Tiny Machine Learning Virtualization for IoT and Edge Computing using the REXA VM

Towards Learning Technical Systems

Stefan Bosse [1,2]

Christoph Polle [3]

[1]University of Bremen, Dept. Mathematics & Computer Science, Bremen, Germany

[2]University of Siegen, Dept. Mechanical Engineering, Siegen, Germany

[3]Faserinstitut Bremen (FIBRE), Bremen, Germany

Overview

Tiny Machine Learning is a new approach that is being used for data-driven prediction, classification, and regression on microcontrollers using local sensor data.

Overview

Tiny Machine Learning is a new approach that is being used for data-driven prediction, classification, and regression on microcontrollers using local sensor data.

But even simple sensor data acquisition, aggregation, and processing is a challenge in distributed sensor network environments, the IoT, mobile networks, and other distributed strongly **heterogeneous** networks.

Overview

Tiny Machine Learning is a new approach that is being used for data-driven prediction, classification, and regression on microcontrollers using local sensor data.

But even simple sensor data acquisition, aggregation, and processing is a challenge in distributed sensor network environments, the IoT, mobile networks, and other distributed strongly **heterogeneous** networks.

The goal is to process sensor data locally and derive compressed relevant information features (e.g., damages, attacks, ...) with final global feature fusion.

Overview

To overcome issues and limitations with software and ML deployment in strong heterogeneous computer networks, the real-time capable low-resource Virtual Machine REXAVM is introduced.

Overview

To overcome issues and limitations with software and ML deployment in strong heterogeneous computer networks, the real-time capable low-resource Virtual Machine REXAVM is introduced.

REXA VM provides Virtualization of basic ML operations and models including but limited to: Decision Trees, ANN, CNN

Overview

To overcome issues and limitations with software and ML deployment in strong heterogeneous computer networks, the real-time capable low-resource Virtual Machine REXAVM is introduced.

REXA VM provides Virtualization of basic ML operations and models including but limited to: Decision Trees, ANN, CNN

REXA VM and its ML operations can be deployed on low-resource microncontrollers like the STM32 ARM Cortex M-series starting with 20 kB of RAM and 32 kB ROM only!

Introduction

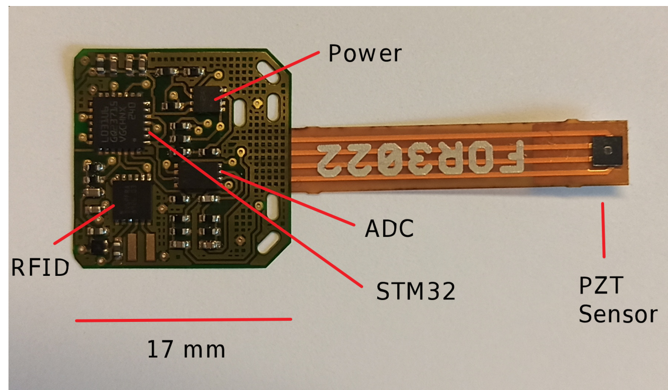


Fig. 1. Let's start here: A material-integrateable sensor node for damage diagnostics in Fibre-Metal Laminates using Guided Ultrasonic Waves (STM32 ARM Cortex M0, RFID, ADC) [IMSAS Bremen, B. Lüssem et al., 2023]

Host Platforms and Efficiency

Efficiency of data processing is always an important objective to optimize, especially for material-integrated sensor networks. The efficiency of data processing systems can be compared by the following normalized performance factor ε:

$$\epsilon = \frac{C \cdot M}{A \cdot P}$$

C: Data processing system's computational power in instructions per second (MIPS)

M: Memory capacity (RAM/ROM) in k Bytes

A: Entire chip area in mm^2

P: Electrical power consumption in mW.

