Hindawi
Scientific Programming
Volume 2017, Article ID 1046161, 24 pages
https://doi.org/10.1155/2017/1046161

Research Article

A Language and Preprocessor for User-Controlled Generation of Synthetic Programs

Alton Chiu, Joseph Garvey, and Tarek S. Abdelrahman

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada

Correspondence should be addressed to Tarek S. Abdelrahman; tsa@eecg.toronto.edu

Received 16 August 2016; Revised 3 December 2016; Accepted 14 December 2016; Published 2 August 2017

Academic Editor: Frank Hannig

We describe Genesis, a language for the generation of synthetic programs. The language allows users to annotate a template program to customize its code using statistical distributions and to generate program instances based on those distributions. This effectively allows users to generate programs whose characteristics vary in a statistically controlled fashion, thus improving upon existing program generators and alleviating the difficulties associated with ad hoc methods of program generation. We describe the language constructs, a prototype preprocessor for the language, and five case studies that show the ability of Genesis to express a range of programs. We evaluate the preprocessor's performance and the statistical quality of the samples it generates. We thereby show that Genesis is a useful tool that eases the expression and creation of large and diverse program sets.

1. Introduction

Large sets of programs are important in a number of areas of computer science and engineering. For example, in supervised machine learning (ML) for performance autotuning, a sufficiently large number of training programs are needed to represent the desired program space. Similarly, in compiler testing, successfully running test programs through a compiler increases confidence in its functionality and correctness. Finally, in software testing, the adequacy of the testing strategy of a program is measured by testing a large number of faulty mutant versions of the program [1]. The percentage of mutants for which errors are detected is used as a measure of the adequacy of the testing.

However, the number of real programs available for use is often limited. For compilers, it can be difficult to build up a diverse set of programs that contain enough functionality combinations and error scenarios. Similarly, benchmark suites used to evaluate performance of software and systems [2, 3] usually consist of only tens of programs and are usually too small to build sufficiently large and diverse training sets for ML models. Finally, a large number of mutant programs are needed to increase confidence in a testing strategy. Thus, *program generators* are often used to produce synthetic programs for use in such situations.

There are several existing program generators [4–6]. However, these generators suffer from limitations, in particular, the lack of user control over the generated code [4], inflexible and restrictive use cases or target languages [6–8], and difficulties with associated tools [6]. Ad hoc methods of generating large program sets, such as the use of Perl or Python scripts, also have their own limitations; the resulting scripts are difficult to write, maintain, and extend.

Thus, in this work, we design, implement, and evaluate *Genesis*, a program generation language that addresses the above shortcomings. Genesis facilitates the generation of synthetic programs in a statistically controlled fashion. It allows users to annotate a template program to identify and parameterize those segments of the program they wish to vary, the values each parameter may take, and the desired statistical distribution of these values across generated programs. The Genesis preprocessor uses the annotations to generate programs based on a template program, with the values of each parameter drawn from its corresponding distribution.

Genesis is unique in that it allows the generation of synthetic code with controlled statistical properties, which is important in some application domains. The constructs of the language provide a simple yet flexible means of varying template code. They also allow for the hierarchical composition of generated code segments. This facilitates the generation of large numbers of programs that are arbitrarily long with only a handful of constructs. It also makes it easy to create, modify, and extend existing Genesis programs. Genesis is target-language agnostic in that it can be used with template programs written in various programming languages.

The goal of this paper is to provide a detailed description of the Genesis language and to demonstrate its utility through a number of case studies of problems in which large program sets are needed. In addition, the paper provides an evaluation of the performance of the Genesis preprocessor. The paper is organized as follows. Section 2 gives an overview of Genesis with a simple example to illustrate its basic use. Section 3 gives a detailed description of the constructs of Genesis language. Section 4 describes five case studies of using Genesis. The current implementation of the Genesis preprocessor prototype is described in Section 5 and its evaluation in Section 6. Finally, Sections 7 and 8 review related work and provide some concluding remarks, respectively.

2. Overview of Genesis

The Genesis preprocessor takes two inputs: a *template program*, expressed in a standard programming language, such as C, Java, or C++, and a *Genesis program*, expressed using the Genesis language, as shown in Figure 1. The template program contains *references* to Genesis *features*, which are code snippets that are to vary across generated *instance programs*. The Genesis program defines the features using code mixed with Genesis *names*. When a feature referenced in the template program is *processed* by the preprocessor, the names in its definition are replaced by values sampled from user-specified distributions, producing an actual code snippet that replaces the feature reference. The following example helps to demonstrate this process.

```
for (int i = 0; i < n; ++i) {
     :
    t1 = x[c1*i+s1];
     :
    t2 = x[c2*i+s2];
     :

}
```

The loop in this example, extracted from a GPU kernel, makes two reads to an array, `x`, in each iteration. The memory access constants `c1`, `s1`, `c2`, and `s2` affect memory performance, and it is desired to use them as features to train a machine learning model. Thus, we wish to generate a number of training programs that have different values of these constants. For the sake of this example, it is desired to uniformly distribute `c1` and `c2` over the range 1 to 4 and `s1` and `s2` over the range 0 to 7.

A Genesis program and a template program that could be used to generate such training programs are shown on the left side of Figure 1. The template program is essentially the code from the example but with the memory accesses replaced by references to the feature `mem_access`. The feature itself is defined in the Genesis program, delimited by `begin genesis` and `end genesis`, as the code snippet `x[${coef}*i+${offs}]`. The two Genesis names `coef` and `offs` are used in this code snippet. The values of `coef` and `offs` are taken from the distributions `coef_dist` and `offs_dist` as indicated by the `sample` constructs in their definitions. The distributions themselves are defined by Genesis' `distribution` construct declared in the Genesis program.

The `generate` statement in the Genesis program instructs the preprocessor to generate 15 instances of the template program. In each instance, the preprocessor processes the feature `mem_access` twice, sampling the values of each name from its respective distribution. The right side of Figure 1 shows some examples of the programs produced.

3. The Genesis Language

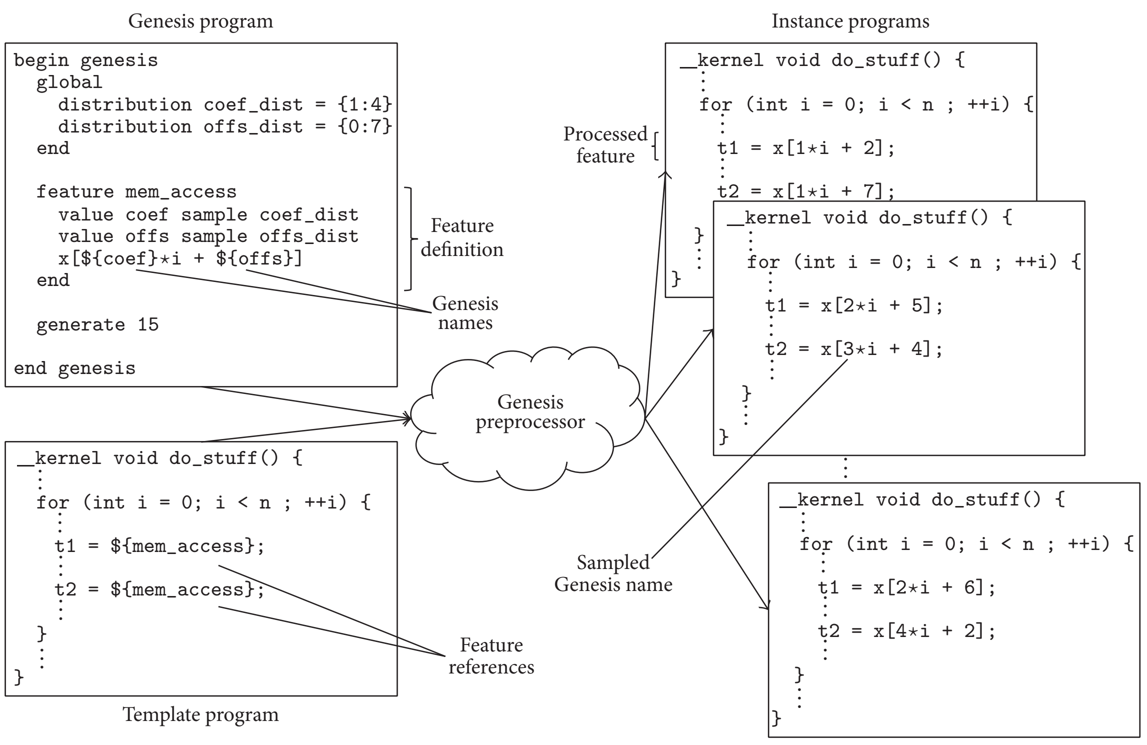
3.1. Design. There are several design concerns that we faced when designing Genesis. We briefly discuss some of these concerns and rationalize the decisions we made.

One important design concern in Genesis is the choice of its programming paradigm. We opt to use the imperative paradigm [9] because the domains we expect Genesis to be used in (i.e., compiler testing, automatic performance tuning, etc.) mostly employ imperative languages, such as C, C++, or OpenCL. Thus, the use of an imperative paradigm for Genesis makes it easier to adopt it in these domains. Nonetheless, fundamentally, there is no limitation preventing it from being used with functional and/or declarative target languages.

A second design concern is whether to have Genesis as a standalone language or embed it within a host language, such as C. The latter option has the advantage of providing a rich type system for Genesis variables and entities. However, it would severely limit the portability of the language. By designing Genesis as a preprocessor with simple data types, it becomes applicable to many target programming languages or even possibly nonprogramming ones (e.g., our image layering applications described in Section 4.4).

Yet another design concern in Genesis is that of variable scoping. We adopt a simple scoping scheme. Genesis variables and entities defined in a feature are *local* to that feature and can only be used within it. In contrast, Genesis variables/entities that are defined in the global section of a Genesis program (see Section 3.3 for the description of Genesis sections) are *global* and can be used anywhere in the Genesis program. Finally, variables defined within the program section of Genesis are local to the current program being generated. The choice of this scoping scheme leads to natural semantics for the sampling of variables, as discussed in Section 3.3.

