AI-Assisted Design-Space Analysis of High-Performance Arm Processors

Joseph Moore
University of Bristol
Bristol, United Kingdom
zi23956@bristol.ac.uk

Tom Deakin
University of Bristol
Bristol, United Kingdom
tom.deakin@bristol.ac.uk

Simon McIntosh-Smith
University of Bristol
Bristol, United Kingdom
s.mcintosh-smith@bristol.ac.uk

***Abstract*—This work quantifies the impact of microarchitectural features in modern high-performance Arm CPUs. To combat a parameter space that is too large to traverse naively, we employ a decision tree regression machine learning model to predict the number of execution cycles with 93.38% accuracy compared to the simulated cycles. We build on previous work by specializing our design to real-world HPC workloads and modernize our approach with updated search spaces, improved simulation frameworks, and over 180,000 simulated data points. We find empirically that vector length typically has the largest impact on HPC code performance at 25.91% of our performance weighting, followed by memory performance across all levels of the memory hierarchy, and the size of the reorder buffer and register files. Our results motivate deeper exploration of these parameters in both hardware design and simulation, as well as advancing the modelling of architectural simulation through the use of machine learning.**

I. Introduction

Hardware/Software co-design has become increasingly prevalent in an age where Moore's law no longer guarantees large performance gains with each processor iteration [1]. With less performance on the table from conventional architectural improvements, the design of high-performance CPUs must be carefully crafted to perform well for a set of target applications, ranging from general-purpose compute tasks to specialized applications or benchmarks, such as SPEC CPU or SPEChpc. Fujitsu's A64FX is a strong example of the success of this method, designed in collaboration with RIKEN R-CCS to target performance for a finite set of HPC applications [2].

The prototyping undertaken to design such CPUs often occurs in simulation at a variety of granularities. Register-Transfer Level (RTL) simulations provide excellent modelling at the cost of simulation speed, while many higher-level microarchitectural simulators lack either the speed or cycle accuracy that would be required to distinctively assess performance differences of subtly different CPU designs.

Design-space optimisation of CPU design is not a new topic, but previous research has typically focused on outdated ranges of parameters [3]–[7], or narrowly focused on few parameters in a small search space [8], limiting the scope of the research to exclude future-generations of hardware designs. For larger parameter spaces, we have to be more clever about modelling performance, as brute force approaches become untenable. Machine learning (ML) can aid this search, requiring only a sample of the search space to accurately predict target variables, or by guiding the parameter search towards optimal values. This has previously been coupled with simulation tools to assess various configurations, but again focuses on a narrow scope of parameters with the intention of optimisation of die size, cost, or energy to solution [9].

This work aims to provide a modernized approach to the problem of analysing the performance impact of several microarchitectual parameters. This provides insight into the complex relationships between various design decisions and guides future research towards where significant performance improvements can be unlocked by highlighting where bottlenecks lie. In particular, this problem is motivated by the goal of optimizing only the features that have a large impact on the compute performance of a CPU for the codes that it will be used on.

This problem is difficult due to the size of the parameter space coupled with the slow speed of simulation compared to hardware, meaning smaller subsets of the space must be investigated at one time. We tackle this problem through the use of a surrogate ML model, learning a reduced representation of our simulation framework from a sample of the data which allows us to accurately reason about the full parameter space without the constraint of having to simulate it all.

Using over 180,000 simulated data points across our design-space and four HPC workloads, including a selection from SPEChpc 2021, we train a Decision-Tree Regression model for each application to predict the number of execution cycles depending on the microarchitectural configuration provided. After validating the accuracy of our model, we examine the average weights of each parameter that define our model and score each parameter based on an importance percentage to quantify the predicted impact they have on performance. We then refer back to our original dataset to provide context for our model's predictions.

The contributions of this work are (i) the significant collection of simulated data across the CPU architecture parameter space for multiple HPC codes, (ii) the mapping of this dataset to an accurate surrogate ML model to an explainable latent-space representation, (iii) interpreting the architectural parameters in CPUs with the largest performance impact of HPC codes based on our representation, (iv) probing the simulated data to provide theoretical underpinnings that reason our findings.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00186

II. Related Work

Previous work in the late 2000s set the scene for ML-assisted design space analysis in microarchitectural design. In particular, P.J.Joseph et al. [3] presented the use of Linear Regression Models to analyse 26 microarchitectural parameters and conveyed the significance of their impact on SPEC CPU2000 codes. This study was limited in that the parameter space searched was still small, with only two values tested for each parameter, and the values are no longer representative of the size of modern HPC processors that we see today.

Lee et al. expanded upon this work in 2006 [4] by considering a larger range for each parameter and used detailed statistical analysis of each variable, as well as spline functions to model non-linearity. This study, however, focused on the power use and instructions per second rather than the total cycles per application. Further, all analysis is based on the regression model's output without referring back to the collected data to demonstrate the cause of the impact that each parameter has. Again, many chosen ranges of the parameters no longer represent modern chip designs, with no vectorisation explored whatsoever.

Further work expanded on these core ideas, with İpek et al. [5] proposing the use of Artificial Neural Networks to predict their full defined parameter space with as little simulation data as possible, while Dubach et al. approached the same problem with an architecture-centric approach that allowed unseen codes to be predicted after minimal simulations based on a model trained on several other applications [6]. A continuation of this work saw Dubach et al. predicting the best microarchitectural design based on the compiler optimisations used [7]. These works are limited in the range of values modelled both in terms of modern relevancy, and focus on CPU workloads that are not comparable with real-world HPC workloads. They instead aim to tackle a different problem: the prediction of a defined parameter space from minimal samples rather than an analysis of the source of bottlenecks. In our case, we are not bound by limited data due to the significant gains in simulation frameworks and the performance of machines since these works were published, and instead favour masses of data that lead to deeper insight that is guided by our ML-predicted impacts.

III. Simulation Environment

Our need for a simulation environment for this study is obvious - though our choice for a higher-level execution-driven simulator is motivated by the infeasibility of developing an RTL design for each configuration, something expected to yield more accurate results. We define our simulation environment to be both the simulation of a single configurable core model, supported by a simulated memory backend model.

The core model is simulated via The Simulation Engine, also known as SimEng, developed by the University of Bristol High Performance Computing Group [10]. SimEng provides a fast, easy to use, and cycle-approximate open-source tool to rapidly explore the microarchitectural design space. The speed across each run that is required to generate significant data would have been unachievable in other simulators of this granularity, leading SimEng to be the obvious choice for this research.

The memory subsystem is simulated via the Structural Simulation Toolkit (SST), developed by Sandia National Laboratories [11], which has an existing integration with SimEng. This allows us to effectively model an L1 and L2 cache, along with main memory. We later design our benchmarks to cover a range of smaller L1 and L2 resident problems, as well as larger test cases to simulate real-world problem cases of a variety of codes. The scope of this study targets a single core with this memory model, which, assuming a multi-core environment in which all cores work under saturation of the main memory controller, reflects the same performance impact of memory-bound codes that one would see in real world multi-core problem sets.

Where SimEng and its integration with SST particularly shines in this study is the ability for model-driven design. We orchestrate each run through automated generation of the core's configuration file as well as the SST memory model file, followed by dispatching multiple instances of SimEng at once and collecting the returned statistics from each run. This allows for a rapid simulation workflow that makes the significant amount of data required for this study achievable, where many other alternatives are difficult to instrument in such a broad manner.

