Automatic Calibration of Micro-Architecture Description Models

[†]Zhuoran Zhao, [‡]Gary Morrison, [†]Andreas Gerstlauer
[†]Electrical and Computer Engineering Department, The University of Texas at Austin
[‡]Freescale Semiconductor Inc.
[†]{zhuoran,gerstl}@utexas.edu, [‡]ra0801@freescale.com

ABSTRACT

Cycle-accurate platform simulators are key components of virtual prototyping for system level design. High-level architecture description language (ADL) based instruction-set simulation (ISS) models are often used to verify functionality and estimate system performance during early design stages. However, development of such abstract models to faithfully cover every micro-architecture detail is a tedious and error-prone process, often resulting in performance mismatches between the cycle-accurate model and RTL. Thus, time-consuming manual calibrations of high-level architecture models against existing RTL code across a large set of benchmarks are typically needed.

To address this problem, we propose an automated framework for calibration of architecture models against RTL to automatically discover and generate accurate platform simulators for rapid early exploration and development. Our framework is based on open-sourced ADL environments from industry, making use of open-source evolutionary algorithm (EA) libraries to tune pre-defined parameters in a given template and automatically derive accurate architecture models for a given RTL processor. During the evaluation stage of the calibration, multi-tiered fitness assessment is used to reduce the calibration time.

We verify our flow by calibrating parameterized processor model templates against RTL reference traces. Results show that our framework can sucessfully generate architecture model in an average of 8.1 hours with more than 98% timing accuracy.

1. INTRODUCTION

Virtual prototyping is widely used in embedded system design methodologies for quick functional validation and performance estimation at early desgin stages. Typical virtual prototypes incoroporate cycle-accurate processor simulators to get performance feedback and direct early exploration and development. Cycle-accurate instruction-set simulation (ISS) models automatically derived from high-level, easily retargetable architecture description languages (ADLs) are often used in this context. Although ADL-based flows already provide useful abstractions to quickly capture processor models, development of such models to faithfully cover every micro-architecture detail is still a tedious and error-prone process. Performance mismatches between the cycle-accurate ISS model and RTL are very common. To resolve such mismatches, manual calibrations of high-level architecture models against existing RTL code across a large set of benchmarks are typically needed, which is usually time-consuming and lacking flexibility.

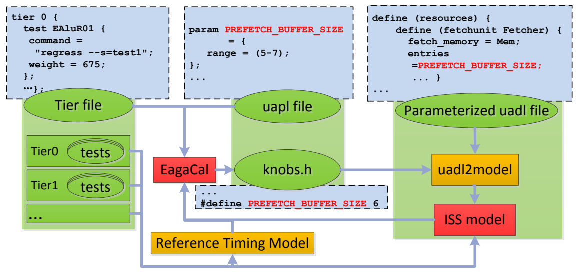

```
tier 0 {
  test EAluR01 {
   command =
    "regress --s=test1";
   weight = 675;
  };
  …};
```
```
param PREFETCH_BUFFER_SIZE
       = {
   range = (5-7);
};
...
```
```
define (resources) {
    define (fetchunit Fetcher) {
      fetch_memory = Mem;
      entries
       =PREFETCH_BUFFER_SIZE;
       ... }
...
```
```
...
#define PREFETCH_BUFFER_SIZE 6
```

Figure 1: Calibration infrastructure overview.

To address this problem, we propose an automated framework for calibration of architecture models against RTL. Our framework automatically discovers and generates accurate platform simulators for rapid early exploration and development. In this report, we document our proposed framework for automated model calibration. Our framework is based on Freescale's uADL (micro-architecture description language) modeling infrastructure [1] and makes use of open-source evolutionary optimization libraries to tune pre-defined parameters in a given template and automatically derive accurate architecture models for a given RTL processor. We have applied our framework to automatically calibrate parameterized template model against PowerPC-based reference models. In all cases, our framework was able to discover the correct parameter set.

2. INFRASTRUCTURE OVERVIEW

Figure 1 shows an overview of our calibration infrastructure. The first step of the calibration is parametrizing the existing uADL models with possible micro-architectural parameters. Most parameterization is accomplished through C preprocessor text substitution performed by the uADL compiler.

