Towards a Domain-Specific Language for Hardware In-The-Loop Testing of Mixed-Signal Embedded Systems

Graham Power
powerg@mcmaster.ca
McMaster University
Hamilton, Ontario, Canada

ABSTRACT

Testing and validation of an embedded system is one of the most complex challenges of the design. While software testing has many tools to help automate this process, the electrical validation is left mainly as manual work. As such, it takes extensive human resources and time to validate successfully. Ideally, an automated solution would be provided. However, that generally means writing code, and we find there is usually little overlap between those responsible for designing the electrical system and those who can write code. So, even more human resources and time are required to write automated tests. In this paper, we propose a domain-specific language to allow electrical systems designers to write automated tests for their designs easily and quickly. We first focus on finding and accelerating an optimal algorithm for validating arbitrary time-series signals. Then, we implement a domain-specific language capable of auto-generating these algorithms and all supporting code. This approach greatly reduces the time and effort required to validate electrical designs for embedded systems by eliminating the need for manual validation and dedicated engineers to write automated tests.

KEYWORDS

domain-specific languages, testing, validation, hardware in-the-loop

ACM Reference Format:
Graham Power. 2024. Towards a Domain-Specific Language for Hardware In-The-Loop Testing of Mixed-Signal Embedded Systems. In *Proceedings of ACM Conference (ECE-718).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

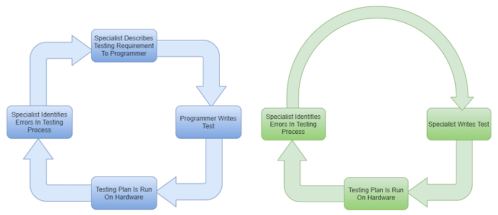
Specialist Describes Testing Requirement To Programmer
Programmer Writes Test
Testing Plan Is Run On Hardware
Specialist Identifies Errors In Testing Process
Specialist Writes Test
Testing Plan Is Run On Hardware
Specialist Identifies Errors In Testing Process

Figure 1: A Comparison of the Testing and Validation Process With the Involvment of a Programmer to Write Automated Tests (Left) and With a Domain-Specific Language Useable by the System Specialist

1 INTRODUCTION

Testing and validation are critical parts of embedded systems design, especially if one wants to be confident in the functionality of the system before releasing it for public use or remote operation such as in a spacecraft [12], leaving a desire for many companies to automate this work without losing confidence in the validation process. Testing automation tools already exist for software applications such as the Unity test framework [10], Pytest [8], and many more. But to our knowledge, no such automation solution exists for mixed-signal validation of embedded systems. This means that dedicated validation engineers are required to write these tests. However, there is a knowledge gap between those who designed the system and know how it is supposed to function and those who know how to write tests to validate its functionality. This leaves for a significant overhead in the testing and validation process of these systems 1. Ideally, the electrical designers would have an easy way to write their own tests, eliminating this significant overhead 1.

This paper aims to provide a domain-specific language to aid in code-generation of automated testing for arbitrary time-series signals as a step towards testing full mixed-signal embedded systems. We will be focusing on validating exponential pulses in our DSL since they both fit the bill for a mixed-signal and validation of these types of signals has real-world applications such as detector validation on spacecraft payloads like the McMaster PRESET mission [1]. For arbitrary time-series signals, we use dynamic time warping as our metric for signal validation as it has been shown to be a good metric for this type of application [4]. However, one challenge is to make dynamic time warping fast without losing accuracy since the original algorithm has quadratic time and space complexity [2].

In this paper, we first review the existing work in accelerating dynamic time warping and in domain-specific languages for hardware testing. We then compare the relative speedups and accuracy of different DTW implementations, both against the existing UCR dataset [6] and a custom exponential pulse dataset. Then, we design and propose a domain-specific language for generating executable code to run these comparisons in an automated way. The design choices behind the DSL are explained, and an example program is executed. Finally, we propose future work to extend the functionality of our domain-specific language.