Host Platforms and Efficiency

Device	**Chip Area**	**Clock/MIPS**	**Power**	**RAM/ROM**	**ε**
Atmel Tiny 20	2.1 mm^2 (1.55x1.4x0.53 mm)	12 MHz	4 mW	0.1 kB/2 kB	3
ARM Cortex M0 (Smart Dust 2002)	0.1 mm^2	740 kHz	70 mW	4 kB/4 kB	0.84
FreeSclae KL03 (ARM Cortex M0+)	4 mm^2	48 MHz	3 mW	2 kB/40kB	168
STM32 F103VC M3	~10 mm^2	72 MHZ	200 mW	48 kB/256 kB	11
STM32 F103C8 M3	~6 mm^2 (meas.)	48 MHZ	100 mW	20 kB/64 kB	6.7
STM32 L031G6U6 M0+	0.25 mm^2 (meas.)	16 MHZ	2 mW	8 kB/32 kB	1280
STM32 L073CZU6 M0+	~1 mm^2	16/32 MHZ	5/12 mW	20 kB/192 kB	678/565
Xilinx Spartan 3-500E	9.6 mm^2 (meas.)	50 MHz	100 mW	45 kB	2.34
Xilinx Spartan 7-S25	~50 mm^2	100 MHz	100 mW	202 kB	4

The Concepts

1. VM with integrated compiler
2. Programs (and ANN models, too) are always delivered in textual format
3. On-the-fly compilation to linear Bytecode (< 600 lines of C code!)
4. No dynamic memory management except by stack operations
5. KISS (< 3000 lines of C code); highly configurable (custom ISA)
6. VM can be directly embedded in IO loops (microcontrollers) cooperating with other tasks

VM Architecture

Memory Model

VM Architecture

Memory Model

Instruction Set Architecture (Bytecode)

VM Architecture

Memory Model

Instruction Set Architecture (Bytecode)

Real-time Features and Scheduling

VM Architecture

Memory Model

Instruction Set Architecture (Bytecode)

Real-time Features and Scheduling

Compiler

VM Architecture

Memory Model

Instruction Set Architecture (Bytecode)

Real-time Features and Scheduling

Compiler

ML Core Operations

Memory Model and Instruction Processing

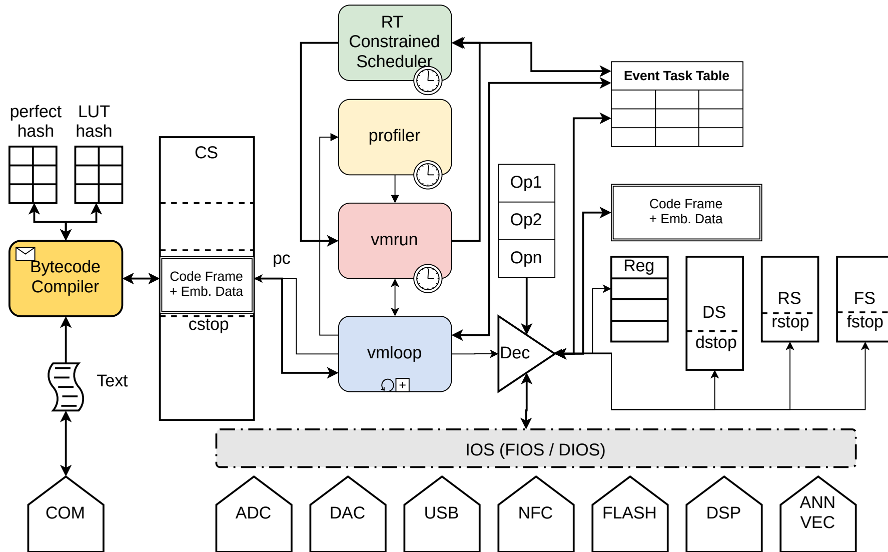


Fig. 2. Multi-stack Computer with mixed-mode code segment (no heap memory), integrated JIT compiler, and Bytecode processor (vmloop)

Instruction Set Architecture

- Most ops are zero-operand instructions (single world) operating directly on the stack(s) or the program counter
- With some exceptions the ISA can be freely defined (via code snippets and macro definitions, discussed in the SDK section)
- Zero-operand operations consume one Byte (see next slide)
- Most instructions have constant and equal execution times (real-time; run-time prediction possible)

