

# Expression and Loop Libraries for High-Performance Code Synthesis

Christopher Mueller and Andrew Lumsdaine  
{chemuell,lums}@osl.iu.edu

Open Systems Laboratory  
Indiana University  
Bloomington, IN 47405

**Abstract.** To simultaneously provide rapid application development and high performance, developers of scientific and multimedia applications often mix languages, using scripting languages to glue together high-performance components written in compiled languages. While this can be a useful development strategy, it distances application developers from the optimization process while adding complexity to the development tool chain. Recently, we introduced the *Synthetic Programming Environment (SPE)* for Python, a library for generating and executing machine instructions at run-time. The SPE is an effective platform for optimizing serial and parallel applications without relying on intermediate languages. In this paper, we use the SPE to develop two code generation libraries, one for scalar and vector (SIMD) expression evaluation and another for parallel and high-performance loop generation. Using these libraries, we implement two high performance kernels and demonstrate how to achieve high levels of performance by providing the appropriate abstractions to users. Our results show that scripting languages can be used to generate high-performance code and suggest that providing optimizations as user-level libraries is an effective strategy for managing the complexity of high-performance applications.

## 1 Introduction

Scripting languages have become common tools for developing many types of applications, including high-performance scientific and multimedia applications. To achieve high performance, developers often use a lower-level language such as C, C++, or FORTRAN for the performance-critical sections. While effective, this approach complicates the development process by adding new dependencies into the tool chain and requiring additional developer skills. Recently, we introduced the Synthetic Programming Environment (SPE) for Python [9]. The SPE enables the run-time synthesis of machine instructions directly from Python, without requiring an intermediate language. It exposes the full processor instruction set as a library of Python functions that can be used to construct new instruction sequences at run time and provides a library for synchronous and asynchronous execution of the generated sequences.

The SPE provides the infrastructure for generating high-performance code at run time. In this paper, we introduce two meta-programming libraries for managing the complexity of generating instruction sequences for serial and parallel applications. The first library provides abstractions for generating instruction streams for arithmetic and logical expressions on scalar and vector (SIMD) data types. The second library abstracts loop generation and provides a flexible set of tools for generating sequential and parallel loops. With these libraries, it is possible to generate high-performance computational kernels using a natural syntax, while maintaining user-level control over the final optimizations.

In the next section we introduce *synthetic programming* and provide the context for our contributions. Following that, we describe the new libraries, starting with the scalar and vector Expression library and proceeding through the Iterator library. We detail code generation techniques used by each component and illustrate their use with two examples. We conclude with a review of related techniques. A basic knowledge of Python syntax is assumed for the discussion, but more complex techniques will be described as they are introduced.

## 2 Synthetic Programming

Synthetic programming is the process of developing programs composed of computational kernels synthesized at run time using the SPE. The computational kernels, or *synthetic programs*, are generated with meta-programming routines called *synthetic components*. By using synthetic components to generate synthetic programs from a high-level language, developers can create high-performance applications without sacrificing the productivity gained from using a high-level language.

Synthetic programs are developed using three components supplied by the Synthetic Programming Environment: `ISA`, `Processor`, and `InstructionStream`. `ISA` components are collections of functions that generate binary coded machine instructions for a particular instruction set architecture. For instance, in the PowerPC ISA, the function `addi(D, A, SIMM)` generates the `addi`, or *add immediate*, machine instruction for adding a constant `SIMM` to the value in register `A`, storing the result in register `D`. A synthetic program is built by adding a sequence of instructions to an instance of `InstructionStream`. For example, the following code generates the synthetic program for the computation  $r_{return} = (0+31)+11$ :

```
c = InstructionStream()
c.add(ppc.addi(gp_return, 0, 31))
c.add(ppc.addi(gp_return, gp_return, 11))
```

`gp_return` is a constant that specifies the register for integer return values. In addition to managing the user-generated instructions, `InstructionStream` also provides a basic register allocator that warns developers of register pressure.

Prior to execution, `InstructionStream` generates an ABI (application binary interface)-compliant prologue and epilogue for the synthetic program, making it a valid “function” for the current execution environment. When the sequence is ready, it is executed by a `Processor` instance:

```
proc = Processor()
result = proc.execute(c)
print result
--> 42
```

`Processor` can execute synthetic programs synchronously, blocking until completion, or asynchronously in their own threads, returning immediately.

