SYNTHETIC PROGRAMMING: USER-DIRECTED RUN-TIME CODE SYNTHESIS FOR HIGH PERFORMANCE COMPUTING

Christopher E. Mueller

Submitted to the faculty of the University Graduate School in partial fulfillment of the requirements for the degree Doctor of Philosophy in the Department of Computer Science Indiana University May 25, 2007
Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Andrew Lumsdaine, Ph.D.

Mehmet Dalkilic, Ph.D.

Dennis Gannon, Ph.D.

Arun Chauhan, Ph.D.

May 25, 2007
# Contents

Chapter 1. Introduction 2
  1. Scripting Languages and High-Performance Computing 3
  3. The Many Core Revolution 9
  4. Synthetic Programming 10
  5. Evaluation and Contributions 12
  6. Conclusion 13

  1. Run-time Code Generation 16
  2. Meta-Programming 19
  3. Special Topics: DSLs and Auto-Tuners 21
  4. Conclusion 23

Chapter 3. The Synthetic Programming Environment 24
  1. Design Goals 24
  2. Design Methodology 26
  3. The Synthetic Programming Environment 27
  4. ISAs and Instructions 28
  5. InstructionStream 33
  6. Processor 35
Abstract

Scripting and interpreted languages are important tools for software engineering and are often used in place of compiled languages for application development. While they enable a high level of developer productivity, their run-time environments limit the overall performance attainable with any given application. To develop performance-critical applications, developers continue to rely on compiled languages. However, recent changes in microprocessor design have begun to push the limits of compiled languages, making it difficult for developers to fully utilize the available hardware resources.

This thesis presents Synthetic Programming, a new approach to generating high-performance code and developing parallel applications from scripting languages. Synthetic programming is based on the synthetic programming environment, a collection of libraries that provide direct access to low-level processing resources from the Python programming language. Using the synthetic programming environment, developer’s can create applications using machine-level instructions directly from Python, enabling a number of optimization strategies not readily available when using compiled languages.

In this thesis, synthetic programming is evaluated for feasibility and applicability. To demonstrate the feasibility of synthetic programming, the synthetic programming environment was developed for three complete instruction sets (PowerPC, AltiVec/VMX, Cell BE SPU) and two operating systems (OS X and Linux). Using the synthetic programming environment on G5 and Cell Broadband Engine-based systems, synthetic programming is demonstrated to be applicable for a broad range of common high-performance problems from SIMD programming to automated loop optimization and multi-core application development.
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 enable a high level of developer productivity through language features designed for usability and large collections of libraries for common tasks. While essential for general application development, scripting languages are based on run-time environments that do not lend themselves to high-performance computing. To develop performance-critical applications, developers continue to rely on legacy compiled languages that lack many of the productivity enhancements available in more modern languages.
This thesis explores a new approach to generating high-performance and parallel code from scripting languages that bridges the gap between performance and productivity. Synthetic Programming provides developers with a collection of libraries that enable fine-grained control over code generation and execution, providing the tools necessary to build high-performance kernels without resorting to compiled languages. The core synthetic programming library, the synthetic programming environment, is a refactoring of assembly language programming for run-time code synthesis from scripting languages [47]. Libraries built on the synthetic programming environment provide abstractions for generating high-performance code and include components for loop optimizations and SIMD algorithm development [46]. Synthetic programming also introduces new methods for developing parallel applications on multi-core processors. By treating high-performance code sections as sub-programs, rather than sub-routines, synthetic programming encourages developers to organize their applications using parallel patterns from the beginning.

The next sections provide the contextual motivations for synthetic programming, first discussing the state of high-performance computing with scripting languages and continuing with a summary of two challenges facing developers of high-performance applications. The final sections introduce synthetic programming and outline the main research questions and contributions addressed in this thesis.

1. Scripting Languages and High-Performance Computing

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 [51, 53]. 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 [7, 11, 59]. While effective, this approach complicates the development process by adding additional layers of abstraction and extending the development tool chain (Figure 1).

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 applications, such as games and vendor-optimized math libraries, many 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-level abstractions or meta-programming features that allow developers to manage complexity for larger applications (e.g., templates in C++, macros in C and Lisp, reflection in Java and Python).

For developers using scripting languages, dropping into assembly is rarely an option. The few in-line systems for scripting languages (e.g., [34, 50, 68]) 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.

For common application areas, compiled language libraries exist for most scripting languages. For example, Numeric Python [7] has for years been the standard library for array and matrix operations from Python. Numeric Python provides a simple, Matlab-like interface that is easy to use and provides an much higher level of performance than equivalent operations developed entirely in Python. However, libraries such as Numeric Python are limited by their designs to specific domains. Using them in an application requires designing the application around the libraries. These libraries also have their own performance challenges that must be taken into account when using them. Numeric Python performs
1. INTRODUCTION

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 development process. The Synthetic Programming Environment replaces the compilation tools with a simple machine-code generator and collection of targeted code generation components, giving developers direct access to the processor from high-level languages (bottom).

Each operation on the entire array before proceeding to the next operation, generating large temporary arrays and causing unnecessary memory transfers for composite operations. Design limitations such as these will always keep compiled libraries from being true general purpose high-performance tools for scripting languages.

DRAFT: May 23, 2007
The mix of compiled languages, scripting languages, and libraries is becoming a development and cognitive bottleneck for high-performance computing. To effectively manage non-trivial high-performance applications, developers must be experts in multiple languages and libraries and also understand the consequences of specific hardware architectures. Synthetic programming begins addressing this problem by removing all but the extreme ends of the toolchain, scripting languages and machine-level programming. With the starting points identified, synthetic programming builds a new model for high-performance computing that leverages the productivity and language features of scripting languages while giving developers direct access to the underlying hardware.

Abandoning compiled languages for high-performance computing is a radical step. However, compiled languages themselves are not necessarily the panacea they claim to be for high-performance computing.

2. Atlas Shrugged?

While they are the standard tools for high-performance software development, compiled languages themselves face new challenges 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 [9, 56]. 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 [21], if not impossible, and as a result, most compilers cannot generate high-performance code without significant guidance from the developer [15].