In order to specify the list of valid parameters and the corresponding ranges, we define a micro-architecture parameterization language (*uapl*) file. To evaluate each genome, i.e. uADL instantiation with a valid set of parameters, we introduce a multiple-tier benchmark hierarchy, which is specified in a *Tier* file. The syntax rules of our *uapl* and *Tier* files are fed into Lex and Yacc [2] to generate the corresponding parser for our calibration tool. Then, the *uapl* and *Tier* parser along with the calibration source code, which mainly includes invocation commands for calibration components and the evolutionary library are linked and compiled to generate our executable calibration tool called *EagaCal*.

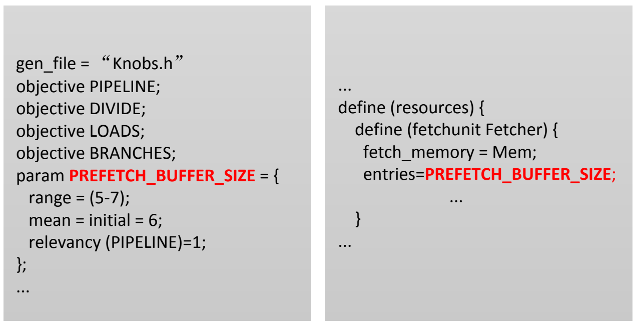
During the start-up of *EagaCal*, the calibration tool will read the *uapl* file to build up the encoding of parameters into a genome and the initial population for the evolutionary algorithm. Furthermore, the *Tier* file is read to specify the benchmark sets for fitness assessment. Once input files have been read, an iterative exploration is performed across the given set of benchmarks until a pre-defined stop metric is met, such as reaching a maximum number of generations or hitting an accuracy threshold.

During the exploration, each genome is translated into an automatically generated knobs file, which is in turn used to compile the corresponding uADL model into an ISS executable using the existing *uadl2model* command. Finally, the ISS' performance accuracy will be evaluated on a selected subset of a given set of benchmarks. These benchmarks must previously have been run on the reference model (usually RTL) to generate a set of reference timing traces, which is one-time effort, and clustered into multiple tiers to be dynamically added into the evaluation benchmark set. During calibration, the timing result of each test case running on the generated ISS model will be compared against the corresponding timing reference to get the performance accuracy.

Finally, the average accuracy of each benchmark weighted by the instruction count will be assigned as the fitness value of the corresponding genome and further fed back into the evolutionary algorithm.

3. UADL-MODEL PARAMETERIZATION

To perform the calibration, uADL models need to be made explorable and tunable through parameterization. Most parameterization can be accomplished through C-preprocessor text substitution. Parameterized uADL models use *#define*d macros like any other program. The value assigned to each macro can be defined in a separate header file, which will be included by the uADL model and externally modified by the evolutionary algorithms.



```
gen_file = "Knobs.h"
objective PIPELINE;
objective DIVIDE;
objective LOADS;
objective BRANCHES;
param PREFETCH_BUFFER_SIZE = {
   range = (5-7);
   mean = initial = 6;
   relevancy (PIPELINE)=1;
};
...
```

(a) uapl file

```
...
define (resources) {
   define (fetchunit Fetcher) {
     fetch_memory = Mem;
     entries=PREFETCH_BUFFER_SIZE;
           ...
   }
...
```

(b) uadl file

Figure 2: uADL parameterization.

To further specify the complete list of parameters and their allowable value ranges (knobs), micro-architecture parameterization language (uapl) files are introduced into our calibration infrastructure. Figure 2 shows abbreviated examples of a code snippet of a parameterized uADL model (Figure 2(b)) and its corresponding *uapl* file (Figure 2(a)). In the original uADL file, the number of entries in the fetch unit can be an integer number in the range of 5-7. In the parametrized uADL model in Figure 2(b), the integer number is replaced with a label PREFETCH_BUFFER_SIZE, and the corresponding range of this label is then specified in the *uapl* file in Figure 2(a). Its actual value is later defined

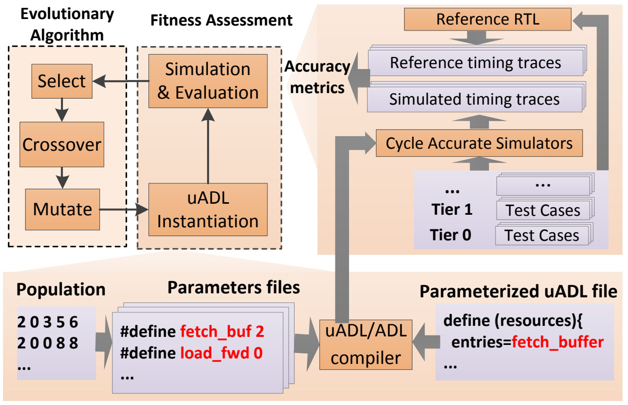


Figure 3: Genetic Algorithm calibration framework.

in a separate header file generated during the calibration process according to the specifications from the *uapl* file.

