

# Runtime Synthesis of Parallel and High-Performance Computational Kernels

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

August 31, 2006

## 1 Introduction

Scripting and interpreted languages have become important tools for software development, with languages such as Visual Basic, C#, Java, Python, and Perl replacing compiled languages such as C, C++, and FORTRAN as the primary tools for application development. Scripting languages offer significant increases in productivity compared to compiled languages, due in large part to simplified type systems, dynamic run-times, and large support libraries. However, these productivity increases have come at the cost of performance. Scripting and interpreted languages are evaluated at run time, limiting overall performance of applications.

For most applications, the loss of performance is acceptable, and often not even noticeable. But for applications that require high performance, especially scientific and multimedia applications, the performance penalty imposed by scripting and interpreted languages limits an application's overall utility. The common solution to this is to develop performance-critical kernels in a compiled language and expose them as functions in the higher-level language. While effective, this approach complicates the development process by adding additional layers of abstraction and extending the development tool chain (Figure 1).

Compiled languages themselves face a new challenge for generating high-performance code. The processor architectures for most commodity and high-end computer systems no longer match the simple models that C and FORTRAN were designed to support. Most systems include multiple processors with multiple execution units and most processors include vector units for SIMD parallelism. Automatically generating optimized code that fully takes advantage of these systems is difficult, if not impossible, and as a result, most compilers cannot generate high-performance code without significant guidance from the developer.

In many cases, however, there is a simple, direct mapping between a performance-critical code section and the available processor resources. But, because of the deep

software stack, code that may use these resources rarely does. In some communities, such as game development, a large portion of the the performance-critical code sections are simply developed in assembly, by-passing the compiler tool chain entirely. Assembly language development, while effective for generating high-performance code sections, is difficult. The language features available in an assembly language to aid development are limited to a few simple constructs, such as labels and names for registers. In contrast, most modern programming languages include extensive language or meta-programming features that allow developers to manage complexity for larger applications.

For developers using scripting languages, dropping into assembly is rarely an option. The few in-line systems for scripting languages either support compiled languages (which may allow in-line assembly) or subsets of existing assembly languages. But, as with standard assembly programming, the support tools are limited. Simplifying assembly programming from scripting languages becomes primarily an exercise in string processing.

Because the mix of compiled languages, scripting languages, and processor features is becoming a development and cognitive bottleneck for high-performance computing, we propose a different approach to creating high-performance applications. Rather than placing multiple layers of abstraction and auto-magic optimizations between the developer and the hardware, we propose removing all but the most extreme abstractions and building a new model for high-performance computing on the resulting system. Our proposed approach, called *synthetic programming*, leverages the productivity and language features of scripting languages while giving developers direct access to the underlying hardware.

We have implemented a prototype system to explore this idea. The *Synthetic Programming Environment (SPE)* [34] is a meta-programming library for run time code synthesis. Our prototype is implemented in Python for the PowerPC [23] and AltiVec [14] instruction sets on Apple's OS X. The SPE exposes the instruction sets to the developer as libraries of Python functions and supplies a set of tools for generating and executing custom instruction streams in single or multi-processor environments. Using the SPE, developers use Python for the majority of the application and implement small *synthetic programs* for the performance-critical sections.

At its core, synthetic programming is similar to using in-line assembly code for optimization. However, because the instruction sets are Python libraries, Python can be used as a meta-programming language for generating synthetic programs. For example, to unroll a synthetic<sup>1</sup> loop, the instructions that make up the loop body can be placed in a function. The function can be called repeatedly generate copies of loop body, effectively unrolling the loop.

Developing more complicated instruction sequences at the instruction level is challenging. To aid in the development, we have begun to explore the use of *synthetic components* for abstracting common instruction sequences. Synthetic components are similar to the code generators used by compilers and can implement complex optimizations. Because synthetic components can be written for a specific problem, they

---

<sup>1</sup>We use "synthetic" as a modifier to denote that the code is composed of instructions generated at run-time.

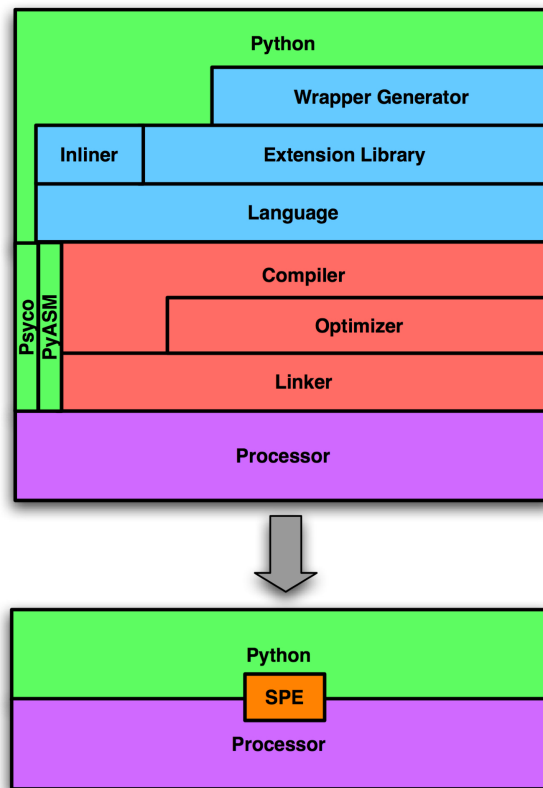


Figure 1: The standard stack of components for developing high-performance applications using a mix of scripting and compiled languages (top). Each layer adds additional dependencies, both programmatic and cognitive, on the develop process. The Synthetic Programming Environment replaces the compilation tools with a simple machine-code generator and set of targeted code generation components, giving developers direct access to the processor from high-level languages (bottom).

can often generate highly efficient code. And, because they are available directly to the application developer, the developer has fine grained control over the optimization, without having to understand the internals of a compiler.

For this thesis, we will expand our study of synthetic programming to explore new optimization strategies and evaluate its utility as a cross-platform tool. We will focus on a core set of real-world applications and use their implementations to drive the design of reusable synthetic components with a focus on a few specific classes of optimizations. To evaluate the portability of synthetic programming, we will port the SPE to a IBM's Cell Broadband Engine (Cell BE) [24], a PowerPC-based system augmented with a novel co-processor, the *synergistic processing unit (SPU)*. The SPU is a SIMD processor that can also execute sequential, control flow instructions. Currently, there is

no effective optimizing compiler for the Cell BE, and we believe the SPE represents a viable alternative.

The applications we will use are inspired by tools from chemical and bioinformatics. Applications for the life sciences, in contrast to traditional scientific applications, are built around irregular data structures and memory-bound algorithms. Techniques for high-performance implementations have not been studied extensively and these applications provide an opportunity to apply the SPE to new problem domains.

