Runtime Synthesis of High-Performance Code from Scripting Languages

Christopher Mueller and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{chemuell,lums}@osl.iu.edu

ABSTRACT
Scripting languages are ubiquitous in modern software engineering and are often used as the sole language for application development. However, some applications, specifically scientific and multimedia applications, often have small sections of code that require a higher level of performance than the host language can deliver. In many cases, the algorithm being optimized is simple and has a clear mapping to hardware resources. But, without introducing an intermediate language, developers generally have no direct methods to implement an optimized solution.

In this paper, we present the synthetic programming environment, a run-time system for synthesizing and executing high-performance instruction sequences directly from scripting languages. Our implementation, available for download, is implemented in Python for PowerPC processors and gives Python developers direct access to system resources for performance critical code. We discuss strategies for creating and managing synthetic programs and provide two real-world examples, an interactive particle system and a chemical fingerprint comparison tool.

Categories and Subject Descriptors
D.2.2 [Software]: Software Engineering—design tools and techniques, software libraries; D.2.11 [Software]: Software Engineering—Software Architectures, Domain-specific architectures; D.3.4 [Software]: Programming Languages—Processors, code generation; D.3.4 [Software]: Programming Languages—Processors, run-time environments

General Terms
Performance, Design, Languages

Keywords
Meta-programming, SIMD, chemical fingerprint, machine code, Python, Synthetic Programming

1. INTRODUCTION
Application developers rely on scripting languages not only to glue together application components, but also to develop entire applications. While typically enabling large increases in developer productivity, the use of scripting languages generally incurs a significant performance penalty at run-time. For most applications, processor, memory, and network speeds are more than adequate for hiding the costs of using scripting languages. However, for scientific, multimedia, and other data intensive applications, the overhead of a scripting language is often unacceptable.

The common solution for exposing high-performance algorithms to scripting languages is to supply libraries developed in a compiled language such as C, C++, or FORTRAN. These libraries often provide tuned implementations of common algorithms and data types for a particular domain. For instance, two common Python libraries, Numeric Python [4] and the Python Imaging Library [18], are both based on native C code. There is still some overhead for marshaling data between library components, but these libraries extend Python in a way that makes high-performance number crunching and image processing applications possible.

While extension libraries provide a good, general purpose solution, they have a few drawbacks. Developers often must refactor portions of their applications and data structures to conform to the types expected by the library. The libraries often extend the dependencies of the application well beyond the standard libraries for the scripting language. This increases the cost of production and deployment and can also lock the version of the scripting language at the version supported by the libraries. Finally, general purpose libraries targeted at multiple platforms rarely take advantage of platform specific features. For instance, many processors contain vector processing units that general purpose libraries do not use. Some compilers may perform an auto-vectorization pass when the library is built, but they can rarely detect all opportunities for vectorization.

For many applications, the performance critical section is isolated and uses well defined algorithms on well defined data structures. For instance, an audio effects system applies signal processing kernels to an audio stream that generally consist of nothing more than simple arithmetic. The difference between a scripting implementation, a native implementation, and a vectorized implementation can be an order of magnitude or more at each level of specialization.
In this paper, we presented a new approach to implementing high-performance kernels in scripting languages. The synthetic programming environment (SPE) is a Python library for generating and executing machine code at runtime. The SPE exposes machine instructions as functions, making the low level instructions as easy to work with as other high-level libraries. It enables developers to take full advantage of available hardware resources to develop highly optimized code directly from Python. The SPE relies only on the Python standard libraries for code generation and uses minimal amount of native code for execution. Our implementation supports the complete PowerPC and AltiVec instruction sets from Python running on Apple’s OS X operating system, but is designed to be easy to port to other architectures and operating systems.

As a simple demonstration of the SPE, the following Python code computes the sum of the elements in an array:

```python
# r_* are processor registers, e.g. r_sum = 3
# c is an instruction stream
n = 1000000
a = array(range(n))

# Load a pointer to the array
c.load_word(r_addr, addr(a))
# Set the counter
c.load_word(r_temp, n))
c.add(ppc.mtctr(r_temp))

# Zero the sum
r_sum = 0
# Set a loop label and load the next value
start = c.add(ppc.lwz(r_current, r_addr, 0))
# Update the sum
next = code.size() + 1
result = proc.execute(c)
```

This example loads each element from the array onto the processor and adds it to the running sum. It uses the PowerPC counter register to store the loop count and the decrement/branch instruction to decrement the counter and loop if the counter is not zero. On a 10 million element array, it takes 20 milliseconds to synthesize the code and compute the sum.

In pure Python, the same algorithm can be implemented as:

```python
for i in a: s += i
```

While shorter and easier to read, this implementation takes 3.99 seconds to compute the sum on the same machine. For data intensive applications, the additional 3.97 seconds can add up quickly. If this was part of an interactive application, the UI designer may consider a progress bar for the pure Python version. Using the synthetic version, it may take more time to create the progress bar than it does to compute the sum.