IV. Assumptions

A. Equivalent Code Execution

Our study analyses the performance of 4 codes: STREAM, miniBUDE, TeaLeaf, and MiniSweep. Each of our codes were compiled statically using the Arm Compiler for Linux v23.04.1, based on LLVM 16.0.2, targeting the `armv8.4-a+sve` instruction set. In this study, one of the features we explore is SVE vector width, so we must ensure that each binary is generated as vector length agnostic to both ensure functional correctness, but to also ensure that all lanes of the vector are fully utilized regardless of the specified width. This is specified in our compilation with the flag `-msve-vector-bits=scalable`.

The same binaries were run for each test per configuration, meaning only vector length imposes a restriction on the instruction stream while all other parameters must exploit increased Instruction Level Parallelism (ILP) for improved performance.

Given the primary objective of this paper is to identify performance trends as a result of a change of microarchitecture (rather than predicting the exact performance one would achieve with a given configuration), we assume that while the compiler cost model may have some influence on the performance, this will be notably smaller than the wider changes to the architecture. In particular, we aim to mitigate the impact has through our choice to compile targeting scalable SVE vector length, making execution more comparable even if it comes at the cost of poorer performance for larger vector lengths. By comparing equivalent code streams, except

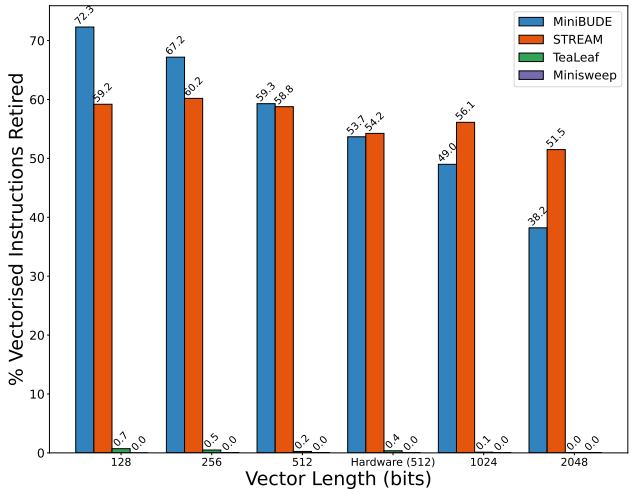

Fig. 1. Percentage of retired instructions that are SVE instructions across multiple vector lengths.

TABLE I
SIMULATED SINGLE-CORE CYCLES COMPARED TO HARDWARE CYCLES ON MARVELL'S THUNDERX2 FOR OUR CHOSEN APPLICATIONS IN SIMENG WITH SST

	Simulated Cycles	Hardware Cycles	% Difference
STREAM	25,078,088	26,665,221	5.95%
MiniBude	42,436,227	48,778,524	13.05%
TeaLeaf	19,966,725	14,607,184	36.69%
MiniSweep	6,529,912	10,374,617	37.05%

for changes induced by vector length, we allow for a like-for-like comparison of CPU configurations that negates the potential weakness of the compiler targeting potentially far-fetched designs that it has not been optimised for. It remains an open area of research to explore how compiler cost models can be generated from processor design specifications which would enable more accurate code generation tailored to each processor model in an automated study such as this.

We make the observation in Fig. 1 that for two of our codes, TeaLeaf and MiniSweep, the vectorisation of these codes by the compiler was poor. We measured the vectorisation percentage as the percentage of retired instructions that have at least one Z (SVE vector) register as a source or destination register. We also validate these findings on physical A64FX hardware, which has an SVE vector length of 512 bits, using `perf stat` and the `SVE_INST_RETIRED` event counter.

Due to the poor vectorisation of these codes, we focus any analysis of the impact of vector length on the remaining two codes, MiniBude and Stream. It is worth noting that the poor SVE usage in TeaLeaf and Minisweep is compiler and application dependent, so the impact of vector length may still be negligible in several other applications where the compiler can not effectively vectorize, regardless of how impactful it is on highly vectorized binaries.

B. Accurate Simulation

Our analysis of simulated data to project future trends of microarchitecture design assumes accuracy of our simulation tools. In Table I we validate the accuracy of SimEng combined with SST by running our benchmarks on a physical Marvell ThunderX2 node and comparing this to a fixed ThunderX2 simulated model. We observe that STREAM and MiniBude are simulated with relatively high accuracy, while TeaLeaf and MiniSweep have larger discrepancies. We expect the larger difference in some of our applications is due to a simplified simulation of the memory backend, with our implementation of SST using basic prefetching algorithms, as well as abstracting out important features of a modern memory subsystem such as memory banking. The absence of this granularity of simulation is likely to behave differently depending on memory access patterns of the program, leading to a range of accuracy in the number of cycles observed.

Despite this, the impact on our conclusions remains minimal, as a model predicting accurately to the data yielded from these simulations remains valid assuming the simulator correctly follows the performance trends of different codes. As the magnitude of the cycle difference remains within a reasonable range for each code, we are confident that our simulation framework does capture these trends, hence leading to an insightful ML model if the model's accuracy is also high.

The Marvell ThunderX2 microarchitectural design was used as a baseline for our variable configurations, being an out-of-order superscalar arm-based CPU, and supporting the armv8.4-a ISA. It does not have support for SVE, limiting a direct comparison of our experimental setup with real hardware. In our generated configurations, SVE support was added by modifying the design of the execution units.

The core model supplied in the SimEng repository was accompanied by an SST configuration designed using a mixture of known values and values acquired from online sources empirically testing cache latencies [12].

It is a difficult problem to demonstrate the accuracy of the simulation across the entire design space, especially given the fact that many of the tested architectures do not exist. We therefore rely on the validation of our baseline model in Table I to represent the capability of our chosen simulation framework to capture performance trends. This gives us confidence to extrapolate these trends accurately within the context of our simulation environment, and expect this to still provide insight into the performance trends of real-world hardware.

V. METHODOLOGY

In this study, we develop a surrogate model that, for a given core configuration, instruction stream, and program input, predicts the number of execution cycles. We introspect this model to gain understanding on the core design parameter space and define the most important predicted parameters. Using this understanding, we further analyse the underlying reasons as to what causes the selected parameters to have such importance on the performance of the core, and use our large simulated dataset to back up these findings.

TABLE II
SIMENG CORE PARAMETERS PRESENTED WITH THEIR RANGES AND STEPS EXPLORED IN THIS STUDY.

Parameter	Range	Step
Vector Length (Bits)	{128-2048}	Powers of 2
Fetch-Block-Size	{4-2048}	Powers of 2
Loop-Buffer-Size	{1-512}	1
General Purpose (GP) Registers	{38-512}	8 starting from 40
Floating-Point (FP)/SVE Registers	{38-512}	8 starting from 40
Predicate Registers	{24-512}	8
Conditional Registers	{8-512}	8
Commit Pipeline Width	{1-64}	1
Frontend Pipeline Width	{1-64}	1
Load-Store-Queue Completiton Pipeline Width	{1-64}	1
Reorder Buffer (ROB) Size	{8-512}	4
Load Queue Size	{4-512}	4
Store Queue Size	{4-512}	4
Load Bandwidth (Bytes)	{16-1024}	Powers of 2
Store Bandwidth (Bytes)	{16-1024}	Powers of 2
Permitted Memory Requests Per Cycle	{1-32}	1
Permitted Memory Loads Per Cycle	{1-32}	1
Permitted Memory Stores Per Cycle	{1-32}	1

TABLE III
SST MEMORY MODEL PARAMETERS PRESENTED WITH THEIR RANGES AND STEPS EXPLORED IN THIS STUDY.