The applications we will focus on are:

- Genome/protein alignment
- Chemical database searching
- Integrative graph mining

The first two applications are common tools used by scientists that currently lack high-performance implementations. Our initial research in both areas has suggested that significant speedups are possible, but that the memory access patterns make it difficult to fully utilize available resources. The kernels tend to be simple and finish before the next data items are loaded from memory.

So-called integrative graphs are large graphs composed of connections between data sets, e.g., protein interaction data may be connected back to toxicity data to form a larger graph. These graphs hold vast amounts of data, but are difficult to mine, in part because the algorithms do not parallelize well. As part of our study of synthetic programming, we will explore novel techniques for speeding up common graph operations on these graphs.

For all applications, we will focus on the following classes of optimizations:

- Common optimizations
- Run-time specialization
- Multiple levels of natural parallelism
- Latency hiding

For synthetic programming to be a viable alternative to compiled languages, common optimizations such as loop transformations [5] should be readily accessible to developers.

Run-time specialization offers a type of optimization not possible with compilers. For instance, protein comparisons are based on look-up tables provided by the user. Efficient instruction sequences can be generated if the tables are known at compile time. The SPE allows for run time synthesis of look-up tables based on user-supplied tables.

Automatic parallelization is available from some compilers using language extensions that allow developers to specify regions that can be parallelized, e.g. [9, 12]. Our initial experiments with auto-parallelization suggest that similar approaches are possible using synthetic components.

Latency hiding is a special case of parallelism that attempts to perform useful work with every cycle while data for the next operation is being loaded in the background. We intend to explore abstractions for making latency hiding techniques available directly to developers. These will include double-buffering and software pipelining components along with special purpose components for graph processing.

When complete, our study of the SPE will provide a compelling argument for removing the compiler from the tool-chain for high-performance computing and exposing the code generation and optimization process as user level libraries. By studying known optimizations and novel optimizations on real-world applications, we will be able to show how developers can effectively use synthetic programming to generate powerful computational kernels that exploit available information on the execution environment and current application. By democratizing the optimization process, the synthetic programming has the potential to lead to a renaissance in high-performance computing.

In the next few sections, we provide a detailed overview of the synthetic programming environment, outline our research plan, and present an overview of related work.

## 2 Synthetic Programming

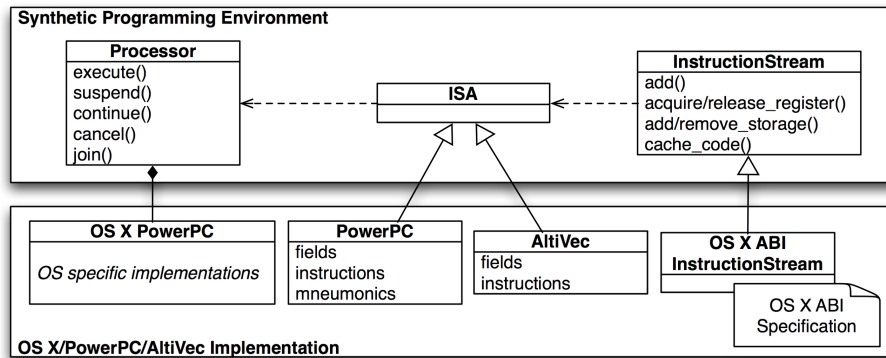


Figure 2: The components of the synthetic programming environment and the OS X/PowerPC implementation.

Synthetic programming [34] is the process of developing programs composed of computational kernels synthesized at run time. 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.

Figure 2 shows the *Synthetic Programming Environment (SPE)*, our prototype library for studying synthetic programming. Synthetic programs are developed using three components supplied by the SPE: 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.

To explore the possibilities for using synthetic components to implement libraries for targeted applications and optimizations, we implemented two libraries for common abstractions, expressions and loops [33]. The new components provided by the `Expression` and `Iterator` libraries abstract the generation of common code sequences for serial and parallel execution of arithmetic expressions and various applications of loop constructs. The following functions demonstrate some of the features of the two libraries:

```
def scalar_arithmetic(c):
    a = var(c, 31)
    b = var(c, 11)

    a.v = a + b

    syn_return(c, a)

def parallel_loop(c, X, Y):
    xi = vec_iter(c, X)
    yi = vec_iter(c, Y)
    zi = zip_iter(c, xi, yi)
```

```

for x, y in parallel(zi):
    x.v = x + y + vmin(x, y)

return

```

`scalar_arithmetic` is equivalent to the earlier example for computing  $31 + 11$ . The `var` factory produces typed variables that allocate registers and generate instructions for various operations. In this example, the expression  $a.v = a + b$  generates the instruction sequence to add the values in  $a$  and  $b$ 's registers. The `syn_return` function places the value in  $a$ 's register in the integer return register.

`parallel_loop` is a more complex example that performs element-wise addition on the arrays  $X$  and  $Y$  using AltiVec SIMD operations.  $xi$  and  $yi$  are iterators that generate the instructions to move data between memory and the AltiVec registers.  $zi$  “zips” the data streams together, ensuring the loop variables  $x$  and  $y$  point to the correct elements in the arrays. The  $+$  operator generates the SIMD add instruction when the expression is evaluated. The `parallel` proxy splits the iteration to execute in multiple threads. `vmin` shows how processor instructions can be abstracted to work in expressions. Here, `vmin` is the element-wise minimum between the two current vector loop variables.

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 Research Plan

Our study of synthetic programming is designed to identify a set of synthetic components that can be used across multiple platforms and isolate components and design strategies that are platform-specific. The results will form the foundation for a set of high-performance code generation libraries and outline the core design and development methodology for synthetic programming.

In the next few sections, we provide overviews of the optimizations we will pursue. We then introduce the applications that will form the practical basis for the study and describe how the targeted optimizations apply to each. Finally, we provide details on the target platforms and how they will advance their study of synthetic programming.

#### 3.1 Optimizations

Modern optimizing compilers perform a number of transformations on different representations of the source code before finally committing to a particular machine-instruction sequence [5, 30]. These transformations are designed to reorder operations in the code so they are executed more efficiently on the target platform. Optimizations can be as simple as constant propagation – identifying values that can be

determined at compile time and using those instead of variables – or as complicated as automatic loop vectorization. Except for the final machine-code layout optimizations, all optimizations performed by a compiler can be performed by the developer. Developers typically avoid writing code for specific optimizations to keep the code simpler and let the compiler perform the difficult work. The downside of this approach is that the compiler cannot always determine if an optimization is safe or may not even identify that a code section can be optimized. In this case, the a possibly easy to implement optimizations may go unused.

Synthetic programming takes a different approach to optimization. Instead of implementing optimizations behind the scenes, synthetic components directly implement different optimizations. Developers using the SPE have a choice of different optimization techniques and can vary their use depending on the current run-time parameters. In this regard, synthetic programming is most similar to pragma- and library-based language extensions for high-performance computing. Tools such as OpenMP [9] and High Performance Fortran (HPF) [12] provide a combination of library calls and compile-time pragmas that let developers signal to the compiler that a particular class of optimizations is allowed.