In this paper, we describe the design and implementation of the synthetic programming environment for Python running on a PowerPC under OS X. Using this environment, we present techniques for integrating synthetic code into Python applications. Finally, we discuss two real-world applications, an interactive particle system and a chemical fingerprint comparison tool, that use synthetic programming to achieve a high-level of performance.

2. THE SYNTHETIC ENVIRONMENT

The synthetic programming environment models different aspects of the code generation and execution process as high-level components. The three main components are InstructionStream, ISA, and Processor (Figure 1). These components present a simplified view of the operating system, instruction sets, and execution resources while giving the user fine grained control over the resources exposed by each component.

The first two components, the InstructionStream and ISA, are responsible for code generation. The ISA contains the machine instructions for a particular architecture and exposes them as Python functions. The arguments to ISA functions are the operands for the underlying machine instructions. The functions return a binary coded instruction that can be inserted into an InstructionStream. In the sum example in the Introduction, the ISA component is ppc and the InstructionStream component is c.

In addition to maintaining the sequence of instructions, InstructionStream ensures that the generated code conforms to the operating system’s application binary interface (ABI). The ABI defines the protocols used to pass parameters between functions and specifies who is responsible for maintaining the consistency of registers across function calls and context switches. As the instruction sequence is constructed, InstructionStream monitors register usage and generates the additional code required to support the ABI. InstructionStream also provides support for register allocation and managing user defined resources, such as memory pools.

The last component, Processor, executes the generated code. Processor exposes different methods for executing code and returning values back to Python. Techniques include methods for returning integer and floating point values and also managing asynchronous execution of generated code using threads. In the sum example, the execute() method returns the integer value stored in register r3.

In our implementation, we support two processor families, the PowerPC 7400/7410 and the PowerPC 970, more com-
Figure 1: The components of the synthetic programming environment and the OS X/PowerPC implementation.

Commonly known by their Apple brand names, the G4 and G5, respectively. The versions of these processors supplied by Apple support two distinct ISAs, the PowerPC ISA [12] and the AltiVec ISA [8]. The former is the standard, sequential instruction set used by most applications. The AltiVec ISA is a collection of vector operations for implementing streaming, single-instruction, multiple data (SIMD) applications. Direct access to the AltiVec instructions from Python was the original factor behind the development of the synthetic programming environment. The ABI is the OS X ABI [3] for the 32-bit PowerPC architecture.

A design goal for the synthetic programming environment was to implement as much as possible in Python. In addition to enabling the use of Python language features, the intent is to keep the system accessible to Python programmers. Rather than requiring C extensions or custom parsing routines for a special assembly language, programmers can use Python directly to implement new features and optimizations. Thus, with the exception of the execution methods used by the Processor component, all parts of the system are implemented in Python using only standard Python libraries.

In the next few sections, we detail the design and implementation of the different components in the synthetic programming environment. A working knowledge of the Python programming language is assumed for the discussion.

2.1 Processor
The Processor component manages the execution of an instruction sequence and is the only component that requires support from native code. The public interface to the Processor consists of five main methods:

```python
class Processor:
    def make_executable(code)
    def execute(code, mode='void'|'int'|'fp'|'async')
    def suspend(tid)
    def continue(tid)
    def cancel(tid)
    def join(tid)
```

`make_executable(code)` takes an instruction stream and performs the necessary operations to make it executable on the target platform. On OS X, this entails a call to the memory management function `MakeDataExecutable()`.

`execute(...)` executes an instruction stream using one of four modes. The simplest, `void`, executes the instruction stream as if it were a `void` function call. The next two modes, `int` and `fp`, execute the stream as if it was a function that returned an `int` or `double` value, respectively. Using the ABI convention, the instruction stream places an integer value in general purpose register r3 to return an integer value or floating point register fp1 to return a floating point value. If the stream does not explicitly place a value in one of the return registers, the value returned by `execute()` is undefined. Under the first three execution modes, `execute()` blocks until the stream is finished.

The final mode executes an instruction stream asynchronously in its own thread. It returns immediately, with the thread id (tid) as the return value. The remaining Processor methods are used to control execution of the thread. `suspend()` halts execution of the thread, `continue()` resumes execution of a suspended thread, and `cancel()` terminates a thread. `join()` is used to wait for a thread to finish. Asynchronous mode makes it possible to fully utilize all available proces
sors from Python for data intensive applications. It is worth noting that, aside from join, synchronization primitives are not supplied at this level. In the usage section, we introduce some techniques for coarse grain synchronization.

The Processor methods are implemented in C and use pthreads for asynchronous execution. Each native function takes the address of the instruction stream as its argument and casts the stream to a function of the requested type to call it directly. For example, the integer version of execute() is implemented as:

```c
typedef int (*Stream_func_int)();
int execute(int addr) {
    return ((Stream_func_int)addr)();
}
```

The Python Processor code handles execute dispatching and is responsible for extracting the address of the instruction stream and ensuring it is properly aligned. This helps keep the amount of native code to a minimum.