Besides the example shown in Figure 2, more advanced parameterizations such as pipeline configuration are also employed in our infrastructure.

4. EVOLUTIONARY CALIBRATION

An evolutionary algorithm (EA) is a generic population-based metaheuristic optimization algorithm [3]. In our case, we use a Genetic Algorithm (GA), a subclass of EA, to construct our calibration infrastructure. GAs typically consist of selection, genetic operators and termination. In this section, the corresponding implementations and customizations in our calibration framework will be illustrated.

The overall setup of our evolutionary calibration framework is shown in Figure 3. As the value of each knob in the calibration is selected from a set with finite discrete elements, we encode the choices of knobs into integer genomes. Before the selection stage, each integer genome in the current population is translated into header files with micro-architecture parameter values. The header files are then compiled with the parameterized uADL file to generate the simulator instances, which are run through multiple tiers of benchmarks to obtain the performance differences in comparison to the reference traces. The average accuracy of each test case weighted by its total instruction count is assigned as the fitness value of each genome. After this, tournament selection is applied in the GA framework to generate the next evolutionary generation from the current one.

After the selection stage, a uniform crossover operator is used to randomly choose the crossover points among parent genome pairs and generate off spring genomes, followed by a mutation stage to randomly change each gene within a valid range (as shown in the *Evolutionary Algorithm* block in Figure 3).

The calibration framework will evolve the population until it reaches a convergence point. In our case, we identify the convergence by the improvement of the best individual. If the best fitness value has no improvement for a threshold number of consecutive populations, the algorithm will be terminated and the results will be reported.

In the entire calibration framework, the fitness assessment stage is the most time consuming process. To deal with this problem, we further utilize parallelized compilation and multi-tiered fitness assessment to reduce the calibration

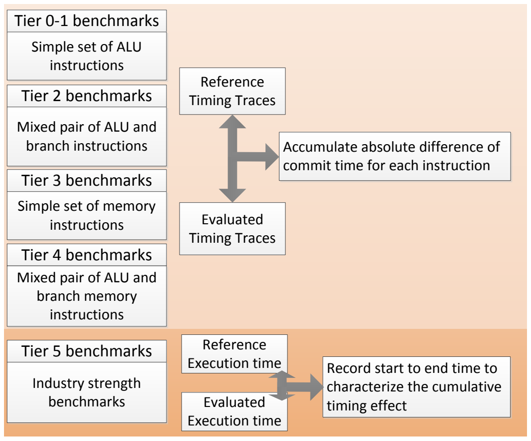

Tier 0-1 benchmarks
Simple set of ALU instructions
Tier 2 benchmarks
Mixed pair of ALU and branch instructions
Tier 3 benchmarks
Simple set of memory instructions
Tier 4 benchmarks
Mixed pair of ALU and branch memory instructions
Reference Timing Traces
Evaluated Timing Traces
Accumulate absolute difference of commit time for each instruction
Tier 5 benchmarks
Industry strength benchmarks
Reference Execution time
Evaluated Execution time
Record start to end time to characterize the cumulative timing effect

Figure 4: Multi-tiered benchmark hierarchy.

time, as will be described in more detail in the following sections.

4.1 Parallelized compilation and simulation

Although the evolutionary algorithm has inter-generation dependency, intra-generation parallelism can be exploited to reduce the evaluation time. In our calibration framework, the compilations and simulations for different genomes are executed in parallel to reduce the evaluation time for one generation. The user can specify the parallelism window size and even migrate the compilation tasks among several machines with the interface provided by our framework.