Parameter	Range	Step
Cache Line Width (clw)	{32-512}	Powers of 2
L1 Latency (Cycles)	{1-10}	1
L1 Clock Speed (GHz)	{1-5}	0.5
L1 Associativity	{1-16}	Powers of 2
L1 Size (KiB)	{16-2048}	Powers of 2
L2 Latency (Cycles)	{6-50}	1
L2 Clock Speed (GHz)	{1-5}	0.5
L2 Associativity	{1-16}	Powers of 2
L2 Size (MiB)	{0.25 - 64}	Powers of 2
Ram Timing (ns)	{40-250}	10
Ram Clock (GHz)	{1-5}	0.5
Ram Size (GiB)	8	N/A

A. Parameter Space

We design a SimEng core to vary all core parameters in the front end of the CPU and the memory subsystem that may impact the number of cycles a code takes to execute. The design of the execution units, ports, reservation stations, and instruction execution latency are fixed to limit the scope of this study. Our CPU configuration includes seven execution units with a single unified reservation station shared between them with a width of 60 and a dispatch rate of four instructions per cycle. Many of the seven execution units support multiple different instruction groups. Three of them are exclusive to load and store instructions, two support NEON and SVE instructions with one additional predicate-only port, and three support a mixture of integer, floating point, and branch instructions. These designs were chosen to reflect a generic yet modern CPU architecture with SVE support.

In Table II we define our ranges to explore from the minimum viable values up to a maximum of realistic values for current and near-future core designs in the context of the number of execution units we model. In order to reduce noise in our dataset, we typically use greater steps to explore parameters with larger ranges.

Table III presents our SST-simulated memory backend, modelling a single core design with L1 and L2 data caches, hitting RAM on a last-level-cache miss. We do not model L1I cache in this study, leading to a 0 cycle latency for instruction fetch.

For each run through our set of benchmarks, a new set of parameters is generated across a continuous uniform distribution. All parameters are independently generated, with the exception of Load and Store Bandwidths, and L2 size and latency. These parameters' lower bounds are dependent on other parameters to ensure a functional and realistic CPU design, where Load and Store Bandwidths must be large enough to load and store at least data as large as the vector length, and L2 cache must be larger and higher latency than our L1D cache.

Rather than focusing on minimising the number of samples required for an accurate model, we take advantage of the simulation speed of these benchmarks, roughly two minutes per benchmark for each configuration, as well as the large compute resources available. While the collected 180,000 data points only cover a fraction of the entire search space, we later find that this sample is sufficient to model the performance trends due to the high accuracy of our ML model's predictions for the number of cycles on unseen data points.

B. HPC codes used

We benchmark each configuration on four codes with different performance characteristics, representative of many codes that one would see in HPC. Two of our codes, TeaLeaf and Minisweep, are included in SPEChpc 2021, though ran on smaller inputs to achieve shorter simulation times. SPEChpc 2021 provides a suite of applications, designed to give realistic and comparable performance measurements of modern HPC systems [13]. This makes it a useful suite to represent the performance of possible CPU designs in the context of HPC, hence us using a subset of the applications.

TeaLeaf is a miniapp that solves linear heat conduction equations and is typically a memory-bound application, while Minisweep is a miniapp modelling radiation transport and is compute or communication bound, depending on the number of ranks used. In our case where we model a single core, Minisweep acts as a compute bound application due to its relatively high arithmetic intensity [14]. We also include STREAM as a sustained memory benchmark to represent heavily memory bound codes [15], as well as miniBUDE, a molecular dynamics code used for drug discovery, as a further compute bound application [16].

The four codes have been chosen for a few reasons: there is a trade off between the number of codes we use and the amount of data we can collect in a given timeframe; any codes used must be supported within our chosen simulation framework, which currently only has prototypical dynamic linking support, ruling out MPI-reliant codes; these codes attempt to characterise some core aspects of performance within HPC, using a mixture of compute and memory bound applications, each with different memory access patterns and compute kernels. Increasing the application diversity would potentially deepen the insights achievable by this study, though also runs the risk of diminishing the accuracy of our findings

TABLE IV
PARAMETERS SET FOR EACH APPLICATION RAN ACROSS ALL CONFIGURATIONS.

Application	Input options	Input Values
STREAM	Programming Model	OpenMP (single thread)
	Stream Array Size	200000
MiniBude	Programming Model	OpenMP (single thread)
	Benchmark Name	bm1
	Atoms	26
	Poses	64
	Iterations	1
TeaLeaf	Programming Model	OpenMP (single thread)
	Dimensions	2D
	Number of cells along {X, Y}	{32, 32}
	Domain {xmin, xmax}, {ymin, ymax}	{0, 10}, {0, 10}
	Solver Method	Conjugate Gradient
	Initial Timestep	0.004
	End Step	5
	Max Iterations	10000
MiniSweep	Programming Model	OpenMP (single thread)
	Global number of gridcells along {X, Y, Z}	{4, 4, 4}
	Total number of energy groups	1
	Number of angles for each octant direction	32
	Sweep Iterations	1
	Sweep blocks used to tile the Z dimension	1

due to the increase in compute time required to collect data. This study characterises performance based on some core attributes generally seen across the HPC space, though in reality workloads are dependent on machine and are hard to fully represent within the scope of a single study. We believe that our chosen applications provide enough coverage to accurately portray trends on the core-level of performance.

All of our applications have been compiled statically without MPI, and instead with exclusively an OpenMP backend. As we are simulating only a single core, much of this backend is negated, and we instead focus our analysis on the single-core performance, as well as the memory performance of our codes.

The input data for each application presented in Table IV has been tuned to both be small enough to yield a short simulation (between 10-50 million retired instructions on average per run, leading to a simulation time of between 1-5 minutes when run with one instance per core), yet be representative of the performance characteristics one would expect on a larger code. For some codes, the smaller input size means that the code becomes increasingly L1 or L2 bound rather than RAM bound, but this is recognized in the analysis of the results and does not significantly impact the insight we can gain from this study. This concession must be made in order to generate enough data at this resolution to ensure an effective ML model that can accurately model our simulated codes across the high dimensional parameter space, which allows us to gain insight from each parameter.

For STREAM, an array size of 200,000, or 4.6 MiB, will either be L2 or RAM bound depending on the configuration. For MiniBude, we use the provided dataset `bm1`, with a relatively small amount of poses at 64 and only run for a single iteration. In the case of TeaLeaf, we simulate more cells in each dimension at 32 instead of the default of 10 and run for 5 timesteps. For Minisweep, we run for a single timestep on a small grid of 4x4x4 cells, but still follow the documented guidance on what will maintain the performance characteristics of the code. These choices of parameters provide confidence that the inputs chosen allow our runs to be representative of the application's performance.

All applications used by this study have built in validation mechanisms, comparing the generated output to an expected output. Only runs that pass this validation are considered in our results to ensure there are not any known issues with the binary or the simulation environment.