### 2.2 InstructionStream

InstructionStream is the main component used to build an instruction sequence and provide ABI conformance. It also provides a basic interface for managing memory and processor data resources. InstructionStream includes the following methods:

```python
class InstructionStream:
    def add(inst):
        def __setitem__(i, inst) # [] operator
        def acquire_register(type='gp'|'fp'|'vector')
        def release_register(rid, type = ...)
        def add/remove_storage(s)
        def reset_storage(s)
        def cache_code()
```

The add() method is the method used to build up the instruction stream. It takes a single, binary coded instruction and returns the instruction's position in the stream. The position can be used later to compute relative addresses for branch instructions or to replace the instruction using stream[i] = inst, which calls __setitem__(i, inst).

The next two sets of methods manage register allocation and memory allocated for use with the instruction stream. When building a sequence of instructions, the developer will inevitably need to use registers to store data on the processor. acquire_register returns a register from the pool of available registers for exclusive use by the caller. When the register is no longer needed, it is returned to the pool with release_register.

The instruction stream loads data from memory using the address embedded in the load instructions. To ensure that the memory buffers used by the stream are valid, the add/remove_storage() methods allow the developer to attach objects to the instruction stream, incrementing their reference counts and avoiding garbage collection until they are removed or the storage is reset.

The final method, cache_code(), freezes the code and generates the prologue and epilogue required by the ABI. On OS X, generating the prologue consists of checking the list of used registers for any callee save registers and adding the code to save the registers to memory before the main instruction stream is executed. It also sets flags for any used vector registers to ensure context switches do not corrupt their contents. The epilogue reverses the process and adds an instruction to return execution to the calling procedure. Note that our system currently only supports leaf functions, or functions that do not call any other functions. Leaf functions are not required to perform additional stack management operations, simplifying the prologue and epilogue implementations.

The actual instruction stream is implemented using a Python array of type `1I`, or unsigned integer. This corresponds to the native 32-bit instruction size for PowerPC instructions.

### 2.3 ISAs

The PowerPC and AltiVec ISA components are implemented as Python modules with one function for each instruction in the ISA. The modules also contain convenience functions for commonly used assembly language mnemonics. Both components are built dynamically at module load time from tables of instructions and fields and can be built in production or debug mode. In debug mode, each time a instruction is generated, its binary and symbolic representations are printed to stdout.

The underlying module for ISA synthesis (Figure 2) is generated in general purpose and can be used to generate different ISAs from descriptions of the instructions and fields. At the lowest level, machine instructions for most architectures are a sequence of bits forming one or more words. The bits are subdivided into fields of one or more consecutive bits. Fields can be further categorized as opcodes with a constant value identifying the instruction or parameters that vary each time the instruction is generated. Some fields may also have special formatting requirements and are implemented as subclasses of Field. For instance, the SplitField field type holds a 10-bit number with the first and last 5-bits swapped.

To handle the common ISAs in use today, the ISA synthesis module contains two main classes. The Field class represents a field in an instruction. A field instance has two properties, a name and a bit range. An instance of the Field class is callable, allowing it to appear as a user as a function. The __call__() method takes as its argument the value for the field and returns a word (on 32-bit PowerPC, this is a 32-bit unsigned integer) with the value shifted to the proper bit location. Field instances can also generate a code template of the form (value << shift) that is used by the Instruction objects to build efficient instruction generators.

The Instruction class creates a callable object that appears to the user as a Python version of a machine instruction. Its constructor takes a list of Fields and constants and builds its __call__() method dynamically from code templates supplied by the Field objects. For instance, the add instruction, a function for adding a constant to a value in a register, is composed of four fields: an opcode with value 14,
Figure 2: The components of the ISA Synthesis Module. The Field class encapsulates the bit range of an Instruction field. Subclasses handle special types of fields. An instance of Instruction is a callable object that returns a properly formatted machine instruction for the given arguments. The Synthesize functions create Fields and Instructions from tables, simplifying ISA encoding and generation. Three representations of the addi instruction illustrate how the assembly, binary, and SPE versions relate to each other.

The code for creating and using the addi instruction is:

```python
addi = Instruction((14, D, A, SIMM))
inst = addi(3, 1, 42)
```

D, A, and SIMM are Field objects that contain the proper bit shifting information for destination, source, and constant fields, respectively. The call to `addi()` returns a properly formatted machine instruction that adds 42 to the value in general purpose register r1 and stores the result in r3.