Finally, an important design concern is the typing of variables. While it is possible to envision a rich type set and/or dynamic typing of variables that is common in various languages, we elect to use a simple typing scheme in which variables take one of four types: integer, float, string, or Boolean. The type of a variable is inferred from the values assigned to it. The choice of these types is driven by our initial



```
begin genesis
  global
    distribution coef_dist = {1:4}
    distribution offs_dist = {0:7}
  end

  feature mem_access
    value coef sample coef_dist
    value offs sample offs_dist
    x[${coef}*i + ${offs}]
  end

  generate 15

end genesis
```

```
__kernel void do_stuff() {
  ⋮
  for (int i = 0; i < n ; ++i) {
    ⋮
    t1 = ${mem_access};
    ⋮
    t2 = ${mem_access};
    ⋮
  }
  ⋮
}
```

```
__kernel void do_stuff() {
  ⋮
  for (int i = 0; i < n ; ++i) {
    ⋮
    t1 = x[1*i + 2];
    ⋮
    t2 = x[1*i + 7];
  }
  ⋮
}
```

```
__kernel void do_stuff() {
  ⋮
  for (int i = 0; i < n ; ++i) {
    ⋮
    t1 = x[2*i + 5];
    ⋮
    t2 = x[3*i + 4];
    ⋮
  }
  ⋮
}
```

```
__kernel void do_stuff() {
  ⋮
  for (int i = 0; i < n ; ++i) {
    ⋮
    t1 = x[2*i + 6];
    ⋮
    t2 = x[4*i + 2];
    ⋮
  }
  ⋮
}
```

Figure 1: Overview of Genesis.

use studies of Genesis to generate programs for autotuning and compiler testing. These four types are found sufficient to express a large set of target programs of interest and, thus, we opt to simplify the language and limit variable types to one of the these four.

3.2. Genesis Constructs. Genesis provides several constructs for describing instance programs. Genesis constructs are designed to describe different code patterns, while keeping the Genesis program readable to the user. The appearance of a Genesis construct in a target program instructs the Genesis preprocessor to interpret it as part of a Genesis program and not to have it appear in the output instance program. Thus, these constructs must not conflict with reserved words and variables in the target program. We avoid such conflicts in two ways. First, we introduce an escape character (the backslash "\") that can be used to treat the construct as part of the target program and not as a Genesis construct. Second, Genesis constructs that conflict with common programming constructs (e.g., `if` and `for`) are named with a `gen` prefix, as will be described below.

The remainder of this section describes the Genesis constructs and illustrates them with examples. For simplicity, lines in code snippets beginning with `print` are generic print statements in some target language and are not specific to Genesis.

(i) The **`distribution`** construct specifies values and their corresponding probabilities. For example,

```
distribution a_dist={1{0.7};2{0.2};
4{0.1}}
```

defines a distribution named `a_dist` with values: 1, 2, and 4, each with the probability shown in the curly braces next to it. If the probabilities are omitted, a uniform distribution is used. It is also possible to define uniformly distributed ranges of values, for example,

```
distribution s_dist = {1:10}
```

By default, distributions can contain integers and strings. It is possible to use the modifier `real` to allow real distributions, as in the following example:

```
distribution s_dist = {1:10;;real}
```

This distribution, `s_dist`, allows real values between 1 and 10.

Distributions can be defined using Genesis values. For example,

```
value upperBound  sample {5:10}
distribution b_dist =
{1:${upperBound}}
```

defines a distribution named `b_dist` created with a bound using `upperBound`, a previously sampled Genesis value. Distributions are set once their definition line is processed and do not change during processing. Distributions defined using Genesis values do not change if the Genesis value changes later on in processing.

(ii) The **value** construct defines a Genesis entity whose value is sampled from a distribution. Values can be propagated as constants to the instance programs or can be used in the definition of other constructs. An example of a value line is

```
value stride  sample s_dist
```

This declares the Genesis entity with the given Genesis name stride, whose value is sampled from the distribution s_dist using the **sample** construct. Thus, assuming the s_dist is defined above, stride equally likely takes a value from 1 to 10 each time it is sampled. A reference to stride in a feature is replaced by the sampled value when the feature is processed. The assigned distribution can also be defined inline

```
value stride  sample {1:10}
```

This distribution is functionally the same as the previous example. This allows for a simpler declaration but removes the ability for reuse of the same distribution. Instead of sampling from a distribution, a Genesis value can also enumerate one. Thus,

```
value stride2  enumerate s_dist
```

makes stride2 take on every possible value of s_dist, one per instance program. A program instance that reaches an **enumerate** construct in processing is split into multiple program instances from that point onward, one for each value in the enumerated distribution. As a result, values sampled before an enumerate will be held constant across these programs, while values sampled after may differ across these programs. Alternatively, arrays can be used for multiple sampling:

```
value stride[5]  sample s_dist
```

This results in 5 strides, referred to in the code as stride[0], stride[1]... up to stride[4].

Lastly, values can be set without sampling from a distribution. For example,

```
value stride=2
```

declares the Genesis entity named stride and sets its value to 2, equivalent to a Genesis value sampling from a distribution containing only the value 2.

(iii) The **varlist** construct defines a pool of variables for use in a processed feature and hence is a part of the instance program. Along with the varlist construct, the created pool of variables itself is also called a *varlist*. A varlist is analogous to a distribution as entities which can be sampled from. An example of a varlist line is

```
varlist my_vars[5]
```

This defines a pool named my_vars of size 5. Five variables in the target language, named my_vars1 to my_vars5, can be sampled from this varlist using Genesis variables. The names of the variables in the varlist can be changed using a name modifier, as shown:

```
varlist my_vars[5]  name(temp)
```

The given Genesis name of the varlist remains my_vars, and this Genesis name is used to refer to this varlist. It contains 5 variables ranging from temp1 to temp5. It is possible to create a pool of variables using an existing varlist. For example,

```
varlist other_vars  from my_vars
```

defines another pool of variables named other_vars containing all the variables in the my_vars varlist. This allows manipulation of two separate varlists with the same set of my_vars variables. Varlists can be referenced with an argument to query information from the varlist. This includes the size of the varlist (using (size)), the name used for the variables in the varlist (using (name)), and a specific variable name for a variable in a varlist (using a number). For example,

```
value stride1  sample a_dist
varlist my_vars[5] name(foo)
```

The first varlist reference outputs 5, the varlist's size. The second reference outputs foo, the name used in all the variables in the varlist. The third reference outputs foo4, the specific name of the 4th variable in the varlist.

The section in which varlists are declared indicates the reinitialization rate of the varlists. Varlists declared in the global section are created once for the entire set of programs. The size of the Varlist and state of variables are maintained between instance programs in this case. Varlists declared in the program section are reinitialized at the point of its declaration, and thus, it returns to a full varlist with all its variables for each instance program. Varlists declared in a feature are local to that feature only and are reinitialized for each processing of the feature.

(iv) The **variable** construct defines a Genesis name whose value is sampled from a varlist and is propagated as a variable name to the target program. For example,

```
variable dest  from my_vars
```

defines a Genesis entity named dest. Its value is sampled from the previously defined varlist named my_vars. For each sample, the variable used in the instance program is a variable from my_vars1 to my_vars5. An occurrence of ${dest} in a feature is replaced by this variable name when the feature is processed.

(v) The **feature** construct defines a code snippet that is built up using Genesis names or possibly other features. For example,

```
feature computation
    variable dest,src1,src2 from
    my_vars
    ${dest}=${src1} * ${src2};
end
```

defines a feature named computation that has the code snippet ${dest} = ${src1} * ${src2};. The **variable** construct defines three Genesis variables sampled from my_vars. Thus, each time the feature computation is processed, the variables dest, src1, and src2 are sampled to select three variables from my_vars1 to my_vars5. The sampled values replace the corresponding variable references in the code snippet.

A feature is used in the template program or in other features. A feature is processed on demand for each feature reference. The resultant feature instance is substituted into that feature reference only, and each feature reference is substituted by a newly generated feature instance.

A code snippet spanning multiple lines returned by a feature can be condensed to a single line using a **singleline** modifier before the name of the feature. Multiple references to the same feature can be compacted by using square brackets. For example,

```
${computation[5]}
```

processes computation five times and replaces this reference with the five instances. A previously sampled Genesis value can be used instead of an integer.

Features can also be stored and represented by a Genesis name. In this case, features are explicitly processed and stored, and any reference to this Genesis name causes the already processed code to be substituted similar to a Genesis value or variable. For example,

```
feature stored_comp process
computation
```

processes a computation and stores the code snippet in stored_comp. Thus, when a reference to stored_comp is found, the code snippet previously processed is substituted, without any further sampling of its values and variables. Thus, using stored features allows a user to separate processing from replacement, allowing multiple replacements as necessary from a single processing of a feature.

Features can also have arguments, passed by value. For example,

```
feature access(offset)
    my_vars1 = arr[${offset}];
end
```

defines a feature called access, where offset is passed in, and its value is substituted into the code snippet in the same manner as a Genesis name. When storing a feature, the arguments must be supplied when the feature is processed.

(vi) The **generate** construct defines how many program instances to generate. For example,

```
generate 5
```

indicates that 5 instance programs should be generated. The generate construct allows the definition of global distributions:

```
generate 5 with
    a_dist={1{0.7};2{0.1};4{0.1};
    8{0.1}}
    b_dist={1:6}
end
```

(vii) The **genmath** construct allows the evaluation of expressions and updating of previously sampled values. Consider the following example:

```
value testValue sample {1:5}
...  ${testValue}";
genmath testValue = ${testValue}+5
...  ${testValue}";
```

This samples a testValue value from 1 to 5. After replacing the value in the following line, it increases the value of testValue by 5. The testValue reference in the last line is then replaced by the updated value.