C. Machine Learning Model

One goal of this paper is to implement a surrogate model for our simulation framework specifically for our chosen binaries and datasets. Surrogate modelling is a form of model reduction that allows us to map a higher order system into a lower order space that is computationally less expensive to evaluate [17]. This form of modelling has previously been used for simulation-based optimisation to replace simulators of complex dynamical systems [18], and more commonly used recently in weather modelling [19].

The purpose of the study is to map our simulated model onto an explainable surrogate, allowing us to gain insights by interpreting an alternative representation of our model. This approach has previously been explored in other domains for both neural networks and traditional ML [20], though this paper expands on the prior work implementing models in microarchitectural design with the use of a more modern parameter space and a more expansive search, training our model with notably more data than what we are aware has been done before.

We implement four decision tree Regression models, one for each application, which are trained to predict the number of executed cycles from the thirty variable input features. A decision tree regressor lends itself nicely to this problem, as (i) we want to predict the number of execution cycles within a large, positive, and continuous range, (ii) complex parameter relationships lead to non-linear trends that can be modelled within the tree, (iii) they are highly interpretable as the decision tree describes how the prediction is made which can easily be followed, as well as providing metrics of feature importance which is the goal of this study, and (iv) they are significantly faster to train on large amounts of data than other regression models such as Support Vector Regression (SVR), as their training time complexity is $O(n \log n)$ rather than $O(n^2)$. We train a separate model per application to allow for a more flexible approach, making it easier to introduce a new application without retraining the model on all previous data. One may think that a unified model would be more robust, though a decision tree regressor trained on multiple applications would likely branch based on a given application due to the contrasting performance trends of each application, leading to a larger and less interpretable model without necessarily improving learned trends.

We implement our model using Scikit-Learn in Python [21], using their pre-defined decision tree regressor models. Our dataset contains 180,006 rows, which are split by a randomized 80/20 split for our training and validation sets respectively.

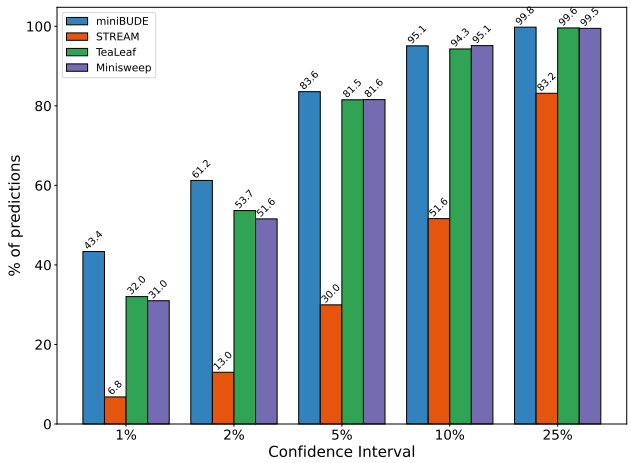

Fig. 2. Percentage of cycle predictions generated by our model that are within the specified confidence interval of the true simulated value.

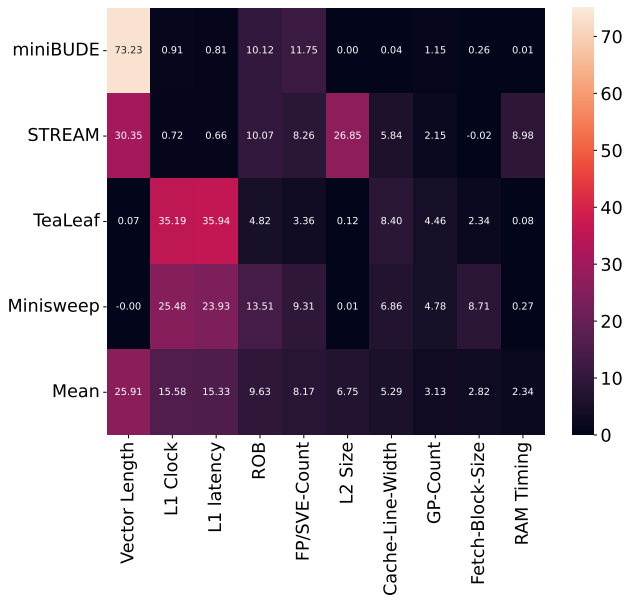

Fig. 3. Ten greatest feature importance percentages (ordered descending by mean) presented for each application (positive implies increasing a parameter yields fewer cycles).

Our decision tree is constructed with minimal constraints on the creation of new leaves - there are no maximum numbers of leaves, a single sample can be considered as a new leaf, and there is no maximum depth to the tree. The criterion to measure the quality of each split is based on the mean squared error, with the split at each node chosen to be the best found. These choices were made as they yielded the highest performing model when judged by the metrics we defined to be important to this problem, discussed further in our results.

We train our model by minimizing mean squared error between our predicted cycles per application, and our simulated truth cycles per application in our training dataset. This criterion is important in this problem as our model is learning over a high dimensional parameter space, leading to a lot of variation within a single choice for a single parameter. Using mean squared error over mean absolute error avoids finding a minima of the loss function by predicting the mean of each parameter, and instead, we penalize the model heavily for avoiding outliers.

VI. Results

A. Model Accuracy

We assess our ML model on a metric of percentage of predicted results within specified confidence intervals of the true value, testing on an unseen split of 20% of the total data. Results in this case refer to the number of cycles per application. This metric is relevant as we care about most predictions being in a close range to the true value, while outliers do not penalize our results heavily as only a few vastly incorrect results would not impact the model's weightings, the focus of this study.

Fig. 2 demonstrates that our model is accurate, with the majority of predictions falling within 2% of their true values for three applications, and nearly all predictions falling within 25%. It is notable that the model used to predict STREAM's cycles yielded poorer accuracy than the other applications. Despite the poorer performance, the overwhelming majority of predictions fall within 25% of the true cycles, suggesting that the trends learned by our model are still correct, though less accurate than the other applications. We also find that between all applications, the mean accuracy of all results is 93.38%, meaning the average prediction is 6.62% away from the simulated true result.

These results provide confidence that our models correctly model the trends in the data, and that each the weightings for the parameters are learned by the model accurately reflect reality for the chosen applications.

B. Feature Importance

When interpreting our model's learned parameters, we use permutation feature importance to estimate the contribution of each input variable. This method randomly shuffles the values of each feature before predicting our output variable and scoring the model with the mean absolute error criterion. This method is repeated 10 times, taking the mean error as the permutation feature importance. Finally, we contextualise this data by expressing the importance as the percentage of the summed error increase across all features. This metric allows us to quantify the impact of each parameter on the number of cycles as it represents how much prediction error increases when randomising a feature's values.

In Fig. 3, we present the largest mean feature importances. We find that vector length has a significant impact, proportional to the percentage of executed instructions that are vector instructions. This is most apparent in the compute-bound application MiniBude, where vector length has by far the largest impact on the number of cycles executed. Despite

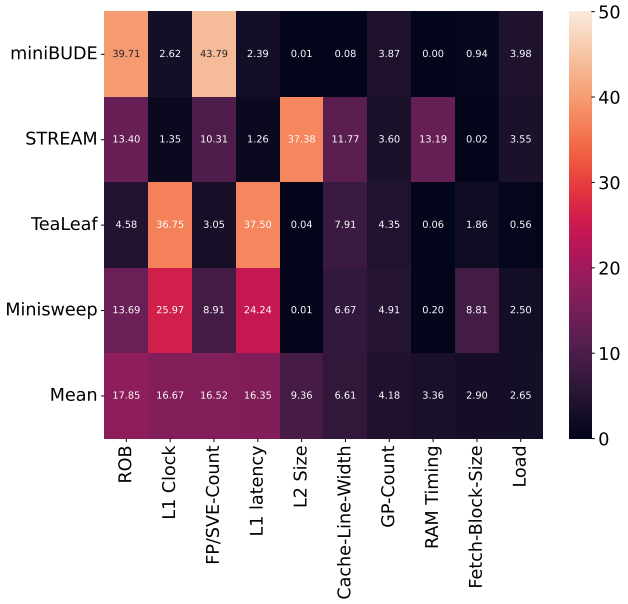

Fig. 4. Ten greatest feature importance percentages (ordered descending by mean) for each application when vector length is constrained to 128 (positive implies increasing a parameter yields fewer cycles).