To help illustrate the challenges faced by modern compilers, consider the case of matrix-matrix multiplication, probably the most studied problem in high-performance computing. The core algorithm for matrix-matrix multiplication is a simple triply nested loop. For three arrays, A (n x m), B (m x p), and C (n x p), a naïve implementation in C can be expressed as:

\[
\text{for}(i=0; i < n; i++)
\]
\begin{verbatim}
for(j=0; j < m; j++)
    for(k=0; k < p; k++)
        C[i*p + k] += A[i*m + j] * B[j*p + k];
\end{verbatim}

While this clearly expresses and illustrates the algorithm, it is far from a high-performance implementation. Simple optimizations, such as lifting $A[i*m + j]$ out of the innermost loop and pre-loading the value, can have profound impacts on its performance. But, even with a well-designed C algorithm, the overall performance is still limited by the final code the compiler generates.

Matrix-matrix multiplication is one of the few algorithms that can attain near-peak performance on a given computer system. When properly mapped to the hardware, it can aggressively exploit the memory hierarchy to reuse data in memory, L1-cache, and registers, utilize SIMD instructions to update multiple values in one instruction, and make progress towards the solution with almost every clock cycle. However, coaxing a C or Fortran compiler to generate the most efficient instruction sequence from user supplied code is nearly impossible.

Most compilers are written to be portable across a range of architectures. To make the compilation process as portable as possible, most optimizations are applied on abstract representations of the code before the code is committed to a specific platform \cite{41}. The code generation stages are far removed from the actual representation supplied by the user. While the early transformation stages may be able to recognize specific implementations of the algorithm and optimize accordingly, there are still many representations a user may discover, making it impossible for the compiler to properly optimize user-supplied code in every instance.

Because matrix-matrix multiplication is an important operation in scientific computing and system benchmarks, many optimized libraries exist that provide sub-routines for the operation. The standard library for matrix operations is the BLAS library \cite{20}. There are two common approaches for implementing fast BLAS on a given platform. The first is to have an expert implement assembly language versions of all the functions in the library. This avoids problems introduced through language abstractions and enables the use of
techniques specific to a single processor version. Of course, this is time consuming and requires a human programmer to carefully design and optimize the code.

An alternate strategy introduced with the Atlas library [69] is to auto-tune BLAS using the platform’s compiler in conjunction with an objective optimization algorithm. The Atlas auto-tuner takes short code segments that implement the algorithms and mixes and matches them until a fast implementation is automatically identified. This removes the human from the process and allows developers to acquire fast algorithms for their specific hardware environment.

A hybrid approach, used by Goto-BLAS [26], is to use a combination of hand-tuned assembly code and a parameter search. The Goto-BLAS algorithms are built around blocked algorithms that operate on matrix blocks at the memory and register level. The parameter search finds the best sizes of the blocks for a given architecture while the assembly code fine tunes the inner loop instructions. This approach generally achieves the highest performance.

This discussion reveals some interesting aspects of high-performance computing. First, while languages like C and Fortran are used for high-performance computing, to achieve true high-performance on the most studied problems, developers still resort to programming at the machine level. What is more troubling, however, is the amount of effort required to optimize a simple program in the general case. Hand-coded assembly, using the traditional tools, is a time-consuming processes that does not lend itself to rapid development. Auto-tuners, while effective, are themselves complex software applications that require maintenance and additional development to adapt to new platforms. And, even with an optimized matrix-matrix multiplication, there is no guarantee a developer’s problem will work with the interfaces supplied by the library. The cost of converting data formats or application architectures may be too high to justify using the optimized version.

Synthetic programming was designed to address these challenges. Access to the hardware is essential for creating high-performance code. The synthetic programming environment is anchored around a library for exposing the hardware directly to the developer. Programming at the hardware level has traditionally been a time-consuming process. By
using a high-productivity scripting language as the host for the synthetic programming environment, synthetic programming makes rapid development of high-performance code for specific platforms possible. Scripting languages also greatly simplify the task of converting data and refactoring applications, taking the sting out of integrating high-performance code into an existing application.

3. The Many Core Revolution

While the mismatch between language expressiveness, instructions sets, and system architectures has always challenged high-performance application developers targeting commodity hardware, a recent shift in processor design is bringing another important problem into the spotlight. To help account for the slowing increase in clock speeds, processor designers are starting to scale out processors to include multiple processing units in a single chip [38, 70]. These so-called multi-core processors represent the first time commodity computers are inherently parallel computers. And, following the trend started in the mid-1990s, most commodity workstations include multiple processors, leading to “many core” systems [39] that combine multi-core processors, graphics processors, and other dedicated processors in a single system that with tremendous processing power. Developing applications that utilize the parallel execution environments in many core systems is difficult.

Despite decades of research into parallel computing, there is no generally accepted language or paradigm for developing parallel applications. While there are many possible reasons, the simplest explanation is that there has never been a compelling need to provide a practical solution to the problem. Parallel computing has traditionally been limited to small communities that are comfortable with small, targeted solutions such as MPI or OpenMP. The broader community of computer programmers has never had users with parallel hardware and thus never needed a solution.

The many core revolution is changing that. For the first time, every computer sold by a Apple, Inc., is a parallel computer [5]. As these systems become more widely deployed, more users will expect applications to fully utilize the processors. But, as noted above, there is no easy way to development parallel applications. The existing paradigms are restricted to
small problem domains, and few high-level languages support parallelism beyond providing threading APIs. To support the general class of applications, new paradigms for parallel programming are required.

As with machine-level programming, synthetic programming takes a new approach to parallel programming, with a focus on application development for multi-core processors. Instead of treating synthetic programs as sub-routines or function calls, synthetic programming treats synthetic programs as sub-programs. While subtle, the distinction is important for framing how developers approach application architecture. As sub-programs, synthetic programs can easily be considered to execute concurrently with the main program. In contrast to thread libraries and other parallel programming models, synthetic programming stops short of adding synchronization and memory consistency models and instead focuses only on execution control for synthetic programs. Decoupling execution from data management frees developers to explore methods for data consistency other than the methods encouraged by the execution library. While not necessarily the final solution for all multi-core programming problems, synthetic programming provides a new foundation for exploring the problem space.