2 BACKGROUND

Before designing and implementing a domain-specific language for hardware-in-the-loop testing, we must choose an appropriate metric to use in our signal comparison. So, we will first look at different approaches to signal comparison and their effectiveness before exploring existing domain-specific languages.

2.1 Signal Comparison Approaches

Thonnessen et al. [14] proposed multiple algorithms for signal validation for hardware-in-the-loop testing, such as tolerance domain and set match. While the algorithms are highly accurate, they were designed primarily for pulse-width modulated (PWM) signals or other simple digital signals. As such, they will not be reliable in a mixed signal validation environment.

Earlier, Zhang et al. [15] proposed the WCOMP tool for mixed-signal validation in memory design. While the tool does provide algorithms such as pattern comparison, which uses Wiener filtering, they specifically target signals found in memory design, so it is unclear how effective these tools will be on arbitrary time series.

In a comparison work, Bagnall et al. [4] attempted to compare 19 algorithms for time series classification over 85 different problems using the University of California, Riverside time series classification archive [6]. They found that two algorithms consistently outperformed the others in terms of classification accuracy: Collective of Transformation Ensembles (COTE) and Dynamic Time Warping (DTW). Although COTE outperformed DTW with an average accuracy gain of 8% over DTW, it is incredibly computationally intensive. Bagnall et al. point out that COTE is bounded by the Shapelet Transform, which has $O(n^2m^4)$ time complexity. This is, unfortunately, unusable for the real-time signal validation required by hardware-in-the-loop testing. The second best algorithm was DTW, which has a much better, but still unideal, time complexity of $O(nm)$. It is for this reason that DTW will be used as the basis of signal validation.

Many attempts have been made in the past to accelerate DTW. Mueen and Keogh have a seminar presentation on extracting optimal performance from DTW [2]. They provide many approaches to improve both the time and memory complexity DTW, some of which come at the cost of accuracy. These approaches include limiting the warping distance, deriving a lower bound on the distance reported by DTW for time series classification, early abandon of the DTW computation, and storing only partial results in the DTW cost matrix to reduce its size. These approaches and some novel ones we present will be compared in later sections of this paper.

Improvements in DTW algorithms, which trade off accuracy for speed, have also been made. Silva and Batista proposed a pruning method to replace early abandonment with an exact approach that does not trade speed for accuracy [13]. However, they only compare their speedup against DTW algorithms, such as the basic DTW and an unrealistic OracleDTW and not the early abandoned implementations they aim to replace.

The University of California, Riverside, has also spent many years generating and curating an archive of time-series data to compare the accuracy of different classification algorithms. As has been done in previous comparisons [4]. We will use this dataset to validate that our attempts to accelerate DTW do not come at the cost of accuracy loss and that DTW is a valid classification method for exponential pulses.

2.2 Domain-Specific Languages

Many domain-specific languages for testing embedded systems have been proposed. And while none perfectly fit our application, parts of them can be built upon to provide a complete solution for hardware-in-the-loop testing for embedded systems.

Maldonado and Garcia propose a domain-specific language to enable cloud testing of mobile applications [9]. Their platform provides a clean and concise syntax, allowing for fewer lines of code to be written without the user having any cloud computing knowledge. Though this DSL is designed for testing on mass-market devices and not custom electronics, it cannot be applied to most use cases. However, their language provides a clean and simple way to describe the hardware being used in the test and has served as the inspiration for our languages' hardware description syntax.

Arrieta et al. propose automated test generation for industrial elevators [3]. This paper provides an incredibly detailed description of the syntax used for their language and writes very closely to a human sentence as opposed to code. This is ideal for our application, where the writer of the tests rarely has any programming background. Unfortunately, their target application is specifically industrial elevators. Although this work shows promise as something that could be generalized to support testing on arbitrary hardware, the authors have yet to do so. Their syntax, with the exception of the hardware description, was used as the basis of our language's syntax.

Barriga et al. provide a much more generalized approach to hardware testing [5] than [3]. They provide a language for testing Internet of Things (IoT) devices with arbitrary sensors and actuators. Unfortunately, they only support simulation of these devices, which is not hardware-in-the-loop testing. Their work can be used as a reference for how a more application-specific language like [3] can be generalized to support testing of arbitrary hardware.

