MicroGrad: A Centralized Framework for Workload Cloning and Stress Testing

Gokul Subramanian Ravi
ECE, UW-Madison
gokulsrm@gmail.com

Ramon Bertran
IBM Research
rbertra@us.ibm.com

Pradip Bose
IBM Research
pbose@us.ibm.com

Mikko Lipasti
ECE, UW-Madison
mikko@engr.wisc.edu

Abstract—We present MicroGrad, a centralized automated framework that is able to efficiently analyze the capabilities, limits and sensitivities of complex modern processors in the face of constantly evolving application domains. MicroGrad uses Microprobe, a flexible code generation framework as its back-end and a Gradient Descent based tuning mechanism to efficiently enable the evolution of the test cases to suit tasks such as Workload Cloning and Stress Testing. MicroGrad can interface with a variety of execution infrastructure such as performance/power simulators as well as native hardware. Further, the modular abstract workload model approach to building MicroGrad allows it to be easily extended for further use.

In this paper, we evaluate MicroGrad over different use cases and architectures and showcase that MicroGrad can achieve greater than 99% accuracy across different tasks within few tuning epochs and low resource requirements. We also observe that MicroGrad’s accuracy is 25-30% higher than competing techniques. At the same time, it is 1.5-2.5x faster or would consume 35-60% less compute resources (depending on implementation) over alternate mechanisms. Overall, MicroGrad’s fast, resource efficient and accurate test case generation capability allow it to perform rapid evaluation of complex processors.

I. INTRODUCTION

Analyzing the capabilities, limits and sensitivities of complex modern processors in the face of constantly evolving application domains is arduous and time consuming. Intelligently generating test cases which can efficiently perform the above analyses will enable quick turnaround times, thereby accelerating the final third of the Innovate-Build-Analyze cycle.

We are particularly interested in two challenging tasks under this umbrella of test case generation:

1) Workload Cloning: which extracts key execution characteristics of a real world application and models them into a synthetic workload.

2) Stress Testing: that maximizes micro-architectural activity of a given processor, specifically to achieve worst-case estimates of execution metrics like performance and power.

We present MicroGrad, an open-source centralized automated framework that is able to efficiently analyze processors based on the scenarios described above. MicroGrad derives its name from (1) Microprobe [5]: a flexible code generation framework which forms MicroGrad’s back-end and (2) Gradient Descent: which is the tuning mechanism used by MicroGrad to efficiently enable the evolution of the test cases to suit the tasks described earlier.

In the past, there has been considerable work in the domains of workload cloning [1]–[3], [13], [15], [16] and stress testing [5], [8]–[10], [14]. Despite this, open source frameworks for these goals have been scarce. Meanwhile, the need for these tools is rapidly increasing with the momentum for open source hardware. As the open source space grows, we will require systematic tools to characterize and stress-test the abundant varied designs and implementations.

To our knowledge, two open-source frameworks available in this space are: Microprobe [5] which can generate user-defined test cases, and GeST [10] which uses Genetic Algorithm (GA) based evolution on an instruction-level model to generate stress tests. MicroGrad goes above and beyond the capabilities and use cases of the above, by providing a fast automated framework for a variety of purposes, all generated with a common centralized tuning mechanism and code generation back-end. Further, MicroGrad can interface with a variety of execution platforms such as performance/power simulators as well as native hardware, in order to evaluate the processor architecture’s execution efficiency. Importantly, the modular "abstract workload model" approach to building MicroGrad allows it to be easily developed upon - allowing for new use cases, improved tuning algorithms, as well as easy interfacing with new execution hardware and simulators. An overview of MicroGrad is shown in Fig.1.

To the best of our knowledge, all prior approaches to cloning and stress testing have been either GA-based or expert driven. Thus, the Gradient Descent based tuning mechanism is a key novelty and highlight in the MicroGrad framework. The tuning mechanism is implemented over a gradient descent algorithm, which iterates through a sequence of "workload generation knobs" configurations (i.e. inputs to the Microprobe framework) and evaluates a specified processor execution metric for those configurations. It gradually moves the code generation configuration in the direction of the steepest execution metric gradient, i.e. one which achieves the best metric improvement for every step change in the configuration, until the optimum configuration / convergence is reached. Note that the execution metric is dependent on the use case - it could be a single high-level statistic like IPC or power consumption in the case of Stress Testing or a combination of both high-level and low-
level statistics such as branch mispredictions, cache miss rates and IPC for Workload Cloning. Overall, the tuning mechanism allows for fast and efficient convergence to the prescribed goal and is observed to considerably outperform competing tuning approaches. Moreover, with its abstracted model, it is easier to deploy compared to expert-driven approaches.