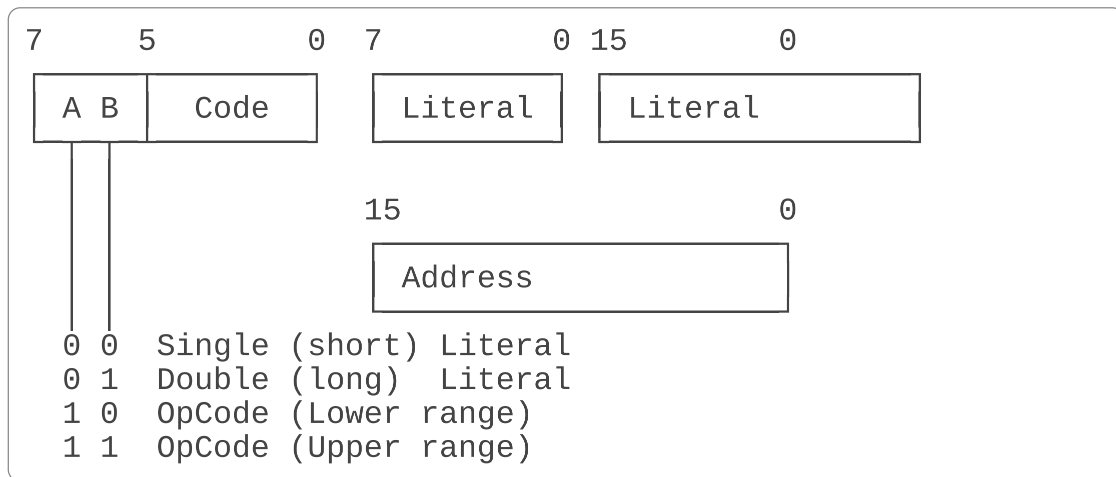
But the widely used and well known FORTH programming language will be used commonly (or any sub-set; there is no real standard)

FORTH

- Reverse Polish Notation (stack language)
- "Write once and forget (read never)" issue
- But keeps compiler simple (low resources and compilation times)

```
var x
10 20 + x !
x @ . cr
: vecmean
  0
  100 0 do
    data i cell+ @ +
  loop
;
vecmean . cr
```

Bytecode Format



Def. 1. REXA VM Bytecode Format (1 Byte: Post-fix operation, 2 Bytes: Short word, 4 Bytes: Double word, 3 Bytes: Code + Address)

Real-time Scheduling

A sensor node, in particular, has to process a set of tasks characterised by different priorities, arrival times, deadline, and execution times:

1. Signal sampling and generation (triggering)
2. Event detection
3. Communication (on-chip, on-board, or externally wireless)
4. Computation
5. Energy Management (energy savings and optimisation)
6. Service requests and processing (deadlines)
7. Watchdog (sensor and node failure detector)

Real-time Scheduling

- These tasks must be scheduled under time and performance constraints.
- Assuming only one physical control path (one processor), the tasks must be scheduled in slices by one main scheduling loop.
- Self-powered sensor nodes introduce additional energy constraints
- The tasks can be classified as
  - event-based (short-running),
  - data-driven (long running), and
  - communication (event- and data-driven) tasks.

Execution Loop

The VM Bytecode execution and source-code compilation can be performed incrementally limiting the number of instruction steps or compiled tokens satisfying soft real-time constraints.

- VM loop is a monolithic switch-case block mapping instruction codes on code snippets executing the operations (optimized to linear goto LUT, **Constant time**)

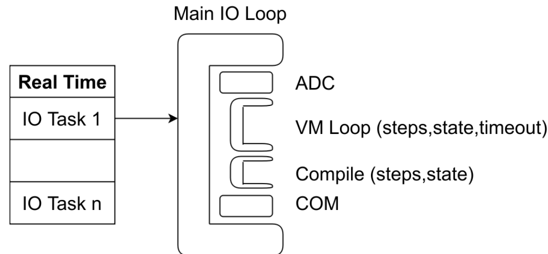
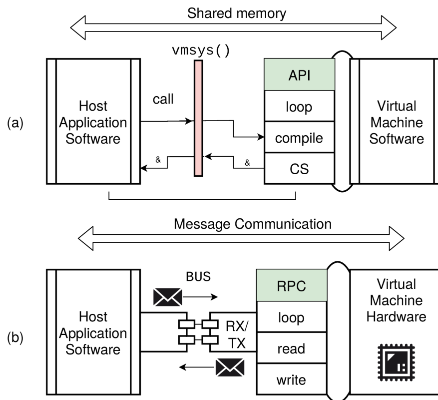