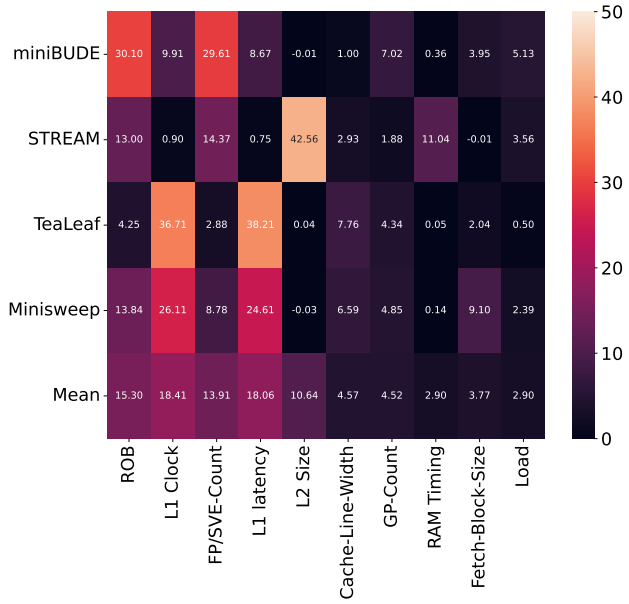

Fig. 5. The same ten feature importance percentages as Fig. 4 for each application when vector length is constrained to 2048 (positive implies increasing a parameter yields fewer cycles).

having a similar vectorisation percentage, vector length has a dampened impact on STREAM, where the L2 cache size has roughly an equal impact. This finding suggests that the Data-Level-Parallelism unlocked by large vector lengths is still bound by the memory bandwidth of the CPU. In more memory bandwidth-bound applications, this effect will amplify and focus more importance on the speed of the memory-subsystem.

For Tealeaf and Minisweep, the severely limited vectorisation leads to minimal impact of vector length, as expected. Instead, greater feature importance is placed on the latency to fetch data from the L1 cache via the parameters L1 Clock Speed and L1 Latency. We expect that for larger inputs of Tealeaf, the importance of cache clock and latency would shift from L1 towards higher levels of the memory hierarchy, while for Minisweep, which has a relatively high arithmetic intensity, the constraint of memory speed is likely to remain in lower levels of cache when run on a single rank.

Due to the large feature importance percentage of vector length being specific to MiniBude and STREAM in this case, to ensure a fair comparison of other features we also analyse the importance of all other features when vector length is constrained. This also provides greater insight into what other parameters vector length relies on for these two applications to unlock the maximum benefit of an increased vector size.

We observe in Fig. 5 that, in the case of a long vector length of size 2048 bits, MiniBude becomes increasingly constrained by L1 cache speeds compared to a short vector length of 128 bits presented in Fig. 4, while the Reorder Buffer (ROB) and Floating-Point/SVE Registers are relieved of pressure as fewer SVE instructions are in flight at any one time. The increase in impact of the number of General-Purpose Registers and Fetch-Block-Size suggests that compute-bound codes where the compute kernel has high levels of ILP start to have a loose bottleneck of both fetching more instructions, but also during register renaming when many instructions are in flight. This impact is also visible in Minisweep, where parameters impacting the ability to push instructions through the front end have a notable effect on our results.

The Cache-Line-Width is also predicted to have a large impact in all applications where increased memory-bandwidth yields fewer cycles. This result is slightly misleading, as increasing the Cache-Line-Width in this case also increases the L1-L2 and L2-RAM bandwidth by the same amount; L1-core bandwidth is defined by the Load/Store Bandwidth parameters. The bandwidth is impacted because each memory request has the same latency, yet yields more data, thus representing the impact of increasing our memory bandwidth. It is interesting to note that a vector length of 2048 dampens the impact of this parameter in highly vectorised codes, as memory requests for vector instructions are accessed in parallel from different banks if the vector length is larger than a single cache-line. SST models an infinite number of memory banks unless explicitly specified, meaning that a shorter cache-line has little impact on the latency of SVE loads/stores, reducing the visible impact in this study. This is different to what we'd expect to see in hardware design, where the increase of Cache-Line-Width is expected to come at the expense of inter-cache latency, and there are only finite memory banks which the loaded data may not be distributed across.

Using the ML guided approach in selecting the most im-

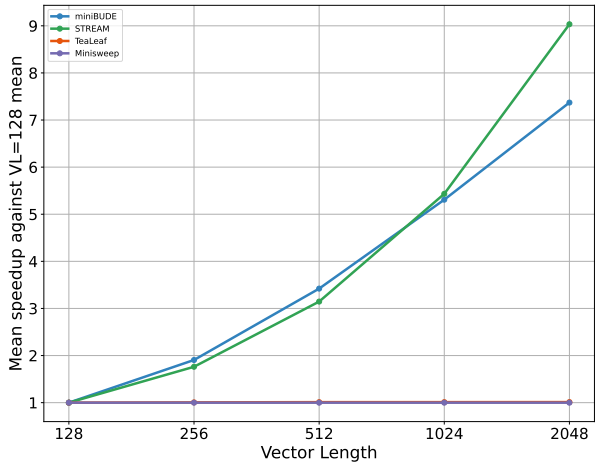

Fig. 6. Mean speedup observed of varying vector length compared to the mean number of cycles a vector length of 128 yields. Only results with a Load-Bandwidth greater than 256 are presented to ensure a fair comparison, given this is the minimum a result with vector length 2048 has.

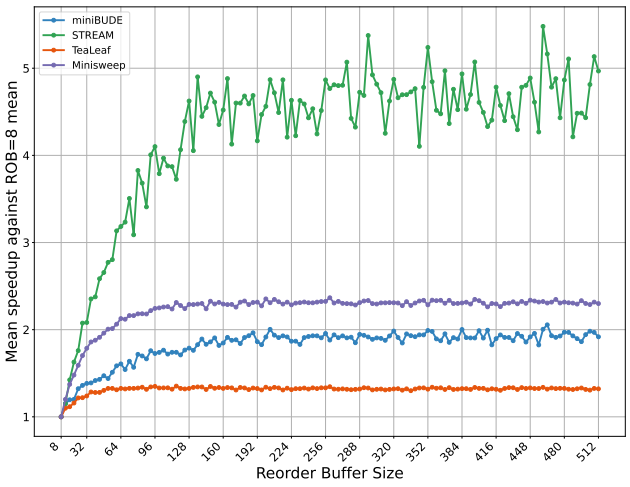

Fig. 7. Mean speedup observed of varying ROB size compared to the mean number of cycles a ROB size of the minimum of 8 yields.

