RGen: A Tool for Generating RISC-V Compiler, Simulator, and Application Support

Derek Zijie Tu
derek.t@rioslab.org
RIOS Lab, Tsinghua University

Zhangxi Tan
xtan@rioslab.org
RIOS Lab, Tsinghua University

ABSTRACT

The landscape of technology is changing and it is becoming increasingly harder for proprietary architectures such as x86 and ARM to meet ever evolving demands. Instead it is important to look towards architectures that are easily malleable and scalable to meet the use cases of the industry, architectures such as RISC-V. RISC-V is an important vehicle for driving technological developments forward. One of the most important features of RISC-V is its modularity,. New instruction extensions to be easily implemented for it and increase RISC-V's performance in many domains. However, there are few ways to test such proposed instruction extensions on real world applications.

In this work an extension to Sail, RGen, is presented. RGen aims to provide an infrastructure for generating compiler, simulator, and real-world application support through machine readable specifications. A new extension for the RISC-V ISA, the matrix extension, is explored and implemented in the Sail language in order to test and validate RGen. The language extension proposed by RGen, RGenIR, is a high-level descriptor and is added as an extension to the base Sail language. RGen is used to generate code for a compiler, simulator and real-world application. Using the designed instructions as inputs, RGen is evaluated in terms of a code generation tool. Furthermore, the generated support allows for the new instructions to be run in PyTorch and evaluated on an emulated RISC-V environment. Further work is also discussed to explore and enhance RGen's scope, including comments towards limited code generation, testing environments, and making it open source.

KEYWORDS

Domain-specific Language, Code Generation, Compilers, Simulators

1 INTRODUCTION

When a new instruction extension is designed, a new machine specification must be written for it. For proprietary architectures, the specification might be sent to a vendor, and an engineer might have to wait for the vendor to finish an implementation before they can continue to the application layer. Many pain points can occur when both communicating with the vendor for implementations and the time loss waiting for the vendor. On the other hand, RISC-V allows the engineer to be completely self-reliant and implement the instruction extension themselves. Instruction extensions can be quickly designed, implemented, and validated on the application layer because the engineer is able to work closely with all parts of the system.

However, a great amount of time is lost translating the specification over to implementation. While more mature architectures such as ARM have tools to parse their specifications[5], RISC-V does not. Each time an implementation needs to be handwritten, it can lead to time losses from human errors such as incorrectly porting over bit fields, instruction mnemonics, execution functions, etc. If the engineering time taken to implement, test, and validate the instruction extension takes too long then the window of opportunity to utilize the instruction extension can be lost. Thus, it is important to have a machine readable specification in order to reduce the time lost from the implementation step of instruction design and allow for more effort to be put towards testing and validating the usage of the instruction extension. Alongside, there needs to be ecosystem support for the specification.

It is important to survey solutions that will help assess the performance of an instruction in real-world applications. Fully supported native RISC-V environments are rare, even more so those that can be easily modified to help test the performance of newly proposed instructions. This work uses sequential models for simulated environments which, while not representative of real hardware, can be used as a replacement for validation and co-design support of hardware development[10, 4, 12]. Unfortunately, the current software environment of RISC-V has few open source tools that can generate support for compilers, simulators, and applications from instruction semantics.