The current implementation supports the PowerPC (scalar) and AltiVec (vector) ISAs and runs on Apple's OS X. Data is passed between the host program and the synthetic program using pointers to memory, such as native Python or Numeric Python `arrays`, and values can be passed to synthetic functions using registers following the ABI conventions. Load and store instructions move data between memory and registers, and loops and conditional code are synthesized using the branch and compare instructions.

### 3 The Expression Library

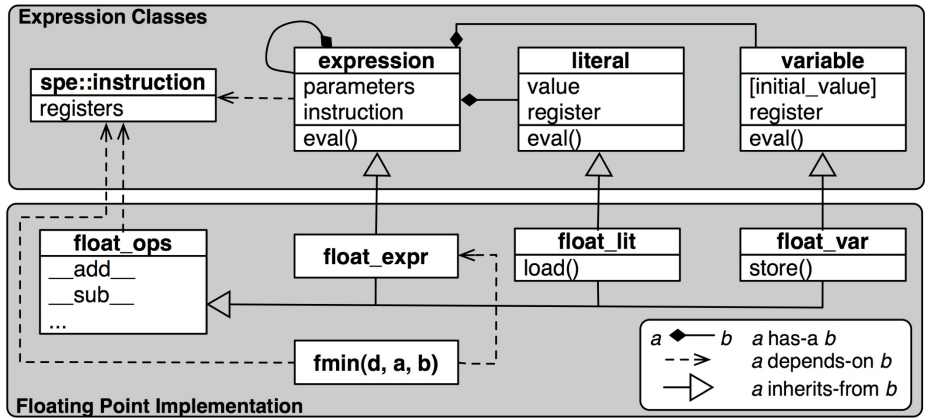
At the lowest level, all machine instructions operate on values stored in registers or on constant values encoded directly into the instruction. In a basic synthetic program, the developer refers to registers directly and explicitly manages movement of data between the processor and memory system. Arithmetic is performed by generating a sequence of instructions that operate on two or three values at a time, and it is up to the developer to ensure complex expressions are evaluated properly. Using the PowerPC ISA, the expression  $a = a*b + a*c$  could be written as:

```
ppc.mullwx(t1, a, c) # a, b, c are registers
ppc.mullwx(t2, a, b) # t1, t2 are temp registers
ppc.addx(a, t1, t2)
```

While this expression was simple to convert to an instruction sequence, in a more complex expression, simple operator precedence rules and register reuse policies are difficult to enforce, leading to code that is difficult to debug and maintain.

Because expression evaluation is at the heart of most performance-critical code sections, the Expression library introduces a set of objects for managing expression evaluation for scalar and vector data types.

The main objects are illustrated in Figure 1. The base classes, `variable`, `literal`, and `expression` implement the Interpreter design pattern and manage register allocation, initialization, and expression evaluation. Python's underlying expression evaluation engine handles operator precedence. The base classes are typeless and type-specific subclasses generate instructions as the expressions are evaluated. In the diagram, the floating point class hierarchy shows how the subclasses share operator implementations, ensuring that floating point expressions are evaluated consistently. Free functions that return `expression` instances allow developers to expose operations that do not map directly to operators. These functions can be used directly in larger expression statements and may abstract sequences of instructions. This is useful for binary operators such as `fmin`, which is actually a three instruction sequence, and various ternary vector instructions.



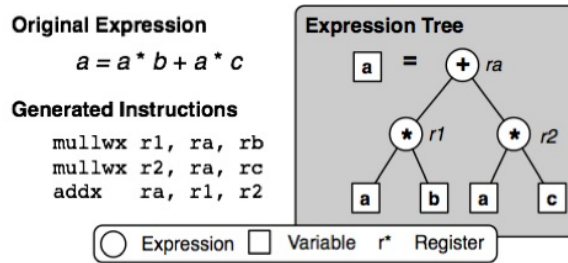
**Fig. 1.** Expression Class Hierarchy and Floating Point Implementation. The expression classes implement the Interpreter pattern for run-time code synthesis. The base classes manage evaluation and type-specific subclasses provide methods to generate code for specific operations.