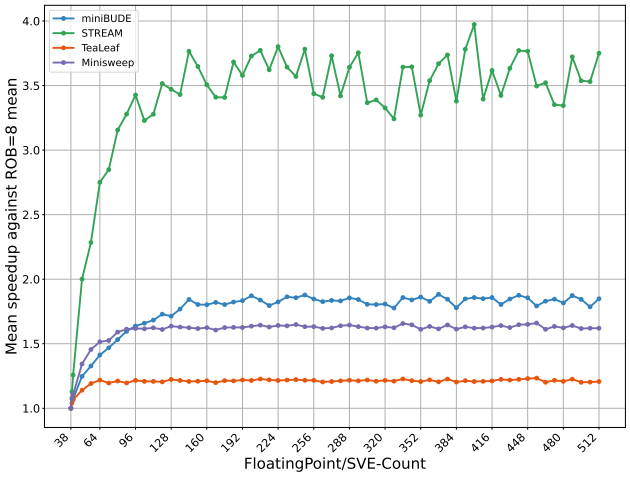

Fig. 8. Mean speedup observed of varying the number of Floating-Point/SVE Registers compared to the mean number of cycles a Floating-Point/SVE Registers count of the minimum of 38 yields.

pactful parameters, we analyse the large quantity of simulated data collected to provide insight into the cause of a high performance weighting.

When exclusively investigating vector length in Fig. 6, it is clear that it can have a large impact on performance. Our results show that we yield a speedup of 7-9x when increasing our vector length by a factor of 16, with the larger speedup in the case of STREAM. We expect this is due to the increased Data-Level-Parallelism, as although we only compare configurations with the same Load/Store Bandwidth, we are bound by at least one request per vector. As our constraints ensure we can always load a full vector, we load more data per memory request in larger vector lengths which saturates more of our memory bandwidth. We also reduce pressure on the front-end of the CPU, allowing for increased ILP.

Fig. 7 shows that varying the ROB size only effects performance up to a certain threshold, depending on the application and the remaining configuration. A larger ROB allows for more ILP, assuming our execution units are not fully saturated. Once these are saturated, no performance improvement is realized and the additional power and die space would be better utilized in memory improvements. We find the largest impact in STREAM, predominantly made up of SVE load instructions, as instructions will remain uncommitted in the pipeline for longer in memory-bound applications due to memory latency, thus requiring a larger ROB to hold instructions and micro-operations to fully saturate both execution units and memory bandwidth. We find that for this layout of execution units, a ROB size greater than 152 yielded minimal improvements in any of our applications.

The number of Floating-Point/SVE Registers presented in Fig. 8 paints a similar picture; in HPC codes which are typically vectorized and contain a large number of floating-point instructions, we require a large number of registers to not be bottlenecked by the register-rename part of the instruction pipeline. We find that a count less than 144 can lead to this bottleneck, meaning that new instructions must wait for commits in order to continue processing. Again, more than this yields minimal speedup as our bottleneck likely shifts to the backend of the CPU.

VII. Conclusion

This paper presents an updated approach to a previous method of exploring the parameter space of architecture through the use of an explainable surrogate ML model. We gain insight into the single-core design of a CPU for a range of commonly used applications, and successfully learn the performance trends in a high-dimensional simulated dataset.

Our machine learning model permits us to more accurately extrapolate across the large search space, allowing us to model the space with a fraction of the data requirements. Our chosen simulation framework SimEng, as an execution-driven simulator modelling the core design, and SST, modelling our memory backend, allows us to quickly and accurately gather our initial data of over 180,000 configurations tested. This approach is still limited to applications the model has been trained on, and cannot yet adapt to unseen codes as the model must learn the characteristics of each code to accurately predict otherwise.

Our use of a decision tree regression model enables quantifying the impact of each parameter as a feature importance score. These scores allowed us to identify that vector length is predicted to have the most significant positive impact on code performance, continuing for large vector lengths up to 2048, but is highly dependent on how vectorized the code is. If the code is highly vectorized, this can have a large speedup of up to 9x in some cases due to the increased Data-Level-Parallelism saturating more memory-bandwidth as well as the relieved pressure on the core's front-end, while there is a negligible impact of vector length on poorly vectorized codes. In cases with limited memory-bandwidth, we do not expect to see as strong scaling.

Bottlenecks in the front-end of the CPU are often caused by the lack of space in the Reorder Buffer, Floating-Point/SVE Registers, or General-Purpose Registers. We find that these can limit performance by up to a factor of five in memory-bound applications due to limiting ILP, specifically on high-latency instructions such as loads and stores that hit higher levels of cache or RAM. If the front end is designed sufficiently large however, no further performance gain is realized as the bottleneck shifts to the backend, either to the execution units or the memory. While the quantity needed to fully saturate the backend is specific to the layout of the execution units, we found that the number of Floating-Point/SVE Registers should be roughly equal to the size of the Reorder Buffer, and a value of 152 was enough in this case.

It can not be understated how much impact the memory makes to the performance of the CPU. While the vector length and front-end parameters certainly contribute to the performance of the chip, they simply shift these bottlenecks to the memory backend if the code was not already highly memory-bound. We find that the latency of L1 cache and RAM have a large impact on performance depending on the size of the memory footprint of our code. The size of our L2 cache had a stark impact on STREAM, as this size can be large enough to turn codes which are loosely RAM bound into L2 bound, drastically reducing memory latencies, regardless of subtle differences to the latency of L2 cache. Any way to improve memory bandwidths, whether this is through clock speed, latency in cycles, or the cache line width, will improve the performance of the code. In a world which is already dominated by memory-bound codes, this study proves that despite having performance gains available to unlock in other areas of CPU design, the unavoidable problem of memory latencies and bandwidths must be faced, else the performance bottleneck will continuously shift onto our memory subsystem; it always comes back to memory [22].

While performance-per-core is important and is the focus of this study, the reality is that HPC relies on parallel execution across multiple cores and nodes. Even on a node level, this study abstracts away the memory contention behaviour exhibited in multi-core systems. For multi-node systems, one must also consider the cost of communication between ranks, as well as the potentially different memory footprint of a code supporting MPI communication. The results gained from this study must therefore be taken in the context of only a single core. While we expect many of the uncovered trends to remain, this work lays the foundation for future work into the impacts of parallel execution. Our ML modelling approach would support this work, simply relying on the mass collection of data that accurately represents the behaviour of multi-core or multi-node systems on a benchmark suite.

The output of this work is primarily three-fold: (i) our ML modelling framework reinforces many of the relationships between code performance and architecture features for simpler kernels such as STREAM, while providing deeper insight into less transparent relationships in the cases of MiniBUDE, TeaLeaf, and Minisweep, (ii) these uncovered relationships guide hardware designers and researchers to where maximal performance gains can be unlocked in HPC codes, through empirical evidence over an incredibly large search space, and (iii) we have demonstrated an updated modelling approach to learn and interpret our data, providing a decision-tree regression model to accurately predict HPC code performance based on a modern CPU configuration, thus mapping an expensive modelling problem onto a reduced representation. This modelling approach can be easily applied to new codes, or a new system design which takes into account multi-core or multi-node performance.