To create a module for an ISA, the register types and instructions are encoded in tables containing the parameters for the constructors and the name of the field/instruction. Each element in the table is used to instantiate an instance of the Field or Instruction and the name is used to attach the new instance to the module. For example, the entries used to generate the Field objects and function for the addi Instruction are:

```python
Fields = (
    ...,
    ('A', Field((11,15))),
    ('D', Field((6,10))),
    ('SIMM', MaskedField_16((16,31))),
)
```

The term “synthetic programming” has been used at least two times in software literature. In 1985, a paper titled Synthetic Programming [7] described an automated program generation system. The term was also used in the early 1980s to describe a method for entering hidden instructions...
purpose synthetic programs at run-time from a collection of code generating components. A Python program that uses synthetic programs is called the host program. We begin with a brief overview of the PowerPC and AltiVec ISAs.

3.1 ISA Overview
The PowerPC ISA includes instructions for integer and floating point arithmetic, loads and stores; integer logical and bit-wise operations; instruction stream branches; and accessing special purpose registers. All instructions operate on operands stored in registers or immediate values encoded directly in the instruction. The ISA specifies 32 32-bit integer and 32 64-bit double precision floating point registers along with a condition register for storing the results of comparisons, a float-point status register for tracking floating point exceptions, a link register for branches, and a count register to hold a single loop counter. PowerPC instructions span one 32-bit word and are fixed length.

On 32-bit PowerPC platforms, memory addresses are 32-bits long, making it impossible to directly encode an absolute memory address in a single instruction. Instead, different methods for computing absolute and relative addresses from operands stored in registers and small immediate operands are provided. To load a full 32-bit address into a register, calls to the addi() (add immediate) instruction along with a shift-add, addis(), can be used. Because this is so common, we provide it as a convenience function in the PowerPC ISA module. ppc.load_word(code, r, w) loads the 32-bit value w into register r.

The AltiVec ISA complements the PowerPC ISA with single-instruction, multiple data (SIMD) operations. SIMD operations, also referred to as vector operations, perform the same operation on multiple data items using a single instruction, enabling fine grained, data-level parallelism. For instance, two AltiVec registers may hold 4 values each from two input arrays. A vector add instruction will perform element-wise addition on the vectors, replacing 4 add operations with one. The AltiVec ISA specifies 32 128-bit registers that can be partitioned into 4, 8, or 16 integers (i.e., int, short, char), 4 single-precision float point numbers, or various packed pixel formats.

Because the PowerPC and AltiVec ISAs have a large number of registers, it is often possible to implement complex algorithms entirely on the processor, only referring to memory to load the next values from a data stream. This makes the ISAs ideal for use with the synthetic programming environment. Other ISAs, such as Intel’s IA-32 have far fewer registers available, making machine-level algorithm implementation more difficult.

3.2 Data Sharing
In order to perform non-trivial operations, developers need to share data between the host program and the synthetic program. While the ABI provides guidelines for passing parameters in function calls, synthetic programs have another mechanism available. Because the executable code is generated at run-time, run-time constants, including memory addresses for dynamically allocated memory, can be encoded directly into the instruction stream using the operations with immediate operands. For instance, the following sequence of Python code creates an array and a sequence of instructions that loads its address and first element into a register:

```python
a = array(range(10))
code.add_storage(a) # create an array
code.add_storage(rd) # inc ref count
rd = code.acquire_register() # address register
ppc.load_word(code, rd, addr(a)) # load the address
rv = code.acquire_register() # value register
code.add(ppc.lwz(rv, rd, 0)) # load the value
```

Using this method, an arbitrary number of parameters or data streams can be passed between a synthetic program and the host program via dynamically allocated memory. Note the call to add_storage(). This ensures that the array is not garbage collected before the synthetic program is executed.

3.3 Progressive Specialization
Throughout the lifetime of an application, the same synthetic programs may be executed multiple times, potentially using different data sources. Because the Processor component only supports simple function calls without parameters, a new synthetic program may be needed for each set of parameters. To handle this situation, and also enable other run-time optimizations, the InstructionStream supports progressive specialization using the bracket operator, []. Any instruction in the stream can be replaced with another instruction, effectively specializing the stream for the current data and execution run.

The most common use of progressive specialization is to replace the address of a data stream, but it can also be used to update constant data sizes, branch conditions, numeric instructions, or chain together partially specialized instruction sequences. For example, after adding a value to every element in an array, the following code sample changes the synthetic program to subtract 42:

```python
...create array and acquire registers...
add_op = code.add(ppc.addi(r1, r2, 42))
...setup stream...
proc.execute(code) # execute with add
code[add_op] = ppc.subi(r1, r2, 42)
...finish stream...
proc.execute(code) # execute with sub
```

Modifying the instruction stream has the side effect of invalidating the cached code. The prologue and epilogue are destroyed and regenerated by the next call to InstructionStream.cache_code().

3.4 Synthetic Components
To simplify the creation of synthetic programs, common instruction sequences and code patterns can be abstracted into synthetic components, in the same way that objects and functions abstract traditional code components. We have
experimented with two different approaches for developing synthetic components. The first approach, used for simple sequences, is to simply provide a function that generates a common sequence of instructions. The load_word() function is an example of this technique. By convention, the first argument to a synthetic component function is the InstructionStream.