(viii) The **add** and **remove** constructs modify a varlist in order to affect future samplings. For example,

```
variable dest1,src1,src2 from my_vars
${dest1} = ${src1} * ${src2};
remove dest1 from my_vars
variable dest2 from my_vars
${dest2} = ${src1} * ${src2};
add dest1 to my_vars
```

prevents dest1 and dest2 from sampling the same variable by removing dest1's sampled variable from the my_vars varlist before dest2 is sampled. The **add** readds dest1's sampled variable back to my_vars so that it can be selected by future samplings.

(ix) The **genif** construct is used for conditional generation of code snippets. Consider the following example:

```
value conditionValue sample {1:3}
```

```
genif ${conditionValue}==1
    ${computation}
end
```

The above code samples a value from 1 to 3 for `conditionValue`. If the value sampled is 1, then `computation` is processed and placed into the instance program. Otherwise, this section of the Genesis program is processed but produces no code as a result. The `genif` construct does not generate if statements in the instance program and is only used to control the flow through the preprocessor.

Using **`genelsif`** constructs after a **`genif`** statement allow for a second condition block that is only evaluated if the first **`genif`** statement is evaluated to be false. Also, **`genelse`** constructs allow a code section to be processed if all preceding **`genif`** and **`genelsif`** statements were evaluated to be false.

(x) The **`genloop`** construct facilitates repetitive generation. Consider the following example:

```
genloop loopvar:1:5
    ${access(${loopvar})}
end
```

This code produces 5 references to the feature `access`, each with a different value from 1 to 5 passed in as an argument. Note that this does not produce a loop in the instance program, but instead 5 consecutive versions of the code are produced when the `access` feature is processed.

The `genloop` construct can also test Boolean conditions, similar to a C while loop. Consider the following example:

```
genloop ${testValue} < 5
    genmath testValue = ${testValue}
    +1
    ··· ${testValue};
end
```

This repeatedly generates code snippets that reference `testValue`. During each iteration, `testValue` increases its value by one. This code stops processing when `testValue` is greater than 5 when the Boolean condition is checked at the beginning of the genloop iteration.

(xi) The **`geninclude`** construct allows Genesis code in another file to be used in the current Genesis program. Usually, this construct is used with premade library files provided with Genesis, which implement useful feature definitions that may be useful across multiple Genesis programs of the same target language. For example,

```
geninclude gen_c.glb
```

makes those features defined in `gen_c.glb` available, a library containing features that declare and initialize variables in C programs. For example, `varlistdeclare` is defined in `gen_c.glb`, which initializes C variables in an indicated varlist.

(xii) The **`genassert`** construct makes an assertion of a Boolean expression, similar to a `genif` statement. If the expression is evaluated to be true, processing of the instance program continues normally. However, if it is false, processing stops for the current instance program and the program is deleted. The preprocessor then continues processing the next instance program. For example,

```
value xCoord  sample {1:5}
value yCoord  sample {1:5}
genassert ${xCoord}*${yCoord}!=1
```

defines two Genesis values, sampled from 1 to 5. The genassert construct calculates the product and asserts that it is not 1 (i.e., 1 is not sampled for both values). If the product is 1, the generation of that instance program is aborted, and that program is not included in the final set of instance programs.

3.3. Genesis Processing Flow. There are three sections in a Genesis program: the *global* section, the *program* section, and the *feature definition* section. The global section contains Genesis constructs that are processed once for the entire set of generated program instances. The program section contains Genesis constructs that are processed once for every instance program. The feature definition section contains all the definitions of features. A feature is generally processed once each time the feature appears in the template program. Genesis names defined in the global and program sections can be used in any feature. However, names defined within a feature cannot be used outside that feature. This process is illustrated graphically in Figure 2.

When a Genesis program is read, the preprocessor begins with the global section, processing each statement sequentially. Once the end of the global section is reached, the generate statement is processed, creating multiple instance programs, each a copy of the template program. For each of those programs, the program section is sequentially processed. When this processing is complete, each instance program is scanned for feature references, and these features are processed as described earlier. Processing all these references results in the final, generated set of instance programs.

3.4. Genesis Sampling. The location of the declaration of a Genesis entity affects the duration for which the entity keeps its sampled value. This can be illustrated with the Genesis program shown in Program 1. In this example, `globalValue` is declared in the global section on line (3). Other Genesis values are declared in the program section on lines (7)–(9). `featureValue` is declared in a feature `varSet` on line (13). Each sampled entity is referenced inside `varSet` on lines (15)–(19), replaced with its sampled value when processed.

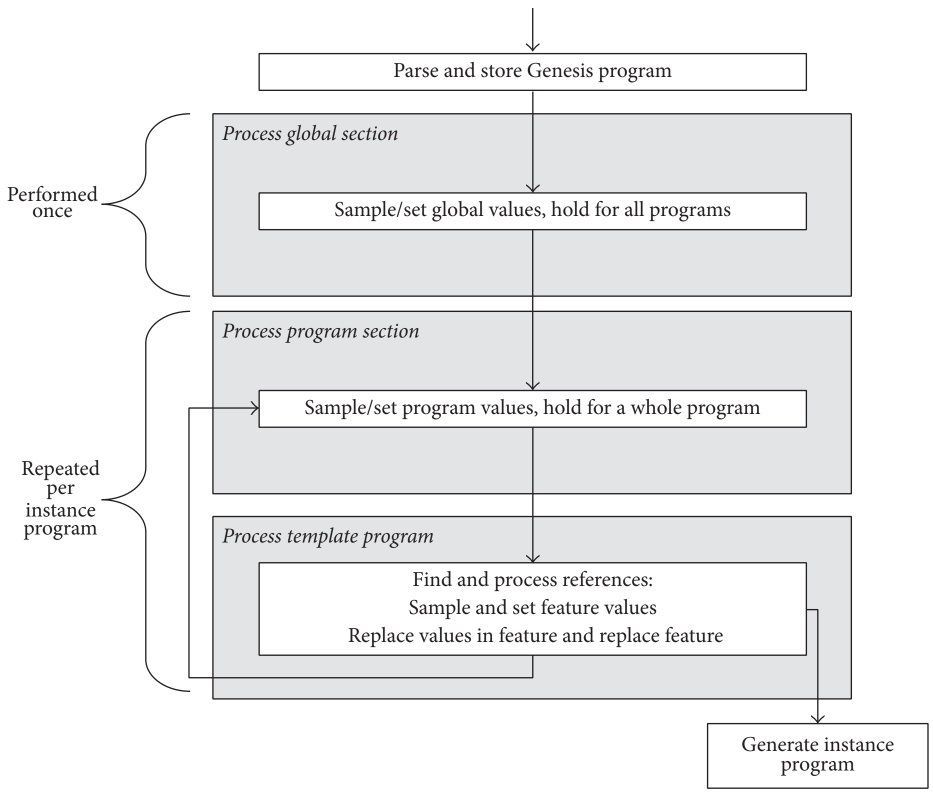

Parse and store Genesis program
Process global section
Sample/set global values, hold for all programs
Performed once
Process program section
Sample/set program values, hold for a whole program
Repeated per instance program
Process template program
Find and process references:
Sample and set feature values
Replace values in feature and replace feature
Generate instance program

FIGURE 2: Processing flow of sections in Genesis.

With the `generate 2` statement on line (22) and the value `enumerator` in the program section on line (8) enumerated through 2 values, 4 instance programs are generated in 2 sets of 2 programs each.

Thus, the global value `globalValue` is sampled once and held constant through all 4 instance programs. Next, `setValue` is sampled once per program set. While that value is held constant, `enumerator` generates two program sets. For each value of `enumerator`, `holdValue` is sampled independently for each set. Next, while processing the template program, `featureValue` is sampled once for each feature reference to `varSet`. Thus, `featureValue` can be different in each different feature reference within the same instance program.

3.5. Using `enumerate`. The `enumerate` construct breaks away from the notion of random sampling by allowing a Genesis `value` to take on each value in a distribution exactly once, one per instance program. When `enumerate` is used in the Genesis program, the number of generated programs by a `generate <number>` construct depends on the location of the enumerated value.

Enumerated values can be placed in either the global section or the program section, both of which affect the flow of Genesis differently. Figures 3 and 4 illustrate the difference between the two using a Genesis value enumerated through 3 values and a `generate 3` statement. When a value being enumerated is in the global section, as shown in Figure 3, the preprocessor first processes the value using `enumerate` before the `generate` construct, and the entity takes on all 3 possible values. When the global section finishes processing, the preprocessor reads the `generate` construct with each of the possible enumerated values. The preprocessor generates 3 programs with each possible value, creating a total of 9 programs. In this case, the preprocessor generates 9 total sets of programs, with each set having 1 instance program and with each enumerated value creating 3 sets.

When a value is being enumerated in the program section, as shown in Figure 4, the preprocessor processes the `generate` construct first, and the number in the `generate` statement determines the number of instance program sets to generate. For each instance program, the preprocessor processes the program section once, and thus, when the preprocessor processes the enumerated value for each instance program, it turns that instance program into a program set. Each program set contains a program for the 3 possible values in the enumeration, resulting in 3 total sets of 3 instance programs each.

Thus, the total number of programs generated is

$$N = E_G * G * E_P, \qquad (1)$$

where N is the total number of programs generated, E_G is the number of enumerated values a Genesis value can take in the global section, G is the number in the generate statement, and E_P is the number of enumerated values a Genesis name can take in the program section.

```
begin genesis
   global
      value globalValue sample sampleDist
   end

   program
     value setValue sample sampleDist
     value enumerator enumerate enumeratorDist
     value holdValue sample sampleDist
    end

    feature varSet
     value featureValue sample sampleDist
     #These will be outputted in each instance with Genesis names replaced
     #SAME ALWAYS:             ${globalValue}
     #SAME THROUGH SET:        ${setValue}
     #ENUMERATED:              ${enumerator}
     #DIFFERENT PER PROGRAM:   ${holdValue}
     #DIFFERENT IN PROGRAM:    ${featureValue}
     end

     generate 2 with sampleDist = {1:100}, enumeratorDist = {1:2}
 end genesis
```

PROGRAM 1: Example of sampling in Genesis sections.