In this thesis, we will verify our claims that synthetic programming is a viable option for implementing different optimization strategies. The next sections introduce the optimizations we will pursue for this study.

### 3.1.1 Common Optimizations

The most useful compile-time optimizations for high-performance applications center around loop and expression transformations and methods for removing abstraction, such as constant propagation and function in-lining. Synthetic programs are *leaf functions*, i.e., functions that do not call other functions, and as a result do not contain many abstraction penalties. While synthetic components abstract common operations, the actual instructions are included directly in the instruction stream and have no execution-time performance consequences.

Synthetic programming manages other forms of abstraction penalties, specifically those imposed by virtual functions for data access, by buffering the data from objects in predictable memory locations that can be easily accessed using memory operations. As with instruction generation, this imposes a performance overhead on the application. However, in this case, careful object design can eliminate the overhead entirely. One strategy is to store object attributes in regular data structures and have attribute accessors index into the structures. The object interface remains for the low-performance code and while the high-performance code can directly access data items.

Managing abstraction penalties demonstrates hows the programming model for synthetic programs does not map directly to the model used for compiled applications. But, there are many cases where language features and optimizations can be implemented as synthetic components. We have explored a few of these and propose an expanded list for this thesis.

In our prototype, we used the Python expression evaluation engine along with operator overloading to provide a clean syntax for generating arithmetic code sequences. The `scalar_arithmetic` example above demonstrates this technique. Scalar vari-

ables could reference fixed memory location, but we did not include support for indexed memory or operations on blocks of memory. For completeness, we will add indexing operations and a more powerful variable type for matrix operations.

We used Python iterators to abstract different types of iteration and optimizations. These were:

- `ctr` register, decrement, and increment-based iteration
- Integer sequence generation
- Scalar and vector 1D array iteration
- Iterator combining, e.g. ‘zip’
- Loop unrolling
- Loop parallelism

The loop model used to implement many of these is based, in part, on an unpublished design study we performed for abstracting loop nests for parameter sweeps. The resulting design pattern, PSWEEP [32], is more flexible than the loop model we currently use in the SPE. We will extend the SPE loop model to be more faithful to the PSWEEP model. Using the updated model, we will augment the loop transformation components to include support for loop fusion, and also apply the new model to loop parallelization and software pipelining, described below. The existing components will also be updated to reflect the PSWEEP model and the design differences will be evaluated.

Our parallel components currently implement a simple form of block decomposition. While the current implementation is only useful for parallel execution, block decomposition can also be applied to sequential algorithms to improve cache reuse. As we expand our basic expression types to support multi-dimensional arrays, we will also extend the loop optimizations to include multiple techniques for block-decomposition, including cyclic, block-cyclic, and triangular block structures, for both sequential and parallel execution.

The final result of the our study of common optimizations will be a set of components and methodologies for handling common optimizations available from compilers for high-performance code.

### 3.1.2 Run-time Specialization

Synthetic programming supports optimizations that can be performed at compile time but also enables run-time specialization of code sequences. Run-time specialization is a type of partial specialization that takes code specified at development or compile-time and modifies it according to the current run-time problem characteristics. For compute intensive operations, the extra cost of performing the final specialization at run-time should be less than the cost of executing the code without the optimizations. In these cases, run-time specialization can pay huge dividends.

The ability to perform run-time specialization brings synthetic programming beyond the world of traditional compilers and in the realm of just-in-time compilers and

other dynamic code generation tools. Run-time specialization also forms the foundations for more advanced optimizations and application architectures. While synthetic programming can be considered a run-time specialization tool because all code is generated at run time, for the context of this work, we will constrain the definition of run-time specialization to cover code generated based on the run-time data characteristics. That is, information not available at development time. We will focus on two types of run-time specializations, data size specialization and register encodings for look-up tables.

Many algorithms operate on vectors that vary in size and type between problem instances. For instance, a feature vector for a data item used in a data clustering algorithm may contain only two elements in one instance but two thousand in another. The core of most data clustering algorithms is a comparison operation between feature vectors, and optimizing this operation is an important step for a high-performance implementation. Depending on the size of the vector, the comparison operation may be implemented as a single operation, a loop that iterates over features, or an unrolled loop that gives the processor scheduler a chance to execute some operations in parallel. The best strategy can be determined by a few experiments and the results encoded into the synthetic component for the comparison. At run time a specific optimization strategy can be selected based on the length of the feature vector. We will focus on synthetic components that abstract this form of run-time specialization based on data size.

A related type of optimization is the run-time encoding of look-up tables into registers [3]. Look-up tables are used in every computation in many algorithms. Keeping them as close to the processor as possible is important to limit memory traffic. Like feature vectors, these data structures have variable run-time characteristics. But, unlike feature vectors, a many classes of look-up tables are small and can be encoded entirely in registers, eliminating the need to move them between memory and the processor between iterations. And, even if not enough registers are available, run-time compression strategies can decrease the size of the tables, reducing the cost of memory transfers. For this study, we will focus on strategies for encoding look-up tables in vector registers. Both the AltiVec and SPU ISAs include vector *permute* instructions that enable single instruction look-up operations, making a potentially costly operation almost free.

### 3.1.3 Parallelism

The biggest challenge faced by modern compilers is the proliferation of parallel processing environments in commodity systems. Intel and IBM's processor lines are heavy with multi-core offerings that include SIMD execution units. And at the lowest level, most of these processors include multiple execution units for specific types of operations, including integer and floating-point arithmetic, that enable instruction-level parallelism (ILP) in some cases. While compilers tend to organize instructions in a such way that a processor's scheduler can extract some amount of ILP, compilers have a very hard time automatically extracting program and data-level parallelism. For complex applications, this is understandable. But, for simpler applications, compilers are presented with too little semantic information to apply even the simplest parallel transformations without additional input from the developer. At this point, the compiler is no longer providing a value added service, as it does when it translates human-readable

code into machine-code, but rather acting as a complicated intermediary between the developer and the machine resources.

Library and languages features for expressing program level parallelism, such as pthreads and Java threads, solve this problem by by-passing or augmenting the compiler and encouraging the developer to be explicit about parallelism in applications. Higher-level systems such as application servers go one step further and present a coarse-grained approach to parallelism that lets developers focus on the serial aspects of their code and ignore the parallel execution environment. All of these approaches have been successful and are used on a regular basis in application development.

Synthetic programming can support both library and language constructs for expressing parallelism at *all* levels, from program, to data, to instruction. Because synthetic programming is a meta-programming technique, it can go further than traditional systems for expressing parallelism. By breaking up application code into natural units of abstraction, developers can recombine the units using synthetic components that apply specific optimizations. For example, a loop can be optimized by pipelining or unrolling, or a combination of both. By properly abstracting the loop, the loop components can be repeated for an unrolled loop or unrolled and interleaved for a pipelined loop. While it is difficult for a compiler to identify situation for applying these optimizations, a developer with knowledge and ready access to meta-programs for generating them can easily apply them.