The goal of this work is to tackle these problems and support and speed up instruction extension design by connecting multiple open source projects. The Sail language is used as the origin point for code generation and takes on the role of a machine readable specification. RGen adds an extension to the Sail[3] language (RGenIR) for high-level descriptions of operations. Then, the backend of Sail is extended to compile and generate files that will aid users in regards to compilation toolchains (LLVM[8]), simulation environments (QEMU[1]), and real-world applications (PyTorch[9]). This gives any defined instruction extension full validation support. It also supports testing for performance by introducing the instruction extension to an environment in which it may be used often. In order to test and validate RGen, a simple design of a matrix extension is proposed and used in generated PyTorch operations to be evaluated on a RISC-V environment.

```
union clause ast = MACCO_MM : (regidx, regidx, regidx)

mapping clause encdec = MACCO_MM(ms2, ms1, md)
  <-> 0b000001 @ 0b0 @ ms2 @ ms1 @ 0b001 @ md @ 0
      ↪ b1011111

function clause execute(MACCO_MM(ms2, ms1, md)) = {
  let mbits : int = get_mtype_bits();
  ...
  let ms1_val : vector('n, dec, bits('m)) = read_mreg(
      ↪ num_elem, ms1);
  ...
  result : vector('o, dec, bits('m)) = undefined;
  foreach(i from 0 to (num_elem - 1)) {
    foreach(j from 0 to (num_elem - 1)) {
      let ms1_val_idx = signed(ms1_val[i]);
      let ms2_val_idx = signed(ms2_val[j]);
      let mult = ms1_val_idx * ms2_val_idx;
      let md_val_idx = md_val[i*num_elem + j];
      let add = md_val_idx + mult;
      result[i*num_elem + j] = add;
   };
  };
 write_mreg('o, md, result);
 RETIRE_SUCCESS
}
}

mapping clause assembly = MACCO_MM(ms2, ms1, md)
  <-> "macco.mm" ^ spc() ^ mreg_name(md) ^ sep() ^
      ↪ mreg_name(ms2) ^sep() ^ mreg_name(ms1)
```

Figure 1: Example RISC-V Sail definition for a *macco.mm* instruction.

2 MATRIX EXTENSION DESIGN

A simple matrix extension is proposed as an example target extension for RGen for testing and validation. In the implemented matrix extension any matrix-vector register (VR) can be freely designated to any matrix-accumulator register (MR). This decision was made to reduce implementation complexity and keep the design space open for modifications. Each VR represents a register within the file of registers that contains 512-bits of elements. Each MR represents a register from the file of 64x64 byte registers where the results of instructions are accumulated. Registers are defined in respect to bits because the matrix registers should adapt to the needs of the application.

Memory (load/stores) and arithmetic (outer/inner-product) operations are implemented for testing. These operations are based on the reverse-engineered Apple-AMX instructions found by Cawley[2]. The instruction semantics, at the time of writing, for the matrix extension instruction Matrix-Multiply Accumulate (*macco.mm*) can be seen in Figure 1. The definition is shortened for clarity, however it can still be seen that new matrix specific functions, such as matrix registers and matrix register manipulating functions, are added to the Sail definitions. In total, 22 new instructions are defined and implemented.

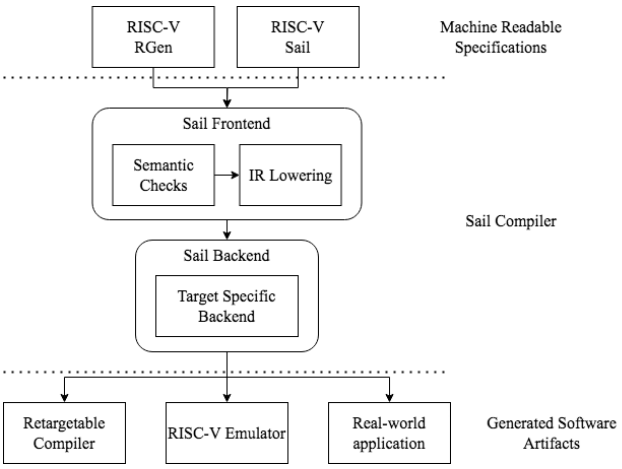

Figure 2: RGen's architecture with respect to inputs, Sail, and output artifacts.

3 RGEN ARCHITECTURE

RGen is an infrastructure that provides code generation support for a compiler, RISC-V emulator and real-world application. Since RGen introduces completely new types and targets to the Sail compiler, all sections of it must be extended. The architecture of RGen, including its inputs, middle-end and outputs can be seen in Figure 2.