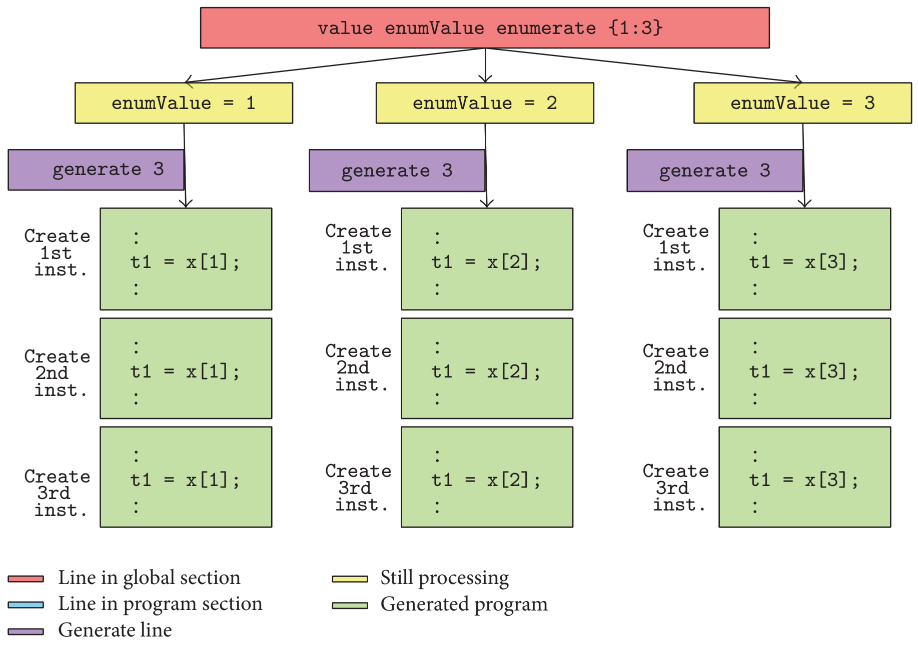


FIGURE 3: Effect of `enumerate` in global section.

Each program set need not contain the same number of programs when using enumerate. For example, for

```
distribution firstDist = {1:5}
value upperBound  sample firstDist
distribution dist2 = 1:${upperBound}
value enumValue  enumerate dist2
...
generate 5
```

the number of possible values `enumValue` is enumerated through is unknown until `upperBound` is sampled. If this code is in the global section, `upperBound` is set once, and each enumerated value of `enumValue` generates 5 programs. However, if these enumerated values are put in the program

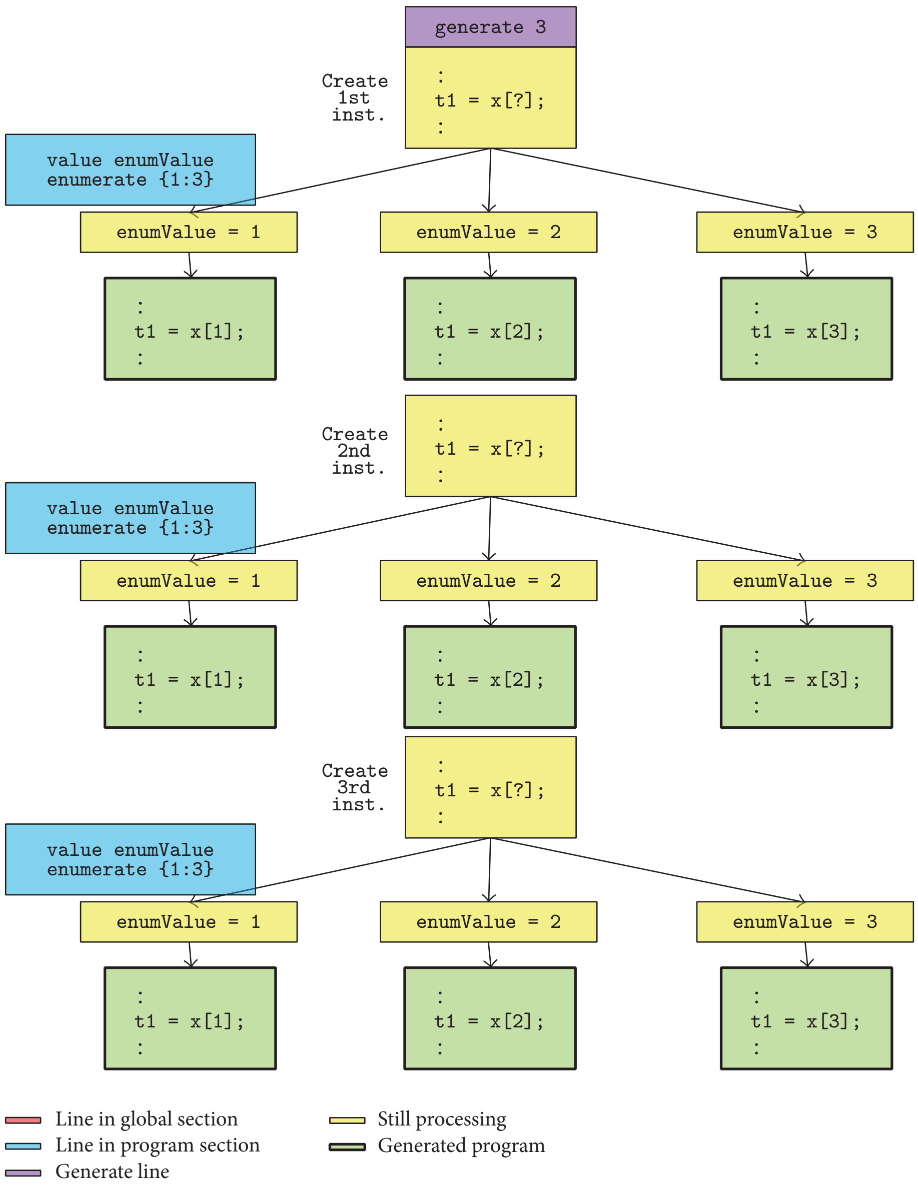


FIGURE 4: Effect of `enumerate` in program section.

section instead, 5 instance program sets are generated, with the number of programs in each set sampled independently. In these cases, the total number of programs generated is

$$N = E_{P1} + E_{P2} + \cdots + E_{Pn} \quad (n = E_G * G), \tag{2}$$

where N is the total number of programs generated, E_G is the number of enumerated values a Genesis value can take in the global section, G is the number in the generate statement, and E_{Pi} is the number of enumerated values a Genesis value can take in the program section during the ith iteration.

One can think of the `generate` construct as a special case of `enumerate` in which the enumerated values are unused. Thus, it is possible to generate the same set of programs using only the `enumerate` construct. Nonetheless, we opt to keep `generate` as "syntactic sugar" to simplify the common case where `enumerate` is not necessary.

4. Case Studies

In this section, we present five case studies to show the utility of Genesis in different application domains. The case studies demonstrate the Genesis language constructs, their use to hierarchically define and compose code segments to generate a rich set of synthetic codes, and the ease by which a Genesis program can be extended to modify the manner in which the code is generated.

4.1. Image Filtering. The first case study deals with the generation of image filtering applications for training in performance autotuning on GPUs. These applications typically have two perfectly nested loops that sweep over two-dimensional images. Each element of an output image is computed as a function of a subset of the pixels in an input image. Specific image filtering applications differ in the subset and the function used to compute the output.

This case study focuses on memory performance, which is affected by the number and pattern of image accesses and the pixel computations in the loop nest. Thus, we model the body of the loop nest as one or more read *epochs* followed by a write *epoch*, where an *epoch* is a sequence of computations followed by a memory access. We wish to generate a number of such programs where the number of read epochs, the number of computations per epoch, and the pattern of memory accesses all vary.

Program 2 shows the Genesis program used for this purpose. The Genesis program defines five features. The first describes a `computation`, which samples four different variables from the varlist `temp`. The code snippet in this feature computes a value using three of the sampled variables and assigns it to the variable sampled by `dest`.

The `read_access` feature describes a memory read that samples a destination variable and three values. The three values and the loop iterators (`it0` and `it1`) determine the array element to read, which is stored in the destination variable. Similarly, the `write_access` feature describes a memory write, where a variable will be stored in a memory location determined by the loop iterators, the inner trip count, and a sampled `offset` value.

With these building blocks, an `epoch` feature can be described. This feature consists of a number of computations followed by a read or write access. The value `numcomps` is sampled and, using a `genloop`, references the `computation` feature `numcomps` times, after the set of computations is either a `read_access` or a `write_access` depending on the value of the `epoch_type` argument.

The template program, shown in Program 3, is a skeletal OpenCL kernel that contains the loops that sweep over the image and reference `epochs`, a feature containing multiple references to the `epoch` feature. The template program also contains two references to features that are defined in the library `gen_c.glb`: `varlistdeclare`, which initializes variables in a varlist, and `keeplive`, which touches every element in a varlist to keep it live and writes to the supplied location. The end result is the creation of 1000 instance programs, each consisting of multiple read `epochs` and a write `epoch`. Each instance program contains a variable number of epochs, number of computations in each epoch, and pattern of memory accesses.

4.2. Static Program Characteristics. This case study is inspired by the work on cTuning with its MilepostGCC compiler [10], an autotuning compiler that extracts characteristics of a program [11, 12], and uses them with a machine learning model to tune programs for performance. Many of these characteristics come from low-level properties of a program's intermediate representation such as the number of basic blocks (BBs), the number of instructions per BB, the number of back edges, and the number of BBs with two successors. Thus, the goal of this case study is to use Genesis to generate a large number of programs with varying values of these characteristics as inputs to this tuning problem. We focus only on varying the type and number of instructions per BB, the number of BBs, and the number of successors to each BB. For presentation purposes, each BB has a series of instructions, namely, `sum`, `copy`, or `load-from-memory`, and ends with a `goto` to the next BB.

Program 4 shows the Genesis program that can be used to generate 1000 instance programs from the template program shown in Program 5. The instructions that can be sampled are described in features on lines (14)–(35). The `can_be_defined` varlist keeps a list of `temp` variables that are used in the instance program and can be sampled as `dest`. The `add` and `remove` constructs in the instruction features manipulate `can_be_defined` to ensure that no dead code will be produced. The instruction sampling is performed in the `singleinsn` feature on lines (37)–(46), where a random instruction type is chosen using a sampled value and multiple `genif` statements.