The SPE exposes data parallelism directly with vector expressions and iterators and program-level parallelism through asynchronous execution. The loop optimizations and models we are developing form the basis for important classes of parallel execution. In our prototype, basic block-level parallelism was a trivial extension of the synthetic loop components. As part of our plan to study common optimizations, we outlined the proposed extensions to our block-level components. To continue our study of techniques for user-aided parallel code generation, we also will extend the types of coarse grained parallelism to include other round-robin style execution and the master-worker paradigm.

Managing parallel execution is one instance where the interplay between synthetic programs and the host language plays an important role in algorithm design. Code developed in the host language is responsible for dispatching synthetic programs to execution resources and handling complex data management and communication tasks (possibly using additional synthetic programs). In addition to the simple parallel strategies described above, we will study architectures for more complex parallel applications. We expect these architectures will suggest extensions to the core SPE for different execution modes and may require synthetic components for abstracting synchronization primitives.

#### **3.1.4 Latency Hiding**

Even with support for micro- and macro-optimization strategies, most data intensive applications are limited by a more mundane problem: the speed of the memory bus. Memory latency on most processors is such that a non-trivial number of instructions can execute in the time it takes to load data from memory. For many data mining kernels, the number of compute instructions fits well within this range. As a result,

the algorithms that use these kernels execute at the speed of the memory bus, not the processor. Latency hiding is the general strategy of hiding the cost of loading data by performing useful work on one set of data while the items for the next set are being loaded. Latency hiding is typically implemented by hand and the productivity cost is justified only in certain circumstances, particularly those where interactivity is important.

Beyond algorithms that simply do not have enough work to perform per load based on their current kernel, there are algorithms on referential data structures that will *never* perform enough work between loads to fully utilize the processor. Graph algorithms, for instance, often traverse a graph and mark each node as it is visited. For graphs of non-trivial sizes, these are entirely memory bound algorithms. Node marking takes only a few instructions and a few more instructions calculate memory addresses to move data between memory and the processor. But, the total number of instructions is far less than the number that can execute between loads. Exploring techniques for hiding latency in graph algorithms is extremely difficult in compiled languages. However, we believe synthetic programming may provide an environment that simplifies latency hiding for even the most extreme memory-bound algorithms. Additionally, synthetic programming also eases the task of exploring register algorithms on partial graphs, a novel idea that may yield significant performance increases for graph processing.

Our initial results modeling loop optimizations suggest that similar strategies will be possible for more complicated optimization strategies usually performed by hand. For this thesis, we will focus designing synthetic components for expressing different types of latency hiding, including software pipelining, double buffering, and ‘register’ algorithms for graphs. The first two are known techniques with limited tool support, and the latter is a novel idea made possible by synthetic programming.

An algorithm is pipelined by decomposing it into discrete execution stages and organizing the implementation so that data is passed from stage to stage, with all stages executing simultaneously. Simultaneous execution can either be in the form of true parallel execution, or instruction level parallelism extracted by the processor. By decomposing the algorithm this way, when one stage is stalled, other stages can still make progress. For our study, we will explore component designs that encourage staged decomposition of synthetic programs and evaluate the utility of software pipelining for different algorithms in different execution environments.

Double-buffering is a simple form of software pipelining that breaks algorithms into two stages, data transfer and data processing. In addition the the algorithm split, memory is divided into two buffers – hence the name – one buffer for the current processing tasks and one buffer for the current transfer task. Double buffering is common in graphics applications, where images are drawn to the back buffer and displayed from the front buffer. This ensures that images that are not fully rendered will not be displayed. Double-buffering is also an effective technique for latency hiding in cache-hierarchies. If the cache is divided into two buffers, the processor can operate on the data in one buffer while the other is being filled. This strategy is especially important for the Cell BE SPU processors, which have a small cache for storing local data and compete with all other processors for the memory bus. Synthetic components for managing memory buffers will be an important tool for developing applications on the Cell BE.

The final technique we will explore is a speculative study of methods for extracting parallelism from graph algorithms. We suspect that by encoding sub-graphs in data structures that span a small number of registers, useful work can be performed on the sub-graphs that yields partial results towards a full answer for common graph algorithms.

## 3.2 Applications

The design studies for the synthetic components and architectural patterns will be informed by real-world application studies, with the applications based on tools for processing chemical and biological data collections. Implementations of chemical and bioinformatics algorithms are rarely developed as true high-performance applications, at least not with the same rigor as numerical algorithms for traditional computation science applications. As a result, studying these applications opens up the opportunity to expand the literature base for high-performance life science applications, rather than simply duplicate existing scientific codes.

For each application, we have engaged in pilot studies to identify strategies for optimization. The identified optimizations correspond to many of the optimizations we outlined above. In the next few sections, we introduce the applications and optimization strategies we will pursue, highlighting how synthetic programming will enhance the optimization process.

### 3.2.1 Genome and Protein Comparison

The Human Genome Project and the many related genomic and proteomic efforts have generated a large amount of sequence data. Gene and protein sequences are represented as long strings of characters, with a four character alphabet for DNA (genes) and a twenty-four character alphabet for the amino acids that encode proteins. As a result of their representation, the algorithms for processing sequences are primarily derived from string algorithms. However, due to noise in the sequences and the unknown semantics encoded by combinations of characters in the sequences, direct string comparisons are not useful for extracting information. To account for noise and uncertainty, sequences alignment algorithms assign a probabilistic score to a comparison between two sequences that measures the quality of the alignment between the sequences. One way to calculate scores is to compute a sliding average of element-wise comparison scores, with individual element-wise scores assigned from precomputed look-up tables. The look-up tables, called scoring matrices, are derived from statistical analyses of sequence collections and vary depending on the species and application.

Scoring matrices are at the heart of most sequence alignment algorithms. Because a score is calculated for every pairwise comparison performed, optimizing the look-up operation will affect the overall performance of the implementation. For genomes, one common optimization is to pre-compute the four possible (one for each letter in the alphabet) sets of scores for an entire sequence and use these new data arrays in the computation. However, this approach increases pressure on the memory bus and introduces unpredictable branches into the algorithm that must be resolved before a memory request is made. For DNA sequences, this approach is manageable. But

for protein sequences, the 24 character alphabet requires 24 sequence-length arrays, adding 20 additional branch options and also fragmenting the cache. Given this, pre-computed score sequence are not the best option for memory-aware, high-performance comparisons.