To illustrate how the Interpreter classes work, consider the example in Figure 2 for evaluating the expression  $a = a * b + a * c$ .  $a$ ,  $b$ , and  $c$  are instances of `variable` with the `*` and `+` operators overloaded. When an operator is evaluated by Python, the overloaded implementation delays evaluation and returns an instance of `expression` that contains references to the operator and operands. In this example, the operands for `+` are `expressions` and the operands for `*` are `variables`. The root of the parse tree is the `+` `expression`.

Up to this point, no code has been generated and registers have not been allocated. All Interpreter classes have an `eval` method. Subclasses override `eval` to generate the appropriate instructions for their data type and registers. When the `expression` tree is assigned to a `variable`, the `variable` triggers the expression evaluation, which in turn generates the instruction sequence to evaluate the expression using a depth-first traversal of the tree. Each `expression` has a register associated with its results that lives for the lifetime of the expression evaluation. After the assignment, the tree is traversed again to free up temporary resources. Note that the final `expression` uses the register from the assigned `variable`. The `variable` passes this to the `expression`'s `eval` method.

Expression evaluation is the same for scalar and vector variables of all types, but expressions cannot contain mixed type variables. As the expression tree is created, the variables are checked against each other and other expressions in the tree. An exception is thrown if there is a type error. Literals that appear in the expression are transformed into the appropriate subclass of `literal`.

`variable` instances can be used in expressions or directly with ISA functions. The attribute `variable.reg` contains the register assigned to the variable and is a valid argument anywhere a register is accepted. When the variable is no longer needed, registers are freed with a call to `variable.release_registers()`.



**Fig. 2.** Expression Evaluation. As Python evaluates the expression, variables return instances of expression objects that correspond to the operator. When the final tree is assigned to a variable, it is recursively evaluated, generating the instruction sequence for the expression.

### 3.1 Scalar Types

Two scalar variable types are supplied in the Expression library. `int_var` and `float_var` implement signed integer and double-precision floating point operations, respectively. `int_var` supports  $(+, -, *, /, <<, >>, \&, |, \wedge)$  and `float_var` supports the standard arithmetic operations  $(+, -, *, /)$ .

Scalar variables are initialized using different instruction sequences, depending on the initial value. Integers use an *add immediate* instruction if the value fits into sixteen bits, otherwise they use an *add immediate* followed by a *shift/add* to load the full 32-bit value. Double-precision floating point values are loaded from an `array` backing store that contains the value as its only element. The floating point load instruction loads the value from the backing store into the register. Because the load instruction is expensive, floating point literals should be initialized outside of loops.

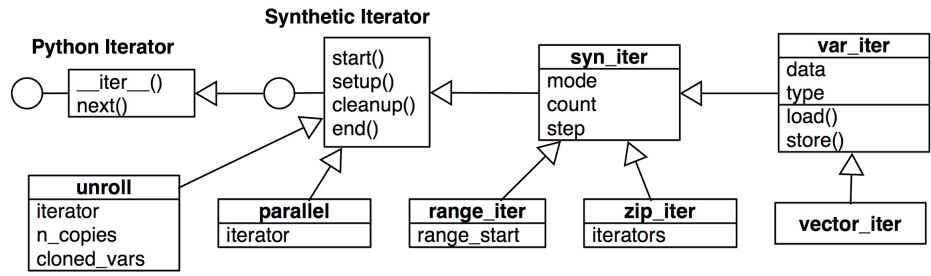
### 3.2 Vector Types

Vector data types support single-instruction, multiple data (SIMD) parallelism using the AltiVec instruction set. AltiVec vectors can be partitioned into a number of types, including signed and unsigned bytes, shorts, and integers and single-precision floating point numbers. The integer types support  $(+, -, <<, >>, \&, |, \wedge)$  and the floating point types support  $(+, -, /)$ . In addition, most of the AltiVec instruction set is exposed as expression functions that are type-aware. For example, the integer vector min operation has six forms, one for each integer type, `vmin[s][b,h,w]`. The type-aware expression function `vmin` abstracts the typed versions and allows the developer to use it as part of an expression. For instance,

```
z.v = vmin(x, y) + 4
```

adds 4 to the minimum value from the element pairs in  $x$  and  $y$ .

Vectors are initialized from a backing store in the same way as floating point scalars. Users can also pass in an array containing the initial values. This array will continue to be used as the backing store when the `store()` method is called to save a vector back to memory.