The above code can be easily extended to generate a set of programs where the number of BBs with two successors will vary. The Genesis program in Program 4 is augmented with the features in Program 6. Lines (1)–(10) describe the top block with two successors. The group number is passed in as an argument and used as a label on line (2). A number of instructions are created on line (4). Lines (5)–(9) are the code that gives this block two successors, where it can branch to one of the two blocks succeeding it. The condition on line (5) can be changed depending on the application.

Lines (12)–(18) describe a block with 1 successor. It follows a similar format to the block with two successors. An additional argument is passed in to determine which of the two successor blocks is being created. Thus, no if statements are needed before the goto statement on line (17) as was needed on line (5). Lines (20)–(27) describe the new `codebody` feature that replaces the one in Program 4. The number of blocks with two successors is sampled. That sampled value is used as a bound to a `genloop` statement, which creates many basic block groups. In each group, the top block is created on line (23), the bottom left block is created on line (24), and the bottom right block is created on line (25).

4.3. Stencil Code Generation and Optimization. This case study is rooted in autotuning of stencil computations on GPUs. We wish to create OpenCL kernels with a variety of stencil types and apply different optimizations, configured in different ways, to each kernel. Genesis can be used to independently accomplish each of these two goals, but what makes this example interesting is how both goals are accomplished simultaneously. In particular, changing the optimization parameters should not change the type of stencil, and, as such, while exploring a variety of optimizations, the stencil parameters must be held constant.

Stencil computations sweep through an array and for each element of that array they perform a set of reads at specific offsets from the element in question, they calculate a weighted sum of the read values, and they write the result to the corresponding element of an output array. The stencil parameters that are to be varied in this example are the number of spatial dimensions of the arrays, the number of elements in the stencil (size), how far each read

```
begin genesis

  geninclude gen_c.glb

  global
      distribution epochdist = {1:10}
      distribution numvardist = {8;16;32}
      distribution compdist = {1:20}
      distribution coefdist = {0:7}
      distribution offsdist = {0:15}
  end

  program
      value numepochs sample epochdist
      value numvars sample numvardist
      varlist temp[${numvars}]
  end

  feature computation
      variable dest,src1,src2,src3 from temp
      ${dest}  =  ${src1} * ${src2} + ${src3};
  end

  feature read_access
      variable dest from temp
      value coef1,coef2 sample coefdist
      value offs_r sample offsdist
      ${dest}  =  arr_in[${coef1} * it0 + ${coef2} * it1 + ${offs_r}];
  end

  feature write_access
      variable src from temp
      value offs_w sample offsdist
      arr_out[inner_tc*it0 + it1 +${offs_w}]=${src};
  end

  feature epoch (epoch_type)
      value numcomps sample compdist
      genloop i:1:${numcomps}
        ${computation}
      end
      genif ${epoch_type} eq "read"
        ${read_access}
      genelse
        ${write_access}
      end
  end

  feature epochs
      genloop i:1:${numepochs}
        ${epoch("read")}
      end
      ${epoch("write")}
  end

  generate 1000
end genesis
```

Program 2: Genesis program for image filtering.

```
void filter(unsigned int outer_tc, unsigned int inner_tc, global float *arr_in,
      global float *arr_out, global int *result){

   ${varlistDeclare(int, temp)}

   for (int it0 = get_local_id(0); it0 < outer_tc; it0 += get_local_size(0)){
     for (int it1 = get_local_id(1); it1 < inner_tc; it1 += get_local_size(1)){
       ${epochs}
     }
   }

   ${keepLive(*result, temp)}

}
```

Program 3: Template program for image filtering.

element can be from the center element (radius), and the weights. The optimization parameters that will be explored are the workgroup size and the number of workgroups, which control the division of work across GPU threads, as well as whether or not the kernel uses local memory, an on-chip cache that is shared across threads in a workgroup.

In the distribution definitions for this example, declared in the global section shown in Program 7, the first four distributions correspond to the properties of the stencil itself while the next five distributions relate to the optimization configurations.

The goal is to produce a variety of programs sampled from the first four distributions and to apply every combination of the values from the second set of distributions to each program. In order to do this, values taken from the first set of distributions use the `sample` construct, while those from the second set use the `enumerate` construct, as shown in the program section in Program 8. Hence, the first set of values will be kept constant in order to preserve the stencil parameters while the second set of values enumerate through all the optimization parameters.

In this way, the sampled values of `dim`, `size`, `radius`, and the various offsets and weights will remain constant while all combinations of the values for the other five parameters are generated. These values are then used in various features such as the `reads` feature shown in Program 9. The values for the offsets and weights will remain the same every time this feature is processed for a given base program, but depending on the value of `use_local`, a different final argument will be passed to the `read` feature thereby producing varying final code.

When the Genesis preprocessor is run with these inputs and, for example, a `generate 5` statement, it creates 360 instance programs consisting of 5 different base programs each with 72 different configurations. An example of two of the instance programs is shown in Program 10 and 11. In this case, both instance programs are from the same base program but in Program 10 local memory was not used while in Program 11 it was. As can be seen, despite their different optimizations, the version that uses local memory performs the same stencil calculation as the version that does not, albeit with some extra indirection. Note that, in this example, for brevity, only some of the Genesis code was shown.

4.4. Image Layering. This case study is motivated by face detection software [13] that use machine learning techniques to detect faces in images. A large set of images with faces of different sizes, shapes, and location within an image are needed to train a machine learning model. Genesis can be used to synthetically generate such images using a set of *face images* as building blocks. A *target* synthetic image can be generated by placing a variable number of face images in the target image at different positions and with different scale. The face images can be viewed as layers on the top of one another and on the top of a background target image. Thus, based on their location, the face images can partially occlude one another as faces are layered, with the top layer being the most visible.

Program 12 shows an example Genesis program that can be used for this purpose. The example assumes that each background image is a 1024×1024 pixel image but makes no assumptions on the face images used to overlay. The template program has a single line with a reference to the top-level feature `createImage`, indicating that the entire code should vary:

```
${createImage}
```

The distributions are laid out in the global section on lines (3)–(9) of Program 12. These distributions control the number of background images, the number of faces to overlay, the filename of the face image, the locations the face images are placed on the target image, and the size of the face image. The feature definition of `createImage` on lines (38)–(43) contains four lines: a reference to the `loadImage` feature, a value `numberFaces` determining the number of faces to load, a reference to the `overlayFace` feature (using the sampled value `numberFaces` to indicate how often faces are overlaid), and a reference to the `storeImage` feature. The three features referenced are for loading an image, placing a face onto an image, and storing an image, respectively.

Loading an image as a background (feature `loadImage` on lines (14)–(25)) is done by first sampling a value from

```
begin genesis
  global
    distribution insn_type_dist = {"sum","cp","ld"}
    distribution insns_dist = {1:20}
    distribution bb_dist = {2:5}
    distribution offs_dist = {0:7}
  end

  program
    varlist temp[5]
    varlist can_be_defined from temp
  end

  feature suminsn
    variable src1, src2 from temp
    add src1,src2 to can_be_defined
    variable dest from can_be_defined
    remove dest from can_be_defined
    ${dest} = ${src1} + ${src2};
  end

  feature cpinsn
    variable src from temp
    add src to can_be_defined
    variable dest from can_be_defined
    remove dest from can_be_defined
    ${dest} = ${src};
  end

  feature ldinsn
    value offs sample offs_dist
    variable dest from can_be_defined
    remove dest from can_be_defined
    ${dest} = arr[${offs}];
  end

  feature singleinsn
    value insntype sample insn_type_dist
    genif ${insntype} eq "sum"
      ${suminsn}
    genelsif ${insntype} eq "cp"
      ${cpinsn}
    genelsif ${insntype} eq "ld"
      ${ldinsn}
    end
  end

  feature codebody
    value numblocks sample bb_dist
    genloop loopvar:1:${numblocks}
      T${loopvar}:
      value numinsns sample insns_dist
      genloop insn:1:${numinsns}
        ${singleinsn}
      end
      genif ${loopvar}!=${numblocks}
        value dest = ${loopvar}+1
        ${gotoinsn(${dest})}
      end
    end
  end

  feature gotoinsn(dest)
```

Program 4: Continued.

```
        goto T${dest};
    end

    generate 1000
end genesis
```

Program 4: Genesis program for program characteristics.

```
void basic_block_code(float *arr){
    ${codebody}
}
```

Program 5: Template program for program characteristics.

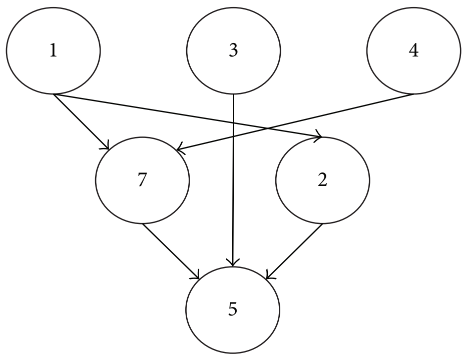


Figure 5: A simple MARE example. Equivalent task graph.

backgroundDist. Depending on the sampled value, the filename from which the background is loaded varies. The feature overlayFace on lines (26)–(33) is referenced multiple times in storeImage. This feature samples two locations, a height value and a width value. It also samples a size multiplier and a number to indicate which face to load. These values are then placed into an abstract place command and returned and replaced in storeImage. This feature is referenced multiple times to load and place multiple layered faces.

The abstract command to store the image to file, generated by feature storeImage on lines (35)–(37), is performed at the end of the generated commands. The feature is defined as a single resultant code snippet with no references and thus is the same across all instance programs. The definition requires no sampling, showing that features do not need varying parts if so desired. When the preprocessor reads the Genesis program and template program, it generates 1000 image layering instance programs as indicated by the generate statement.