A new language extension RGenIR is introduced as an input to Sail in order to give it new functionality. RGenIR allows Sail to quickly generate support for a real-world application, a feature it did not have before.

The frontend and backend components of Sail are also enhanced. The Sail frontend is extended to understand new types and patterns in order for it to correctly parse RGenIR. The Sail backend is extended in order for it to correctly translate the given definitions to a multitude of formats, each one different from the other.

3.1 RGenIR Design

Currently, the Sail Language is tailor made for instruction semantics. However, many user-level applications do not require that level of detail. As a result, RGenIR was devised as a way to provide Sail with the functionality to generate support for user-level applications with minimal effort.

The RGen language extension for Sail, RGenIR, follows the same syntactic rules of the base Sail Language. However, it adds a new *rgenir* clause, new patterns, and new type representations in the form of the *Tensor* type and *Scalar* type. The structure of the RGenIR takes inspiration from the TableGen definitions of MLIR[6]. It goal is to provide a simple definition to generate support for whole-tensor operations. The syntax of *rgenir* can be described using Figure 3.

An RGenIR definition is defined per-operation and will always start with an *rgenir clause* declaration followed by an *RGENIR_DEF* name that is used to group up similar RGenIR definitions. Two RgenIR definitions with the same

```
rgenir clause RGENIR_DEF = Sail_OpName <"sail_ir_name",
    ↪ [Operation, Properties]> {
  summary : string = "Describe high level functionality";
      ↪
  description : string = "Describe the operation in
      ↪ terms of inputs.";
  inputN : Tensor/Scalar(int/real) = undefined
  execute_* : string = "data_type @ inputN op inputN";
  output : Tensor/Scalar(int/real) = undefined;
  DEF_END;
}
```

Figure 3: Structure of an RGenIR definition.

RGENIR_DEF will be grouped together, but ones with a different *RGENIR_DEF* will not. This is to enhance user debuggability by putting the RGenIR definitions of a similar type closer together. Furthermore, it reduces code generation complexity of the backend.

The *RGENIR_DEF* is followed by a per-operation RGenIR definition. Following the declaration is the per-operation constructor, starting with the formal name of the operation *Sail_OpName*. The parameters for the operation are a comma-delimited list in order of: the intermediate representation level name of the operation, *sail_ir_name*, and another comma-delimited list describing the properties of the operation. These can be tied to backend specific implementations. The pattern of this section of an RGenIR definition is similar to the one used by MLIR[6] in TableGen. The MLIR pattern is simple and descriptive, giving a reader all the information they need about the operation in a high-level description. by using similar patterns, RGenIR can eventually be used to generate similar support for functions that use this MLIR pattern such as Torch-MLIR[11].

After the constructor is the body definition of RGenIR which is a high-level description of the operation. This section of the RGenIR definition uses the formal syntax defined by the Sail Language. In particular, it makes use of the Sail's syntax for variable declaration.

The variable definitions starts with a *VAR_NAME* must be of: *summary*, *description*, *inputN* (where *N* is an integer), *execute_** (where the asterisk is replaced by the extension name), *output*, or *DEF_END*.

RGenIR only supports the following types for variables: *string*, *Tensor*, or *Scalar*. *Strings* denote lines of text, while *Tensors* and *Scalars*, are types used for describing the execution of the operation itself. *Tensor* and *Scalar* types have a parameter (enclosed within round brackets) denoting which Sail primitive type (*int* for set of integers or *real* for set of real numbers) its data type is.

