.nn.gru," 2025. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.GRU.html

[5] C. Gabreau, M.-C. Teulières, E. Jenn, A. Lemesle, D. Potop-Butucaru, F. Thiant, L. Fischer, and M. Turki, "A study of an acas-xu exact implementation using ed-324/arp6983," in *12th European Congress Embedded Real Time Systems-ERTS 2024*, 2024.

[6] S. Braun and I. Tashev, "Data Augmentation and Loss Normalization for Deep Noise Suppression," in *Proc. of International Conference on Speech and Computer (SPECON), Lecture Notes in Computer Science, vol. LNCS12335*. Springer, 2020, pp. 79–86. [Online]. Available: http://arxiv.org/abs/2008.06412

[7] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, and D. Prokesch, "Patmos: A time-predictable microprocessor," *Real-Time Systems*, vol. 54(2), pp. 389–423, April 2018.

[8] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl, "A method cache for Patmos," in *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*. Reno, Nevada, USA: IEEE, June 2014, pp. 100–108.

[9] S. Abbaspour, F. Brandner, and M. Schoeberl, "A time-predictable stack cache," in *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[10] E. J. Maroun, E. Dengler, S. Dietrich, Chistian Hepp, B. Herzog, Henriette Huber, J. Knoop, D. Prokesch, P. Puschner, P. Raffeck, M. Schoeberl, S. Schuster, and P. Wägemann, "The platin multi-target worst-case analysis tool," in *22th International Workshop on Worst-Case Execution Time Analysis (WCET 2024)*, 2024.

[11] H. Benmaghnia, M. Martel, and Y. Seladji, "Code generation for neural networks based on fixed-point arithmetic," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 5, pp. 1–28, 2024.

[12] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[13] M. Schoeberl, "Exploration of network interface architectures for a real-time network-on-chip," in *Proceedings of the 2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)*. United States: IEEE, 2024, 2024 IEEE 27th International Symposium on Real-Time Distributed Computing, ISORC ; Conference date: 22-05-2024 Through 25-05-2024.

[14] C. Pircher, A. Baranyai, C. Lehr, and M. Schoeberl, "Accelerator interface for patmos," in *2021 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, 2021.

[15] A. Nicolson and K. K. Paliwal, "Masked multi-head self-attention for causal speech enhancement," *Speech Communication*, vol. 125, pp. 80–96, 2020.

[16] J. Lin, A. J. d. L. van Wijngaarden, K.-C. Wang, and M. C. Smith, "Speech enhancement using multi-stage self-attentive temporal convolutional networks," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 29, pp. 3440–3450, 2021.

[17] I. Fedorov, M. Stamenovic, C. Jensen, L. Yang, A. Mandell, Y. Gan, M. Mattina, and P. Whatmough, "Tinylstms: Efficient neural speech enhancement for hearing aids. arxiv 2020," *arXiv preprint arXiv:2005.11138*.

[18] Q. Zhang, D. Wang, R. Zhao, Y. Yu, and J. Shen, "Sensing to hear: Speech enhancement for mobile devices using acoustic signals," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 5, no. 3, pp. 1–30, 2021.

[19] I. Chatterjee, M. Kim, V. Jayaram, S. Gollakota, I. Kemelmacher, S. Patel, and S. M. Seitz, "Clearbuds: wireless binaural earbuds for learning-based speech enhancement," in *Proceedings of the 20th Annual*

International Conference on Mobile Systems, Applications and Services, 2022, pp. 384–396.

[20] L. Wang and A. Cavallaro, "Deep learning assisted time-frequency processing for speech enhancement on drones," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 5, no. 6, pp. 871–881, 2020.

[21] S. K. Roy, A. Nicolson, and K. K. Paliwal, "Deeplpc: A deep learning approach to augmented kalman filter-based single-channel speech enhancement," *IEEE Access*, vol. 9, pp. 64 524–64 538, 2021.

[22] Y. Ju, W. Rao, X. Yan, Y. Fu, S. Lv, L. Cheng, Y. Wang, L. Xie, and S. Shang, "Tea-pse: Tencent-ethereal-audio-lab personalized speech enhancement system for icassp 2022 dns challenge," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2022, pp. 9291–9295.

[23] I. De Albuquerque Silva, T. Carle, A. Gauffriau, and C. Pagetti, "Extending a predictable machine learning framework with efficient gemm-based convolution routines," *Real-Time Systems*, vol. 59, no. 3, pp. 408–437, 2023.

[24] Kraiskil, "onnx2c," 2025. [Online]. Available: https://github.com/kraiskil/onnx2c

[25] ONERA, "Acetone," 2025. [Online]. Available: https://github.com/onera/acetone

[26] E. Foundation, "Ai edge computing," 2025. [Online]. Available: https://projects.eclipse.org/projects/technology.aidge

[27] PlasmaControl, "keras2c," 2025. [Online]. Available: https://github.com/PlasmaControl/keras2c