For more complex synthesis patterns, classes and class hierarchies can be used to manage abstraction. For example, consider a set of nested loops. The basic pattern for nesting loops is simple. Each loop has some initialization code, body code, loop condition code, and cleanup code. A Loop class captures each component of a single loop using overloaded methods. A second class, LoopNest then accumulates the individual Loops in the proper nesting order and generates the entire code sequence. The chemical fingerprint example uses this approach (Figure 4).

We have also abstracted specific types of loops. The simplest, a CountLoop takes a loop body and a size n and creates a loop that iterates n times. Properties on CountLoop allow the object to share the count register with the loop body object, essentially in-lining the body operation.

### 3.5 Asynchronous Execution

The Processor provides a basic set of methods for executing synthetic programs asynchronously. As with any multi-threaded application, this feature should be used carefully. Its primary purpose is to enable Python applications to take advantage of multi-processor and multi-core systems for naturally parallel problems. A synthetic thread can communicate progress with Python by updating values in a shared memory location. Memory shared between the host and the synthetic program can also be used to implement synchronization primitives using atomic read and write instructions, e.g., lwzu and stwu on PowerPC.

### 3.6 Debugging

The biggest challenge with any machine-level code, especially one that accesses data indirectly using pointers/memory addresses, is debugging. Synthetic programs are no exception. The core components of the synthetic environment provide a few tools to aid in debugging. First, the ISAs can be generated in a debug mode. Each time an instruction is executed, human-readable, binary, and decimal representations are printed to stdout. For instance, in debug mode a call to ppc.addi(3, 1, 42) outputs:

```
addi (D, A, SIMM) [ D = 3 A = 1 SIMM = 42 ]: ...
  939524096 | (long(D) << 21) | (long(A) << 16) ...
| long(SIMM) & OxFFFF
 001110000110000100000000000101010
945881130L
```

The first line contains the instruction name, its parameters, and the actual code template used to create the function. The next two lines contain the binary and decimal representations of the instruction word. The binary format is especially useful for inspecting bit ranges to ensure the instruction is properly formatted. Most Illegal Instruction errors are a result of incorrectly formatted instructions.

If the ISA modules are not in debug mode, the binary form of the instruction sequence can still be printed using the print_code(pro=False, epi=False) method on InstructionStream. The keyword arguments turn on and off output of the automatically generated prologue and epilogue. Along with the instruction sequence, print_code() also outputs the address and size of the buffer containing the instruction sequence.

When the instruction printouts fail to provide a clue to an error, synthetic programs can be debugged directly using a debugger such as gdb. To start the program in gdb, execute `% gdb python` and use the run command to execute the host program. When the program crashes, the instruction sequence, memory locations, an even registers can be examined directly. If the program is not crashing, but still malfunctioning (e.g., running an infinite loop), the program can be forced to crash by inserting the special Illegal() mnemonic from the PowerPC ISA at the head of the instruction stream. Once the program crashes, it can be resumed and executed stepwise within gdb.

### 3.7 A Note on Safety

Because synthetic programs are written at the machine-level, they lack common safety features taken for granted by developers using higher level languages. Like C pointers and in-line assembly code, there is no bounds checking and any valid user instruction can be executed directly on the processor. Thus, a synthetic program has access to the entire Python run-time and, as we have learned from buffer overruns, the graphics buffers on OS X. As with any low level programming system, it is up to the developer to carefully select and inspect synthetic components to ensure they are performing as desired. For most high-performance kernels, this is simple, but for more complex applications, a more robust security system should be developed.

### 4. EXAMPLES

In the next two sections, we describe two sample applications of synthetic programming. The first is a particle system simulation, originally developed as a simple example of Numeric Python and PyOpenGL [1] integration. The Numeric code for updating the particle system was ported directly to a synthetic program, providing a level of scalability far beyond the original version. The second example is a more complex application for computing the similarity matrix for a collection of chemical fingerprints. In this example, successive implementations are compared to demonstrate the costs and benefits of different approaches to managing synthetic components. Both programs are available for download with the synthetic programming environment.

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). Times were acquired using the Python time module’s time() function.

#### 4.1 A Particle System

**synparticles** demonstrates a port of a Numeric Python application to the Altivec ISA using synthetic programming.
Figure 3: A “time-lapse” image of the particle system application. Two versions were implemented, one uses Numeric Python to update the points and the other an AltiVec-based synthetic program.

synparticles implements a simple, interactive particle system (Figure 3). Particles are emitted as the mouse moves around the application window. The current speed and direction of the mouse determines the 2-dimensional velocity vector for the emitted particle and the particles move around the screen, slowly succumbing to gravity and air resistance.