**Fig. 3.** The Iterator Hierarchy. The synthetic iterator protocol extends Python’s iterator protocol to provide additional hooks into loop evaluation. `syn_iter` generates code to manage loops using the counter register or general purpose registers, as selected by the user. The subclasses override methods to implement iterator specific functionality. `unroll` and `parallel` are Proxies that restructure synthetic iterators for high-performance code generation.

### 4 The Iterator Library

Variables and expressions provide the foundation for developing syntactically clear high-performance synthetic programs. Synthetic iterators supply the abstractions for processing collections of variables and vectors in single or multi-threaded environments. In the same way expressions use features of Python as meta-programming constructs, the synthetic iterators are based on Python’s underlying iterator protocol.

Python iterators are objects that manage element-wise iteration over sequences. The iterator protocol requires two methods, `__iter__()` and `next()`. `__iter__()` returns an iterator for the object, and `next()` is called to retrieve each element, raising `StopIteration` when the sequence is exhausted. All sequence iteration in Python is handled by iterators, and Python `for` loops expect iterators as their sequence parameter.

Synthetic iterators use the Python iterator protocol to provide a concise representation for the generation of instruction sequences for implementing machine-level loops. The synthetic iterator hierarchy is shown in Figure 3. The synthetic iterator protocol extends the Python iterator protocol with hooks for managing different aspects of code generation.

All synthetic iterators work in same basic way. To demonstrate, consider the `syn_range` iterator:

<pre> Loop code:  sum = var(c, 0) rng = syn_range(c, 20)  for i in rng:     sum.v = sum + i </pre>	<pre> Call sequence:  __iter__(): start() 1st next(): setup() ... body ... 2nd next(): cleanup() end() raise StopIteration </pre>
--	---

In this example, `syn_range` generates integers from 0 to 19 and makes them available in the variable `i`. The pseudo-code on the above shows the Python and synthetic iterator protocols and their order of evaluation.

When the iterator is created by the `for` loop, `start()` acquires the registers, generates initialization code and returns a reference to the iterator. On the *first* iteration of the `for` loop, i.e. when `next()` is called the first time, the `setup()` generates the loop prologue, initializing the loop counter and branch label, and returns a `variable` for the current value. Then the loop body is executed, and the expression code is added to the instruction stream. Note that any instructions from the ISA can be used here, not just expressions. On the *second* iteration of the `for` loop, `cleanup()` generates the loop epilogue with the branch instruction and any loop cleanup code. It also resets the counters, in case the loop is nested. Finally, `end()` frees loop registers and other resources. While still at the beginning of the second iteration, the iterator raises `StopIteration`, ending the Python loop.

In the next few sections, we detail the design and implementation of the iterator hierarchy.

#### 4.1 `syn_iter`

`syn_iter` handles the mechanics of sequential loop generation and is the base class for most other iterators. It supports three different modes of iteration: counter based (CTR), register decrement (DEC), and register increment (INC). `syn_iter`'s constructor takes the iteration count, step size, and mode. The generated loop performs  $(count \div step)$  iterations of the loop body.

CTR iterators generate the most efficient loops. CTR loops use the PowerPC `ctr` register to hold the loop count and the `bdnz` (decrement counter, branch if non-zero) instruction to branch. CTR iterators do not require any general purpose registers, but only one CTR loop can be active at any time. The iterator variable (e.g., `i` in the above example), is set to `None` for CTR loops.

DEC iterators work in a similar manner as CTR iterators, decrementing a value and terminating the loop when the value reaches zero. However, DEC iterators keep their counter in a general purpose register, making it available as the iterator variable for the current loop.

INC iterators are the opposite of DEC iterators, starting the counter at zero and looping until the counter reaches the stop value. This requires two registers, one for the counter and one for the stop value. The current loop count is available as the loop variable.

In all modes, `syn_iter` can be directed to excluded the branch instruction, allowing more complicated iterators to have fine-grained control over loop generation.

#### 4.2 `syn_range`

`syn_range` is the simplest subclass of `syn_iter`. Its arguments follow the semantics of Python's `range` and support `start`, `stop`, and `step` keyword parameters. `syn_range` uses the INC mode and overloads the constructor to pass the correct

count value to `syn_iter`. The iterator variable contains the current value from the generated sequence.