Fig. 3. Nested execution loops in an embedded system

System Call-gate Interface

The system call-gate interface is a unified communication and execution interface to the REXA VM run-time environment and compiler. There are two complementary versions of the system call-gate interface addressing software and hardware implementations of the VM:



SM: A shared-memory architecture

MP: A message-based architecture

Fig. 4. System Call-gate Interface connecting a sensor node root application software to an isolated VM instance (a) via shared memory and a single system call function (b) via message-based communication and a serial link or signal bus

Pocket GUW Laboratory

- Example of an application using the call-gate interface: A digital oscilloscope equipped with the REXA VM

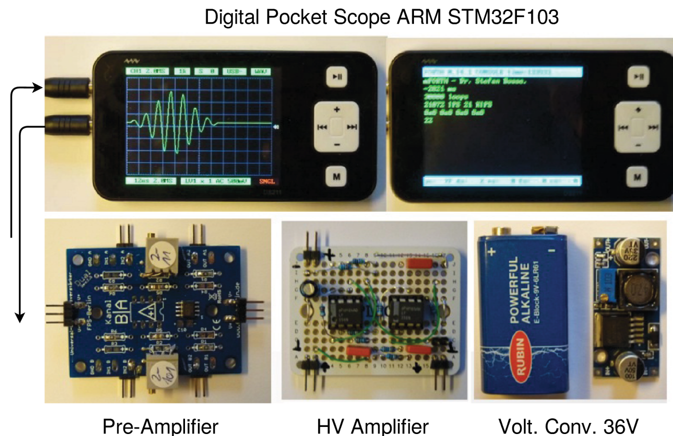


Fig. 5. The pocket GUW laboratory only using low-budget and low-quality devices for GUW-based damage detection in Fibre composite materials. The DSO implements REXA-VM and communicates via an USB virtCOM port with an external computer. https://arxiv.org/abs/2302.09002v1

Compiler

Highlights

- Just-in-time and incremental compiler
- In-place compilation Program Text ⇒ Bytecode via Code Segment frames
- Low memory requirements
- Use of hash and indexed Lookup Tables (LUT) for core instruction codes and user defined data and code (function words)
- Only static tables used with constant memory requirement

Vector Operations

- Only integer arithmetic is supported (by low-resource and low-power microcontrollers)

An ANN (and CNN) consists of two parts:

1. The data, i.e., for parameter, input, and output variables;
2. The structure and functions processing the data.

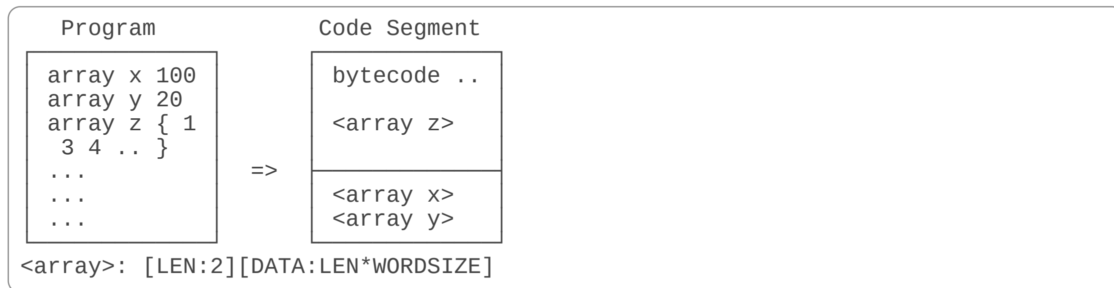
The ANN can be functionally decomposed into the following vector and matrix operations assuming integer approximation:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^p \approx \mathbb{I}^n \rightarrow \mathbb{I}^p, f = g \circ f_{l-1} \circ f_{l-2} \circ . . \circ f_1, f_i(\vec{x}) = a\left(\hat{w}_i \vec{x} + \vec{b}_i\right)$$

$$g(\vec{z}) = \begin{cases} z & \text{regression} \\ \frac{1}{1+e^{-z}} & \text{binary classification} \\ \frac{e^{z_j}}{\sum_k \left(e^{-z_k}\right)} & \text{multi-classification} \end{cases}$$