In our earlier study of DNA sequence comparison [34], we used pre-computed score matrices and focused on extracting different levels of parallelism from the algorithms, using the SIMD processor to accumulate scores and multiple processors to process different blocks of the sequence comparison matrix. Attempts to extend the algorithm to protein and explore other optimization strategies were stalled by the complexity of managing the code given the tools available. To continue this study, we will first reproduce our results using synthetic components and then extend the study to include more advanced optimizations. These will include a comparison of pre-computed score vectors to scores computed on-the-fly and an extension to support protein sequence comparison. For the latter, we will study approaches for run-time specialization of the look-up table code and in all cases we will use the parallel components for parallel execution.

The optimization strategies we develop for genomic and protein data will be applied to a Cell BE port of the BLAST [2] sequence search program. As part of our design study, we will compare the SPE implementation of BLAST against a C implementation to explore the differences between the two approaches to developing applications.

### 3.2.2 Chemical Database Searching

The drug discovery process is a search for chemical compounds that affect biological processes in specific species. Due to the combinatorial explosion in the number of possible compounds and the cost of synthesizing and studying physical compounds, drug development is an expensive and tedious process. However, as more researchers study individual compounds, more data is available to help narrow the search for viable drug candidates. At pharmaceutical companies, it is not uncommon to find databases with raw data on tens of millions of synthesized compounds<sup>2</sup>. These databases, and their publically managed counterparts [21, 35], are useful tools for identifying promising drug candidates. Scientists can cluster compounds [53], identifying groups of similar compounds, or search databases for compounds with similar properties to a new compound [54]. The data acquired from these studies can give clues about the properties of other compounds and help in the design of more expensive laboratory studies.

There are a number of different methods for comparing chemical compounds [54]. The most familiar representation for compounds is the two dimensional Kekule diagram, which provides a concise overview of the elements that make up the compound and their relationships via bonds. While useful for humans, this representation is difficult for computers to process, especially for comparisons. Comparing two compounds is an instances of sub-graph-isomorphism, a known NP-complete problem. Instead, most algorithms reduce compounds to feature vectors that contain entries for significant structural or chemical properties. One such vector may contain entries for the

---

<sup>2</sup>Synthetic chemistry is the sub-field of chemistry that focuses on synthesizing compounds from smaller building blocks. The use of “synthetic” in synthetic programming was borrowed from synthetic chemistry.

so-called “rule-of-five” properties [29], five properties that tend to determine if a compound is a viable drug candidate. Other, more complicated feature vectors use hundreds or even thousands of bits to identify the presence or absence of a particular feature, say a benzene ring. In both cases, comparing compounds is reduced from an NP-complete problem to a more tractable feature vector comparison.

Feature vectors are compared using any number of similarity or dissimilarity measures. Different measures are useful for different types of feature vectors, though most studies tend to use Euclidean distance for real-valued vectors and the Jaccard/Tanimoto metric for bit vectors. A compound collection can either be compared against a small set of compounds in a simple similarity search or against itself as one iteration of a clustering study. As the size of compound databases increases, the cost of the comparison operation is becoming more noticeable.

For simple searches, a small databases can be searched in microseconds, but for larger databases the running time can be seconds or minutes. For clustering steps, however, the running time moves from linear to quadratic, and a linear time of one second for a 1000-compound database increases to over 8 minutes for a full comparison. With clustering algorithms typically requiring  $O(\log n)$  iterations, clustering studies can easily take hours or days for a single parameter set. At these time-scales, even small improvements in the performance of the comparison kernels have out-sized effects.

While the individual metrics have an effect on the performance of a larger search, most metrics require only a few instructions to execute. Instead, the run-time performance is dominated by the memory access patterns. Thus, to optimize chemical searches, it is import to design implementations that minimize the movement of data between memory and the processor. For scalar data, this is a simple application of algorithms designed for matrix operations. But for feature vectors that may be larger than the available register set, more complicated strategies are necessary.

Designing efficient memory-bound algorithms is challenging in compiled languages, where the final memory access patterns are implemented by the compiler. However, developing algorithms using the SPE makes all memory accesses explicit. And, the SPE also enables fine-grained control over the implementation of the similarity metrics, making it possible to simultaneously optimize on two dimensions. For this portion of our research, we will focus on optimizations for chemical similarity computations with a focus on synthetic components for memory access patterns in the context of variable data sizes and comparison kernels.

In our initial study [34], we developed components for abstracting linear and triangular data processing patterns. We also developed components for optimizing kernels for feature vectors based on the length of the input vector, studying approaches for unrolling vector comparison loops for the Jaccard/Tanimoto metric. We will expand these to generalize across other metrics, such as Euclidean distance and the Perlman correlation coefficient, to study the interplay between memory components, computational kernels, and feature vector size for both serial and parallel execution.

### 3.2.3 Graph Mining

Chemical and sequence comparisons for the raw data for bioinformatics. But, the real knowledge is captured in graph data structures that integrate information from multiple homogeneous data sets. Integrative graphs, as they are sometimes called, combine data from chemical, genomic, proteomic, taxonomic, and other biological databases and connect heterogeneous elements based on experimental or theoretical evidence. For instance, a graph may contain genes and small molecules with edges between the data sets when the small molecule inhibits the expression of the genes. It is believed that these graphs will be invaluable tools for exploring biological knowledge.

Because graphs are inherently relational data structures, graph algorithms and data structures are difficult to optimize for high-performance execution. Most operations traverse edges, requiring a memory access to determine the next operation. To further complicate matters, different data structures are useful for different algorithms, suggesting that a graph may need to take on many physical forms while as it is processed.

To evaluate the SPE on difficult to process data structures, we propose a speculative study into SIMD algorithms for graphs. While this may produce a null or negative result, it should help identify the breadth of algorithms to which the SPE can be applied. Our graph study will focus on two common operations, path finding – is there a path between node a and node b – and cluster identification – what nodes are connected to each other in some maximal fashion. Both problems have many known solutions, and, depending on how it is defined, cluster identification can be an NP-complete problem. Our study will focus on attacking these problems by encoding the graph into a form appropriate for vector registers and exploring algorithms based on vector select and permute instructions.

### 3.3 Cell BE Port

IBM's Cell Broadband Engine (Cell BE) [24] is built on a novel system architecture that combines a general purpose processor, the PPU, with 8 dedicated vector processors, called SPUs<sup>3</sup>. The Cell BE is intended for data intensive applications that can benefit from multiple levels of parallelism. Because the architecture is new, effective optimizing compilers do not exist. The synthetic programming model closely mimics the programming models for the Cell BE. The models suggested by IBM uses the PPU as a control processor and the SPUs as the processing workhorses. This mirrors the synthetic programming model of using Python as the controller and synthetic programs as the compute engines.