The value for *Tensor* and *Scalar* types is always *undefined*. The value for *strings* depend on if it is a *summary*, *description*, or *execute_* string*. *Summary* and *description* variable declarations are for describing the operation. The asterisk within an *execute_** declaration must be replaced with an associated instruction extension name. Furthermore,

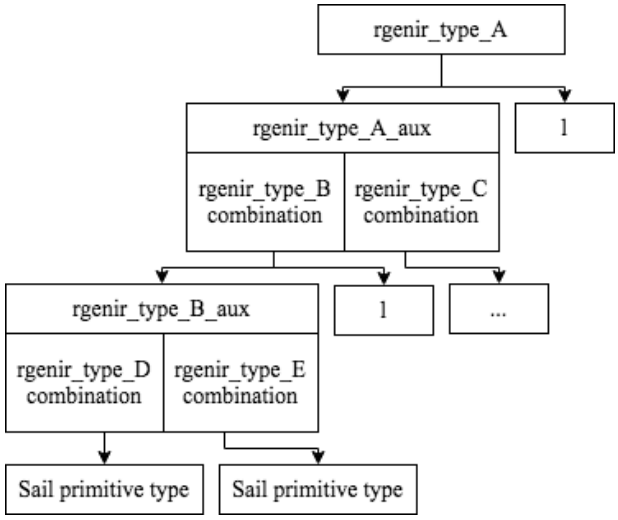

Figure 4: Tree relation of RGenIR types

*execute_** strings must contain the data type of the operation and the mathematical *op* relationship between its *input(s)*.

The syntax of the body of RGenIR was built this way to reuse much of the functionalities offered by the Sail language and reduces the amount of extra engineering work needed to modify it. The defined types were designed to ensure the operation definition conveys as much as possible in as little as possible.

3.2 RGenIR Sail Pattern Matching

In order to correctly pattern match for the RGenIR the Sail frontend must be extended. Firstly, different types relating to RGenIR are added to Sail.

These defined types are used to abstract different possible patterns for the Sail frontend. It allows for reusability and reduces code duplication. Currently, RGenIR types are simple and can be substituted with already defined Sail types. However, they have been defined in order to provide a clean split from the base code. This way, as RGenIR evolves and requires patterns that the original patterns do not want to support, it can easily add them without worrying about contaminating older Sail types.

The types defined for RGenIR have a tree-like relationship as seen in Figure 4. Each declared type has two forms. The first is the auxiliary type which is what the Sail compiler interacts with for semantic checking. The second is the non-auxiliary wrapper that is used to ensure each type has location information that can be used to trace the pattern (denoted by the *l* node). Each auxiliary type can have multiple patterns defined for it for pattern definition reusability and readability. The patterns themselves can consist of further high-level patterns, such as a combination of other RGenIR types, or a primitive Sail base type, such as strings.

Pattern matching is done using simple string matching logic where the Sail frontend parser will look through all the

possible combinations of a defined type to ensure correctness and does this in a tree-traversing fashion. Looking at Figure 4 as an example, for an RGenIR *A* type, the parser will expect to find a pattern containing RGenIR *B* type and RGenIR *C* type. An RGenIR *B* type is expected to have child patterns of RGenIR *D* type and RgenIR *E* type. Both of these types will have base Sail types as children that are the leaf nodes of the tree. Like with all tree-traversing algorithms, the pattern matching algorithm will stop at the leaf nodes and ensure the section of the string it is currently is the correct type. After confirming the patterns at the leaf node, the algorithm will propagate the results back upwards.

The implementation of type-checking is also done in a tree-like manner and follows the same logic as the base Sail types. Each type will be deconstructed into the types of its child nodes, which will also be deconstructed. This continues until all leaf nodes in the Sail AST are visited and checked to ensure type correctness. Once this process is done the original parent node will be returned and marked as correctly type checked.

4 RGEN SAIL BACKEND EXTENSION

The Sail backend is modularized so any new target outputs can be written as extension backends to Sail. Since the implementation of each backend is independent of the other implementation files are less bloated and much easier to manage. Each Sail backend will take the parseable AST from the Sail frontend and grab the relevant information needed to generate their respective files. The full Sail definition (example shown in Figure 1) is used to construct the generic ASTs used for code generation. Instruction encode and instruction decode information is generated through the defined *encdec* and *assembly* mappings. The functionality of an instruction is generated through the *execute* function of the generic Sail AST.