4. Synthetic Programming

Synthetic programming was developed to provide a new approach to high-performance and multi-core computing using scripting languages [47]. The design of the synthetic programming environment encourages developers to experiment with different approaches for mapping algorithms to processing resources and generating optimized code sequences.

Synthetic programming itself is a methodology for building applications that combine high-level language code with user generated machine code. The synthetic programming environment exposes the underlying processor instructions as first-class functions in the host language and provides components for building and executing instruction sequences built using the instruction interfaces. The instruction sequences, called synthetic programs, can contain any instruction available for the processor, allowing developers to create highly optimized kernels for high-performance tasks. An application can contain many synthetic
programs, all of which can be executed an arbitrary number of times synchronously or asynchronously, allowing the application to make full use of data and processing resources.

At the most basic level, the synthetic programming environment is a complete replacement for the assembly language toolchain. Instead of an assembly language, the synthetic programming environment supplies a library that includes functions for all machine instructions. Mnemonics, typically used to select between various forms of a given instruction, are discarded in favor of keyword arguments that modify bit flags on the instruction, tying the developer’s intentions directly to the instruction call. The assembler is superseded by a run-time component that collects instructions and renders them directly to memory as they are created, by passing intermediate file formats. Using just the these components of the synthetic programming environment, developers can create complex synthetic programs in much the same way they would use assembly to optimize a C program. However, the synthetic programming environment, used in conjunction with the host language, gives developers more opportunities for specializing performance-critical code based on the run-time requirements of the application than is possible with an assembly-based approach.

While designed to be more accessible than assembly languages, working with the base synthetic programming libraries can still be as verbose and error prone as raw assembly programming. The real benefits of synthetic programming start to emerge as components are built on top of the base libraries. Called synthetic components, these components abstract specific code generation or optimization and can considered micro-compilers for specific tasks. Synthetic components have distinct advantages over general purpose languages and compilers. Most significantly, synthetic components can take advantage of run-time characteristics of the data and environment when generating code, enabling optimizations not possible using a statically compiled library. Component library designers also have more freedom when mapping domain specific constructs to the hardware, which is particularly useful in cases where processor features have outpaced the abilities of optimizing compilers.
The synthetic programming environment and synthetic components, along with the host language, form the raw materials for rapid high-performance application development. When used on real problems, synthetic programming has been demonstrated to be an effective approach to development.

5. Evaluation and Contributions

This thesis evaluates synthetic programming from two perspectives: feasibility and applicability. To be feasible, synthetic programming requires a run-time environment that can generate and execute arbitrary instructions sequences on a given platform from a scripting language. Feasibility also requires that synthetic programming be platform neutral. While any particular application will be platform specific, the run-time environment and abstractions must not differ significantly from platform to platform.

With a run-time environment in place, the next level of evaluation determines whether or not synthetic programming is applicable to a broad range of common high-performance applications. Synthetic programming must be demonstrated to be more flexible than hand coding assembly programs. It should provide abstractions and patterns that simplify machine-level programming and provide services not readily available from compiled languages. In this regard, it must be demonstrated that complex kernels and applications can be developed using the synthetic programming environment that take advantage of the run-time nature of synthetic programming.

The results presented in subsequent chapters demonstrate both the feasibility and applicability of synthetic programming, with Chapter 3 fully demonstrating its feasibility and the remaining chapters providing detailed examples of how synthetic programming can be applied. The main contributions, broken out by chapter, are:

- Chapter 2 introduces many of the concepts that inspired synthetic programming along with surveys of similar techniques.
- Chapter 3 details the implementation of the synthetic programming environment across three different instruction sets – PPC, Altivec/VMX, and Cell SPU – on two operating systems – Linux and OS X. The synthetic programming environment is
demonstrated to be platform neutral from the developer’s perspective, with platform differences evident only in the choice of instruction set.

- Chapter 4 introduces synthetic programming as a paradigm for software development. It defines the terminology and describes the basic procedures used to develop applications using the synthetic programming environment.
- Chapter 5 shows how synthetic programming can borrow features from the host language to present abstractions that hide the details of machine-code generation using common programming syntax for both sequential and parallel applications.
- Chapter 6 details three case studies of progressive specialization with synthetic programming and compares synthetic implementations of the applications with compiled language versions. The applications are a array min kernl, matrix-matrix multiplication, and a chemical fingerprint comparison program.
- Chapters 7 discusses the implementation of a particle system that uses a synthetic program for targeted optimization of the physics loop and shows how to use synthetic programs in conjunction with external high-performance libraries.
- Chapter 8 describes a novel architecture and implementation of the BLAST algorithm for the Cell BE processor using synthetic programming. The new algorithm is the first true multi-core decomposition of BLAST and the first implementation to utilize the Cell BE’s SPU processors.
- Chapter 9 shows how the combination of run-time code generation and interactive development tools can be used to provide novel ways of interacting with the processor and debugging applications. A fully interactive GUI for executing individual instructions on the Cell BE SPU is presented along with a very compact debugger for SPU synthetic programs.

6. Conclusion

Synthetic programming represents a departure from traditional high-performance application development. The tools and techniques presented in this thesis combine the best practices in rapid application development with techniques for low-level code generation
to address the needs of users developing high-performance kernels and applications designed in inherently parallel environments. By examining low-level programming in the context scripting languages, synthetic programming presents a new paradigm for developing high-performance and parallel programming that removes the dependency on the complex collection of legacy tools currently used in this space.
Synthetic programming has its roots in two approaches for constructing optimized applications: run-time code synthesis and library-supported meta-programming. Run-time code synthesis tools generate new machine code at run time to optimize applications. They can work behind the scenes, as with just-in-time compilers, or provide libraries that allow developers to integrate run-time information into the code generation process. Meta-programming is a more general technique that covers all aspects of developing programs that generate programs. Most meta-programming systems provide templates for algorithms or common operations and generate new source-language code prior to the machine code
In addition to run-time code synthesis and meta-programming, there are some special-purpose techniques that are related synthetic programming. These include domain-specific languages and auto-tuners, both of which have had an important research impact on developing high-productivity environments for high-performance and scientific computing.