Shin et al. propose a domain-specific language for hardware-in-the-loop testing of cyber-physical systems [12]. This is exactly the type of language we need, though they seem to only support simple comparisons between numbers such as equals, less than, greater than etc. and not signal matching for arbitrary time-series. For mixed-signal systems, we need the ability to compare time-series signals as well.

Finally, Anjorin et al. proposed a domain-specific language for the validation of electronic circuit designs [11]. However, they only do design rule checks and other pre-fabrication tests to aid in the CAD process. They do not provide a way to check the system's functionality after fabrication.

3 METHODOLOGY

In this section, we first attempt to accelerate DTW. We describe the acceleration approaches and the reasoning behind them, as well as our approach to validate their speedup and effectiveness. Then, we will describe the implementation of a domain-specific language to facilitate the use of DTW in hardware-in-the-loop testing for embedded systems.

3.1 Dynamic Time Warping Implementation

To determine the best implementation of DTW for our language, we have compared ten different implementations of DTW. Mainly from [2] and [13]. The full list of algorithms compared is as follows:

- Original DTW (No Improvements)
- Swapped Loops

- Reduced Bitwidth
- Partial Cost Initialization
- Row/Column Threaded
- Row/Column Threaded Row/Column Interleaving in the Cost Matrix Memory
- Limited Warping
- Memoization
- Early Abandon
- Pruned DTW

3.1.1 *Original DTW.*
This is the original DTW algorithm with no improvements. The algorithm algorithm implementation is shown in Algorithm 1.

Algorithm 1 Original DTW Algorithm

```
$D(0:n, 0:n) \leftarrow inf$
$D(0,0) \leftarrow 0$
for $i = 0; i < n; i++$ do
    for $j = 0; j < m; j++$ do
        $cost \leftarrow (x(i) - y(j))^2$
        $lastMin \leftarrow min(D(i-1, j), D(i, j-1), D(i-1, j-1))$
        $D(i, j) \leftarrow cost + lastMin$
    end for
end for
$result \leftarrow D(n, m)$
```

3.1.2 *Swapped Loops.*
Like in GEMM, this proposed optimization involves switching the order of the loops in Algorithm 1. Unlike GEMM, however, swapping the loops in this case should reduce the spatial locality of the accessed cost matrix. However, since this is a relatively trivial change to make to the algorithm, it was added anyway.

3.1.3 *Reduced Bitwidth.*
The exact results of the DTW algorithm are not crucial in signal matching. As long as the relative results remain the same, one should still be able to classify a time series accurately. In this approach, instead of the standard double type used in C++, we will be using the Float type, which has half the bit-width. The overall algorithm is still the same as Algorithm 1. This should greatly improve the algorithm's speed since more of the cost matrix can now sit in a single cache line, reducing the number of cache misses from generating the matrix.

3.1.4 *Partial Cost Initialization.*
When building the cost matrix in DTW, all but the first row and column of the cost matrix are overridden. Since they are never accessed before being overridden, we don't have to initialize the cost matrix fully. This new approach can be seen in Algorithm 2

3.1.5 *Row/Column Threaded.*
DTW is fairly difficult to parallelize since the cost matrix calculations are almost entirely dependent on what came before. However, since any value of the cost matrix is dependent directly on the three values which are up and to the left of it, we should be able to calculate the cost matrix along a row and down a column in parallel without breaking this data dependency as shown in Figure 2

Algorithm 2 DTW Algorithm with Partial Cost Initialization

```
$D(1:n, 0) \leftarrow inf$
$D(0, 1:n) \leftarrow inf$
$D(0,0) \leftarrow 0$
for $i = 0; i < n; i++$ do
    for $j = 0; j < m; j++$ do
        $cost \leftarrow (x(i) - y(j))^2$
        $lastMin \leftarrow min(D(i-1, j), D(i, j-1), D(i-1, j-1))$
        $D(i, j) \leftarrow cost + lastMin$
    end for
end for
$result \leftarrow D(n, m)$
```