4.1 RGen LLVM Support

RGen offers support towards the LLVM through instruction legalization and selection. Helping LLVM understand how to legalize and select the instructions can be achieved by introducing full instruction semantics such as bit-fields, mnemonics, and argument string format. RGen does this by automatically generating LLVM's TableGen definitions[7].

For this work, three main files are generated: an instruction format TableGen definition, a scheduler TableGen definition, and finally an instruction information TableGen definition. The instruction decode (*encdec* mappings) information is used to generate the instruction format TableGen definitions while the instruction mnemonic inputs (*assembly* mappings) are used to generated both the scheduler TableGen definitions and the instruction information TableGen definitions.

Adding an instruction extension to the LLVM backend also involves adding the generated files to existing ones, as well as ensuring the LLVM understands any new types that may have been added with the extension. Many of these additions are handwritten because they require little effort. For this work, the matrix extension is added as a RISC-V subtarget feature by defining it in the *RISCV* TableGen file and *RISCVSubtarget* header file. The generated instruction scheduler definitions are added to the general *RISCVSchedule* TableGen file, and the generated instruction information is added to the *RISCVInstrInfo* TableGen file. Finally, because the matrix extension introduces a new matrix register type (MR) to the LLVM, information about the matrix registers is added to the *RISCVRegisterInfo* TableGen files as well as the *RISCVDisassembler* code.

4.2 RGen QEMU Support

There are two main sections within the QEMU execution loop that must be extended in order to enhance QEMU for it to correctly simulate the added instructions. These two sections are extending QEMU's decoder and Tiny Code Generator (TCG). Generating support for QEMU's decoder requires RGen to fully translate over the instruction semantics such as bit-fields, as well as the instruction mnemonic. Furthermore, the format of each instruction argument string must be correctly parsed in order to generate the correct instruction argument string alias required by QEMU.

Generating support for QEMU's TCG requires RGen to correctly set up all the instruction parameter based TCG variables as well as generating all the support to allow the instructions to modify the simulated CPU state. This includes creating helper methods that meticulously follow the QEMU code generation workflow. Furthermore, the helper methods must be generated with functionally correct code that compiles without further assistance from the engineer.

The instruction decode information from Sail (*encdec* mappings) is used to generate QEMU instruction decode files, while the instruction execution information from Sail (*execute* functions) is used to generate the QEMU TCG code.

4.3 RGen PyTorch Support

Enhancing PyTorch and adding the relevant matrix operations as PyTorch Tensor operations requires generating support for certain areas of PyTorch's interface and data flow. These two areas are PyTorch's Python API generator and the associated C++ backend implementations of each operator. Unlike code generation for LLVM and QEMU, RGen uses RGenIR for PyTorch support generation as PyTorch operations do not require instruction-level semantics. RGen generates a one-to-three mapping of input definitions to PyTorch operations. Each input definition has different parameters and fulfill different roles within the PyTorch ecosystem and RGen must ensure each operation is generated in a correct manner. Each C++ implementation follows the same basic logic: ensures the given data types within each input tensor is correct, sets up the tensors so that the data within them can be manipulated, and finally manipulates the data within each given tensor in accordance to associated PyTorch operation.

Table 1: Lines of code generated of target artifacts for matrix extension instructions.

Instruction/Op	Number of Handwritten Lines	LLVM	QEMU	PyTorch
macco.mm	30	26	61	x
fmacc64o.mm	30	26	62	x
mvv.mm	23	25	49	x
loadz.mm	36	25	53	x
storez.mm	42	25	49	x
All instructions (22)	696	593	1226	x
Sail_MACCOMMOp	10	x	x	41

Table 2: Code generation results for RGen targets

Code Generation Target	Code Generation Time(s)	Code Size (%)
LLVM	11.613	0.017
QEMU	12.124	0.83
PyTorch	0.23	0.0057

5 TESTING AND EVALUATION

RGen's code generation has been evaluated in terms of lines of code generated, code generation time, generated code size, and stopwatch runtime. All work was done run on a GNU/Linux x86_64 distribution with an Intel(R) Xeon(R) Silver 4214 CPU at 2.20GHz. Results related to lines of code generated can be seen in Table 1. Results for code generation time and generated code size can be seen in Table 2.