This chapter reviews many approaches to run-time code synthesis and meta-programming and their influence on synthetic programming and provides an introduction to additional topics that are related to the goals of synthetic programming.

1. Run-time Code Generation

Run-time code generation, also known as dynamic code generation, is an active area of research covering many different approaches to optimization. In most cases, run-time code generation is a behind-the-scenes process, with very little interaction with or direction from the developer. As a result, the semantic information used to optimize blocks of code or instructions is limited to the information passed directly to the system. In some cases this is limited to the machine code of the program, while other systems allow the developer to provide hints through library calls or code construction APIs.

Many run-time code generation systems have internal libraries that are similar to the modules supplied by the synthetic programming environment. However, prior to the development of the synthetic programming environment, no systems made a concerted effort to expose the entire processor and operating system run-time environments at the library level for general use by developers.

This section explores different approaches to run-time code generation, starting with low-level approaches and proceeding to techniques that give the user more control over the optimization process. It concludes with a discussion of inlining systems and their relation to synthetic programming.

1.1. Just-in-Time Compilation. The most common type of run-time code generation comes in the form of just-in-time compilers, or JITs [8]. 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 signifi-
cantly from a JIT. In the synthetic programming, while the code is generated at run-time,
its generation is directed by the developer, who may have more semantic information avail-
able to direct optimization.

1.2. Dynamic Compilation. Dynamic compilation frameworks are libraries for con-
structing workflows that are compiled at run-time. The workflows tend to be domain spe-
cific, often targeted at matrix operations or image processing kernels. Dynamic compilation
frameworks combine code generated at compile-time with code generated at run-time to
optimize workflows based on the run-time characteristics of the data and environment.
Typically designed to be platform neutral, these frameworks target general classes of opti-
mizations that do not depend on specific hardware requirements.

A few full-featured dynamic compilation frameworks have been developed for C. DyC [27]
is a 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 gener-
ation system. ‘C (pronounced tick-C) [52] 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 the notion of synthetic components, is the TaskGraph
Library for C++ [12]. 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.

libsh [43, 44] and its commercial incarnation RapidMind [1] are run-time compilation tools for generating kernels that run on graphics cards and the Cell BE’s SPUs. Along with the synthetic programming environment, they are the only run-time code generation systems available for the Cell BE. RapidMind is a C++ library that provides a vocabulary of parallel operations and patterns. The “nouns” and “verbs” are objects and methods that allow the developer to provide suggestions for what code can be optimized for parallel execution. RapidMind’s internal technology is similar to CorePy, but designed primarily for use only by RapidMind developers [64].

1.3. Inlining Systems. Inlining systems are libraries or preprocessors that allow developers to include code from another language in a host language. The most common type of inlining system is the inline assembler found in most compiled languages. Inline assemblers allow developers to include assembly instructions directly in high-level languages, such as C or Fortran, and share data between the host and assembly languages. Inline assembly code is compiled directly into the application and is used primarily to access processor features not directly available through language or to optimize short operations.

Multiple inlining systems have been developed that allow developers to access other languages directly from Python. The two most notable systems are Weave [34] and PyASM [50]. Weave allows the developer to inline 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 inlining language. PyASM is essentially an inline assembler for Python.

Inlining systems differ from the synthetic environment in an important and fundamental manner. Inlining systems rely on an intermediate language for expressing new code. These can encumber the run-time system with additional dependencies 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.

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 many high-level programming models. This discussion constrains 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.

2.1. C++ Template Meta-Programming. C++ templates have had a large influence on high-performance computing. Templated functions and objects let users 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 [67] that can help guide their optimization using a technique called template meta-programming. The original template meta-programming library, Blitz++ [66], 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) [62] and the Boost Graph Library (BGL) [61]. 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 [24] 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 code that is difficult to read. 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.

2.2. Macro Systems. 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. The most common use of macros in high-performance environments is to abstract simple code sequences that should always be inlined, 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. Their popularization of dynamic run-time systems make run-time specialization and code generation possible. Ableson and Sussman’s classic programming language textbook [2] introduces many of the core ideas that form the foundations for advanced macro systems and run-time code manipulation. While these languages are rarely associated with high-performance computing, their influence on synthetic programming is evident in synthetic programming’s treatment of code as run-time data.

1While C++’s template system is technically Turing complete, using it as a complete macro language is not practical.
The synthetic programming environment leverages Python's dynamic run-time system 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.

3. Special Topics: DSLs and Auto-Tuners

Run-time code synthesis and meta-programming provide the raw intellectual materials for developing more complicated optimization systems. The most general application of these ideas has been in the realm of domain-specific languages: small, restricted languages meant to capture the semantics of a particular application domain. Auto-tuners take a wildly different approach to generating optimized libraries. Rather than relying on rules and algorithms, they separate algorithms into kernels and perform an exhaustive combinatorial search using meta-programming techniques to find optimal implementations of algorithms on specific platforms.

3.1. Domain-Specific Languages. The domain-specific language [65] (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 [14] implements a streaming language as an extension of ANSI C, and Sh [45] 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, 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, etc.).

Another important class of DSLs is related to Telescoping Languages [35]. 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 [54]. 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.

3.2. Auto-Tuners. Auto-tuners, while currently limited to a small collection of libraries, are an influential technique for optimizing code. Auto-tuners are based on the idea that while algorithms are should be platform independent, their optimal implementation must be platform dependent. Auto-tuners take an existing algorithm and break it into a number of small code segments that represent basic operations. When the algorithm is installed on a platform, the code segments are combined in different ways to find the ordering of segments that produces the most efficient code on the target platform.

The Atlas [69] and FFTW [23] systems are adaptive code generation tools for 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 for user-designed abstractions, adaptive generators, including ones modeled after Atlas and FFTW are possible applications for synthetic components.