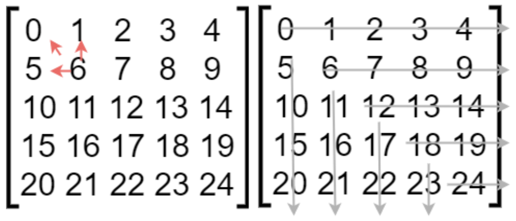

Figure 2: The Data Dependencies in the DTW Cost Matrix (Left) and the Order in Which Threads Computing Separate Rows and Columns Could be Run Without Breaking This Data Dependancy

3.1.6 *Row/Column Threaded Row/Column Interleaving in the Cost Matrix Memory.*
While this threading can potentially increase DTW's speed, all column threads exhibit poor cache locality, especially for larger matrix sizes. This can be solved by storing the matrix row/column interleaved. In this storage scheme, we would store the entire first row contiguously in memory, followed by the first column. This order would be repeated for all rows and columns down and to the right in the matrix. While the algorithm for calculating DTW would not change, we would need a new function for calculating the index in the memory to access given the row and column of the matrix. This function is shown in Algorithm 3

Algorithm 3 DTW Algorithm with Partial Cost Initialization

```
if $i > j$ then
    $index \leftarrow ((n+1)*(n))/2 + (j*n+i-2*j-1) - (j*(j-1))/2$
else
    $index \leftarrow (i*n+j-i) - (i*(i-1))/2$
end if
```

3.1.7 *Limit Warping.*
The process of limiting the warping of DTW is described quite nicely in [2]. Simply, the process involves limiting the amount of distance from the main diagonal of the matrix you are allowed to go before the cost becomes infinite. To achieve this, the algorithm limits the inner loop of the basic DTW algorithm to only go +-w away from the outer loop index i, where w is the warping window.

3.1.8 *Early Abandon.*

Early abandon is a method for accelerating the time to compare many time series using DTW to find the most optimal one. With early abandon, you set some lower bound for the final cost, and you are happy to accept this as a match without making further DTW comparisons. The exact value of the lower bound varies by implementation and is discussed in detail in [2].

3.1.9 *Memoization and Pruned DTW.*

Since both these implementations are from other papers, full details on their implementations can be found [2] and [13], respectively.

3.1.10 *Domain Specific Language Implementation.*

In this section, we will describe the domain-specific language we will implement, its features, syntax, and justification behind their choices. We will start by describing what the language should be like and how the syntax should look. We will then describe the method for parsing this language to produce an Abstract Syntax Tree (AST).

3.1.11 *Language Features.*

Since the language will require matching time series signals to expected values, users must be able to express an arbitrary signal to use in a comparison. Allowing a mathematical representation of these signals seems the most optimal since almost any signal can be represented with a math equation. To achieve this, we must allow for basic and even complex math to be described. To achieve this, we will implement a MATHFUNC keyword.

Next, we will need to support a description of the hardware being tested. We will do this with the HARDWARE keyword, which allows users to describe the type of hardware connected. Every hardware definition has the following: A package name used by the compiler to determine which library functions to use. A name, preceded by the NAME keyword, gives a name to the variable. A protocol preceded by the ON keyword gives the communication protocol to use only when the package library supports multiple protocols. A pin(s), preceded by the ON keyword and protocol if it exists, gives the physical pins of the device we are using to run the test code. A conversion function, preceded by the USING keyword, provides a math function which will be used to convert all inputs/outputs to/from this hardware.

We also want the users to be able to create simple variables, both for use in mathematical equations and to provide context to the code, such as the name of the test or the author. To do this, we will provide two keywords: the CONSTANT keyword for numerical constants and the CONTEXT keyword for strings used to describe the code.

We also want to be able to control the output of some GPIO pins to aid in executing the test. This will be done with the SET keyword.

Finally, the heart of our language is the ability to compare signals. This is done with the ASSERT keyword. The user can then pass in a reference signal and a self-described signal to compare against, as well as a custom tolerance for the comparison if desired. We will support comparing both simple values with the EQUAL keyword and time-series with the MATCHES keyword.