Different output filename names can be realized by modifying the storeImage to keep a global counter value and use genmath to increment it after every reference. Using a value defined in the global section counter, the modified feature storeImage looks as follows:

```
feature storeImage

    genmath counter = ${counter}+1
    load outputFile to "output${counter}
    .jpg"

end
```

4.5. Task Graphs. This case study is motivated by studies on using Dynamic Voltage and Frequency Scaling (DVFS) to conserve energy in applications [14, 15]. In many of these studies, the applications are modelled as a *task graph* in which nodes represent computations and edges represent dependence among these computations. Given a task graph, computations not on the critical path are slowed down using DVFS to save energy (e.g., [15–18]). Often, the proposed techniques are sensitive to the structure and properties of the task graph. Thus, it is desirable to have a large set of task graphs that are diverse in their topology, task execution times, and dependence to better assess a proposed technique. Genesis provides a flexible and convenient way to generate such task graphs.

We express task graphs using the MARE programming model [19], which is used to express tasks and their dependence on Qualcomm SoC platforms. A MARE program consists of tasks, each of which must be created, have its dependencies on previous tasks expressed, and then be launched. This process is demonstrated in Program 13, which provides a snippet of MARE code used to realize the task graph shown in Figure 5.

Genesis is well suited for the task of generating synthetic MARE programs as it allows a user to easily create task graphs with varying depth, width, and connectivity. Program 14 shows an excerpt from a Genesis file used to produce such programs. On lines (1) and (4), the depth of the graph and the width of each layer are sampled from user-defined distributions. On line (10), the number of fan-in for a given node is sampled from another user-defined distribution.

Genesis also makes the problem of handling task dependency simple. As a level of the graph is built, its tasks are each represented as variables that are added to the varlist this_level (line (34) of Program 14). Once an entire level has been completed, that varlist is added to two other varlists (lines (38)–(43)), one tracking all tasks and one tracking those without fan-out (as any newly created tasks have no fan-out). When a new task is created, its fan-in can be chosen among all those tasks from previous levels by simply sampling from the varlist of all tasks (line (17)). By removing the sampled task from the no-fan-in varlist at this time (line (20)), we can

```
feature basicBlockWith2Successors(groupNumber)
    T${groupNumber}:
    value numinsns sample insns_dist
    ${singleinsn[numinsns]}
    if (dest1 > 0) {
       ${gotoinsn(${groupNumber}L)}
    }  else
       ${gotoinsn(${groupNumber}R)}
    }
end

feature basicBlockWith1Successor(groupNumber,type)
    T${loopValue}${type}:
    value numinsns sample insns_dist
    ${singleinsn[numinsns]}
    value dest = ${loopValue} + 1
    ${gotoinsn(${dest})}
end

feature codebody
    value numBlocksWith2Successors sample bb_dist
    genloop loopValue:1:${numBlocksWith2Successors}
       ${basicBlockWith2Successors(loopValue)}
       ${basicBlockWith1Successor(loopValue,L)}
       ${basicBlockWith1Successor(loopValue,R)}
    end
end
```

Program 6: Genesis feature creating BBs with sampled values.

```
global
    distribution dim_dist = {1:2}
    distribution size_dist = {1:9}
    distribution radius_dist = {0:5}
    distribution weight_dist = {1:3}

    distribution wg_size_y_dist = {1, 4, 16}
    distribution wg_size_x_dist = {1, 4, 16}
    distribution num_wgs_y_dist = {8, 16}
    distribution num_wgs_x_dist = {8, 16}
    distribution use_local_dist = {0, 1}
end
```

Program 7: Stencil code distributions.

also track which tasks have no fan-out. This allows for the creation of a join task at the end of the program which uses all remaining tasks with no fan-out to ensure the results of all tasks are used. The creation of this joining task is shown in Program 15.

5. Implementation

Genesis was implemented as a standalone preprocessor in Perl, and thus, Genesis is not limited to a specific target language. Using a scripting language such as Perl as opposed to a proper lexer and parser reduced development time while keeping the implementation flexible as the language evolved.

The preprocessor works in three phases. During the first phase of *file parsing*, the preprocessor reads a Genesis program and builds an internal representation of the constructs present. Each line is stored in a separate array based on its Genesis construct type, such as value or variable, and given a distinct ID. Each feature is stored in memory, with each Genesis line in that feature represented by the construct type and ID. The template program is also read and stored during this phase.

In the second phase of *instance generation*, the information stored is used to generate the desired number of instance programs. First, the global section is processed. Then, for each of the generated instance programs, the program section is

```
program
    value dims sample dim_dist
    value size sample size_dist
    value radius sample radius_dist
    value n_radius = −1∗${radius}
    distribution offset_dist = {${n_radius}:${radius}}
    value y_offset[${stencil_size}]
    value x_offset[${stencil_size}]
    value weight[${stencil_size}]
    genloop i:1:${stencil_size}
        genmath y_offset[${i}] sample offset_dist
        genmath x_offset[${i}] sample offset_dist
        genmath weight[${i}] sample weight_dist
    end

    value wg_size_y enumerate wg_size_y_dist
    value wg_size_x enumerate wg_size_x_dist
    value num_wgs_y enumerate num_wgs_y_dist
    value num_wgs_x enumerate num_wgs_x_dist
    value use_local enumerate use_local_dist
end
```

Program 8: Stencil code sampling and enumeration.

```
feature reads
  genloop i:1:${size}
    genif ${use_local} == 1
      ${read(${y_offset[${i}]},${x_offset[${i}]},${weight[${i}]},local_in)}
    genelse
      ${read(${y_offset[${i}]},${x_offset[${i}]},${weight[${i}]},input)}
    end
  end
end

feature read (y_offset,x_offset,weight,array)
  temp += ${weight} ∗ ${array}
  genif ${dims} == 2
    [y+${y_offset}]
  end
  [x+${x_offset}];
end
```

Program 9: Feature reads with offsets and weights.

processed and a copy of the template program is created. The code in each copy is scanned for any feature references as regular expressions. For each feature reference, the feature is processed and using similar regular expressions, the resulting code snippet replaces the reference. Random sampling of Genesis entities is done using the `rand()` function provided by Perl. Once all feature references in the template program are detected and replaced, the instance program is written to an output file in the third and final phase: *file output*. The last two phases are done iteratively to generate the set of instance programs.

The preprocessor can produce a comment block at the beginning of each generated file that includes the sampled values for each Genesis entity used to generate that instance program. Since Genesis is language agnostic, the user must provide a comment character when running Genesis to produce this comment block in each file. The preprocessor can also display statistical information, such as how often each value in a distribution is sampled for a Genesis entity across all instance programs. Using these values, the preprocessor can also output an analysis of the sampled values using Pearson's chi-squared test [20], which helps the user of Genesis determine the amount of deviation an actual set of sampled values has from its declared distribution.

In some cases, instance programs can fail to generate. For example, an instance program can fail to generate if a

```
__kernel void stencil(global double (*input), global double (*output), local double *
     local_in) {
  int x_gid = get_global_id(0);
  int x_lid = get_local_id(0);

  int x_base = x_gid / 4 * 16;
  int x_start = x_base + x_lid * 1;

  for (int x_block = x_start; x_block < x_base + 16; x_block+= 4) {
     int x = x_block;
     double temp = 0;
     temp += 1 * input[x+1];
     temp += 2 * input[x+0];
     output[x] = temp;
   }
}
```

Program 10: Example of generated stencil code. Instance program without local memory.

```
__kernel void stencil(__global double (*input), __global double (*output), __local double
     *local_in) {
  int x_gid = get_global_id(0);
  int x_lid = get_local_id(0);

  int x_base = x_gid / 4 * 16;
  int x_start = x_base + x_lid * 1;

  :
  //For brevity, the code that loads from global memory into local memory is omitted
  :

  for (int x_block = x_start; x_block < x_base + 16; x_block+=4) {
    int x = x_block;
    double temp = 0;
    temp += 1 * local_in[x-x_base+1+2];
    temp += 2 * local_in[x-x_base+0+2];
    output[x] = temp;
  }
}
```

Program 11: Example of generated stencil code. Instance program with local memory.

user attempts to sample from an empty varlist. When this happens, the program is not generated and that program instance number is skipped. The preprocessor then continues onto the next instance program. Our Perl preprocessor implementation reports the number of programs generated, the number of program sets, and which programs failed to generate.

Our implementation provides logging information to the terminal, at various levels of verbosity, controlled by the user. Further, it reports usage errors as well as errors that cause the generation of an instance program to fail. It also reports a host of warnings [21]. The implementation allows for the user to specify a naming scheme for the instance output programs: an output filename followed by a sequence number for each instance. The current implementation prototype does not allow for the target program to be split across multiple files. However, this is not a fundamental limitation of Genesis and can be incorporated into a future release.

6. Evaluation

In this section, we describe our evaluation of Genesis. We verify the correctness of our implementation using a large number of test programs [21]. In addition, we conduct an evaluation of the performance of the Perl preprocessor using the case studies of Section 4. We also assess the statistical quality of data sampling of Genesis values to demonstrate how faithful the sampled data is to the declared distributions.

We collect the runtime and sampling data by running Genesis programs and template programs through the

```
begin genesis

   global
      distribution backgroundDist = {1:3}
      distribution numFacesDist = {1:10}
      distribution facesDist = {1:1000}
      distribution locationDist = {0:1023}
      distribution sizeDist = {1:10}
   end

   program
   end

   feature loadImage
      value background sample backgroundDist
      value background_image
      genif ${background} == 1
         genmath background_image = "grass.jpg"
      genelsif ${background} == 2
         genmath background_image = "field.jpg"
      genelsif ${background} == 3
         genmath background_image = "house.jpg"
      end
      load "${background_image}" to outputFile
   end

   feature overlayFace
      value heightValue sample locationDist
      value widthValue sample locationDist
      value sizeValue sample sizeDist
      value face sample facesDist
      place facefile${face}.jpg at height ${heightValue} and width ${widthValue} with size
         ${sizeValue}x
   end

   feature storeImage
      store outputFile to "output.jpg"
   end

   feature createImage
      ${loadImage}
      value numberFaces sample numFacesDist
      ${overlayFace[${numberFaces}]}
      ${storeImage}
   end

   generate 1000
end genesis
```

Program 12: Image layering Genesis program.

preprocessor on an Intel Core i7-4930K CPU running at 3.40 GHz, with 32 GB of memory and running Perl 5.18.2.