4. Conclusion

Synthetic programming is built on many ideas from borrowed from different areas of computer science for many different applications. Whereas most approaches attempt to hide the complexity of the platform from the user, the synthetic programming environment embraces the hardware platform and provides a set of building blocks that can be used by any developer to explore optimization strategies and high-performance library design. This fundamental difference sets synthetic programming apart from all previous approaches and has the potential to democratize code synthesis and optimization research.
The Synthetic Programming Environment

Synthetic programming is supported at the language level by a synthetic programming environment. The synthetic programming environment is responsible for providing access to the processor’s instruction set and the system’s execution environment along with supplying collection of code generation libraries to abstract run time code synthesis. This chapter presents the design and implementation of the synthetic programming environment for Python.

1. Design Goals

In order to be a generally viable solution for run-time code synthesis scripting languages, the synthetic programming environment must meet a few criteria. First, with the exception of the each supported processor’s instruction set, the components that make up the synthetic
programming environment must be portable at the framework level. That is, to use the synthetic programming environment on a new platform, a user should only be required to learn the new instruction set. The actual code synthesis, execution methods, and data sharing strategies should be the same across platforms.

Next, the implementation of the synthetic programming environment should rely on a minimal amount of platform-specific code to support the code generation and execution costs. The platform-specific code should be isolated in well-defined components within the framework. This ensures that the synthetic programming environment can extended to support new platforms with minimal development effort.

Finally, the synthetic programming environment should be implemented in the host language and should respect and utilize common idioms of that language. At the simplest level, this requires the synthetic programming environment use language features, such as functions or objects, to expose the machine instruction sets, rather than simply creating another assembly language. At a more fundamental level, the underlying support code, such as ABI compliance, should be implemented in the host language and not a compiled language. While some features will inevitably require compiled implementations, reliance on compiled code should be minimal. This requirement serves three purposes. First, on emerging platforms such as IBM’s Cell BE, compiler support is limited. Removing the reliance on compiled language for optimization or other advanced features ensures that the compiler tool chain does not become a portability bottleneck. More importantly, synthetic programming is designed to use scripting languages for the entire code synthesis process. Relying too heavily on a compiled language for functionality negates the general thesis that scripting languages are sufficient for the task of run time code synthesis. Finally, using the host language for most functionality removes the cognitive dependency on external compiled languages, allowing developers without experience with systems level languages to explore the entire code synthesis stack.
2. Design Methodology

To help ensure these goals were met, the synthetic programming environment was developed using an agile development process designed to evolve the design quickly by integrating requirements identified through actual system and application requirements. Requirements for the synthetic programming environment came from two primary sources: platform-specific implementations and applications built on top of the synthetic programming environment. The synthetic programming environment was developed using small iterations that focused on the requirements of the current application or platform. This approach allowed the design to evolve as requirements were identified without becoming overwhelmed by features that were not motivated by any particular use case. It also ensured that design changes were instituted for specific reasons.

While the synthetic programming environment evolved continuously over the course of its implementation, its design path can be broken into a three major phases, with different platforms and applications providing the requirements in each phase.

The first phase consisted of the initial implementation, simply called the SPE, for the PowerPC and AltiVec instruction sets on G4 and G5 processors running OS X. The primary applications used to drive the development were the particle system (Chapter 7) and chemical fingerprint (Chapter 3) applications. In this phase, the basics concepts of user defined semantics, synthetic programs, and data sharing using memory regions were developed.

The next phase focused on scaling the SPE to new platforms and building more complex applications using synthetic programming techniques. In this phase, the SPE was ported IBM’s Cell BE processor, requiring ABI support for Linux, along with support for a custom run-time system on the Cell BE’s synergistic processing element vector processing units. The application that drove the Cell BE design was a vectorized, multi-core implementation of the BLASTP algorithm.

The final phase unified the PowerPC and Cell BE SPE implementations under a common framework and finished the design of the component libraries. The final implementation of the synthetic programming environment is the CorePy library for Python. CorePy meets
the original goals of a interface and implementation-portable library implemented primarily using the host language.

The remainder of this chapter presents the design and implementation of the ISA, processor, and instruction sequence components of the synthetic programming environment for Python. The next chapter covers the code synthesis components, while the remaining chapters detail the applications. Important decisions are presented from the perspectives of both the SPE and CorePy implementations of the synthetic programming environment and the applications that motivated specific features to provide the context and justification for design choices.

3. The Synthetic Programming Environment

The synthetic programming environment models different aspects of the code generation and execution process as high-level components. The four main components are InstructionStream, ISA, Processor, and the memory classes. These components present a simplified view of the operating system, instruction sets, execution resources, and available memory 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 first-class Python callable objects. The arguments to ISA objects are the operands for the underlying machine instructions. Instantiated ISA objects can be directly rendered into binary encoded machine instructions or inserted into an InstructionStream as part of a larger instruction sequence.

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 basic register allocation and management of 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.

The next sections 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.

4. ISAs and Instructions

ISAs are the main user interface to a processor’s underlying instruction sets. From the user’s perspective, an ISA is a collection of callable objects that implement a full instruction set. For example, an instance of the addi PowerPC instruction, implemented in CorePy in the module ppc, is created by simply calling the instruction with the proper arguments:

```python
# Evaluate (0 + 31) + 11
i1 = ppc.addi(gp_return, 0, 31)
i2 = ppc.addi(gp_return, gp_return, 11)
```

i1 and i2 are assigned references to instances of the addi subclass of the Instruction class that can be added to an InstructionStream for execution.

ISAs are implemented as modules that contain the callable objects for each instruction. The instruction classes are implemented as subclasses of the Instruction class. In the SPE, instructions were implemented simply as functions that returned the binary encoded instruction. To provide a more flexible system for generating instruction sequences, CorePy changed the implementation to be based on an extensible class hierarchy.

One of the design challenges for instructions was selecting the order of the operands for all ISAs supported by the synthetic programming environment. Instructions have two natural operand orderings, the machine ordering and the assembly ordering. In some cases,
these are identical, but in other cases they are not. Strategies for machine ordering may
differ from platform to platform. The Cell BE SPU instruction set places immediate (i.e.,
constant) operands first in the binary encoding while the PowerPC instruction set places
them last. Using the machine order makes rendering the binary instruction fast and easy to
maintain and was the solution in the SPE. However, as applications evolved to support syn-
thetic programs that used multiple ISAs, the different orderings led to developer confusion.
Thus, to present a consistent interface across all ISAs, the assembly order was selected for
CorePy as the default ordering.