### 4.3 Scalar and Vector Iterators

`var_iter` and `vector_iter` iterate over arrays of integers or floating point values, supplying the current value as a scalar or vector variable of the appropriate type. The arrays can be native Python or Numeric Python `arrays`. For example, the following code implements an add/reduce operation:

```
data = array.array('I', range(100))
sum = var(c, 0)
for value in var_iter(c, data):
    sum.v = sum + value
```

When the iterator is initialized, `var_iter` modifies the count and step values to conform to the length of the array and size of the data type. The first time through the Python loop, the iterator generates the code to load the next value in the array for the iterator variable. `vector_iter` subclasses `var_iter`, overloading the memory access methods to use vector instructions that handle unaligned vectors. It also adjusts the step size to account for multi-element vectors.

### 4.4 zip\_iter

The previous iterators all support iteration over one sequence. Often, values for a computation are pulled from multiple sequences. Python's `zip` iterator pulls together multiple iterators and returns loop variables for one element from each iterator. `zip_iter` performs the same function on synthetic iterators. The following code uses a `zip_iter` to perform the element-wise operation  $R = X * Y + Z$  on the floating point iterators  $X$ ,  $Y$ ,  $Z$ , and  $R$ :

```
for x, y, z, r in zip_iter(c, X, Y, Z, R):
    r = vmadd(x,y,z)
```

`zip_iter` works by disabling the branch operations for its wrapped iterators and generating its own loop using the smallest count value from the iterators. For each step of code generation, it calls the appropriate methods on each iterator in the order they are supplied.

### 4.5 The unroll Proxy

Loop unrolling is a common optimization for loop generation. An unrolled loop contains multiple copies of the loop body between the start of the loop and the branch instruction. Executing the body in rapid succession reduces the overhead from branching and gives the processor more opportunities for instruction-level parallelism. While modern branch prediction hardware has lessened the impact of loop unrolling, it is still a valuable technique for high-performance computing.

`unroll` is a Proxy object that unrolls synthetic loops by allowing the Python iterator to iterate over the body multiple times, generating a new set of body instructions each time. Between iterations, the the loop maintenance methods

are called with the flag to exclude branches instructions. On the final iteration, a single branch instruction is generated along with the loop epilogue. The following is a simple loop and pseudo-code for the generated code:

Loop code:

```
rng = syn_range(c, N)
for i in unroll(rng, 3):
    sum.v = sum + 2
```

Generated code:

```
start()
  setup(); body; cleanup();
  setup(); body; cleanup();
  setup(); body; cleanup();
end()
```

`unroll` has two options that allow it to generate truly high-performance code. If the `cleanup_once` flag is set, the cleanup code is only generated once per unroll iteration, rather than once for each body iteration. The counter is updated appropriately by `unroll`. The second option allows the user to supply a list of variables that are replicated at each unrolled iteration and reduced once each actual iteration. In the above example, `sum` depends on itself and creates a stall in the processor pipeline. However, if the `sum` register is replicated for each unrolled iteration and the running sum computed at the end of an iteration, the processor can issue more instructions simultaneously to the available integer units, maximizing resource usage. The complete high-performance sum:

```
for i in unroll(rng, 16, cleanup_once=True, vars = [sum]):
    sum.v = sum + 2
```

achieves near peak integer performance on a PowerPC 970.

#### 4.6 The parallel Proxy

Most scripting languages, Python included, are single-threaded and only ever use a single processor on multi-processor systems. However, many scientific and multimedia applications have natural parallel decompositions. To support natural parallelism, the `parallel` Proxy class provides an abstraction for generating instruction sequences that divide the processing task among available processors. `parallel` is designed for problems that divide the data among resources with little or no communication between executing threads. While communication is possible, it is better to decompose the problem into a sequence of synthetic programs, with communication handled at the Python level.