**Fig. 1: MicroGrad Overview**

**Summary of contributions:**

1) We present *MicroGrad*, an open-source automated framework for workload cloning and stress testing. To our knowledge, *MicroGrad* is the first open tool for automated cloning and further, the only open tool for fast-exploratory stress testing with an abstract workload model.

2) *MicroGrad* is the first proposal to perform intelligent test generation via a Gradient Descent based tuning mechanism, which is shown to outperform other tuning mechanisms and is easier to deploy than expert-driven approaches.

3) *MicroGrad* extends the potential for the Microprobe framework which has a wealth of features for code generation.

4) The modular and abstracted approach to building *MicroGrad* allows the seamless integration of new use cases, execution platforms and tuning algorithms.

II. BACKGROUND

A. Workload Cloning

There are multiple challenges with using real-world applications for architecture benchmarking, such as intellectual property hurdles, effort involved in porting the application to suit the execution framework, as well as long run times. While the advent of standardized benchmark suites have improved the testing ecosystem, there are still several time/resource challenges especially posed to architecture simulation in academic research as well as industry product development. Simulation times are often intractable, even on today’s most efficient simulators running on the fastest processing systems.

Workload Cloning is a general technique to mimic real-world applications or benchmarks via miniature synthetic workloads and has been pursued in multiple prior works [1]–[3], [13], [15], [16]. The technique distills key behavioral characteristics of the original application/benchmark and models them into a synthetic workload. The resultant workload abstracts away any proprietary application characteristics, it is usually significantly shorter in execution time and it can be suitable compiled to make it amenable to both native hardware as well as simulation frameworks. The integral components of the Cloning workflow are discussed below.

1) Application Characteristics: Specific characteristics of an application are captured and used to generate the synthetic workload. These are characteristics which influence the instruction distribution, control flow as well as memory patterns of the application. These characteristics can be divided into microarchitecture-independent and microarchitecture-dependent. The former includes instruction distributions, register dependency distance etc. and memory footprints while the latter includes branch misprediction and cache miss rates and others. While some prior works [3] have used both microarchitecture dependent and independent characteristics in conjunction, others have used a wider range of solely microarchitecture independent characteristics [16], but which are then significantly impacted by compiler optimizations. In this work we use the former i.e. a combination of both microarchitecture dependent as well as independent characteristics which allow optimal capturing of both static and dynamic characteristics of an application on a specific processor architecture.

2) Target Metrics: The generated clones are expected to accurately meet specific target metrics. A full system designer might require the clone to mimic the real application in terms of low-level target metrics such as L1/L2/TLB miss rates, branch misprediction rates, register usage, instruction distribution as well as high-level target metrics like IPC, power, energy or thermal characteristics. In this paper our tool evaluation focuses on cache miss rates, branch mispredictions, instruction distributions, IPC and Power.

3) Generation Mechanism: Prior clone generation mechanisms have comprised of a number of steps, each attempting to feed specific application/benchmark statistics into a model, so as to attempt to generate the required characteristics in the application. These steps include: generating the synthetic workload spine using instruction distribution, memory access pattern modeling, branch predictability modeling, register assignment, and finally code generation [3]. While these steps might individually achieve satisfactory accuracy for their low-level target metric (such as branch misprediction rate), performing them in a sequential manner (a sort of greedy approach) means that there is limited control over other high-level target metrics of interest like IPC.

In our work, we take a more synergic approach to clone generation. By estimating gradients and following the steepest curves, our tuning mechanism is able to inherently sacrifice the accuracy on some specific low-level target metric (for example, L2 cache miss rate) if required, if it aids in optimal achievement of other low-level and high-level target metrics, thereby creating a clone with higher fidelity. Further, our
approach allows a flexible generation time vs. cloning accuracy tradeoff. For instance, a 95% accuracy 1-metric target would take considerably less generation time in comparison to a 99% accuracy N-metric target.