The operand ordering dilemma led to a very flexible approach to operand handling
in CorePy. Rather than rely on a fixed ordering, the CorePy Instruction class supports
arbitrary orderings for instructions, selected by an ordered list of operands passed to the
constructor. Additionally, all operands can be passed using a keyword arguments list, rather
than as positional arguments. As an example, the following calling sequences for the addi
instruction are equivalent:

```plaintext
1 # Assembly order
2 ppc.addi(r1, r2, 42)
3
4 # Keyword operands
5 ppc.addi(SIMM = 32, A = r2, D = r1)
6
7 # Order from user-supplied list
8 ppc.addi(r2, 42, r1, order = (A, SIMM, D))
```

The instructions exposed in an ISA module correspond to the machine instructions
for the processor, not the assembly instructions. Most assembly instructions are actually
mnemonics for machine instructions that set bit flags using immediate operands. To make
the machine instructions more usable, all immediate operands that used to select variations
of the instruction are set to a default value using the keyword arguments. The default value
is always 0, which typically corresponds to the use case with the least amount of side effects.
Users can override the default value by simply passing in the keyword argument for the bit
flag. The PowerPC addx\(^1\) instruction has two bit flags, Rc and OE, for setting the condition and overflow registers, respectively. To set the flags, keyword arguments are passed to the instruction:

1. normal, no flags set, equiv. to 'add'
   ```python
   ppc.addx(rd, ra, rb)
   ```

2. Set the overflow flag, equiv. to 'addo'
   ```python
   ppc.addx(rd, ra, rb, OE=1)
   ```

3. Set the condition flag, equiv. to 'add.'
   ```python
   ppc.addx(rd, ra, rb, Rc=1)
   ```

By using keywords and default arguments, the ISA implementations maintain the same expressiveness as the full collection of assembly mneumonics.

4.1. ISA Generation. Both the SPE and CorePy use a metaprogramming module to generate the actual Instruction classes from a list of instructions, fields, and other metadata. To create a module for an ISA, the operand field types and instructions are encoded in tables containing the parameters for the constructors and the name of the field/instruction. For example, the entries used to generate the Field objects and class for the addi and fmaddx Instructions are:

```python
Fields = (
    ('A', (Field, (11,15))),
    ('B', (Field, (16,20))),
    ('C', (Field, (21,25))),
    ('D', (Field, (6,10))),
    ('SIMM', (MaskedField_16, (16,31))),
    ('Rc', (Field, (31), 0)),
    ...
)
```

```python
PPC_ISA = {
    'addi', {'binary': (14, D, A, SIMM), 'asm': None }},
    ...
    'fmaddx', {'binary': (63, D, A, B, C, 29, Rc),
```

\(^1\)addx corresponds to the add instruction. The ISA manual lists the instruction as addx to inform the user that the assembly mnemonics may include a period to set the rc bit. Both the SPE and CorePy leave the x on the instruction name.
In this example, the definition for the addi Instruction uses the opcode 14, contains three operand fields that form the arguments to create the instruction, and specifies that the assembly order should be the same as the machine order. \texttt{fmaddx}, floating-point multiply and add, has an opcode of 63, four register operands, an extended opcode of 29, and a bit flag. The \texttt{asm} order list swaps the position of the B and C operands to conform to the assembly ordering.

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 to a user as a function. The \texttt{\_\_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 (\texttt{value \ll shift}) that is used by the MachineInstruction objects to build efficient instruction generators.

At the lowest level, machine instructions for all supported architectures are a sequence of bits forming one 32-bit word. 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.

The MachineInstruction 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 \texttt{\_\_call\_\_()} method dynamically from code templates supplied by the Field objects. To create the final Instruction class, the MachineInstruction class is combined with the remaining elements in the instruction definition list and added to the module’s namespace with the instruction’s name as its name in the module.
4.2. Active Code. Instructions are standalone classes whose instances can be added to any InstructionStream. The normal usage, however, is to add all instructions in a block of code to the same InstructionStream. For example, the following code creates an InstructionStream and adds three instructions from the ppc ISA to it:

```python
1 code = synppc.InstructionStream()
2 code.add(ppc.lwzx(r_value, r_addr, r_index))
3 code.add(ppc.addi(r_value, r_value, 10))
4 code.add(ppc.stwx(r_value, r_addr, r_index))
```

For short synthetic programs this approach is acceptable. But, as the number of instructions increases, typing `code.add()` repeatedly becomes tedious, and the visual noise generated by the method call makes it difficult to read the synthetic program.

ISA modules have a special property, `active_code`, that sets one InstructionStream as the primary instruction stream for all instructions in the ISA. Using the example from above, the code object is set as the active code object for ppc:

```python
1 code = synppc.InstructionStream()
2 ppc.set_active_code(code)
3 ppc.lwzx(r_value, r_addr, r_index)
4 ppc.addi(r_value, r_value, 10)
5 ppc.stwx(r_value, r_addr, r_index)
```

With the active code property set to an InstructionStream, all Instructions add themselves to the InstructionStream when they are instantiated, simplifying the syntax for generating synthetic programs.

Active code presents a challenge for synthetic component libraries. Components can be developed using the `add()` method or the active code property. Using active code makes the libraries themselves more readable, but requires extra work to ensure that active code is properly set. `add()` removes any questions as to which InstructionStream is being used, but makes the code more difficult to work with. Ultimately, it is up to the library developer to decide. Active code was a late addition to CorePy, but since its addition the general
3. THE SYNTHETIC PROGRAMMING ENVIRONMENT

3.1 Approach

The approach has been to use it over add(). The BLAST application was developed using active code for all synthetic components.

5. 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(self, 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 an instance of instruction, adds the instruction to its internal data structures 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).