A sample source code for the algorithm is shown below;

```
CONTEXT NAME = Some Testing Plan Name Here
CONTEXT AUTHOR = Graham Power
HARDWARE BK2194 NAMED scope ON usb com4
HARDWARE SDM3045X NAMED multimeter ON usb com5
HARDWARE GPO NAMED trig_pulse ON DIGITAL 3
HARDWARE GPI NAMED pulse ON ANALOG 5 USING convert_pulse
CONSTANT Eo = 1.5
CONSTANT alpha = 0.015
CONSTANT beta = 0.0056
MATHFUNC convert_pulse (raw) = raw*(3.3/(212))/100/0.05*1000
MATHFUNC pulse_func (t) = Eo*(exp(0-alpha*t) - exp(0-beta*t))
ASSERT multimeter VOLTAGE EQUAL 3.3V TOLERANCE 5
ASSERT multimeter CURRENT GREATERTHEN 10mA AND LESSTHEN 600mA TOLERANCE 10mA
SET trig_pulse HIGH
ASSERT pulse_in MATCHES pulse_func TOLERANCE 10
```

3.1.12 *Language Implementation.*

The language is implemented using a Recursive Descent Parser [7] whose architecture is shown in Figure 3. A recursive descent parser works by parsing down its structure recursively, which gives the expressions at the very bottom of the tree the highest priority when it comes to parsing.

The bottom of our tree is built-in expressions. This is a list of keywords that map to a variable in the global space that is defined by default in the language. This allows us to easily identify and parse things like the CONTEXT keyword above that we have primary expressions, which function just like any other in our language. Primary expressions consist of any variable names, numbers, or parenthesis. Above that, we have two types of math expressions: multiplicative and additive. These allow our language to support all the basic math operations like addition and subtraction. They are split into two separate nodes of the tree as an easy way to enforce an order of operations. Since the multiplicative node is below the additive in the tree, any multiplication and division will always be evaluated first when building the AST.

Next in the tree, we have assignment expressions, which allow us to support the use of the equals sign in our language. This lets users define their own math functions or redefine context variables such as the user name and author.

Finally, at the top of the tree, we have native function calls, object expressions, and variable declarations. Native function calls are how we choose to support the assert function. Object expressions are how we define the hardware being used in the test. The HARDWARE keyword indicates to the parser to create a new hardware object whose parameters are determined by the keywords following the hardware declaration. For simplicity, this language does not allow users to create custom objects.

4 EXPERIMENTAL RESULTS

4.1 DTW Implementation Comparison

The accuracy of all DTW implementations was validated by running against the UCR Archive. The results can be seen in Figure 4. All implementations of DTW had an accuracy of around 5%, beating out Euclidian distance, which had an accuracy of around 12%.

We also attempted to create an entry to the UCR Archive for exponential pulses, a new type of signal for their archive, and one we provided earlier in this paper as an example use case for our domain-specific language. The dataset, when tested against DTW