B. Stress testing

Benchmark suites are usually built to represent the nominal behavior of real world applications and not to mimic worst-case scenarios. However, worst-case scenarios in terms of microarchitectural activity, heat dissipation, power consumption and voltage noise [4], [8], [9], [14], [17], [18] are critical to understand the limits and sensitivities of current generation processors, so that future systems can migrate to the most promising regions of the microarchitectural design space. These worst case scenarios are closely tied to the microarchitecture, and must be created in accordance. Thus, stress tests are used to create these worst-case scenarios for a given target execution metric and a specific processor microarchitecture. Considering the complexities and non-linear relationships within the modern processor, manually crafting such stress tests is usually time consuming and tedious. Consequently, automating their generation is of critical importance for a rapid design cycle.

1) Generation Model: As highlighted in prior work [10], there are two prominent design models for stress-test generation: a) based on an abstract-workload model and b) based on instruction-level primitives. In the abstract-model [8], [9], [14] the stress test generation process involves tuning a vector of workload generation parameters/knobs such as instruction mix, register dependency distance, memory footprint / stride patterns and branch transition patterns. The vector is then used to generate the assembly (or high level language) code. On the other hand, for the instruction-level frameworks [10], [17], [18], the tuning is performed directly on the instruction assembly, with per-instruction control.

The key advantage with the abstract workload model is that knobs are well defined, can be selected to be only a few in number, and can potentially have exclusive mapping to particular execution characteristics, significantly reducing the complexity of the tuning required to achieve the maximum stress. The advantage of the instruction-level model is that it provides deterministic and finer granularity of control i.e. on a per-instruction basis. In this work, we adopt the abstract workload model, which provides suitable abstractions to allow for a more modular framework suited to multiple use cases, evaluation frameworks and tuning algorithms.

2) Tuning Mechanism: The role of the tuning mechanism is to nudge the generated test case towards maximum stress (as per the specified stress metric). Prior works built on both the generation models described above have predominantly utilized genetic algorithm (GA) based tuning. GAs tune towards a target metric by applying operators inspired by natural evolution. These operators include: selection of fittest individuals, crossover of features, mutation and guaranteed and elitism prioritization [8]–[10], [14]. The GA parameters used by prior work [10] are shown in Table I. To our knowledge, MicroGrad is the only stress test generation scheme to stray away from GA based tuning. We find that a gradient descent based tuning approach, with stochastic randomness to jump out of local minimas, as well as adaptive step sizes (larger to smaller over time), enable considerably faster (i.e. less number of tuning epochs) and more accurate convergence compared to the GA based approach. For the abstract workload model specifically, our insight is that important GA operators like crossover are rather ineffective, while they are much more valuable in an instruction-level model. On the other hand, the gradient descent approach of following the steepest path to maximizing the metric of interest is very effective when local minimas can be avoided.

It is also interesting to note that the compute cost for a GD based tuning epoch is proportional to the number of knobs of interest, which could be low in the context of many use cases. On the other hand, the compute cost in a GA epoch is proportional to the population size, which is often fixed throughout and therefore usually conservative. Thus every GD epoch is often faster and/or consumes less compute resources in comparison to the GA approach. Our results demonstrate up to a 2.5x benefit for the GD approach.

III. THE MICROGRAD FRAMEWORK

An overview of the MicroGrad framework was shown in Fig.1. MicroGrad is built in a modular manner, allowing ease of use as well as flexibility for further development. Additional use cases and metrics of interest, custom evaluation platforms, as well as improved tuning algorithms, can be developed and integrated conveniently into the framework.

A. Framework Inputs

The inputs to MicroGrad are provided in the form of a configuration file. These inputs are use case dependent and those for our target use cases are described below.

1) Workload Cloning: ① The input specifies the target execution platform, the architecture configuration, the required cloning accuracy, as well as a maximum epoch limit for tuning. If unspecified, defaults are used. ② Further, characteristics of the application (which requires cloning) should be provided and there are multiple ways to do so:

| Parameter          | Value |
|--------------------|-------|
| Population Size    | 50    |
| Individual Size (# knobs) | 25    |
| Mutation Rate      | 3%    |
| Mutation position  | Random|
| Mutation type      | Random|
| Crossover Operator | 1-point|
| Crossover Rate     | 100%  |
| Crossover Position | Random|
| Elitism            | True  |
| Tournament Size    | 5     |

TABLE I: GA parameters
• The numerical values of the application’s metrics of interest (which the clone is expected to match) can be directly provided as input. MicroGrad would then tune the clone to match these values.
• The application binary and its input data can be provided along with specification of the metrics of interest. By default, MicroGrad uses instruction distributions, cache miss rates, branch misprediction rates and IPC as the metrics of interest.
• Application Simpoints [21] can be provided, so as to generate a clone for each simpoint individually. The combination of simpoints and clones can expand the evaluation space of the original application, with potentially one clone for each interesting phase of the application.