Internally, instructions are stored using a Python list for instances of Instruction and a Python array of unsigned 32-bit integers for the binary encoded instructions. On all platforms, the instruction size was 32 bits. In the SPE, instructions were rendered immediately into their binary form and no further information was retained in the InstructionStream. This complicated future optimization and debugging passes, require a complex disassembly mechanism to extract the instruction type and operands. CorePy's InstructionStream uses the dual approach of storing the rendered instruction along with the original Instruction instance, greatly simplifying more complex code synthesis, optimization, and debugging tasks.

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`.

An `InstructionStream` can have multiple register pools, depending on the ISAs the `InstructionStream` supports. Register pools are added to `InstructionStream` when `InstructionStream` is subclassed for a specific architecture using a declarative class attribute. Register pools return instances of the `Register` class, subclassed to associate them with a specific hardware register type. `Register` subclasses are typically opaque types that do not add functionality to the base class. For instance, the PowerPC `InstructionStream` includes three register pools, one each for the general purpose, floating-point, and vector register files. The `Register` subclass definitions and `RegisterFiles` class attribute for the PowerPC `InstructionStream` are:

```python
1 2 3 4 5 6 7 8 9 10
class GPRegister(spe.Register): pass
class FPRRegister(spe.Register): pass
class VMXRegister(spe.Register): pass
class InstructionStream(spe.InstructionStream):
    ...  
    RegisterFiles = (('gp', GPRegister, range(2,10) + range(14, 31)),
                      ('fp', FPRRegister, range(1,32)),
                      ('vector', VMXRegister, range(0,32)))
```

The `RegisterFiles` attribute is a list of tuples, with each register file descriptor tuple composed of a textual key, the register type, and a list of the available registers.

The instruction stream loads data from memory using the address embedded in the load instructions. To ensure that the memory buffers and other resources allocated in Python and 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 the synthetic programming environment 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.

6. 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
1 class Processor:
2     def make_executable(code)
3     def execute(code,
4         mode='void'|'int'|'fp'|'async')
5     def suspend(tid)
6     def continue(tid)
7     def cancel(tid)
8     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()`. On Linux, all memory regions are executable by default, making this method essentially a no-op. However, recent security enhancements to Linux from the SELinux project will require that future implementations of the synthetic programming environment on Linux implement this function to ensure proper code execution.

`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 were a function that returned an `int` or `double` value, respectively. Return values can only be supported on platforms where there are clearly defined ABI rules for passing values back to the caller. For instance, on a PowerPC
running OS X, the ABI convention has the instruction stream place an integer result 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. However, on a Cell BE SPU,
instruction sequences are executed asynchronously in a separate thread and the concept of
a function return value is not defined.

Under the first three execution modes, execute() is synchronous and 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() termin-
nates a thread. join() is used to wait for a thread to finish. Asynchronous mode makes it
possible to fully utilize all available processors from Python for data intensive applications.
On a Cell BE with eight available SPUs, eight SPU instruction streams can executing simul-
taneously. It is worth noting that, aside from join(), synchronization primitives are not
supplied at this level. Chapter 4 introduces some techniques for coarse grained synchro-
nization.

The Processor methods are implemented in C++ and use pthreads for asynchronous
execution on both Linux and OS X. The Cell BE SPU implementation uses libspe1.2 to
create SPU threads. 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() on OS X 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 extract-
ing the address of the instruction stream and ensuring it is properly aligned. This helps
keep the amount of native code to a minimum.
The final components that make up the basic functionality of the synthetic programming environment are the memory management classes, `MemoryDescriptor` and `AlignedMemory`. While the synthetic programming environment relies on the underlying operating system and Python classes for memory allocation, the memory management components provide consistent interfaces for abstracting blocks of memory.

The original SPE did not formally abstract memory. Instead, each application managed memory its own way. On G4 and G5 processors running OS X this was sufficient, primarily due to a design feature of OS X’s underlying memory allocation system and its default Python build. On OS X, all allocated memory is naturally aligned to on 16-byte boundaries, and the default Python build is built in “release mode”. The former ensures that no alignment issues arise with system allocated memory, avoiding many performance and memory loading pitfalls. The latter is more subtle. Python’s memory debugging system adds 8-byte sentinels to beginning of the allocated memory, shifting the address by 8-bytes. For the Cell BE port, where the Python build used the memory debugging flags, all memory was returned with the additional 8-byte offset. Without modifications to the underlying Python library code, guaranteeing aligned memory was not possible.

Further complicating matters for memory management is the user-managed local store on the Cell BE SPU cores. The local store is essentially a 256 kb blank slate of memory that holds the program and data and is managed by the programmer. This gives the developer incredible flexibility for managing what is essentially the SPU’s L1 cache. The `MemoryDescriptor` class evolved out of the need for a consistent mechanism to describe regions of memory in the local store and main memory.

In addition to their practical uses, the memory classes serve as fundamental resource abstraction necessary for giving developers full access to the hardware. Both C and Fortran have similar approaches via `void` pointers and common blocks, respectively.
7.1. Memory Descriptor. MemoryDescriptor abstracts memory using a typecode, an address, and a size. While address and size (in bytes) is sufficient for describing memory regions, the common usage is to use memory descriptors to describe data allocated in Python or Numeric Python array objects. Including typecode in the definition of MemoryDescriptor keeps the arithmetic for computing block sizes based on types in one location, eliminating a possible source of errors.

A basic MemoryDescriptor is formed with one call to the constructor:

```python
n_elts = 64
md = memory_desc('i', 0, n_elts)
```

This example creates a MemoryDescriptor of type signed integer at memory address 0 with 64 elements. The typecodes for memory are based on Python’s array class typecodes and cover the common integer and floating point types. To be consistent with array allocation semantics, the size property of MemoryDescriptor is the number of elements in the array. The actual number of bytes described is \( n_{\text{elts}} \times \text{sizeof(type)} \), in this case, \( 64 \times 4 = 256 \) bytes.