`parallel` works in conjunction with the `ParallelInstructionStream` class. `ParallelInstructionStream` extends `InstructionStream` and reserves two registers to hold the thread rank and group size parameters for the current execution group. `parallel` modifies the count, stop, and address values for the loops it contains to iterate through the block assigned to the current thread. The complete code sequence for a simple parallel loop is:

```
c = ParallelInstructionStream()
proc = synppc.Processor()
```

```

data = array.array('I', range(100))
rank = int_var(c, reg=code.r_rank)

for i in parallel(var_iter(c, data)):
    i.v = i + rank

if MPI:
    t1 = proc.execute(c, mode='async', params=(mpi.rank,mpi.size,0))
else:
    t1 = proc.execute(c, mode='async', params=(0,2,0))
    t2 = proc.execute(c, mode='async', params=(1,2,0))

proc.join(t1); proc.join(t2)

```

In this example, each thread adds its rank to the value in the array. Two threads are created with rank 0 and 1, respectively. The first 50 elements in `data` remain the same, while the second 50 elements are increased by 1. The `join` method blocks until the thread is complete.

## 5 Experiments

Synthetic programs are intended for small computational kernels that are executed in the context of a larger application. For compute-intensive tasks, they should provide a noticeable performance increase over a pure Python implementation and similar execution times as kernels implemented in low-level languages. Surprisingly, we have found that in most cases, because additional semantic information is available for specific optimizations, synthetic programs often outperform equivalent low-level versions.

In this section, we present two examples of synthetic programs along with performance results. The first example walks through different implementations of an array min function and compares scalar, vector, and parallel implementations against native C and Python versions. The next example implements the update function for a interactive particle system application and shows how expressions and iterators significantly simplify synthetic programs.

The timing results were obtained on a Dual 2.5 GHz PowerPC G5 with 3.5 GB RAM running Mac OS X 10.4.6. The Python version was 2.3.5 (build 1809) with Numeric Python 23.8 and PyOpenGL 2. The native extensions were built using gcc 4.0.1 (build 5250) with -O3 and SWIG 1.3.25. Native versions were also tested with IBM's XLC 6.0 compiler. The results did not differ significantly between the compilers and the results for gcc are reported. Times were acquired using the Python `time` module's `time()` function.

### 5.1 Min

The `min` function iterates over an array and returns the smallest value in the array. We implemented four synthetic versions, a sequential and a parallel version

for both scalar and vector arrays, and two native versions, one in Python and the other in C++. For the each example, the kernel was executed 10 times and the average time was recorded. All kernels operated on the same 10 million element array from the same Python instance.

The first two synthetic programs implement the `min` function using scalar and vector iterators and the `fmin` and `vmin` expression functions. `vmin` maps directly to a single AltiVec operation and `fmin` is implemented using one floating point compare, one branch, and two register move operations. The synthetic code for each is:

```
def var_min(c, data):
    m = var(c, _max_float)
    for x in var_iter(c, data):
        m.v = fmin(m, x)

    syn_return(c, m)

def vec_min(c, data):
    m = vector(c, _max_float)
    for x in vector_iter(c, data):
        m.v = vmin(m, x)
    m.store()
    return m
```

The scalar implementation uses the floating point return register to return the value. The vector version accumulates results in a vector and returns a four element array containing the minimum value from the four element-wise streams. The final min is computed in Python<sup>1</sup> and is an example of mixing Python operations with synthetic kernels to compute a result.

The parallel versions of the vector and scalar examples extend the sequential implementations by including code to store results based on the thread's rank. The code for the parallel scalar version is:

```
def par_min(c, data, result):
    min    = var(c, _max_float)
    rank   = int_var(c, reg=c.r_rank)
    offset = var(c, 0)

    for x in parallel(var_iter(c, data)):
        min.v = fmin(min, x)

    offset.v = rank * synppc.WORD_SIZE
    min.store(address(result), r_offset=offset.reg)
    return

mn = min(result) # final reduction
```

The vector version is implemented similarly, with additional space in the result array and a multiplier of 4 included in the offset. In the parallel examples, the synthetic kernels execute in two threads simultaneously and communicate results using the `result` backing store. To avoid overwriting the other thread's result, each thread calculates its index into the result array using its rank. As with `vec_min`, the final reduction takes place in Python.

<sup>1</sup> In all cases, any extra computation in Python was included in the execution time.

	Time (sec)	Speedup	Parallel Speedup	M Compares/sec
<b>Python</b>	1.801	1.0	–	5.6
<b>var_min</b>	0.033	55.2	–	306.6
<b>par_min</b>	0.017	106.4	1.93	590.9
<b>vec_min</b>	0.014	125.0	<i>2.26</i>	694.2
<b>par_vec</b>	0.010	177.5	1.42	986.0
<b>C++</b>	0.068	26.4	–	147.8

**Table 1.** Performance results for different array min implementations on a 10 million element array. The parallel speedup for `vec_min` is relative to `var_min`.