4.2 Multi-tiered fitness assessment

In the stage of fitness assessment, we set up a multi-tier benchmark hierarchy, shown in Figure 4, to evaluate each considered uADL instantiation. The first four tiers consist of bottom-up, directed-random timing tests in which the evaluation will accumulate absolute differences in issue and commit times for each instruction between uADL and RTL, or between uADL and reference uADL. To be more specific, *Tier 0 and 1* contain basic tests of the most fundamental ALU individual instruction classes and *Tier 2* contains combinations of those instruction classes, followed by *Tier 3* with simple memory instructions sets and *Tier 4* with mixed pairs of ALU, memory instructions. The last tier is introduced for the purpose of top-down, industry benchmark based regression using compiled C code that, generally, represents realistic customer applications. Different from the first five tiers, the uADL instantiation is tested in terms of start-to-end time to characterize the total, cumulative timing effect of the entire model as a whole. By contrast, the goal of bottom-up testing in the lower tiers is to study in detail the instruction-by-instruction timing for different instruction classes. In both cases, cycle count differences are divided by the total cycle counts to compute the final relative error metric (in percent error) of each test case.

The motivation of such a benchmark hierarchy is to calibrate the target model over different aspects while being able to quickly prune the design space using multi-tier evaluation. Moreover, dynamically adding tiers to the calibration framework can also avoid running too many benchmarks during the early generations, which largely reduces the evaluation time during calibration.

This idea is shown in Figure 3 and Figure 4. The algorithm will evaluate the population starting with *Tier 0*, until the best fitness individual is not improved after a threshold number of consecutive generations, *Tier 1* is added into the benchmark set and the whole population will be reevalutated by *Tier 0* and *Tier 1*. The other tiers will be added to the benchmark set in the same way and the algorithm will be terminated when no improvement occurs once all the tiers have been added.

Table 1: Summary of uADL Parameters

Objectives	Parameters
Pipeline	Prefetch buffer size; Forward execution path is existing for ALU, compare, load, multiplication and devide instructions or not;

Memory Operations	Memory address bandwidth;
	Memory read latency;
	Memory write latency;

Multiplication	Latency of multiplication with:
	1-8 leading zeros;
	9-16 leading zeros;
	17-24 leading zeros;
	25-32 leading zeros;

Devide	Divided by 0 is treated as special case or not;

Table 2: Calibration algorithm Parameters

Population Size	20
Crossover Rate	0.5
Mutation Rate	0.4
Parallelism Window Size	4
Population Steady Threshold	4
Maximum Generations	80

For fitness assessment, the corresponding simulator instance of a genome will go though all the test cases from the tiers included in the current benchmark set, and the negative average running error percentage of all test cases weighted by total instruction counts will be calculated and assigned as the genome's individual fitness.

After running a genome with n test cases, we compute the final fitness as the negative weigthed average of error percentages (E), where each test case is weighted by its instruction count (W) specified in the tier file:

$$fitness = -\frac{W_1 * E_1 + W_2 * E_2 + \ldots W_n * E_n}{W_1 + W_2 + \ldots W_n}.$$

Currently we are using a single objective algorithm to verify the calibration performance against reference RTL traces. Any multi-objective appraoch will probably find its usefulness if calibration for more advanced model introduces more issues, but is a subject for future work.

5. EXPERIMENTS AND RESULTS

We use Evolving Objects (EO) [4], a template-based, ANSI-C++ evolutionary computation library, to implement our calibration framework. Calibrations were performed on a quad-core Intel i7 workstation running at 2.6 GHz, on which we applied a 4-way parallelization of the compilation process. We applied our framework to calibrate an artificial parameterized model against reference RTL traces. Five tiers of benchmarks were used without applying any industry-strength top-down benchmarks in Tier 5.

A summary of the tunable uADL parameters in our expreiments are shown in Table 1. During the calibration, individuals in the initial generation are randomly generated from their corresponding value sets, and further evolve within the valid range.

The calibration algorithm parameters used in our experi-

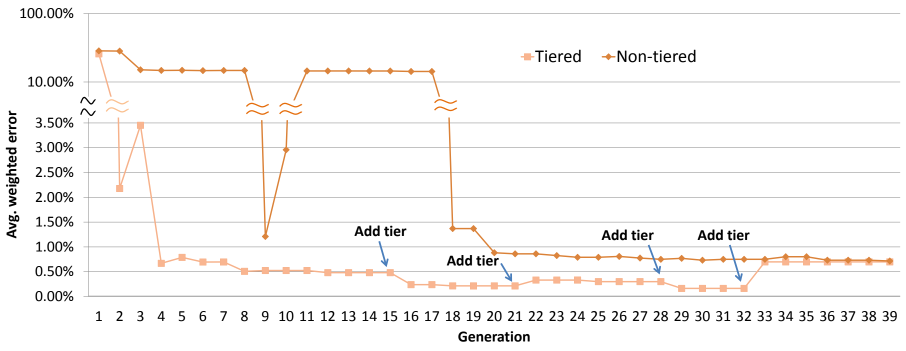


Figure 5: Fitness by generations.