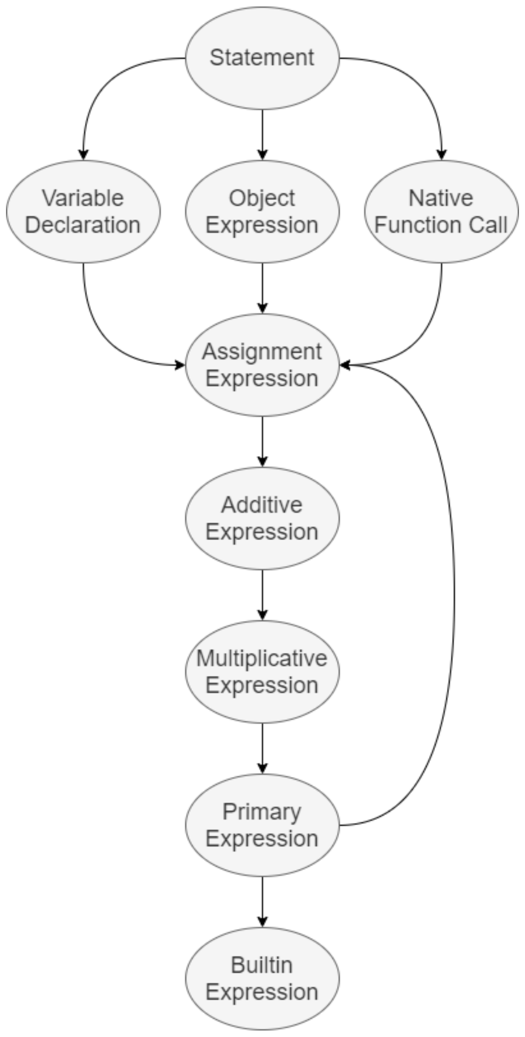

Figure 3: The Structure of the Recursive Descent Parser Used in our Domain-Specific Language Implementation

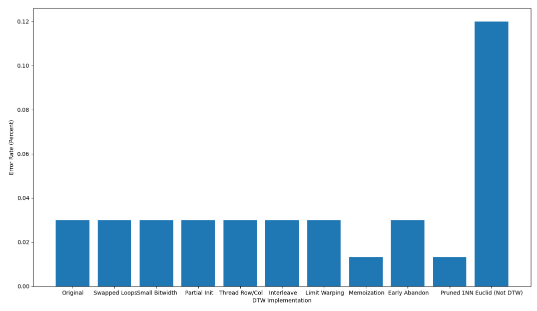

Figure 4: Accuracies of Different Implementations of Dynamic Time Warping

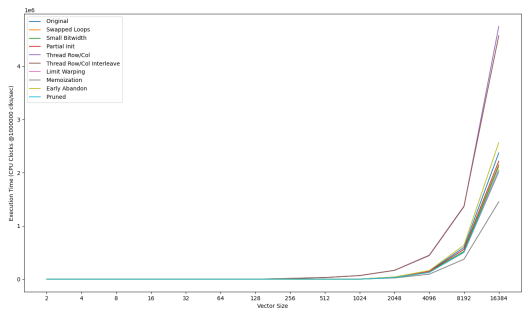

Figure 5: Execution Time of Different Implementation of Dynamic Time Warping. Timing Results are Gathered Using the clock() Function of the Standard C++ Library, which Returns the Number of CPU Clock Ticks

and Euclidian distance, resulted in an error rate of around 50% for DTW and 70% for Euclid distance. DTW still outperformed Euclid distance, again showing it is best suited for these types of time series classifications, perhaps with the exception of COTE for non-real-time applications.

The last comparison of DTW algorithms was for their speedup from the base algorithm. The best implementations by far were memorization, limited warping, partial cost matrix initialization, and smaller bit-widths. Luckily, these implementations are not mutually exclusive and so could all be combined into a single implementation for use by our domain-specific language. While these implementations are faster, they still appear to scale quadratically, suggesting that running comparisons on very large vectors should be avoided for the sake of performance.

Surprisingly, our threaded application was much worse than even the original DTW implementation. This is possibly because the overhead of threading outweighs any benefit gained from having two threads running at a time. Though storing the matrix row/column interleaved did provide a performance gain over just plain multithreading, suggesting that that approach did improve the cache locality.

4.2 Domain-Specific Language Implementation

The implementation of our domain-specific language consists of the following parts: A lexer to tokenize source code, a parser which generates an AST using recursive descent parsing, and a runtime which attempts to interpret and execute the AST live in Python. During the final phase of the project, I was able to implement support for all but native function calls in the parser. Unfortunately, the runtime does not support native function calls or object expressions. As a result, while I am able to generate a full AST for most of the language's proposed features, I cannot actually execute them in Python. Examples of the AST and results generated by this domain-specific language can be found in the code artifact discussed in Appendix A. The ASTs generated by the language are very verbose and would not fit neatly into a project report.

5 FUTURE WORK

In the future, I would like to extend both the AST parser and runtime to support the execution of native function calls so I can show a full example of the language using DTW to compare an input signal to a mathematical model. I would also like to add support for executing the language in C or C++, so it can be run on embedded systems like an STM board and not just a Raspberry Pi.