Vector Operations

- ANN models can be decomposed in chained vector operations!
- Vectors are initialised arrays (model parameters) or initialised arrays (input, intermediate, and output data)
- Vector (array) data is embedded in the Code Segment (no heap!)



Def. 2. Initialized arrays embedded in-place in code frames and non-initialized arrays stored at the end of the compiled code frame

Vector Operations

- For ANN and CNN models, a set of scaled vector (array) operations is provided (commonly W=16 Bits signed integer).
- Most vector operations are using $2W$ arithmetics internally (e.g., 32 Bits) with final down (or up) scaling of results
- Scaling parameters must be computed by a model analyzer

Operation	Description
`array`	Create an initialised or unintialised array (vector)
`vecscale`	Scale a vector (negative scale value: division, positive: multiplication)
`vecadd vecmul`	Elementwise vector addition and multiplication
`vecfold`	Folding operation (ANN FC layer for multiple neurons)
`vecconv`	Multi-purpose convolution and pooling operation (CNN)
`vecmap`	Elementwise application of a function (e.g., relu or sigmoid), used for ANNs and CNNs
`vecreduce`	Vector reduction (scalar output), e.g., minimum or maximum search, sum, product

ANN Example

Model Data

```
( Layers: 14,8,2 #neurons:24 )
array input 14
( Layer I )
array wghtsI { 329 -499 ... 10 400 }
array biasI { -764 389 ... -907 -405 }
array scaleI { -3 9 ... 5 9 }
array actI 14
( Layer H1 )
array wghtsH1 { 622 -790 ... 708 248 }
array scaleH1 { 0 5 ... -4 7 }
array actH1 8
( Layer O )
array wghtsO { 869 939 ... 785 910 }
array biasO { 252 -565 }
array scaleO { 4 5 }
array output 2
```

Model Computation

```
( Input data is stored in input )
( Output data is stored in output )
: forward
  ( Layer I )
  input wghtsI actI scaleI vecmul
  actI biasI actI 0 vecadd
  actI actI $ sigmoid 0 vecmap
  ( Layer H1 )
  actI wghtsH1 actH1 scaleH1 vecfold
  actH1 biasH1 actH1 0 vecadd
  actH1 actH1 $ sigmoid 0 vecmap
  ( Layer O )
  actH1 wghtsO output scaleO vecfold
  output biasO output 0 vecadd
  output output $ sigmoid 0 vecmap
;
```