This work could be further explored by going further to also experiment with the design of the execution units and investigating how large the CPU backend needs to be to resolve compute-bound bottlenecks. With Arm ISA extensions such as SME and SME2, the impact of these could also be explored to realize the impact that these have on both compute and memory bound codes in the context of the remainder of the CPU configuration. Future research could also explore the avenue of creating a more complex surrogate model to map more aspects of our simulation framework to an advanced machine learning model. We envision this as an extension to the model that we discover that can interpret new binaries and datasets, to allow for accurate modelling of unseen codes at a fraction of the computational cost or time that execution-driven simulations require. Finally, more work is needed on the topic of compiler cost-modelling to ensure that compilers can always generate code for specific microarchitectural parameters, rather than pre-defined models for few architectures.

ACKNOWLEDGMENT

The authors would like to thank Jack Jones, Finn Wilkinson, and the remainder of the SimEng development team for support in supporting SimEng throughout this study and any proofreading efforts. Thanks to James Cussens for guidance on the machine learning front. This work used the Isambard 2 UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1).

REFERENCES

[1] Waldrop, M.M., 2016. The chips are down for Moore's law. *Nature News*, *530*(7589), p.144.

[2] Sato, M., Ishikawa, Y., Tomita, H., Kodama, Y., Odajima, T., Tsuji, M., Yashiro, H., Aoki, M., Shida, N., Miyoshi, I. and Hirai, K., 2020, November. Co-design for a64fx manycore processor and" fugaku". In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-15). IEEE.

[3] Joseph, P.J., Vaswani, K. and Thazhuthaveetil, M.J., 2006, February. Construction and use of linear regression models for processor performance analysis. In The Twelfth International Symposium on High-Performance Computer Architecture, 2006. (pp. 99-108). IEEE.

[4] Lee, B.C. and Brooks, D.M., 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. ACM SIGOPS operating systems review, 40(5), pp.185-194.

[5] İpek, E., McKee, S.A., Caruana, R., de Supinski, B.R. and Schulz, M., 2006. Efficiently exploring architectural design spaces via predictive modeling. ACM SIGOPS Operating Systems Review, 40(5), pp.195-206.

[6] Dubach, C., Jones, T. and O'Boyle, M., 2007, December. Microarchitectural design space exploration using an architecture-centric approach. In 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007) (pp. 262-271). IEEE.

[7] Dubach, C., Jones, T.M. and O'Boyle, M.F., 2008, October. Exploring and predicting the architecture/optimising compiler co-design space. In Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems (pp. 31-40).

[8] Zaourar, L., Benazouz, M., Mouhagir, A., Jebali, F., Sassolas, T., Weill, J.C., Falquez, C., Ho, N., Pleiter, D., Portero, A. and Suarez, E., 2021, November. Multilevel simulation-based co-design of next generation HPC microprocessors. In 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) (pp. 18-29). IEEE.

[9] Zaourar, L., Chillet, A. and Philippe, J.M., 2023, September. A-DECA: an Automated Design space Exploration approach for Computing Architectures to develop efficient high-performance many-core processors. In 2023 26th Euromicro Conference on Digital System Design (DSD) (pp. 756-763). IEEE.

[10] Jones, J., Wilkinson, F. Weaver, D., Moore, J., Cockrean, A. McIntosh-Smith, S. SimEng, GitHub. https://github.com/UoB-HPC/SimEng

[11] Rodrigues, A.F., Hemmert, K.S., Barrett, B.W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., Cooper-Balis, E. and Jacob, B., 2011. The structural simulation toolkit. ACM SIGMETRICS Performance Evaluation Review, 38(4), pp.37-42.

[12] De Gelas, J., 2018, May. Assessing Cavium's ThunderX2: The arm server dream realized at last. https://www.anandtech.com/show/12694/assessing-cavium-thunderx2-arm-server-reality/6.

[13] Li, J., Bobyr, A., Boehm, S., Brantley, W., Brunst, H., Cavelan, A., Chandrasekaran, S., Cheng, J., Ciorba, F.M., Colgrove, M. and Curtis, T., 2022, July. SPEChpc 2021 benchmark suites for modern HPC systems. In Companion of the 2022 ACM/SPEC International Conference on Performance Engineering (pp. 15-16).

[14] Henschel, R., 2023, February. Overview of SPEC HPC Benchmarks and Details of the SPEChpc 2021 Benchmark. Presentation at PPoPP2023. https://parallel.computer/presentations/PPoPP2023/2023-Robert-Slides.pdf

[15] McCalpin, J.D., 1995. Memory bandwidth and machine balance in current high performance computers. IEEE computer society technical committee on computer architecture (TCCA) newsletter, 2(19-25).

[16] Poenaru, A., Lin, W.C. and McIntosh-Smith, S., 2021, June. A performance analysis of modern parallel programming models using a compute-bound application. In International Conference on High Performance Computing (pp. 332-350). Cham: Springer International Publishing.

[17] Antoulas, A.C., Sorensen, D.C. and Gugercin, S., 2001. A survey of model reduction methods for large-scale systems. Contemporary mathematics, 280, pp.193-220.

[18] Cozad, A., Sahinidis, N.V. and Miller, D.C., 2014. Learning surrogate models for simulation-based optimization. AIChE Journal, 60(6), pp.2211-2227.

[19] Westermann, P., Welzel, M. and Evins, R., 2020. Using a deep temporal convolutional network as a building energy surrogate model that spans multiple climate zones. Applied Energy, 278, p.115563.

[20] Roscher, R., Bohn, B., Duarte, M.F. and Garcke, J., 2020. Explainable machine learning for scientific insights and discoveries. Ieee Access, 8, pp.42200-42216.

[21] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. and Vanderplas, J., 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, *12*, pp.2825-2830.

[22] Wulf, W.A. and McKee, S.A., 1995. Hitting the memory wall: Implications of the obvious. ACM SIGARCH computer architecture news, 23(1), pp.20-24.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

The main contributions of this paper are as follows:

- C_1 Orchestration of the SimEng architecture simulator to collect results of a HPC benchmark suite on 180,000 unique and modern CPU configurations.
- C_2 The creation of a decision-tree regressor surrogate model for our simulation framework on the chosen codes, allowing for accurate prediction of cycles across our architecture search space.
- C_3 The analysis of the architectural parameters delivering the most impact, observed via the introspection of our learned surrogate model.

B. Computational Artifacts

We provide two artifacts for this paper, A_1 for SimEng, our architectural simulator of choice, and A_2 for the remaining infrastructure, including the Machine Learning model, code binaries and build scripts, and result analysis and graphing code.

- A_1 https://github.com/UoB-HPC/SimEng/tree/1c394e791a37e83ad97c48d9ad0009ec9635815d
- A_2 https://doi.org/10.5281/zenodo.13852699

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Tables 1-3
A_2	C_2, C_3	Table 4 Figures 1-8

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This artifact is for SimEng, the simulation framework used in this study. All CPU configurations tested were ran through SimEng using the binaries of the codes chosen, all of which were compiled with Arm Compiler for Linux 23.04.1. The results of the SimEng runs were used to learn the architectural parameter space through our ML model in A_2. Note that while the version of SimEng provided in this artifact will work for several other binaries, to reproduce this study a patch file provided in A_1 must be applied.

Expected Results

SimEng will run a provided binary given a CPU configuration file, and return statistics such as cycles executed, number of instructions, and more upon completion of the simulation. This allows us to define new configurations seen in Tables 2-3, as well as counting the number of SVE instructions passed through SimEng for Table 1 by adding additional code to count this statistic. This additional code is not provided in the artifact, though SimEng documentation provides good explanation of where to begin with this. Using the additional infrastructure in A_2, SimEng can be orchestrated to run different configurations on the defined workflow in parallel, along with the collection of the statistics output by SimEng.