6.1. Preprocessor Performance. We generate instance programs in powers of 10 from 10 to 100,000 using the Genesis and template programs of the case studies in Section 4 and measure the runtime of each run of the preprocessor. Each experiment is run 10 times and results are averaged. Figure 6 shows the runtime as a function of the number of generated instance programs (the number of generated instance programs for the stencil generation case study starts at 1000). The graph shows that runtime scales linearly with the number of generated instance programs in all cases. The image filtering, stencils, and program characteristics case studies contain nested `genloop` constructs, while the image layering Genesis programs contain a single `genloop` construct with fewer Genesis entities. Thus, the time to generate programs for the former group is an order of

```
auto task1 = mare::create_task(task_type_0);
task1->launch(array1);
auto task3 = mare::create_task(task_type_0);
task3->launch(array3);
auto task4 = mare::create_task(task_type_0);
task4->launch(array4);

auto task7 = mare::create_task(task_type_2);
task4->then(task7);
task1->then(task7);
task7->launch(array7,array4,array1);
auto task2 = mare::create_task(task_type_1);
task1->then(task2);
task2->launch(array2,array1);

auto task5 = mare::create_task(task_type_3);
task2->then(task5);
task3->then(task5);
task7->then(task5);
task5->launch(array5,array2,array7,array3);
task5->wait_for();
```

PROGRAM 13: A simple MARE example. MARE code (created using Genesis).

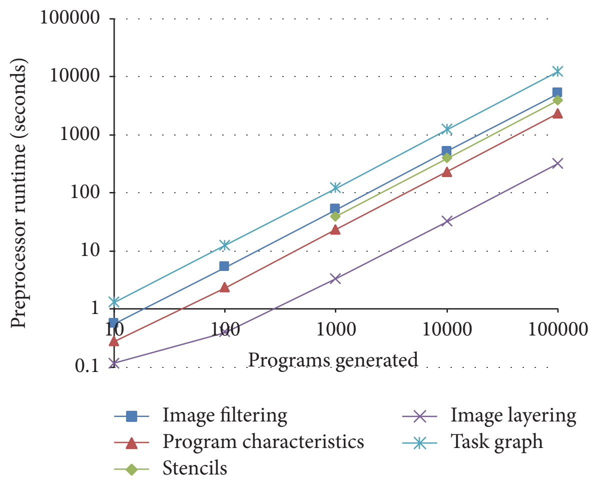


FIGURE 6: Runtimes of the preprocessor.

magnitude higher than the time for the latter. Nonetheless, even for large numbers of generated instance programs, the time remains in the tens of minutes, leading us to conclude that the time taken to generate programs is reasonable.

The time to generate programs can be broken down into three components: reading and parsing the Genesis program, generating instance programs, and writing instance programs to files. This breakdown is shown in Table 1 for the image filtering case study. Reading the Genesis program is done once for each invocation of the preprocessor, and thus the runtime in this phase remains constant and almost negligible. The other two phases grow linearly as the number of programs generated increases and constitute the bulk of the runtime with the instance program generation component dominating. However, this component is also the most amenable to parallelization since the generation of each instance is independent. Such a parallel approach is left to future work.

TABLE 1: Breakdown of program generation time for image filtering.

Programs	File parsing (s)	Instance gen. (s)	File output (s)
10	0.0013	0.52	0.0004
100	0.0013	5.20	0.0043
1000	0.0013	51.37	0.043
10000	0.0013	512.91	0.52
100000	0.0013	5134.51	6.36

6.2. Statistical Sampling. We evaluate the statistical quality of the sampled data using Pearson's chi-squared goodness of fit test [20]. The chi-squared (χ^2) test is an indicator of how well a sampled distribution differs from a declared distribution. A χ^2 value is calculated from the samples, where a higher resultant χ^2 value indicates a greater deviation from the declared distributions, and a lower value gives greater confidence that the sampling came from the desired distribution without bias.

A calculated χ^2 value can be converted to a P value, the probability of observing a sample statistic as extreme as that χ^2 value for many degrees of freedom. The degree of freedom is one less than the number of possible outcomes in a distribution [22]. A P value of 0.05 is commonly accepted as a threshold for significant deviance [22]; a sampling with a P value greater than 0.05 is considered reasonable while a sampling with a P value lower than 0.05 is expected to have some bias. Thus, a calculated χ^2 value can be compared to a

```
value depth sample depth_dist
genloop d:1:${depth}−1
    varlist this_level[0]
    value width sample width_dist
    genloop w:1:${width}
        variable new_task from tasks
        remove new_task from tasks

        genif ${d} != 1
            value num_fanin sample fanin_dist
            genif num_fanin > ${all_tasks(size)}
                genmath num_fanin = ${all_tasks(size)}
            end
            ${create_task(${new_task}, ${num_fanin})}
            varlist fanin[0] name(array)
            genloop i:1:${num_fanin}
                variable input_task from all_tasks
                add input_task to fanin
                remove input_task from all_tasks
                remove input_task from no_fanout_tasks
                ${express_dependence(${input_task}, ${new_task})}
            end
            value array_count
            genmath array_count = ${num_fanin} + 1

            /// Write the launch call (omitted for brevity)
            /// And add the fanin back to all_tasks
            ...
        genelse
            /// Create a node with no fanin (omitted for brevity)
            ...
        end

        add new_task to this_level
    end

    /// Add this_level to all_tasks and no_fanout_tasks
    genloop w:1:${width}
        variable transfer_task from this_level
        add transfer_task to all_tasks
        add transfer_task to no_fanout_tasks
        remove transfer_task from this_level
    end
end
```

PROGRAM 14: Varying depth, width, and connectivity.

critical value, defined as the χ^2 value that has a P value of 0.05 for a given degree of freedom. Thus, samplings that have χ^2 higher than the critical value have a P value lower than 0.05.

We report the results for our first case study with 1000 instance programs. Table 2 gives the seven Genesis values in the case study, their distributions, and their degrees of freedom. For a single run to generate the 1000 instance programs, the resulting χ^2 values and their corresponding P value are shown for the generated Genesis values. In all cases, the P value is much larger than 0.05, leading us to conclude that bias due to sampling is unlikely.

Tables 3, 4, 5, 6, and 7 show the same result over 10 runs of the generation of 1000 instance programs for each of the five case studies. The tables show the maximum and minimum χ^2s calculated across the 10 runs and the critical value of each distribution. For example, in the image filtering case study and over the 10 runs, 66 out of 70 χ^2 values, or 94.3%, are below their critical value, implying an unbiased sampling. Similarly, for the other case studies, almost all of runs result in χ^2 below the corresponding critical value. This leads us to conclude that the statistical quality of sampling from distributions is as expected.

```
variable last_task from tasks
remove last_task from tasks
value num_no_fanout = ${no_fanout_tasks(size)}
${create_task(${last_task}, ${num_no_fanout})}
add last_task to all_tasks
varlist last_fanin[0] name(array)
genloop t:1:${num_no_fanout}
   variable parent_task from no_fanout_tasks
   add parent_task to last_fanin
   remove parent_task from no_fanout_tasks
   ${express_dependence(${parent_task}, ${last_task})}
end
/// Write the launch call (omitted for brevity)
...
${last_task}->wait_for();
```

Program 15: Creation of a final task that depends on all tasks with no fan-out.

Table 2: Test results for one run of the image filtering Genesis program.

Variable	Distribution	Deg. of freedom	χ^2	P value
numEpochs	Uniform 1–10	9	7.26	0.61
numComps	Uniform 1–20	19	21.88	0.29
numVars	8, 16, 32	2	3.30	0.19
coef1	Uniform 0–7	7	3.57	0.83
coef2	Uniform 0–7	7	5.83	0.56
offs_r	Uniform 0–15	15	19.76	0.18
offs_w	Uniform 0–15	15	10.88	0.76

Table 3: Test results for value distributions for image filtering.

Variable	Distribution	Min χ^2	Max χ^2	Critical value	χ^2s over critical
numEpochs	Uniform 1–10	5.02	20.96	16.92	20.96
numComps	Uniform 1–20	8.17	35.05	30.14	35.05
numVars	8, 16, 32	0.18	3.30	5.99	None
coef1	Uniform 0–7	3.19	13.92	14.07	None
coef2	Uniform 0–7	3.07	13.86	14.07	None
offs_r	Uniform 0–15	8.54	20.32	25.00	None
offs_w	Uniform 0–15	8.99	31.20	25.00	29.98, 31.20

Table 4: Test results for value distributions for stencils.

Variable	Distribution	Min χ^2	Max χ^2	Critical value	χ^2s over critical
weight_cube	Uniform 1–10	1.94	10.47	16.92	None
weight_star (1)	Uniform 1–10	4.91	12.73	16.92	None
weight_star (2)	Uniform 1–10	3.82	14.91	16.92	None
weight_star (3)	Uniform 1–10	5.27	17.09	16.92	17.09
weight_diamond	Uniform 1–10	4.12	13.53	16.92	None
weight_thumbtack	Uniform 1–10	4.13	14.48	16.92	None
weight_no_corners	Uniform 1–10	6.00	14.85	16.92	None

Table 5: Test results for value distributions for static program characteristics.

Variable	Distribution	Min χ^2	Max χ^2	Critical value	χ^2s over critical
offs	Uniform 0–7	2.13	12.13	14.07	None
insntype	“sum”, “cp”, “ld”	0.00	4.53	5.99	None
numBlocks	Uniform 2–5	0.38	9.96	7.81	9.96
numInsns	Uniform 1–20	8.00	27.19	30.14	None

Table 6: Test results for value distributions for image layering.