2) Stress Testing: ① The input specifies the target execution platform, the architecture configuration and a maximum epoch limit for tuning. ② Metrics of stress are provided as inputs - this can either be a single high-level metric such as IPC or a single low-level metric like branch misprediction rate or a combination of multiple metrics. By default, IPC is used as the stress metric.

#Instruction fractions
ADD = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
MUL = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
FADD = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
FMUL = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
BEQ = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
BNE = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
LD = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
LW = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
SD = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
SW = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

#Dependency distance
REG_DIST = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

#Memory Footprint
MEM_SIZE = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]

#Memory access strides
MEM_STRIDE = [8, 12, 16, 20, 24, 32, 40, 48, 56, 64]

#Memory temporal locality - how many to repeat
MEM_TEMP1 = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

#Memory temporal locality - how often to repeat
MEM_TEMP2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

#Branch pattern randomization ratio
B_PATTERN = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

Listing 1: Example Knobs and range of values

B. Knob Interface
MicroGrad uses a set of knobs to interface between the Tuning mechanism and the Microprobe framework. The Tuning mechanism nudges the knobs in the directions appropriate for the use case, and these knobs are conveyed to Microprobe which generates the test-case based on these knob values. Further, the generated test case is executed on the evaluation framework, whose outputs metrics are fed back to the tuning mechanism to re-tune the knob values. An example subset of the knobs used by MicroGrad and their range of values are shown in Listing 1. In this example subset, the instruction knobs act as fractions of the overall distribution, another knobs allows control of the register dependency distance, the memory knobs specify footprint, stride and temporal locality, and the branch pattern knob specifies the fraction of randomness in the branch pattern. Other tuning knobs are not shown in the interest of space.

Listing 2: Microprobe passes

C. Code Generation
The tuning mechanism presents knob values to Microprobe [5], [12] to generate the corresponding test case. Microprobe is a flexible code generation framework that provides a high level Python scripting interface to access to a rich set of mechanisms and features to control the code generation process. This enables the users to adapt the code generation process to different use cases without having to deal with all low level details. For instance, it has been used in the past
for power model generation [5], maximum power and dI/dt stressmark generation [4], complete architecture characterizations [7], [23] and also reliability analysis [22].

MicroGrad directly uses Microprobe scripting interface to define the code generation process according to the knobs specified. The test case is then generated by a sequence of code synthesis passes which are applied in accordance with the MicroGrad defined ordering rules. A code snippet highlighting some of the standard Microprobe passes used by MicroGrad are shown in Listing 2. More details on these passes and others can be found on the open source Microprobe tool website [12].

```python
# Calculate new configuration
kc = kc - step_size*grad
return kc

Listing 3: Gradient descent tuning
```

D. Gradient-based Tuning

Each tuning epoch involves tuning the knob configuration by evaluating the execution metrics in the vicinity of the current configuration and making changes to the knobs accordingly. Pseudo-code for the tuning mechanism is shown in Listing 3 and features of the mechanism are discussed below.

1. A new tuning epoch starts with capturing the execution metrics (e.g., IPC, energy, cache miss rates) at the previous epoch’s output knob configuration (random configuration, if first epoch). This is the ‘base’ configuration for this epoch. This involves generating the test case with Microprobe at the base configuration, running the test case on the evaluation platform and measuring the base metrics.

2. The goal at the end of the epoch is to find the new knob configuration which is the steepest move (in terms of the matching the use case requirements) from the base configuration.

3. In order to achieve this, the base knob configuration is independently perturbed by +/- δ in each dimension (i.e., each knob). Each resulting configuration is a ‘gradient-check’ configuration. This results in $2^n$ knobs number of ‘gradient-checks’ per epoch.

4. The execution metrics are then captured at each of these ‘gradient-check’ configurations, again by generating test cases with Microprobe and running the test cases on the evaluation platform.

5. For each case, the ‘gradient-check’ execution metrics are compared to the base and the target metrics to obtain a Loss, which is tied to the use case goal.