Software Development Kit

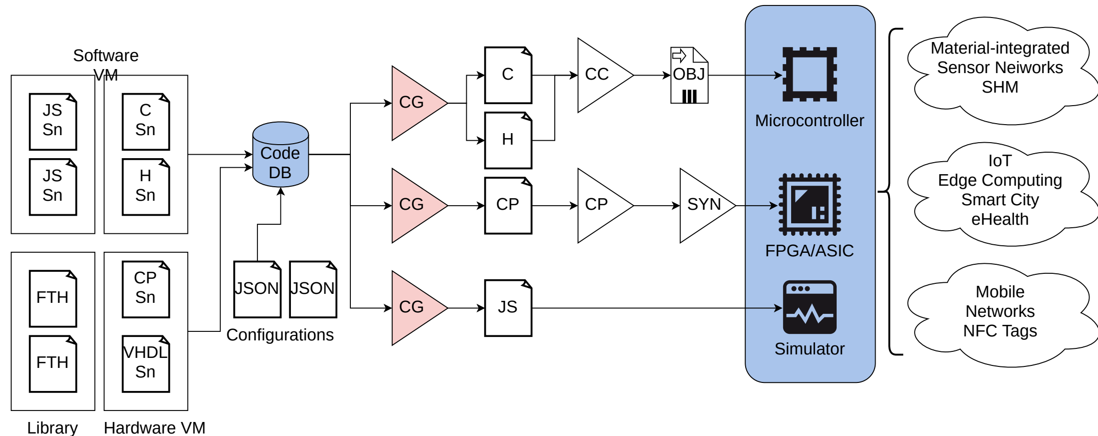


Fig. 6. Overview of the overall concept of REXA-VM development (C-SN: C source code snippet, H-SN: C Header snippet, JS: JavaScript, FTH: Forth VM code definitions, JSON: JavaScript Object Notation, CG: Code generator, CC: C Compiler, CP: ConPro HLS, SYN: RTL synthesis tool)

- The ISA is defined by a collection of code snippets and macro definitions
    - There are different implementations for different host platforms (or OS)
    - There are different implementations for software and hardware VMs
- All code and definitions are stored in a simple JSON data-base that can be accessed by various compiler programs

Git it here and try out: https://github.com/bsLab/rexavm/

Resilience

Resilience and robustness on VM-level

0. KISS: VM architecture is simple and provides inherent safety due to its simplicity;
1. Enhanced error detection and error recovery due to virtualization and isolation of critical architecture components; a pure textual code and data VM input interface increases the probability of detecting communication errors (data corruption);
2. Strict separation of control and data stacks (r-stack is not accessible by user code);
3. Tasks can only access private data directly (data is embedded in their private code frames);
4. Ensemble VM execution (hardware or multi-core software implementation), executing the same code in parallel on multiple VM instances and comparing intermediate states and results (majority decision making; stopping of faulty computations);

Resilience and Robustness on VM-level

5. Check-pointing with optional persistent storage enabling stop-and-go (instead of stop-and-forget) processing (e.g., on irregular and short power cycles);
6. Exception handling;
7. Adaptivity due to incremental code execution (i.e., code updates overwriting older code via the global dictionary);
8. Hardware-Software-Simulation Co-design by unified DB-driven VM code generators enables the operational simulation, profiling, and test of real network nodes with the same operational semantics and discrete timing;
9. Optional special data codings (hardware, simulated in software) for improved error detection and error correction.

Performance

VM

1. Compilation (MCPS, Tokens)
2. Bytecode execution (MWPS, Bytecode Instructions)

Target	Configuration	MIPS	MCPS	Code/Heap
STM32 F103VC3, 72MHz, 256kB ROM, 48kB RAM	CS=1024, DS=256, RS=128, FS=64, Words=101	1.1 / 15k/MHz	0.1 / 1.4k/MHz	8/8 kB
STM32 F103VC3, 72MHz, 256kB ROM, 48kB RAM	CS=1024, DS=256, RS=128, FS=64, Words=64 (no double word arithmetic)	1.1	0.1	7/7 kB
STM32 F103VC3, 72MHz, 256kB ROM, 48kB RAM	CS=4096, DS=1024, RS=256, FS=128, Words=101	1.1	0.1	8/16 kB
STM32 L031, 16 MHz, 32 kB ROM, 8 kB RAM	CS=1024, DS=256, RS=32, FS=32, Words=101	0.24 / 15k/MHz	0.02	7.1/8 kB
i5-7300U, 3GHz 4 GB RAM	CS=16384, DS=4096, RS=1024, FS=256, Words=101	280 / 90k/MHz	27 / 9k/MHz	32/64 kB

VM

Highlights

- 1:70 → About 70 native machine instructions / VM instruction execution (ARM Cortex) or 1:15 (Intel x86)
- 1:700 → About 700 native machine instructions / Word compilation (ARM Cortex) or 1:100 (Intel x86)
- Only 13 nJ / VM instruction (ARM Cortex M0+)
- Only 130 nJ / Word compilation (ARM Cortex M0+)
- Computation times of medium sized ANNs is below 1 Second (ARM Cortex M0+, 16 MHz, typically in the Milliseconds range)
- Compilation times of medium sized programs is below 1 Second (typically in the Milliseconds range)
- Start-up time of VM is below 100 ms (typically in the Milliseconds range)