Variable	Distribution	Min χ^2	Max χ^2	Critical value	χ^2s over critical
background	Uniform 1–3	0.09	5.70	5.99	None
heightValue	Uniform 0–1023	945.63	1087.02	1098.52	None
widthValue	Uniform 0–1023	968.11	1067.90	1098.52	None
sizeValue	Uniform 1–10	2.34	17.11	16.92	17.11
face	Uniform 1–1000	937.60	1031.54	1073.64	None
numberFaces	Uniform 1–10	3.82	19.10	16.92	19.10

Table 7: Test results for value distributions for task graphs.

Variable	Distribution	Min χ^2	Max χ^2	Critical value	χ^2s over critical
depth	Uniform 3–5	0.01	4.90	5.99	None
width	Uniform 2–5	1.07	12.12	7.81	12.12
num_fanin	Uniform 1–3	0.22	6.20	5.99	6.20

7. Related Work

Our work related to program generators. CSmith [4] is a tool to generate C programs and is used to find bugs in compilers through stress testing. The generated programs are not fully described by the user and are generally random. CodeSmith Generator [5] creates visual basic code using templates. However, it does not provide sampling like Genesis and, consequently, does not generate multiple similar versions of a program with different characteristics. TestMake [6] generates test harnesses for programs. In contrast, Genesis generates whole programs that vary in their characteristics.

Christen et al. [23] describe a domain-specific language for describing stencil codes and optimizations that can be applied to them. The language is used in Patus, which is an autotuning framework for stencils. Patus uses the program description to generate stencils optimized in different ways for use in their heuristic search for good performing code. Thus, to some extent, our work bears resemblance to theirs. Nonetheless, Genesis is not limited to stencils, although it has been used to describe stencils and their optimizations in a case study. Further, unlike Genesis, the Patus language does not control the random distribution of optimizations parameters.

Voronenko et al. [8] automate the generation of vectorized and multithreaded linear transform libraries, providing users with optimized code for this domain of applications. Similar to Patus, the specific domain of this work is in contrast to Genesis, which can be used in any domain.

Bazzichi and Spadafora [24] create an automatic generator for compiler testing that produces a set of programs covering the grammatical constructions of a context-free grammar language. However, it does not give the user control over the programs generated beyond selecting a random seed.

Kamin et al. [25] created Jumbo, which generates code for Java during the actual running of the program. Poletto et al. [26, 27] have also added language and compiler support to generate code during runtime. In contrast, Genesis generates code but does it during compilation and not runtime. Genesis also generates multiple programs when it is run taken from statistical samples instead of runtime information.

Genesis uses variables whose values are randomly sampled in order to customize generated programs based on given distributions. Hardware description languages, such as Verilog [28] and SystemVerilog [29], also use randomly generated values for variables. For example, the `rand` keyword in the declaration of a variable in a Verilog program randomly assigns the variable of a value from a specified range with a given distribution. However, unlike Genesis, these variables are used to randomly vary inputs and signals for the purpose of generating test vectors for hardware verification.

Our work also relates to other approaches that describe programs, such as Program Description Language [30], and approaches that customize programs, including lexical [31] and syntactic [32–34] preprocessors. In contrast to all these works, Genesis describes and generates multiple programs whose code is customized using user-specified statistical distributions.

The work presented here extends the authors' initial presentation of Genesis [35] through more detailed description of the constructs and the processing flow of the language, the use of new case studies, and expansion of the experimental evaluation.

8. Conclusion and Future Work

We presented Genesis, a language to express and generate statistically controlled program sets for use in multiple domains and applications. It differs from previous preprocessors by providing the unique ability to sample from distributions. It is not restricted to a specific output language and is also flexible enough to express sets of programs with varying lengths and characteristics. We presented five case studies in different domains to illustrate the utility of Genesis and its ability to easily express programs with different characteristics. We designed and implemented a prototype preprocessor for Genesis, which is released into the public domain as an open source artifact (https://github.com/chiualto/genesis). We evaluated the preprocessor's performance and demonstrated the statistical quality of the samples it generates. We believe that Genesis is a useful tool that eases the expression and creation of large and diverse program sets, which can provide large benefits for its users.

This work can be extended in several directions. More case studies can be used to assess if there is a need to extend the Genesis constructs to increase functionality or usability. The language itself can be extended, for example, by adding return values for features. The efficiency and memory footprint of the preprocessor can be improved, in particular via the parallelization of the program instance generation phase. It may also be beneficial to migrate the preprocessor into a compiler. Finally, language-specific features may be introduced. For example, if the instance programs being generated are known to be written in OpenCL, it might be possible to generate the host program to allow the user to run the programs and get runtime information directly after using Genesis.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was funded by research grants from NSERC and Qualcomm.

References

[1] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[2] C. Bienia, *Benchmarking modern multiprocessors [Ph.D. dissertation]*, Princeton University, 2011.

[3] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pp. 24–36, S. Margherita Ligure, Italy, June 1995.

[4] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, pp. 283–294, San Jose, Calif, USA, June 2011.

[5] CodeSmith Tools LLC, CodeSmith Generator, http://www.codesmithtools.com/product/generator.

[6] A. Markus, "Generating test programs with TestMake," in *Proceedings of the Second European Tcl/Tk User Meeting*, pp. 127–138, TU Hamburg-Harburg, June 2001, http://flibs.sourceforge.net/article_testmake.pdf.

[7] J. Schimmel, K. Molitorisz, A. Jannesari, and W. F. Tichy, "Automatic generation of parallel unit tests," in *Proceedings of the 8th International Workshop on Automation of Software Test (AST '13)*, pp. 40–46, IEEE, San Francisco, Calif, USA, May 2013.

[8] Y. Voronenko, F. De Mesmay, and M. Püschel, "Computer generation of general size linear transform libraries," in *Proceedings of the 7th International Symposium on Code Generation and Optimization (CGO '09)*, pp. 102–113, Seattle, Wash, USA, April 2009.

[9] M. Gabbrielli and S. Martini, *Programming Languages: Principles and Paradigms*, Springer, 1st edition, 2010.

[10] G. Fursin, Y. Kashnikov, A. W. Memon et al., "Milepost GCC: machine learning enabled self-tuning compiler," *International Journal of Parallel Programming*, vol. 39, no. 3, pp. 296–327, 2011.

[11] cTuning.org, "Static Features available in MILEPOST GCC V2.1," http://ctuning.org/wiki/index.php/CTools:MilepostGCC:StaticFeatures:MILEPOST_V2.1.

[12] Y. Chen, S. Fang, Y. Huang et al., "Deconstructing iterative optimization," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 3, pp. 21:1–21:30, 2012.

[13] OpenCV Dev Team, Face recognition with OpenCV, 2014.

[14] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power optimization of variable-voltage core-based systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 12, pp. 1702–1714, 2006.

[15] P. Chowdhury and C. Chakrabarti, "Static task-scheduling algorithms for battery-powered DVS systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 2, pp. 226–237, 2005.

[16] A. Andrei, P. Eles, Z. Peng, M. Schmitz, and B. Al-Hashimi, "Voltage selection for timeconstrained multiprocessor systems," in *Designing Embedded Processors—A Low Power Perspective*, Springer, Berlin, Germany, 2007.

[17] M. Ruggiero, D. Bertozzi, L. Benini, M. Milano, and A. Andrei, "Reducing the abstraction and optimality gaps in the allocation and scheduling for variable voltage/frequency MPSoC platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 3, pp. 378–391, 2009.

[18] A. Benoit, R. Melhem, P. Renaud-Goud, and Y. Robert, "Assessing the performance of energy-aware mappings," *Parallel Processing Letters*, vol. 23, no. 2, Article ID 1340003, 17 pages, 2013.

[19] Qualcomm Technologies, Inc, Multicore asynchronous rutime environment: Documentation and interface specification, 2013.

[20] R. L. Plackett, "Karl Pearson and the chi-squared test," *International Statistical Review*, vol. 51, no. 1, pp. 59–72, 1983.

[21] A. Chiu, *Genesis: a language for generating synthetic programs [M.S. thesis]*, Department of Electrical and Computer Engineering, University of Toronto, 2015.

[22] G. W. Snedecor and W. G. Cochran, *Statistical Methods*, Iowa State University Press, 8th edition, 1989.

[23] M. Christen, O. Schenk, and H. Burkhart, "A code generation and auto-tuning framework for parallel stencil computations," in *Proceedings of the Cetus Users and Compiler Infrastructure Workshop*, October 2011.

[24] F. Bazzichi and I. Spadafora, "An automatic generator for compiler testing," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, pp. 343–353, 1982.

[25] S. Kamin, L. Clausen, and A. Jarvis, "Jumbo: run-time code generation for Java and its applications," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*, pp. 48–56, IEEE, San Francisco, Calif, USA, March 2003.

[26] M. Poletto, *Language and compiler support for dynamic code generation [Ph.D. thesis]*, Massachusetts Institute of Technology, 1999.

[27] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek, "'C and tcc: a language and compiler for dynamic code generation," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 2, pp. 324–369, 1999.

[28] S. Palnitkar, *Verilog R Hdl: A Guide to Digital Design and Synthesis*, Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd edition, 2003.

[29] C. B. Spear, *Systemverilog for Verification: A Guide to Learning the Testbench Language Features*, Springer, 2nd edition, 2010.

[30] S. Caine and E. Gordon, "PDL: a tool for software design," in *Proceedings of the National Computer Conference and Expo*, pp. 271–276, 1975.

[31] W. Turski, "Software engineering-some principles and problems," in *Programming Methodology by David Gries*, pp. 29–36, Springer, New York, NY, USA, 1978.

[32] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rmy, and J. Vouillon, The objective Caml system release 3.12, 2010, http://caml.inria.fr.

[33] D. Rémy and J. Vouillon, "Objective ML: an effective object-oriented extension to ML," *Theory and Practice of Object Systems*, vol. 4, no. 1, pp. 27–50, 1998.

[34] G. Steele, *Common Lisp the Language*, Digital Press, 2nd edition, 1990.

[35] A. Chiu, J. Garvey, and T. S. Abdelrahman, "Genesis: a language for generating synthetic training programs for machine learning," in *Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15)*, Ischia, Italy, May 2015.