ACKNOWLEDGMENTS

I would like to thank my instructor for this course, Dr. Kazem, for meeting with me regularly to provide valuable insights and get me on track with the course project.

REFERENCES

[1] [n. d.]. McMaster PRESET. https://mcmasterneudose.ca/updates. Accessed: 2023-12-14.
[2] Eamonn J. Keogh Abdullah Mueen. 2106. Extracting Optimal Performance from Dynamic Time Warping. Presented at 22nd ACM SIIGDD Conference on Knowledge Discovery and Data Mining, San Francisco, California.
[3] Aitor Arrieta, Maialen Otaegi, Liping Han, Goiuria Sagardui, Shaukat Ali, and Maite Arratibel. 2022. Automating Test Oracle Generation in DevOps for Industrial Elevators. *Proceedings - 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022* (2022), 284 – 288. https://doi.org/10.1109/SANER53432.2022.00044
[4] Anthony Bagnall, Jason Lines, Aaron Bostrom, James Large, and Eamonn Keogh. 2017. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery* 31, 3 (2017), 606 – 660. https://doi.org/10.1007/s10618-016-0483-9 Cited by: 844; All Open Access, Green Open Access, Hybrid Gold Open Access.
[5] Jose A. Barriga, Pedro J. Clemente, Encarna Sosa-Sanchez, and Alvaro E. Prieto. 2021. SimulateIoT: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments. *IEEE Access* 9 (2021), 92531 – 92552. https://doi.org/10.1109/ACCESS.2021.3092528
[6] Hoang Anh Dau, Eamonn Keogh, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Yanping, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, Gustavo Batista, and Hexagon-ML. 2018. The UCR Time Series Classification Archive. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/.
[7] Anthony J. Dos Reis. 2012. *Recursive-Descent Parsing*. 185–214. https://doi.org/10.1002/9781118112762.ch9
[8] Holger Krekel. [n. d.]. *PyTest*. https://docs.pytest.org/en/7.4.x/
[9] Sergio David Romero Maldonado and Jose Joaquin Bocanegra Garcia. 2022. Towards a Domain-Specific Language for Provisioning Multiple Cloud Testing Environments for Mobile Applications. *Proceedings - 3rd International Conference on Information Systems and Software Technologies, ICI2ST 2022* (2022), 178 – 184. https://doi.org/10.1109/ICI2ST57350.2022.00033
[10] Mark VanderVoord, Mike Karlesky, and Greg Williams. [n. d.]. *Unity*. https://www.throwtheswitch.org/unity
[11] Adrian Rumpold and Bernhard Bauer. 2019. A Metamodel and Model-based Design Rule Checking DSL for Verification and Validation of Electronic Circuit Designs. *International Conference on Model-Driven Engineering and Software Development* (2019), 315 – 322. https://doi.org/10.5220/0007381303150322
[12] Seung Yeob Shin, Karim Chaouch, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. 2021. Uncertainty-aware specification and analysis for hardware-in-the-loop testing of cyber-physical systems. *Journal of Systems and Software* 171 (2021). https://doi.org/10.1016/j.jss.2020.110813
[13] Diego F. Silva and Gustavo E.A.P.A. Batista. 2016. Speeding up all-pairwise dynamic time warping matrix calculation. *16th SIAM International Conference on Data Mining 2016, SDM 2016* (2016), 837 – 845. https://doi.org/10.1137/1.9781611974348.94 Cited by: 84.
[14] David Thönnessen, Stefan Rakel, Niklas Reinker, and Stefan Kowalewski. 2018. Matching Discrete Signals for Hardware-in-the-Loop-Testing of PLCs, Vol. 51. 229 – 234. https://doi.org/10.1016/j.ifacol.2018.06.267
[15] Peng Zhang, Wai-Shing Luk, Yu Song, Jiarong Tong, Pushan Tang, and Xuan Zeng. 2007. WCOMP: Waveform comparison tool for mixed-signal validation regression in memory design. *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC* (2007), 209 – 214. https://doi.org/10.1109/ASPDAC.2007.357987

A CODE ARTIFACT

The full source code for this project can be found at https://github.com/GNPower/TestpointDSL.