To port the SPE to the Cell BE, we will provide Cell BE specific versions of the main SPE components. The PPU is a PowerPC processor with a few additional instructions and will require minimum changes to the current PowerPC ISA component and PowerPC InstructionStream. The SPUs implement a new ISA and also have special instruction ordering requirements. A new ISA for the SPU instructions will be implemented along with an InstructionStream that is aware of the SPU's instruction

---

<sup>3</sup>SPUs are actually part of a processing element called the SPE. To avoid confusion between our definition of SPE and the Cell's, we will always refer to the Cell SIMD units as SPU and the Synthetic Programming Environment as SPE.

layout requirements. The most interesting component will be the Processor components. The Processor component will be extended to support code transfer between the PPU and SPUs and include additional SPU management functions analogous to the thread functions currently provided by the SPE.

In addition to the basic SPE port, we will implement a library of synthetic components to abstract common communication and data transfer operations. The Cell BE provides a rich set of machine level operations for direct-memory access (DMA) and inter-processor communication. Our components will focus on abstractions used in the studied applications.

## 4 Related Work

Most compilers, code generation tools, meta-programming systems, and languages are related at various levels to Synthetic Programming. Indeed, Synthetic Programming was inspired by many features of existing systems. Synthetic Programming, however, differs from current technologies in one important aspect: it presents machine-code generation as a user-level task. This shift in perspective changes the way users interact with the processor and affects how they develop and apply optimizations for performance-critical code.

Related work for this study falls into three general categories, run-time code generation, meta-programming, and application-specific related work. The next few sections detail each.

### 4.1 Run-time Code Generation

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 [4]. 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. The Synthetic Programming Environment 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 [17] 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) [39] 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 run time, 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 [25] and PyASM [38]. 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.

The domain-specific language [49] (DSL) movement has also inspired Synthetic Programming. While most DSLs are built around compile-time code generators, the core architectural ideas are similar. DSLs present the user with a set of abstractions for modeling a particular domain. The domain abstractions allow the user to work at a natural level of abstraction for their problem while the underlying transformation and code generation engines handle optimizations. The base SPE can be viewed as a DSL for machine programming and the synthetic components as a DSL for optimizations. However, the synthetic components are not limited to optimization components, making the SPE a general tool for building DSLs.

A few classes of DSLs are worth detailing. With the increasing in processing power on graphics cards, new domain-specific languages have emerged for generating GPU instructions from host languages. BrookGPU [8] implements a streaming language as an extension of ANSI C, and Sh [31] 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. In addition to these peer-reviewed systems, Apple Computer, Inc.'s CoreImage library takes user-specified image processing pipelines and generates efficient implementations at run time based on the available computational resources (e.g., multi-core processor, SIMD unit, high-end graphics cards).

Another important class of DSLs are those related to Telescoping Languages [26]. Telescoping Languages are DSLs that generate optimizing compilers for a specific application domain and have focused primarily on generating optimized libraries for Matlab. Telescoping Languages is an offline system, requiring significant computational resources and time to generate efficient libraries.

A related tool is ROSE [40]. ROSE is a C++ tool for generating pre-processors for domain-specific optimizations. It is a generalized source-to-source transformation

engine that allows domain experts to design optimizations for object-oriented frameworks.

The Atlas [52] and FFTW [15] systems are adaptive code generation tools high-performance library generation. Both systems generate high-performance libraries by trying out different code sequences for linear algebra (Atlas) and FFT (FFTW) libraries. While Synthetic Programming is intended to user-designed abstractions, adaptive generators, including ones modeled after Atlas and FFTW are possible applications for synthetic components.

The general optimizations we will focus on are traditional, well-known high-performance optimizations. [5] provides a comprehensive survey of these optimizations in the context of compiler transformations.

## 4.2 Meta-Programming

Meta-programming systems are related to code generation systems in that both use programming languages and libraries that allow users to generate new code. As with code generation tools, meta-programming systems can be used to implement DSLs and other high-level programming models. For this discussion, we will constrain meta-programming to refer to systems that allow users to directly manipulate the host language. Under this definition, the most significant systems with respect to Synthetic Programming are C++ templates and the various macro languages for Lisp, Scheme, C, and other languages.

C++ templates have had a large influence on high-performance computing. Templated functions and objects let users to substitute types at compile time. This form of polymorphism gives the compiler more opportunities to optimize code sequences and also opens up the possibility of creating so-called *active libraries* [51] that can help guide their optimization using a technique called template meta-programming. The original template meta-programming library, Blitz++ [50], uses compile-time operator overloading for transforming array expressions into more efficient source code sequences. Since its introduction, other meta-programming libraries have been introduced for other high performance applications. Most of these libraries are collected under the Boost project. Two significant scientific computing libraries are the Matrix Template Library (MTL) [46] and the Boost Graph Library (BGL) [45]. Both the MTL and BGL use meta-programming techniques to generate high-performance code from natural programming abstractions.

The design strategies used by C++ meta-programming have been directly applied to Synthetic Programming. The scalar and vector expressions use the Interpreter design pattern [16] to implement a domain-specific language for transforming expressions into equivalent sequences of instructions for the target architecture. This is the same approach used by Blitz++ to reorder operations for matrix operations. Other meta-programming techniques used by the C++ libraries, including loop unrolling in the MTL, are similar to those used by various synthetic components.

C++ meta-programming relies on C++ templates. C++ templates are strictly a compile time feature and are evaluated prior to the compiler optimization and code generation passes. Thus, C++ templates cannot take advantage of run-time information for further specialization. C++ templates also suffer from usability concerns. The template

syntax is verbose, leading to difficult to read code. More significantly, the evaluation algorithms used to process templates are expensive and semantically agnostic, leading to long compile times and long, overly detailed error messages.

Macro systems provide an alternative approach to compile-time meta-programming. Macros are similar to templates in that they are evaluated at run-time, but macros are generally full languages used to create language extensions, rather than clever applications of the type-system<sup>4</sup>. The most common use of macros in high-performance environments is to abstract simple code sequences that should always be in-lined, such as `min/max` operations. More advanced uses include iterator constructs that hide the mechanics of iterating over complex data structures and macros for compile-time loop unrolling.

Like C++ templates, C macros are evaluated at compile time and suffer the same limitations. However, languages with more advanced run-time systems such as Lisp and Scheme, have macro systems that affect run-time code generation. While these languages are rarely associated with high-performance computing and their macro and run-time code generation systems focus on generating new instructions sequences in the host language, it is worth noting their relationship to this work. Their popularization of dynamic run-time systems make run-time specialization and code generation possible. Ableson and Sussman’s classic programming language textbook [1] introduces many of the core ideas that form the foundations for advanced macro systems and run-time code manipulation.

The SPE for Python leverages Python’s dynamic run-time and its well-defined iterator and operator overloading protocols to use Python as a macro language for machine code generation. From this perspective, synthetic components are macros that abstract complex sequences of instructions. Because Python is a general purpose language, complex “macros” can be developed that do more than simple code insertions. More importantly, because the macro language is also a programming language, applications do not need to switch context between the macro language and the main language for development, as is required by C and List macro systems.