6. The gradient of the Loss along each knob dimension is calculated by evaluating how much the loss function changed along each dimension’s δ perturbation.

7. This information is used to obtain the new knob configuration - the knobs with the steepest gradients move by ‘one’ step-size, while the other knobs proportionally move by a fractional of the step size. This becomes the starting configuration for the next epoch.

8. Inspired by adaptive learning rate based gradient methods [19], the tuning mechanism’s step-sizes are larger on earlier epochs and gradually become smaller, allowing for rapid convergence earlier but slower but surer convergence later on.

9. To add robustness to the convergence to help avoid local minima, a random set of knobs are skipped in tuning each iteration, with decreasing skipping probability over epochs.

10. Tuning continues until either convergence, the target accuracy or the maximum number of epochs is reached.
TABLE II: Core Configuration

| Parameter         | Small | Large |
|-------------------|-------|-------|
| Frequency         | 2 GHz |       |
| Front-End Width   | 3     | 8     |
| ROB/LSQ/RSE       | 40/16/32 | 160/64/128 |
| ALU/SIMD/FP       | 3/2/2 | 6/4/4 |
| L1/L2 Cache       | 16k/256k | 32k/1M + prefetch |
| Memory            | 1GB   |       |

E. Metric Evaluation

Once the test case is generated and compiled to meet the requirements of the evaluation architecture, the test case is executed on the platform. MicroGrad is able to interface with a number of platforms such as native hardware, performance simulators (e.g., Gem5 [6]) and power estimation frameworks (e.g., McPAT [20]). In the case of simulators, the architecture configuration can be passed as input to MicroGrad and used in the simulator to express the desired architecture.

In terms of capturing metrics, the requisite metrics are dependent on the use case. A stress test use case might require only IPC / Power, whereas a cloning use case might require low-level metrics like mispredictions and miss rates. When using simulators, the MicroGrad interface enables the required metrics to be read from the output dumps of the simulators. In the case of native hardware evaluation, appropriate hardware counters and their required interfacing can be used in similar fashion.

F. Framework Outputs

MicroGrad completes execution when either the target is met or some execution time/resource constraint is reached. The output at the end of execution is dependent on the use case. In the case of Workload Cloning, MicroGrad outputs the clone binary, details of the corresponding knobs and the metrics based on the evaluation of the clone. With stress testing, MicroGrad outputs the stress test binary, the knobs and the stress metrics. In both scenarios, intermediate data can be stored, so as to understand the tuning/execution progress over the epochs (for example, to improve the tuning algorithm).

IV. EVALUATION

A. Experimental Setup

1) Workloads: To evaluate the cloning use case, we choose 8 benchmarks from the SPEC INT CPU2006 [11] suite and generate clones on simpoints [21] of 100 million instructions. The generated test cases (for both use cases) are made up of roughly 500 static instructions in an endless loop and run for a total of 10 million dynamic instructions.

2) Evaluation Framework: We target the Gem5 [6] architectural performance simulator and the McPAT [20] power estimation framework. While performance numbers and module level statistics can be evaluated from Gem5 alone, power estimation requires the transfer of execution statistics from Gem5 to McPAT, based on which dynamic power is estimated.

3) Target Microarchitectures: We target the RISC-V ISA. We model two cores – Large and Small– to evaluate the performance of MicroGrad on different corners of the architecture design space. The details of each core are listed in Table II. For the power template, we use the default McPAT configurations commensurate with these core sizes.

4) Metrics / Accuracy: For Workload Cloning, we focus on: i) Integer, Branch, Load, Store instructions, ii) L1D, L1I, L2 cache hit rates, iii) Branch misprediction rate and iv) IPC. For Stress Testing, we focus separately on IPC and Dynamic Power. The Loss function utilized by the tuning algorithm calculates log loss over the metrics of interest specified above. Where applicable, we target an accuracy of 99% across the metrics.

B. Workload Cloning

In Fig.2 and Fig.3 we showcase the efficiency of MicroGrad towards Workload Cloning. Fig.2 shows the workload clones generated across the 8 benchmarks on a Large core while Fig.3 shows the same on a Small core. In the figures, the circumferential axis represents different metrics - instructions distributions, mispredictions, cache miss rates and IPC. The radial axis represents the accuracy of the clone’s metric compared to the original benchmark (1 indicates complete accuracy).