You should find that altering the configuration provided to SimEng will alter the number of cycles simulated on a given binary. In particular, this study finds that changing vector length will typically have the largest impact compared to other parameters on the number of simulated cycles.

Expected Reproduction Time (in Minutes)

Artifact Setup time: The expected setup time for SimEng with SST support is 30 minutes, and a further 1 minute to edit the CPU or SST configuration.

Artifact Execution: The expected time to run a simulation in SimEng depends on the binary and your hardware, though in this study most runs would take around 1 minute of compute time each on a single core, simulating at a speed typically around 1 MIPS.

Artifact Analysis: SimEng prints statistics immediately after completion, leading to instant analysis. In this study, it is A_2 that collects these statistics for further analysis that takes time.

Artifact Setup (incl. Inputs)

Hardware: SimEng runs on a single core of a CPU, though this study can utilise multiple cores and multiple nodes by running many instances of SimEng. While no fixed number is required, this study used ten Marvell Thunder-X2 nodes on the Isambard 2 supercomputer, each with 64 cores. Each process can use up to a few GB of RAM (depending on the binary), so to ensure no Out-Of-Memory issues, we would recommend allowing 8GB per process, though this is likely overkill.

Software: SimEng supports multiple compilers, though in this study we used GCC 9.3.0 to compile SimEng, available at: https://ftp.gnu.org/gnu/gcc/gcc-9.3.0/.

We use a modified version of SimEng, using commit 1c394e7 on branch dev as the base, and a patch file for any further changes included in A_2. This branch is also available at: https://github.com/UoB-HPC/SimEng/tree/1c394e791a37e83ad97c48d9ad0009ec9635815d

Datasets / Inputs: SimEng relies on a CPU configuration defined in YAML, an SST configuration defined in Python, and a statically compiled binary (plus any arguments or input data) to be ran. All of these used in this study are described or provided in A_2.

Installation and Deployment: In order to install SimEng with SST, you must install version 12.0.x of SST-Core and SST-Elements as well as SimEng itself. Full guidance is given in the SimEng documentation, available at https://uob-hpc.github.io/SimEng/sst/building_simeng_with_sst.html.

To compile each of these, we used GCC 9.3.0 and CMake 3.24.2, though other compilers that support C++17 should also work.

Artifact Execution

The bulk of the workflow is contained in A_2, which orchestrates our simulation framework in A_1. In particular, T_1 is the generation of a CPU and SST configuration (generated in A_2) to be passed to SimEng, where T_2 runs our benchmark suite (contained also in A_2) through SimEng, providing statistics for these runs specific to our configuration generated in T_1. T_3 is the collection of this data, initially stored by each process in a buffer file, but extracted and placed into a dataset. This collection and further analysis (T_4) is also contained in A_2. This leads to many T_2 tasks requiring one T_1 task each, with all T_2 tasks leading to a single T_3 and T_4.

The configuration provided to SimEng and SST is uniformly randomly generated between ranges defined in A_2. In our data collection, we used 180,000 unique T_1 and T_2 tasks, though this value was chosen based on compute time rather than an optimal or minimal value for successful results. It may be possible to effectively map the design space with only a few thousand results, or one may wish to redefine the design space's boundaries.

B. Computational Artifact A_2

Relation To Contributions

This artifact is for the infrastructure code that orchestrates thousands of SimEng runs. Not only does this generate CPU and SST configurations, but it also contains the binaries ran, some helpers to build binaries if used with their sources, the code to generate a decision-tree regressor model from the generated dataset, and code to analyse results gained from this model.

Expected Results

The script "run_xci.sh" allows one to continuously run SimEng on a given number of nodes and cores. This script will generate CPU configurations given design constraints set in the file config_generator.py, before running this configuration on SimEng for each benchmark specified, storing the outputted statistics temporarily until "collect_data.py" is invoked to scrape the useful statistics and append this entry to a database. Finally, "analysis_.py" contains the code to create, train, and analyse results for our machine learning model. "graph-generation.py" is a supplementary script to plot graphs used in the paper.

Expected Reproduction Time (in Minutes)

Artifact Setup time: The expected setup time to start running the workflow (collecting results) is 2 hours. This is as SimEng needs to be modified with the supplied patch file, and multiple scripts are system-specific so would need to be minorly changed. A further 30 minutes would be required to modify all remaining scripts to be system-specific. If one wanted to recompile the codes provided, an additional 1-2 hours would be required.

Artifact Execution: To generate a dataset of 180,000 entries using the same four binaries and inputs used in the paper, we expect this to take 24-48 hours of compute time on across 10 nodes of Marvell Thunder-X2 CPUs, each with 64 cores each. On a single node, this would be roughly 10-20 days.

Once the dataset has been generated, training the machine learning model is extremely fast, taking less than 1 minute on a standard laptop CPU.

Artifact Analysis: Anaylsis of results should take less than 1 hour. One can introspect the learned machine learning model within seconds, and the bottleneck is writing the code to extract interesting statistics which typically requires no more than Pandas and Matplotlib code. To simply use the existing infrastructure, less than 30 minutes is required.

Artifact Setup (incl. Inputs)

Hardware: While there are minimal strict requirements for hardware, the more CPU cores and RAM one can supply, the more/faster the data can be collected. We ran our experiments with a total of 640 cores and 5TB RAM for 1-2 days. In particular, this study made use out of ten XCI nodes on the Isambard 2 supercomputer hosted at the University of Bristol, hence scripts are currently targeted towards running on these. These should be easily modifiable to work on most hardware.

Software: All benchmark codes were compiled using Arm Compiler for Linux 23.04.1 (https://developer.arm.com/Tools%20and%20Software/Arm%20Compiler%20for%20Linux#Downloads), compiled statically, without MPI, SVE vector length agnostic, and with O3 optimisations. Isambard 2 also made use of the Cray ALPS scheduler, hence scripts are currently tuned to use this. The artifact also requires Pandas, Scikit Learn, Matplotlib, and Seaborn Python packages to be installed for the machine learning model and further data analysis to work.

Datasets / Inputs: This artifact allows one to generate a dataset through the "collect_data.py" script. This is coupled with other scripts through the "xci_launcher.sh" to dispatch runs of SimEng, before recording every configuration option and all the statistics for each benchmark named accordingly.

Installation and Deployment: Python 3 as well as the required modules (Numpy, Pandas, Scikit Learn, Matplotlib, Seaborn) is needed to run this artifact. Any modules can be installed via pip.

Artifact Execution

Most of this artifact is Python or bash scripts, meaning no additional compiler is required unless wanting to recompile benchmarks, in which case Arm Compiler for Linux 23.04.1 was used. In order to execute the experiments, first "xci_launcher.sh", or T_1, must run for a fixed period of time. T_1 continuously creates new architectural configurations, tests them in SimEng against our chosen benchmarks, and collects them into a database. This is done through the single script. T_2 trains the machine learning model by running "analysis.py". This will create a new database detailing the impact scores of each parameter for each code, as well as the model accuracy.

Finally, T_3, or "graph-generation.py", will produce graphs similar to those in the paper. This leads to a simple workflow of $T_1 \rightarrow T_2 \rightarrow T_3$.