### 4.3 Applications

For the applications we selected, different amounts of related work exists. BLAST is a well studied and available in the public domain, encouraging research on different implementations. It is the existence of this large body of related work that will allow us to use BLAST as a case study for synthetic programming. Instead of developing entirely new approaches for a parallel implementation, we can leverage the work of others and focus instead on the novel optimizations that synthetic programming and the Cell BE make available. On the other hand, chemical database search is a relatively simple problem but most libraries are commercial applications. This allows us to study how well synthetic programming can be used to experiment with different optimizations. Finally, as already discussed, graph algorithms have inherent properties that make them difficult to optimize. In this case, we will use synthetic programming

---

<sup>4</sup>While C++’s template system is technically Turing complete, using it as a complete macro language not practical.

to explore novel graph representations and alternative processing methods that may not be possible in traditional programming environments.

The next few subsections highlight related projects that will help inform our studies.

### 4.3.1 Sequence Alignment

Due to the number of whole genomes available as well as the development of high-throughput DNA microarrays [7,10,20], there has been recent significant activity in developing large scale and high-performance sequence analysis algorithms and tools, especially sequence alignment and database searching algorithms. Sequence alignment is essential for identifying conserved regions for understanding evolutionary relationships and for facilitating the detection of functional and structural genes [27,43] by matching mapped regions in one species against unmapped regions in another species [13]

In our original study of high-performance techniques for sequence comparison, we focused on one of the foundational sequence analysis tools, the dot plot. The dot plot is a matrix visualization that illustrates similar regions between two sequences. Our implementation used the AltiVec SIMD instruction set to compute the matrix using a data parallel algorithm and also divided the work for coarse-grained parallel execution. This work helped identify strategies that we will apply to the synthetic BLAST implementation.

BLAST is a multi-step algorithm that searches a database of sequences for matches to a query sequence. It performs a simple, statistical comparison of short sequence segments, called words. If two words meet a certain threshold a more expensive alignment is performed to extend the matched region. Additional computations determine the significance of the aligned regions and the final output includes a full Smith-Waterman [47] alignment of the region. Our strategy for the synthetic BLAST implementation will focus on memory access patterns and SIMD algorithms for both the initial search and the alignment extension steps. Additionally, we will use coarse-grained parallelism to process multiple database sequences at once.

In addition to our dot plot algorithm, a few other SIMD algorithms have been for bioinformatics applications.<sup>5</sup> A version of BLAST developed by Apple and Genentech, AGBLAST [19], uses Apple's Velocity Engine to enhance BLASTN alignments. For large nucleotide word sizes, AGBLAST attains a 5x speedup over the standard implementation. However, the results are highly dependent on the word size and the improvement for more sensitive searches is not as dramatic. Erik Lindahl has also reported [28] an AltiVec enhanced version of HMMER that is 4-6 times faster than the standard C version. A data-parallel version of the Smith-Waterman dynamic programming algorithm for finding optimal local alignments was presented in [42]. Using the MMX vector unit in Intel processors [22], a 6x speedup was achieved.

Parallel implementations of BLAST have also been developed. The main implementation is mpiBLAST [11], which distributes the database across the available processing nodes. ScalaBLAST [37] improves on the mpiBLAST by introducing a pre-fetching system for latency hiding across cluster nodes. The prefetcher schedules database sequence movement while other sequences are being processed, eliminating

---

<sup>5</sup>By SIMD, we refer to data parallel vector processors found in modern workstations, not special purpose SIMD machines such as those from MasPar and Thinking Machines.

a key bottleneck in mpiBLAST. BLAST has also been ported to more exotic architectures, including the Blue Gene/L [41]. This work will help guide our synthetic implementation for the Cell BE.

### 4.3.2 Chemical Database Searching

In contrast the bioinformatics tools, most cheminformatics applications are proprietary and closed source systems. While the algorithms and use-cases are well known, specific implementations are not. Thus, little is known about the underlying techniques used to extract high levels of performance for chemical database searching. Our pilots study on a SIMD implementation of the Jaccard/Tanimoto appears to be the first publicly available description of the algorithm. However, the kernels for most chemical database search routines, including the Tanimoto metric, are closely related to common DSP algorithms, which have many known mappings to SIMD processors, e.g. [36].

### 4.3.3 Graph Algorithms

Graph problems are typically optimized at the algorithm level by selecting a different algorithm or more efficient data structures. Research using the Boost Graph Library [45] shows that the simple, low-level optimizations that C++ templates allow compilers to perform via type-based polymorphism can have a significant impact on performance [18]. This result suggests that even traditional high-performance techniques may be applicable to a broader range of graph problems.

Beyond these techniques, few other techniques have been explored for specializing graph algorithms based on the hardware. The Parallel BGL [18] has begun collecting parallel implementations of graph algorithms that focus on distributed memory algorithms for graphs. Some SIMD algorithms for graphs have been explored for traditional SIMD architectures, where a large amount of processing units are available [44, 48], but no algorithms exist for modern SIMD processors.

## 5 Schedule and Risk Assessment

This primary work for this thesis will be performed between March 2006 and April 2007, with the final thesis presentation targeted for May 2007. The general research schedule is shown in table 1. This is an aggressive schedule for the scope of work proposed and will be adjusted on an on-going basis to ensure we maintain a high-level of quality in our study while providing results suitable for a PhD level dissertation.

Our core design study is near completion and the results suggest that synthetic programming is not only a viable tool for high-performance programming but also an effective tool. However, this project still contains some key risks. Our study of the utility of the SPE on novel platforms depends on the IBM's Cell BE. The Cell BE is a new technology with limited availability. Our ability to study synthetic programming in this environment depends on support from IBM. To this point, IBM has been helpful in providing support, but this still remains a risk.

<b>Project</b>	<b>Planned Dates</b>	<b>Risk</b>
<b>PPC SPE</b>	March-May 2006	Complete
<b>Basic Opts</b>	May-June 2006	Complete
<b>Add'l Opts</b>	Sept 2006-April 2007	Low
<b>Cell SPE</b>	August-Sept 2006	Med
<b>SynBLAST</b>	Sept-Nov 2006	High
<b>Chem. Database</b>	April 2006, Jan 2007	Low
<b>Syn Graph</b>	Jan-April 2006	Med
<b>Final Results</b>	May 2007	Med

Table 1: Research Schedule and Risk Assessment

Of the application studies, the BLAST study represents the highest amount of risk. BLAST is a complex, multi-stage algorithm. Our intent with this study is to understand how synthetic programming can be used to minimize the development costs of such algorithms by using high-level languages for most of the work. BLAST is by far the most complicated application we will attempt and we believe that the risk is worth it for the insights into synthetic programming it should provide.

The graph applications present a minor risk, primarily because they are poorly specified at the moment. However, we will seek to minimize this risk over the next few months as we work with Lilly to identify the exact application and optimizations we will study.