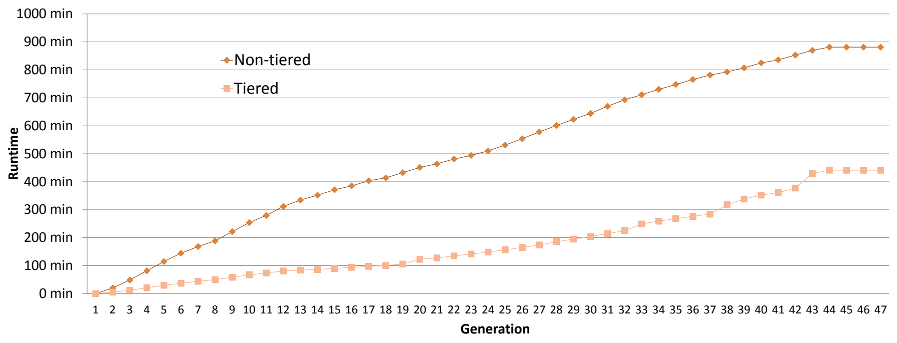


Figure 6: Accumulated execution time by generations.

Table 3: Calibration Runtime Statistics

	Tiered	Non-tiered	Improvement
Avg. runtime	08:21:43	16:10:38	48.3%
Max. runtime	08:24:14	16:21:21	48.6%
Avg. generations	48	48	0%
Max. generations	48	48	0%

ments are as shown in Table 2 Note that the number of *Population Steady Threshold* is used to deicide whether a new tier needs to be added into the benchmark set. In our case, if no improvement of best fitness occurs for 4 consecutive generations, then a new tier will be added. We compared our multi-tiered fitness assessment with the case where all the test cases are organized into one single tier, and recorded the statistics of the algorithm behavior along with the generations evolved. For multi-tiered and non-tiered algorithms, we collected results for 3 independent runs each.

Figure 5 and Figure 6 show a representative calibration progress of applying our calibration framework to our example model both with and without multi-tiered fitness assessment, represented as “Tiered” and “Non-tiered”, respectively. Figure 5 shows the absolute value of fitness for the best individual recorded in each generation. Tiers are added right after generation 15, 21, 28 and 32. From the slope of the curve we can see the new tiers are only added if the population reached a steady state and can not be improved anymore using the current benchmark set. The sudden decrease in fitness after adding a new tier is due to re-evaluation based on the new benchmark set.

Figure 6 shows the accumulated runtime of our framework, Since the multi-tiered fitness assessment will start with a single tier and only add new tiers if necessary, the slope of the accumulated runtime curve starts with a small rate and only increases when new tier is introduced, resulting in a much smaller average evaluation time across all tiers. Thus, although the two approaches will give the same results after a similar number of generations, the multi-tiered fitness assessment can significantly reduce the total calibration time.

Overall, results averaged over all 3 runs for each algorithm(Table 3) show that our mult-tiered approach can reduce the calibration time by an average of 48.3% and in all cases reduced the timing error comparing with RTL traces to less than 2% after 43 generations.

6. SUMMARY AND CONCLUSIONS

In this paper, we presented a framework for automated calibration of high-level cycle-accurate simulation models against RTL and other reference models. Our framework is based on tunable, parametrized uADL models from Freescale. In our framework, the Evolving Objects (EO) library is used with slight modifications as our calibration engine. Different types of test cases are organized into a multi-tier benchmark hierarchy to improve the calibration performance. Results show that our framework can effectively calibrate parameterized uADL model against RTL timing traces and generate architecture model with more than 98% timing accuracy in an average of 8.1 hours.

7. REFERENCES

[1] The architectural description language project, ver 2.0.0. [Online]. Available: http://opensource.freescale.com/fsl-oss-projects

[2] “Lex and yacc,” in *lex & yacc, 2nd Edition*. O’Reilly Media, 1992, pp. 1–25.

[3] “An overview of evolutionary computation,” in *Evolutionary Computation for Modeling and Optimization*. Springer New York, 2006, pp. 1–31.

[4] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer, “Evolving objects: A general purpose evolutionary computation library,” *Artificial Evolution*, vol. 2310, pp. 829–888, 2002. [Online]. Available: http://www.lri.fr/ marc/EO/EO-EA01.ps.gz