The core update algorithm for the particles is as follows. At each time step, the velocity of each particle is updated to account for gravity and air resistance, and the position of the particle is adjusted accordingly. After the update, the new position of each particle is checked against the environment bounds (the display extents). If the particle is at a wall, the appropriate velocity component is negated. If the particle is at the “floor”, $p_y = 0$, its $y$ screen position component is set to 1 to ensure the particle is visible on the screen. The Numeric version directly implements this update algorithm. Where possible, it avoids creating copies of Numeric arrays using the optional output arguments for Numeric operations. On our test system, the Numeric code could handle 20,000 particles at 30 frames per second.

The synthetic implementation replaces the Numeric update functions with a synthetic program that performs the same sequence of operations using 4-wide floating-point vectors. The coefficients of gravity and air resistance, along with the display extents are loaded into vector registers from a memory buffer. Then, a loop iterates $n/4$ times. At each iteration, it loads the current four element position and velocity vectors, containing the $x$ and $y$ velocity and position components for two particles. The velocity and position vectors are updated using the AltiVec instruction to add floating point vectors, vaddfp. To detect bounds using vector operations, a vector compare instruction generates a new vector containing 0 or 1 for each element depending on the outcome of the comparison. The new vector is used as an argument to the vsel instruction, which fills another vector with the elements of two vectors, selecting one or the other based on the 0 or 1 values in the comparison vector. This is used to insert 1 or $-1$ values into a multiplier that updates the velocities if a boundary is struck. The Numeric and AltiVec code for the bounds detection portion of the update is:

**Numeric Version:**
```
# Bounce off the walls
lt = Numeric.where(Numeric.less(points[:,0], 0), -1, 1)
Numeric.multiply(lt, Numeric.where(Numeric.greater(points[:,0], width), -1, 1), lt)
Numeric.multiply(v[:,0], lt, v[:,0])
```

**AltiVec Version:**
```
# Bounce off the zero extents
# (floor and left wall)
av.vcmpgtfpx(v_mask, v_zero, v_point, 0))
av.vsel(v_mask, v_one, v_floor, v_mask))
av.vmaddfp(v_vel, v_vel, v_zero, v_mask))
```

$vcmpgtfpx$ replaces greater and $less$ and $vsel$ replaces $where$.

The synthetic program is generated when the program is loaded and all values are passed using Numeric arrays. To support Numeric arrays, a small C extension was required to return the address of the data member in the Numeric array data structure.

synparticles scaled to over 200,000 particles at 30 frames per second. At this point, the graphics pipeline become the execution bottleneck, a rare occurrence for a Python program.

### 4.2 Chemical Fingerprint Comparison

The next example demonstrates the use of synthetic programming for a scientific application, chemical fingerprint comparisons. Chemical fingerprints are bit vectors that encode properties of compounds in a compact form for comparison with other compounds. Various methods for generating fingerprints exist and the Tanimoto/Jaccard metric for bit vectors is the most commonly used metric for comparing chemical fingerprints [10]. However, generating the full comparison matrix for a collection of compounds is a quadratic operation. A naïve implementation in any language will introduce overheads that severely limit its utility in a production environment. A simple factor of 2 difference in performance may be the difference between obtaining results for a moderate collection of compounds in two weeks instead of one.

To evaluate the use of synthetic programming for similarity
matrix generation, we developed a series of implementations that use the synthetic components for different portions of the algorithm. The pseudo-code for the entire algorithm is:

```python
data # collection of bit vectors
#
for x in data:
    for y in data[index(x):]:
        tanimoto(x, y)

def tanimoto(x, y):
    ab = popc(x XOR y)
    c = popc(x AND y)
    return c / (ab + c)
```

The standard Tanimoto metric computes \( \text{similarity} = \frac{c}{a+b+c} \) where \( a \) is the number of bits true in \( x \) and not \( y \), \( b \) is the number of bits true in \( y \) and not \( x \), and \( c \) is the number true in both. The pseudo-code lists the common method for implementing the algorithm using bit operations, where `popc` computes the population count, or number of ‘1’ bits in a bit vector.

The data set used contains fingerprints from David Wild’s gnova database for the NCI compound data set and was acquired directly from Dr. Wild. Each fingerprint is a 166-bit vector and was stored as an 8-word array to meet the data alignment requirements for efficient memory access - AltiVec loads must be 16-byte aligned. The bit vectors were converted from a string to binary representation in Python and stored using a Python `array` with a typecode of ‘I’.

The next few paragraphs describe the different implementations of the algorithm.

**Pure Python** The pure Python implementation is similar to the pseudo-code listing above. The bit vector operations operate on word-sized chunks of the bit vector and accumulate the result. Because the Python code did not require special byte alignments, only 6 words were compared for each operation. XOR and AND were implemented using Python’s built-in bit-wise operators. `popc` broke each word into 4 bytes and used a look-up table to get the bit count for each byte and summed the results. This is the generally accepted method for performing efficient population counts [15].