For the Large core, over the eight benchmarks, the accuracy across all metrics is close to 1 (average error is less than 1%). Worse case scenario is seen in libquantum wherein there is close to a 5% error in the branch misprediction rate and the data cache (DC) hit rate.

In the case of the Small core, results are similar (average error is less than 2%). The accuracy is marginally less compared to the Large core due to the higher metric sensitivity in a core of smaller size. This is due to program characteristics having a larger impact on the execution flow, since the core is not over provisioned with resources. The worse case error is close to 10% in the case of xalancbmk’s IC hit rate. We note that there is potential for more knobs to be implemented in MicroGrad that can control IC Hit Rates with higher accuracy, which we seek to implement in the future.

The captions of both figures indicate the number of epochs required to create the workload clones. Epochs vary from only 5, to a maximum of 52, clearly highlighting that MicroGrad’s high accuracy is achievable in very few tuning epochs.

The accuracy and fast tuning capability of MicroGrad is heavily influenced by the Gradient Descent tuning algorithm. To showcase this, we compare against a Genetic Algorithm based approach in Fig.4 for the Big core. The GA parameters are taken from prior work and were shown in Table I. For this analysis, we allow the GA based approach to run for the same number of tuning epochs as the GD based approach. The figure shows that the accuracy achieved by GA is considerably lower than the GD approach (note that the ratios on the radial axes are far greater). The average error in comparison to the original benchmarks is roughly 30%, with worst case errors of more than 50%
allows for flexible development on multiple fronts, including algorithm fits seamlessly into the MicroGrad framework. This is
(population size) in GA vs 20 evaluations per epoch (2 x
(b) bzip2 [5], (c) gcc [19], (d) hmmer [52], (e) libquantm [45], (f) mcf [21], (g) sjeng [15], (h) xalancbmk [26]

Next, we discuss MicroGrad’s proficiency in stress testing. Fig.5 shows a compute-focused performance stress test sce-
C. Stress Testing

Next, in Fig.6 we show a compute-focused stress test
Fig. 4: Workload Cloning targeting a "large" core, with Genetic Algorithm. Top Left to Right Bottom: (a) astar [10 epochs], (b) bzip2 [5], (c) gcc [19], (d) hmer [52], (e) libquantum [45], (f) mcf [21], (g) sjeng [15], (h) xalanckbmk [26]

Fig. 5: Performance virus: GD vs GA

Fig. 6: Power virus: GD vs GA

In this paper, MicroGrad is built in a modular manner with techniques. To produce fast and accurate workload clones and stress tests, GA-based solutions in terms of time to a solution (epochs) and efficiency in resource utilization.

In summary, we presented MicroGrad, an open-source centralized framework for workload cloning and stress testing. Key novel features in MicroGrad are its gradient based tuning approach, in combination with an abstract workload model, can generate highly accurate stress tests on different use cases. In addition, the gradient decent tuning outperforms existing GA-based solutions in terms of time to a solution (epochs) and efficiency in resource utilization.

Beyond the specific quantitative benefits that are shown in this paper, MicroGrad is built in a modular manner with clear interface boundaries both internally as well as externally.
This allows it to be a promising springboard for wide future development - be it in terms of the use cases it can support, the evaluation platforms it can execute on, as well as running more optimum tuning algorithms.

For example, MicroGrad can seamlessly support other use cases like bottleneck analysis i.e. sweeping over a specified range of finer execution characteristics -such as cache miss rate– and analyzing its bottle-necking impact on the overall processor execution. The framework also allows for experiments on native hardware and other forms of stress testing like voltage droops. Thus, we envision that with future development, MicroGrad can accelerate the entire Innovate-Build-Analyze cycle as a whole, which is especially critical in the coming open-source hardware era.

REFERENCES

[1] R. H. Bell, R. R. Bhatia, L. K. John, J. Stuecheli, J. Griswell, P. Tu, L. Capps, A. Blanchard, and R. Thai, “Automatic testcase synthesis and performance model validation for high performance processor processors,” in 2006 IEEE International Symposium on Performance Analysis of Systems and Software, 2006, pp. 154–165.

[2] R. H. Bell and L. K. John, “Efficient power analysis using synthetic test-cases,” in IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005, pp. 110–118.