5.1 Lines Of Code Generated

Lines of code generated are measured per target backend. Table 1 shows the amount of lines of code generated for some of the matrix extension instructions using the Sail language for instruction semantics (the targets are LLVM and QEMU in columns three and four respectively). The average ratio of handwritten to generated code is 1:2.6. Arithmetic instructions such as *fmacc64o.mm* and *macco.mm* have higher ratios of 1:2.9 because there are no optimizations done for them and many lines of code are generated 1 to 1. On the other hand, memory instructions such as *loadz.mm* and *storez.mm* have lower ratios of 1:2.2 and 1:1.7 respectively because RGen tries to optimize and combine some lines of code. PyTorch files (column five) were generated using RGenIR and the ratio of amount of handwritten code to generated PyTorch code is 1:4. Both results show that RGen has promise in ensuring users can save a lot of time that would otherwise be spent writing individual implementations.

5.2 Code Generation Time

The code generation times can be seen in the second column of Table 2. Times were recorded using Linux's *time* function and the *real* value was recorded. The longest amount of time is 12 seconds with the fastest being 0.23 seconds. RGen compilation time is fast enough that the amount of time saved by those 12 seconds of compilation is already a great boon, especially when considering the result is support for three very different target artifacts. The amount of time required by RGen to generate code is acceptable for validation, verification, and testing purposes.

5.3 Generated Compiled Code Size

The third column of Table 2 shows the size of the code when it is compiled into its respective target application. Byte size numbers for QEMU and PyTorch were calculated by analyzing the byte numbers for the compiled binaries that used the generated code. The byte size number for LLVM was found by calculating the approximate size of elements and classes added due to the inclusion of the matrix extension. The percentages are then found by calculating how large the generated code is in relation to the full size of the application. The sizes of the compiled code are small and represent a very small percentage portion of the application they are a part of. Generated support for QEMU which is the largest, is still less than 1% of the whole QEMU application. PyTorch in particular has a low amount because the implementation of the Ops are always simple small lines of code, resulting in smaller compiled code sizes. The compiled size of RGen's generated code is acceptable and will not negatively affect the size of the overall application.

5.4 Runtime Metrics

The native RISC-V environment was emulated on QEMU7.2.0 using a Fedora Rawhide 20200108 kernel and a Fedora RISC-V 64-bit Rawhide drive with 64 Gigabytes of memory and 4 cores. The generated matrix extension instructions were run in generated PyTorch Matrix-Multiply Accumulate (MMA) Ops. Each Op was run as follows: 64-bit operations with eight element vectors, 32-bit operations with sixteen element vectors, and 16-bit operations with 32 element vectors. Stopwatch based runtime was calculated by using the Python

Table 3: Runtime (seconds) of generated PyTorch MMA Ops

Data Type	Time (s)
Integer32	0.053
Float64	0.016
Float32	0.0046

time package to calculate the time between starting the Op and ending it. Results are shown in Table 3.

The runtime in seconds of the generated PyTorch Ops all take less than a second, and most of the operations take less than 0.02 seconds. These are promising numbers considering that the PyTorch Ops are being run on a emulated environment, which is naturally going to be slower than a non-emulated one. Since runtimes are low, the support generated by RGen can reliably be used for validation and performance-based evaluation of instruction extensions under real-world applications.

6 FUTURE WORK

Future considerations for this work are: increasing the robustness of the Sail generic AST to C++ translator, enhancing the testing environment, and opening RGen up to the community.