The C++ implementation is a simple loop that uses the C++ Standard Library `min` function. It is directly comparable to `var_min`. The address and size of the data array are passed from Python, and the result is returned as a float using a SWIG-generated wrapper. We experimented with different approaches to dereferencing the array and computing `min`. In our environment, directly indexing the array and using `std::min` was the fastest technique.

The results of our timing experiments are listed in Table 1. In every case, the synthetic versions outperformed the native versions. The parallel implementations achieved good speedups over their sequential counterparts. The parallel speedup listed for `vec_min` is the speedup compared to the scalar implementation.

To understand the C++ results, we examined the assembly code generated for different optimization levels across both `gcc` and `XLC` for the different looping and `min` strategies. The fastest C++ version differed from `var_min` by three instructions. Instead of storing the minimum value in registers, `min` was updated from cache during each loop iteration, using a *pointer* to track the value rather than a register. Attempts to use the `register` keyword in C++ did not affect the compiled code. In-lined comparisons (i.e., using an `if` test instead of `min`) led to the worst performing C++ versions. The generated assembly used registers for results, but used an inefficient sequence of instructions of the comparison, leading to unnecessary dependency stalls.

## 5.2 Particle System

In our original paper on synthetic programming, we demonstrated the technique by implementing the update function of an interactive particle system using a synthetic program. The original version was implemented using Numeric Python, and the update function limited the number of particles to approximately 20,000. The synthetic kernel improved the performance of the application to handle over 200,000 particles, at which point the graphics pipeline became the bottleneck. We reimplemented the update function using synthetic expressions and iterators to evaluate the different approaches to synthetic programming. The code for the new update loop is:

```
for vel, point in parallel(zip_iter(c, vels, points)):
    # Forces - Gravity and Air resistance
    vel.v = vel + gravity
```

	Parameters	Loop/Iters	Algorithm
<b>Numeric</b>	–	–	13
<b>Syn. Altivec</b>	43	20	14
<b>Syn. Expr/Iter</b>	8	3	6

**Table 2.** Lines of code allocated to parameter allocation, loop and iterator management, and the update algorithm for the three different implementations of the particle system update function.

```

vel.v = vel + vmadd(vsel(one, negone, (zero > vel)), air, zero)
point.v = point + vel

# Bounce off the zero extents (floor and left wall)
# and positive extents (ceiling and right wall)
vel.v = vmadd(vel, vsel(one, floor, (zero > point)), zero)
vel.v = vmadd(vel, vsel(one, negone, (point > extents)), zero)

# Add a 'floor' at y = 1.0 so the points don't disappear
point.v = vsel(point, one, (one > point))

```

To compare the synthetic and Numeric versions, we stripped out the comments and white-space and assigned each line of code to be either parameter allocation, loop and iterator management, or algorithm implementation. The results are listed in Table 2. All three versions use the same Numeric arrays to store parameters and iterators, and the Numeric version did not require any additional parameter, loop, or iterator code.

The line counts demonstrate the utility of synthetic expressions and iterators. The original synthetic kernel contained 77 lines of code, 63 of which were used for register management and manual loop maintenance. In contrast, the new synthetic kernel uses only 11 lines to manage the same operations, all of which use a clear syntax. Both the Numeric and original synthetic implementations used similar amounts of code to implement the update algorithm. The Altivec ISA contains many instructions that have direct Numeric counterparts, and the code for both versions is similar. The synthetic expression version, on the other hand, uses only six lines of code to implement the update. While the Numeric version could have been implemented using similar syntax, Numeric’s aliasing rules lead to multiple, unnecessary temporary arrays. Because the expression implementation works on registers, the cost of temporary values is kept to a minimum, allowing a more concise syntax.

## 6 Related Work

The Expression and Template libraries are related to similar technologies for run-time code generation and user-level optimizations. Both libraries rely on object-oriented and generative programming techniques for their implementation.