To run the code on Niagara, take the following steps:

(1) Clone this repository on Niagara
(2) CD to the Code/Code directory in the repository (you should see a niagara.sh file)
(3) chmod +x niagara.sh
(4) Either:
  (a) ./build.sh && ./run.sh
  (b) sbatch niagara.sh

The code will produce the following results:

(1) The accuracy results data in Code/Code/plotGen/ accuracy_results.txt
(2) The accuracy results graph in Code/Code/plotGen/ accuracy_results.png
(3) The timing results data in Code/Code/plotGen/ timing_results.txt
(4) The timing results graph in Code/Code/plotGen/ timing_results.png
(5) The accuracy of the custom UCR Pulse Dataset in either the console or the niagara output job file
(6) The AST resulting from source code in Code/Code/ transpilerDSL/test.hil in Code/Code/transpilerDSL/test.ast
(7) The AST resulting from source code in Code/Code/ transpilerDSL/test2.hil in Code/Code/transpilerDSL/test2.ast
(8) The execution results from source code in Code/Code/ transpilerDSL/test2.hil in Code/Code/transpilerDSL/test2.res

B SELF EVALUATION

B.1 Clear Problem Statement and Gap Analysis

I believe I was able to clearly define the problem, why it exists, and how I hope to solve it. I hope to have improved since the seminar presentation since it seemed then that I had trouble getting my justification for why this is important across. Overall I would give myself 12 out of 15%.

B.2 Deep understanding of existing methodologies...

I presented an in-depth analysis of the fields of both DTW acceleration and domain-specific languages for hardware-in-the-loop testing. It is this part of my project I spent the most time on researching, especially on domain-specific language implementations. I believe this is by far the strongest part of my project and would give myself 10 out of 10%.

B.3 A methodology that is clearly about performance optimization and automation

I have made an attempt to propose both accelerations to the DTW algorithm, which I compare against others, and to propose a domain-specific language for automating code generation. I believe that both of these contributions fit well into the first and second sections of this course, respectively. I think my strongest part here is my proposal of a domain-specific language since, although ASTs and their uses were discussed in detail in the class, how to generate them was something I had to learn on my own from the book I referenced. Admittedly, the weaker side of my methodology was on

DTW acceleration since what I was able to propose as novel only allowed for two threads running in parallel, which is far from the performance gains from something like a full GEMM optimization. For these reasons, I would give myself a 12 out of 15%.

B.4 An extensive experimental result that addresses your problem statement

This is where I was the weakest. Unfortunately, my DTW acceleration attempts did not pan out as the threading overhead was much greater than the speedup expected, especially for smaller vector sizes. I also needed more time to fully implement my domain-specific language, as generating and interpreting an AST proved much harder than expected. I think the project scope was much bigger than I was reasonably able to accomplish in a single semester, especially with two courses, and I should have focused on either the domain-specific language or the DTW acceleration. For these reasons, I would give myself a 10 of 15%.

B.5 Presentation and technical writing quality

I believe the presentation and technical writing quality overall to be quite good. However, there is a glaring exception. I used the sigconf template as requested, and no matter what I tried, I could not get Latex to stop generating blank pages in the middle of the paper. This has driven me crazy trying to figure it out. Besides that, I believe the technical content to be of fairly high quality. For these reasons, I would give myself a 4 out of 5%.

B.6 Artifact packaging

This is the simplest one to evaluate. The code has been tested and runs well on Niagara, producing all the results I expected. So, I would give myself a 10 of 10%.

B.7 Overall

Overall, I gave myself a 58 of 70%, or an 82% on just the project. I put a lot of work into the project over the course of the semester, and I hope it shows in the paper. However, I would not rate myself high mainly because I don't believe this is a conference-quality paper yet, and given more time, I would like to have fully implemented my domain-specific language to make my results more substantial.