The current code generation from the AST to C++ is a simple one-to-one translator utilizing basic logic to filter out AST information that may not be needed. This can result in some unused lines being generated. Furthermore, this disallows for more complex statements to be written in the Sail language. The level of the translation is sufficient for the current iteration of the matrix extension. However, once the matrix extension is expanded (or more complex extensions are tested) and requires more than simple statement logic, the code generation logic will also need to be enhanced as well.

Due to the lack of easily attainable and modifiable hardware-level simulators it is difficult to quickly test performance and energy efficiency of the added extensions. Using software-level simulators only allows for functionality verification and ballpark estimations on the performance values of the instructions. To enhance the testing environment and give more accurate performance numbers future work can entail code generation of hardware languages, and introducing a cycle-accurate simulator. While giving better performance metrics, both solutions can also be helpful in many other areas. Introducing hardware language code generation will also allow for hardware and software co-driven development, enhancing the development environment of both sides.

At the time of writing, RGen is not in a state that allows it to be easily used by the open source community. However, there are plans to open this work to the community.

7 CONCLUSION

This work introduces and describes an infrastructure for realizing a new RISC-V extension from design to running on a real-world application. A simple matrix extension for RISC-V is proposed, defined in the Sail Language, and used to target LLVM and QEMU code generation. The Sail Language is also extended to understand a new type, RGenIR, so high-level descriptors of operations can be used for PyTorch code generation. RGen's code generation is evaluated and found to be acceptable. The matrix extension is run in PyTorch on an emulated RISC-V environment with promising results.

Future work for RGen is discussed and it is acknowledged that there are improvements to be made in regards to the Sail generic AST to C++ translation and testing environment. This work is a step forward for providing quick access to preliminary testing and evaluation of simple extensions. However, there is more work to be done to increase its flexibility and scope and open RGen up to the community. Though there are enhancements to be made, this work has shown promising results for automatic code generation and validation of instructions from a single language source.

ACKNOWLEDGMENTS

I would like to give thanks to my advisor Zhangxi Tan for helping give comments and pointers on the direction of my work.

REFERENCES

[1] Fabrice Bellard. 2005. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (ATEC '05). Anaheim, CA, (Jan. 2005), 41–46.

[2] Peter Cawley. 2022. Apple amx instruction set. (2022). https://github.com/corsix/amx.

[3] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture (Waikiki)*. (Dec. 2015), 635–646. DOI: 10.1145/2830772.2830775.

[4] P. Gupta. 2002. Hardware-software codesign. *IEEE Potentials*, 20, 5, 31–32. DOI: 10.1109/45.983337.

[5] James Lamar. 2019. Galoisinc/arm-asl-parser: parsing tools for arm's asl. (Feb. 2019). https://github.com/GaloisInc/arm-asl-parser.

[6] Chris Lattner et al. 2020. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054. https://arxiv.org/abs/2002.11054 arXiv: 2002.11054.

[7] LLVM-Admin Team. 2003. Tablegen programmer's reference. (2003). https://llvm.org/docs/TableGen/ProgRef.html.

[8] LLVM-Admin Team. 2002. The llvm compiler infrastructure project. (2002). https://llvm.org/.

[9] Adam Paszke et al. 2019. Pytorch: an imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703. http://arxiv.org/abs/1912.01703 arXiv: 1912.01703.

[10] M.K. Purvis and D.W. Franke. 1992. An overview of hardware/-software codesign. In *[Proceedings] 1992 IEEE International Symposium on Circuits and Systems*. Vol. 6, 2665–2668 vol.6. DOI: 10.1109/ISCAS.1992.230672.

[11] Sean Silva, Yi Zhang, and Stella Laurenzo. 2020. Torch-mlir. (2020). https://github.com/llvm/torch-mlir.

[12] Juergen Teich. 2012. Hardware/software codesign: the past, the present, and predicting the future. *Proceedings of The IEEE - PIEEE*, 100, (May 2012), 1411–1430. DOI: 10.1109/JPROC.2011.2182009.