The scalar and vector expressions use the Interpreter design pattern [5] to implement a domain-specific language for transforming expressions into equivalent sequences of instructions for the target architecture. This approach closely

related C++ expression templates [14], and in particular Blitz++ [14]. Blitz++ uses compile-time operator overloading for transforming array expressions into more efficient source code sequences. ROSE [12], a C++ tool for generating pre-processors for domain-specific optimizations, is generalized source-to-source transformation engine that allows domain experts to design optimizations for object-oriented frameworks.

High-level systems for dynamic compilation include DyC [6], 'C (pronounced tick-C) [11], and the TaskGraph Library C++ [1]. DyC is an annotation-based system that allows users to specify sections of code that should be specialized at run time. 'C and the TaskGraph library use mini-languages that allow users to write code that is partially specialized at compile time and fully specialized at run-time. All three of these systems have compile- and run-time components and extend C or C++. The synthetic programming environment is a run-time system that uses Python directly for run-time code generation.

With the increasing in processing power on graphics cards, new domain-specific languages have emerged for generating GPU instructions from host languages. BrookGPU [2] implements a streaming language as an extension of ANSI C, and Sh [8] uses a combination of compile-time meta-programming and run-time compilation to generate GPU code. Both systems support multiple processor architectures and abstract the lowest level code from developers.

The synthetic iterators are built on Python Iterators [15] and the `parallel` and `unroll` iterators use the Proxy design pattern [5] to intercept requests. The iterators are all designed to allow the user to annotate the current operation with guidelines for generating optimal code sequences. High-performance Fortran (HPF) [4] and OpenMP [3] both use similar annotation techniques for specifying parallel sections of applications.

Finally, other systems exist for implementing high-performance code from Python. Weave [7] and PyASM [10] are code in-lining systems for C++ and x86 assembly, respectively. Psyco [13] is a run-time compiler for Python that generates efficient machine code for common Python patterns, but due to Python's high level of semantic abstraction, has difficulty finding optimal code sequences. Synthetic programming complements these systems by supplying a pure Python approach to generating high-performance code. The specialized nature of the expression library allows it to overcome the semantic challenges faced by Psyco.

## 7 Conclusion

The Expression and Iterator libraries provide an important addition to the Synthetic Programming Environment. By providing natural abstractions for common numeric and iterative operations, they allow developers to create high-performance instruction sequences using Python syntax, while at the same time removing the dependency on external tools for code synthesis.

The libraries also demonstrate how the SPE can be used to quickly develop and test optimizations for a target architecture. Whereas traditional compiler development requires extensive knowledge of the compiler framework and long build times, the SPE uses Python as the code generation infrastructure and re-

moves the build step entirely. By exposing the code generation and optimization processes as user-level libraries, developers have more control over the generation of high-performance code.

The libraries will be made available as part of the Synthetic Programming Environment.

## 8 Acknowledgements

This work was funded by a grant from the Lilly Endowment. Discussions with Douglas Gregor were instrumental in developing the `unroll` Proxy.

## References

1. Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H J Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In Christian Lengauer et al, editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*. Springer-Verla.
2. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.
3. Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
4. High Performance Fortran Forum. High performance fortran language specification. version 2.0. Technical report, Rice University, 1992.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 4-5, pages 207–219,243–257. Addison Wesley Longman, Inc., 1995.
6. Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.
7. Eric Jones. *Weave User’s Guide*. Enthought. Accessed May 2006.
8. Michael D. McCool, Zheng Qin, , and Tiberiu S. Popa. Shader metaprogramming. In *SIGGRAPH/Eurographics Graphics Hardware Workshop*, pages 57–68, September 2002.
9. Christopher Mueller and Andrew Lumsdaine. Runtime synthesis of high-performance code from scripting languages. In *Dynamic Language Symposium (DLS2006)*, Portland, Orgeon, October 2006.
10. Grant Olson. *PyASM User’s Guide V.0.2*. Accessed May 2006.
11. M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. ‘C and tcc: A language and compiler for dynamic code generation. *ACM Trans. on Programming Languages and Systems*, 21(2):324–369, 1999.
12. Dan Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2-3):215–226, 2000.
13. Armin Rigo. *The Ultimate Psycho Guide*, 1.5.1 edition, February 2005.
14. Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.
15. Ka-Ping Yee and Guido van Rossum. Pep 234: Iterators. Technical report, Python Software Foundation, 2001.