**Synthetic Bit Ops** The first synthetic implementation uses synthetic versions of the bit operations XOR, AND, and POPC implemented using the AltiVec ISA. Each bit operation is called as a Python function. The functions generate the AltiVec instruction stream to compare the data by filling the addresses and sizes of the \( x \) and \( y \) vectors. The result is returned as a bit vector for AND and XOR and an integer for `popc`. The remaining math for the comparison is implemented in Python. The `popc` implementation is able to take advantage of a special operation in the AltiVec. The `vperm` (vector permute) operation allows the bit count look-up table to be implemented directly in the registers on the processor, removing the need to store the look-up table in cache [2]. `vperm` can perform 16 look-ups in parallel.

**Synthetic Tanimoto** The next synthetic implementation uses the previous bit-wise components but also implements the rest of the Tanimoto operation synthetically. This requires only a few additional instructions to perform the final similarity computation and move data between the floating point and integer registers. The floating point conversion was implemented using the algorithm for unsigned integer to double precision conversion in the PowerPC Compiler Writers Guide [11].

**Progressive Tanimoto** The previous two implementations fully specialize the instruction stream for \( x \) and \( y \) each time a comparison is performed. However, the only value that changes between calls is the address of the \( x \) and \( y \) vectors and only four instructions in the stream use these values (two instructions for each vector). And, when a row in the matrix is being computed, only the \( y \) value changes. Rather than recreate the entire instruction stream for each comparison, this version creates the stream and uses progressive specialization to update the existing stream when the addresses change.

**Synthetic Loop** In this version, the inner loop that increments the \( y \) vector is implemented synthetically. The Tanimoto synthetic component is reused to provide the comparison kernel and the loop components are used to abstract the loop operations.

**Synthetic Matrix** The final synthetic implementation uses a second synthetic loop for the outer loop compute the entire similarity matrix synthetically. In this case, the loop body is the loop from the previous section. The outer loop manages the \( x \) vector and also resets the \( y \) vector at each loop iteration.

Figure 4 illustrates the synthetic components and their relations. Where possible, synthetic components directly share data values using references to objects or registers. Giving the user direct control over when and how to share data avoids the creation of temporary variables to pass values between components, removing one of the so-called abstraction penalties common in object-oriented systems.

**C++ Library** The final implementation is a C++ implementation of the full matrix comparison. It is implemented as a C++ library for Python and computes the entire matrix in C++. The C++ version was implemented using standard C++ idioms in order to give the compiler the best chance at providing an optimal implementation. It was compiled using O3, which performs aggressive sequential optimizations but not automatic vectorization. The `popc` was implemented using a look-up table as in the original Python implementation.

**Experiments** To compare the implementations, the first five (all but the full matrix versions) versions were used to compare the first fingerprint against the next 50,000 fingerprints. The synthetic code executed twice, once with the stream generation and execution enabled and the second time with only the stream generation enabled. This allowed us to measure the cost of synthesizing the instruction streams. The full matrix versions were executed for different input sizes starting with 5000 compounds through
Figure 4: The synthetic components for full Tanimoto comparison algorithm. Loop is a synthetic base class that abstracts loop operations. The subclasses add task specific functionality such as incrementing data pointers and placing results in the proper registers. Because the components are composed once all runtime information is available, they can take advantage of optimizations such as sharing registers and manual loop unrolling. For instance, RowLoop initially acquires the register for vx and propagates it to the other components, avoiding the need for temporary registers.

50,000 compounds.

4.2.1 Results
The results for the single row comparisons are shown in Figure 1. Among the synthetic versions, the results show that the cost of creating the instruction stream can have large impact on the overall cost of using synthetic code. In the Synthetic Bit Ops and Synthetic Tanimoto versions, almost all the time is spent generating instructions. The Progressive Tanimoto version, on the other hand, is identical to Synthetic Tanimoto except that the final specialization only modifies two instructions (the y vector, since x is fixed for a single row). At this point, the synthetic version is 3.2 times faster than the Python version, even though the final specialization is handled in Python. Synthetic Loop shows the full effect of synthetic programming, providing a speedup of 723 over the Pure Python version.

Table 1: Run-times for a single row (50,000 comparisons) for the different implementations of the comparison algorithm, broken up by the time spent synthesizing and executing the instruction sequence. Times are in seconds and speedup is relative to the Pure Python version.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Execute</th>
<th>Synthesize</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure Python</td>
<td>5.182</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Syn. Bit Ops</td>
<td>0.586</td>
<td>297.942</td>
<td>0.017</td>
</tr>
<tr>
<td>Syn. Tanimoto</td>
<td>29.498</td>
<td>86.244</td>
<td>0.044</td>
</tr>
<tr>
<td>Prog. Tanimoto</td>
<td>1.604</td>
<td>0.002</td>
<td>3.23</td>
</tr>
<tr>
<td>Syn. Loop</td>
<td>0.007</td>
<td>0.002</td>
<td>723.50</td>
</tr>
</tbody>
</table>

The results for the full matrix comparisons are in Figure 5. The synthetic version is about 50% faster than the compiler optimized C++ version. In this case, extra semantic information was available to the synthetic code. At the point of synthesis, we selected operations for multi-word bit vectors that use the AltiVec processor. There is not enough information available in the C++ code for the compiler to safely determine that we are working on bit vectors and as such, it can only optimize the code so much.