Vector Operations (ANN)

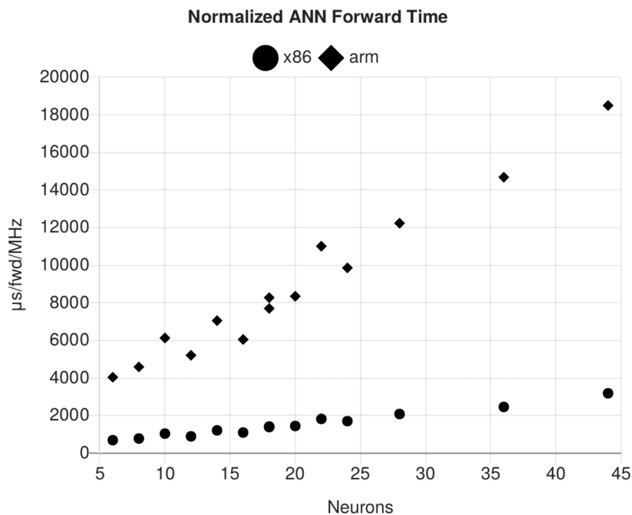


Fig. 7. Normalized computation times for ANNs of different size (with two, three, and four layers) and two different host platforms (Generic i5 x86 @2900 MHz and STM32F103C8 @72MHz) as a function of neurons

Vector Operations (ANN)

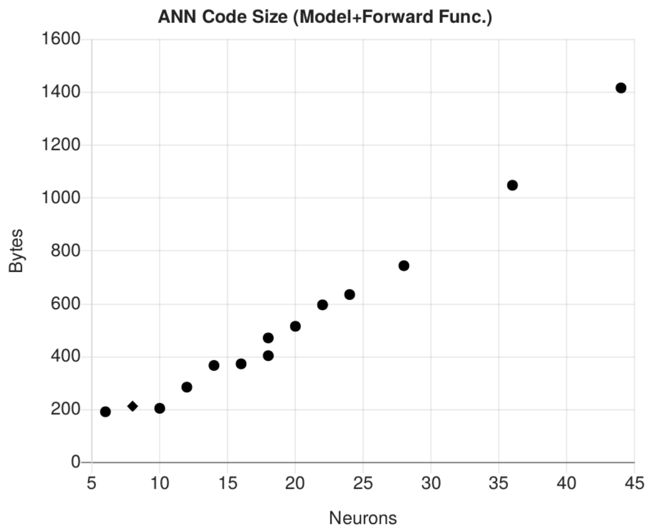


Fig. 8. Code size of ANN as a function of the number of neurons

Summary

A stack-based virtual machine architecture for low-resource, tiny embedded systems was introduced and analyzed. The overhead, even on very low-resource systems, is low with respect to typical running times under energy constraints and tasks to be performed in real-time

Summary

A stack-based virtual machine architecture for low-resource, tiny embedded systems was introduced and analyzed. The overhead, even on very low-resource systems, is low with respect to typical running times under energy constraints and tasks to be performed in real-time

A major feature is the tight coupling of a Text-to-Bytecode compiler with the Bytecode interpreter, ensuring robustness, security, stability, and interoperability in strong heterogeneous environments.

Summary

A stack-based virtual machine architecture for low-resource, tiny embedded systems was introduced and analyzed. The overhead, even on very low-resource systems, is low with respect to typical running times under energy constraints and tasks to be performed in real-time

A major feature is the tight coupling of a Text-to-Bytecode compiler with the Bytecode interpreter, ensuring robustness, security, stability, and interoperability in strong heterogeneous environments.

ML classification and regression models can be computed using integer arithmetic and a set of vector operations with low computation times and memory requirements.

Tiny Machine Learning Virtualization for IoT and Edge Computing using the REXA VM

Towards Learning Technical Systems

Stefan Bosse [1,2]

Christoph Polle [3]

[1]University of Bremen, Dept. Mathematics & Computer Science, Bremen, Germany

[2]University of Siegen, Dept. Mechanical Engineering, Siegen

[3]Faserinstitut Bremen (FIBRE), Bremen, Germany