[3] R. H. Bell and L. K. John, “Improved automatic testcase synthesis for performance model validation,” in Proceedings of the 19th Annual International Conference on Supercomputing, ser. ICS 05. New York, NY, USA: Association for Computing Machinery, 2005, p. 111120. [Online]. Available: https://doi.org/10.1145/1088149.1088164

[4] R. Bertran, A. Buyuktosunoglu, P. Bose, T. J. Siegel, G. Salem, S. Carey, R. F. Rizzolo, and T. Strach, “Voltage noise in multi-core processors: Empirical characterization and optimization opportunities,” in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014, pp. 368–380.

[5] R. Bertran, A. Buyuktosunoglu, M. S. Gupta, M. Gonzalez, and P. Bose, “Systematic energy characterization of cng/smt processor systems via automated micro-benchmarks,” in 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012, pp. 199–211.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” SIGARCH Comput. Archit. News, 2011.

[7] E. J. Fluhrr, R. M. Rao, H. Smith, A. Buyuktosunoglu, and R. Bertran, “IBM POWER9 circuit design and energy optimization for 14-nm technology,” IBM J. Res. Dev., vol. 62, no. 4/5, p. 4, 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8383685/

[8] K. Ganesan and L. K. John, “Maximum multicore power (mampo) an automatic multithreaded synthetic power virus generation framework for multicore systems,” in SC ’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–12.

[9] K. Ganesan, J. Jo, W. L. Bircher, D. Kasenidis, Z. Yu, and L. K. John, “System-level max power (sympo) - a systematic approach for escalating system-level power consumption using synthetic benchmarks,” in In the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2010.

[10] Z. Hadjilambrou, S. Das, P. Whatmough, D. Bull, and Y. Sazeides, “Gest: An automatic framework for generating cpu stress-tests,” 03 2019, pp. 1–10.

[11] J. L. Henning, “Spec cpu2006 benchmark descriptions,” SIGARCH Comput. Archit. News, 2006.

[12] IBM Research, “Microprobe: Microbenchmark generation framework,” https://github.com/IBM/microprobe.

[13] A. Joshi, L. Eckhout, R. H. Bell, and L. John, “Performance cloning: A technique for disseminating proprietary applications as benchmarks,” in 2006 IEEE International Symposium on Workload Characterization, 2006, pp. 105–115.

[14] A. Joshi, L. Eckhout, L. K. John, and C. Isen, “Automated microprocessor stressmark generation,” in 2008 IEEE 14th International Symposium on High Performance Computer Architecture, 2008, pp. 229–239.

[15] A. Joshi, L. Eckhout, R. H. Bell, and L. K. John, “Distilling the essence of proprietary workloads into miniature benchmarks,” ACM Trans. Archit. Code Optim., vol. 5, no. 2, Sep. 2008. [Online]. Available: https://doi.org/10.1145/1401112.1400115

[16] A. Joshi, L. Eckhout, and L. John, “The return of synthetic benchmarks,” in 2008 SPEC Benchmark Workshop, 2008, pp. 1–11.

[17] Y. Kim and L. K. John, “Automated dt/dt stressmark generation for microprocessor power delivery networks,” in IEEE/ACM International Symposium on Low Power Electronics and Design, 2011, pp. 253–258.

[18] Y. Kim, L. K. John, S. Pant, S. Manne, M. Schulte, W. L. Bircher, and M. S. S. Govindan, “Audit: Stress testing the automatic way,” in 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012, pp. 212–223.

[19] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” arXiv preprint arXiv:1412.6980, 2014.

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” SIGARCH Comput. Archit. News, 2011.

[21] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, “Using simpoint for accurate and efficient simulation,” in International Conference on Measurement and Modeling of Computer Systems, 2003.

[22] K. Swaminathan, R. Bertran, H. M. Jacobson, P. Kodva, and P. Bose, “Generation of stressmarks for early stage soft-error modeling,” in 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019, Supplemental Volume. IEEE, 2019, pp. 42–48. [Online]. Available: https://doi.org/10.1109/DSN.2019.00022

[23] T. Webel, P. M. Lobo, R. Bertran, G. Salem, M. Allen-Ware, R. F. Rizzolo, S. M. Carey, T. Strach, A. Buyuktosunoglu, C. Lefurgy, P. Bose, R. Nigaglioni, T. J. Siegel, M. S. Floyd, and B. W. Curran, “Robust power management in the IBM z13,” IBM J. Res. Dev., vol. 59, no. 4/5, 2015. [Online]. Available: https://doi.org/10.1147/JRD.2015.2446872