To ensure the final results are available on schedule, incremental results will be continually documented in research papers.

## References

- [1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.
- [2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215:403–10, 1990.
- [3] Apple Computer, Inc, [http://developer.apple.com/hardwaredrivers/ve/code\\_optimization.html](http://developer.apple.com/hardwaredrivers/ve/code_optimization.html). *Apple Developer Connection: Code Optimization*, Accessed 2006.
- [4] John Aycock. A brief history of Just-In-Time. *ACM Computing Surveys*, 35(2):97–113, June 2003.
- [5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.
- [6] 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.
- [7] A. Berns. Gene expression in diagnosis. *Nature*, 403:491–492, 2000.
- [8] 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.
- [9] 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.
- [10] R. Dalton. Diy(do it yourself) microarrays promise dna chips with everything. *Nature*, 403(234), 2000.
- [11] A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. In *4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with the ClusterWorld Conference & Expo, San Jose, CA*, June 2003.
- [12] High Performance Fortran Forum. High performance fortran language specification. version 2.0. Technical report, Rice University, 1992.
- [13] K. A. Frazer, L. Elnitski, D. M. Church, I Dubchak, and R. Hardison. Cross-species sequence comparisons: A review of methods and available resources. *Genome Res.*, 13(1):1–12, 2003.
- [14] Freescale Semiconductor, Technical Information Center, CH370 1300 N. Alma School Road Chandler, Arizona 85224. *AltiVec(TM) Technology Programming Environments Manual*, 3 edition, April 2006.

- [15] Matteo Frigo. A fast fourier transform compiler. *SIGPLAN Not.*, 39(4):642–655, 2004.
- [16] 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.
- [17] 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.
- [18] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.
- [19] Apple Advanced Computation Group. *Apple/Genentech BLAST*. Apple Computer, Inc., Cupertino, CA, 2002.
- [20] C. E. Barry III and B. G. Schroeder. Dna microarray: Translation tools for understanding the biology of mycobacterium tuberculosis. *Trends in Microbiology*, 8(5):209–210, 200.
- [21] National Cancer Institute. Nci open database compounds. Online: <http://cactus.nci.nih.gov/ncidb2/download.html>, 2000.
- [22] Intel Corporation. *A-32 Intel Architecture Software Developer’s Manual, Volume 1: Basic Architecture, IA-32 Intel Architecture Software Developer’s Manuals*, 2004.
- [23] International Business Machines Corporation, IBM Microelectronics Division 1580 Route 52, Bldg. 504 Hopewell Junction, NY 12533-6531. *PowerPCTM-Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, February 2000.
- [24] International Business Machines Corporation, IBM Systems and Technology Group, 2070 Route 52, Bldg. 330, Hopewell Junction, NY 12533-6351. *Cell broadband engine architecture*, August 2005.
- [25] Eric Jones. *Weave User’s Guide*. Enthought. Accessed May 2006.
- [26] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, , and J. Mellor-Crummey. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *IPDPS ’00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 297, Washington, DC, USA, 2000. IEEE Computer Society.
- [27] W.J. Kent and A.M. Zahler. Conservation, regulation, synteny, and introns in large-scale c. briggsae-c.elegans genomic alignment. *Genome Res.*, 10(8):1115–1125, 2000.

- [28] E. Lindahl. *Altivec HMMer*. The Lindahl Lab, January 2005.
- [29] C. A. Lipinski, F. Lombardo, B. W. Dominy, and P. J. Feeney. Experimental and computational approaches to estimate solubility and permeability in drug discovery and development settings. *Adv. Drug Del. Rev.*, 46(3–26), 2001.
- [30] Kenneth C. Louden. *Compiler Construction: principles and practice*. PWS Publishing Company, Boston, MA, 1997.
- [31] Michael D. McCool, Zheng Qin, , and Tiberiu S. Popa. Shader metaprogramming. In *SIGGRAPH/Eurographics Graphics Hardware Workshop*, pages 57–68, September 2002.
- [32] Christopher Mueller, Douglas Gregor, and Andrew Lumsdaine. A lightweight pattern for managing distributed computational experiments. 2006.
- [33] Christopher Mueller and Andrew Lumsdaine. Expression and loop libraries for high-performance code synthesis. 2006.
- [34] Christopher Mueller and Andrew Lumsdaine. Runtime synthesis of high-performance code from scripting languages. In *Dynamic Language Symposium (DLS2006)*, Portland, Orgeon, October 2006.
- [35] NCBI. Pubchem. Online: <http://pubchem.ncbi.nlm.nih.gov>.
- [36] Huy Nguyen and Lizy Kurian John. Exploiting simd parallelism in dsp and multimedia algorithms using the altivec technology. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 11–20, New York, NY, USA, 1999. ACM Press.
- [37] C. Oehmen and J. Nieplocha. ScalaBLAST: A scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):740–749, August 2006.
- [38] Grant Olson. *PyASM User's Guide V.0.2*. Accessed May 2006.
- [39] 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.
- [40] Dan Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2-3):215–226, 2000.
- [41] Huzefa Rangwala, Eric Lantz, Roy Musselman, Kurt Pinnow, Brian Smith, and Brian Wallenfelt. Massively parallel BLAST for the Blue Gene/L. In *High Availability and Performance Computing Workshop*, October 2005.
- [42] T. Rognes and E. Seeberg. Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(699–706), 2000.

- [43] S. Schwartz, Z Zhang, K.A. Frazer, A. Smit, C. Riemer, J. Bouck, R. Gibbs, R. Hardison, and R. Miller. Pipmaker: A web server for aligning two genomic dna sequences. *Genome Res.*, 10(4):577–586, 2000.
- [44] Tsan sheng Hsu, Vijaya Ramachandran, and Nathaniel Dean. Implementation of parallel graph algorithms on a massively parallel simd computer with virtual processing. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 106–112, Washington, DC, USA, 1995. IEEE Computer Society.
- [45] Jeremy Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.
- [46] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A unifying framework for numerical linear algebra. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 466–467, London, UK, 1998. Springer-Verlag.
- [47] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(195–197), 1981.
- [48] Francis Suraweera and Prabir Bhattacharya. A parallel algorithm for the minimum spanning tree on an simd machine. In *CSC '92: Proceedings of the 1992 ACM annual conference on Communications*, pages 473–476, New York, NY, USA, 1992. ACM Press.
- [49] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.
- [50] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.
- [51] Todd L. Veldhuizen. Arrays in blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
- [52] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.**  
URL: [http://www.cs.utsa.edu/~whaley/papers/atlas\\_sc98.ps](http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps).
- [53] John Willett. *Similarity and Clustering in Chemical Information Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1987.
- [54] P. Willett, J.M. Barnard, and G.M. Downs. Chemical similarity searching. *Journal of Chemical Information and Modeling*, 38(6):983–996, 1998.