5. RELATED WORK
Most compilers, code generation tools, meta-programming systems, and languages are related at various levels to the work presented here. In this section, we focus on the tools and techniques most similar to our approach. These fall into two categories: run-time code generation systems and code in-lining systems.

Run-time and dynamic code generation is an active area of research. The most common type of dynamic code generation comes in the form of just-in-time compilers, or JITs [5]. JITs work behind the scenes in the application’s run-time environment and use information available at run-time to generate new instruction sequences from byte-code or machine-code. The new code is recompiled and optimized for performance. Our approach differs significantly from a JIT. While the code is generated at run-time, its generation is directed by the developer, who may have more semantic information available to direct optimization.

A few full featured dynamic compilation frameworks have been developed for C. DyC [9] is an annotation-based compilation system that allows the user to annotate portions of C code that would benefit from run-time specialization.
When executed, the partially specialized input code is fully specialized based on run-time parameters using a run-time code generation system. 'C (pronounced tick-C) [17] takes a similar approach but introduces a new dynamic language based on a subset of C. `{}` expressions contain code specifications that are partially specialized at compile time and fully specialized at run-time. These two systems both allow specialization based on run-time parameters and perform basic optimizations. In contrast to the synthetic environment, they introduce new mini-languages. As with just-in-time compilers, the final optimizations are not visible to the user and, short of extending the systems, the user cannot add new optimizations.

A higher level system, similar to our notion of synthetic components, is the TaskGraph Library for C++ [6]. A TaskGraph is an object built from a mini-language and stored as an abstract syntax tree (AST). At runtime, the AST can be manipulated and specialized based on the current data parameters. The TaskGraph library transforms the objects to C++ code and uses an external compiler to compile and link the new code at run-time. It does not directly create new low-level code.

Multiple in-lining systems have been developed that allow developers to access other languages directly from Python. The two most notable systems are Weave [14] and PyASM [16]. Weave allows the developer to in-line C and C++ code in Python applications and pass data between Python, C, and C++. At run-time, Weave compiles the C or C++ code and calls it using the Python/C calling conventions. PyASM is similar to Weave but uses a subset of the x86 assembly language as the in-lining language; PyASM is essentially an in-line assembler for Python.

In-lining systems differ from our synthetic environment in an important and fundamental manner. In-lining systems rely on an intermediate language for expressing new code. These can encumber the run-time with additional requirements on external compilers and parsing tools. Additionally, meta-programming becomes a string processing problem. To compose new algorithms at run-time, the user must generate strings in the intermediate language. This adds an additional level of complexity and can complicate debugging.

6. FUTURE WORK

There are two main directions we are pursuing to study synthetic programming: adding support for additional platforms and the developing APIs to abstract platform specific instructions.

Supporting multiple platforms entails implementing the platform specific portions of each part of the synthetic programming environment for different processors and operating systems. The components were designed to separate the operating system dependencies from the processors to simplify porting. For instance, a PowerPC Linux port would require modifying InstructionStream to support the Linux PowerPC ABI and Processor to support execution and thread management on Linux. The PowerPC and Altivec ISAs, however, would stay the same. We are planning support for other PowerPC based systems, including the PowerPC 970 MP and Cell Broadband Engine, and exploring options for IA-32/SSE.

An API for synthesizing common operations would simplify usage and abstract platform specific instructions. For example, Intel’s IA-32 and SSE instruction sets differ significantly from PowerPC and Altivec at the instruction level. But, most code patterns have equivalent representations in each environment. By providing synthetic components for expression evaluation, loops, streaming data, and other applications, the differences between the platforms can be abstracted from the user while still allowing the generation of high-performance code. We are currently exploring techniques based on expression templates [19] and Python generators to provide these abstractions.

7. CONCLUSION

We have presented a system for generating high-performance and highly-specialized code at runtime from Python. In the synthetic programming environment, the developer is given fine grained access to the local processing resources
from a high level language. Using machine instructions as first class objects, developers can use Python’s rich object-oriented and dynamic runtime features to manipulate instruction streams and synthesize computational kernels optimized based on all available runtime parameters. In our example applications, we demonstrated that synthetic programming can be used to produce fine-tuned code without a large amount of effort. By removing the reliance on intermediate languages, developing synthetic applications becomes an exercise in library usage.

The synthetic programming environment is available for download at:

http://www.osl.iu.edu/~chemuell/new/sp.php

8. ACKNOWLEDGMENTS
Jeremiah Willcock provided an early implementation of the PowerPC ISA and helped design the debugging strategies and ABI support. The synjaccard program was developed as a project for David Wild’s Programming for Science Informatics course. This work was funded by a grant from the Lilly Endowment.

9. REFERENCES