MemoryDescriptors are used to model memory in synthetic programs and not all properties are run-time constants, as in the previous example. The typecode is the only required property for MemoryDescriptors. All other properties can be referenced using registers. For instance, if the address and size are run-time parameters stored in registers, a MemoryDescriptor can be initialized as follows:

```python
md = memory_desc('B')
md.set_addr_reg(r_addr)
md.set_size_reg(r_size)
```

Here, the memory contains unsigned bytes and the location and size are stored in registers. Classes that take MemoryDescriptor as arguments must support both constant and register representations of address and size.

7.2. Aligned Memory. The AlignedMemory class complements MemoryDescriptor. Data movement between regions of memory and the processor and memory often must be done using naturally aligned blocks. This is especially important on the Cell BE, where unaligned
transfers between main memory and an SPU's local store result in bus errors. To simplify the task of aligning data blocks, AlignedMemory allocates a block of memory aligned on a user-specified boundary. AlignedMemory has methods for copying unaligned memory into and out of its buffer. This lets the developer use any memory buffer in Python while having a convenient mechanism to force alignment when needed.

AlignedMemory is not intended to be a replacement for more full-featured array classes and hence supports a very limited interface with only three data access methods. copy_to copies data from a memory region into the AlignedMemory instance. copy_from copies data out of AlignedMemory. word_at extracts a word from the AlignedMemory instance's buffer. Like MemoryDescriptor, AlignedMemory is also typed to make working with arrays more straightforward:

```python
1  n = 10000
2  a = array.array('I', range(n))
3  aa = aligned_memory(len(a), typecode='I')
4  aa.copy_to(a.buffer_info()[0], len(a))
5
6  # ... operate on the data in the aligned memory buffer ...
7
8  aa.copy_from(a.buffer_info()[0], len(a))
```

Here, an array is created with 10,000 unsigned integers and copied to an AlignedMemory instance.

8. Conclusion

The synthetic programming environment provides the basic tools necessary for synthetic programming to be feasible. By providing abstractions for machine-level resources, it allows developers to easily develop programs that previously required expertise with assembly language toolchains. From the user's perspective, the synthetic programming environment is portable across architectures, with the ISA being the only component that varies in the user interface from platform to platform. The Processor library presents the user with a simple interface for controlling synthetic programs in multi-core environments, simplifying the task of building multi-core applications. With the synthetic programming environment,
developers can start to explore more complicated code generation and optimization techniques.
Synthetic Programming

The components of the synthetic programming environment provide the foundation for generating high-performance code directly from Python. This chapter introduces the basic techniques used to develop algorithms and pass data between Python and the synthesized instruction streams. This general process of building and progressively specializing instruction streams has been dubbed synthetic programming\(^1\), referring to synthesis of special-purpose synthetic programs at run-time from a collection of code generating components.

\(^1\)The term “synthetic programming” has been used at least two times in software literature. In 1985, a paper titled Synthetic Programming [19] described an automated program generation system. The term was also used in the early 1980s to describe a method for entering hidden instructions on HP-41C calculators [33]. The latter use was popular for a time.
1. Synthetic Programming Basics

In order to effectively discuss synthetic programming techniques, some new terminology has been created. The programming language that the synthetic programming environment is implemented is referred to as the host language. The host language for the SPE and CorePy is Python. Synthetic programs are self-contained instruction sequences created at run time using the synthetic programming environment. A program developed in the host language can contain one or more synthetic programs. An application that uses synthetic programs is referred to as the host application. Synthetic components are reusable code synthesis components developed in the host language that abstract common code generation tasks. The use of synthetic as a modifier on any word other than “program” denotes the object is a code synthesis component. For example, synthetic functions are functions that generate instruction sequences that are used to build up synthetic programs.

An application developed using synthetic programming typically separates the generation and execution of synthetic programs. A possible sequence of operations in an application may be:

(1) Load synthetic programming environment modules, perform application initialization
(2) Load data
(3) Load synthetic components for processing data
(4) Generate synthetic programs based on data characteristics
(5) Process data by executing synthetic programs on multiple processing core
(6) Save results, end program

Synthetic programs can be generated using more information than is available for kernels generated at compile time. In the above sequence, the generation step may customize the kernels based on run time characteristics of the data, such as the size of the actual data set, or a time-dependent property such as the number of available processing cores.

With well-designed synthetic components, synthetic code can read much like normal application code, making it possible for the developer to create synthetic programs using
natural syntax. Two libraries of synthetic components developed to evaluate synthetic programming, the Variable and Iterators libraries, are examples of syntactically concise synthetic components. For example, the following code sample generates a synthetic program for computing a nested summation and returns the result as an integer value:

```python
a = vars.UnsignedWord(0)
for i in syn_iter(code, 5):
    for j in syn_iter(code, 5):
        for k in syn_iter(code, 5):
            a.v = a + i + j + k
util.return_var(a)
```

While this code appears to be a simple Python loop, the variables, iterators, and return function are all synthetic components that generate code with semantics similar to the borrowed syntax.

The basic development strategy for developing applications that utilize synthetic programs is called progressive specialization. The initial application is developed entirely in the host language. As the development iterations progress, candidates for optimization are identified and gradually converted to synthetic programs, following an iterative approach with an emphasis on testing all assumptions. Using progressive specialization, during each optimization iteration only the minimal required amount of a code is be converted to a synthetic program. Performance tests ensure that the optimizations are impacting the overall performance of the application. Chapter 3 contains a detailed case study of using progressive specialization to optimize a portion of an cheminformatics application.

1.1. Working with ISAs. Synthetic programming requires a basic understanding of the target processor's instruction set and the types of operations possible using the supplied instructions. While all ISAs are different, most share common instructions types, including logical instructions for comparing values and performing bit-wise operations, arithmetic instructions for operating on integer or floating point numbers, load and store instructions for moving data from memory to registers, and control flow instructions for branching between
code sequences. This section provides a brief overview of the PowerPC, AltiVec/VMX, and SPU instruction sets.

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.

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.

In addition a core supporting the PowerPC and AltiVec/VMX ISAs, the Cell BE processor includes up to 8 cores that use the synergistic processing unit ISA and a set of channel-based instructions for accessing the memory I/O unit. The SPU ISA is a SIMD instruction set augmented with control flow instructions, enabling it to support complete programs (this is in contrast to the AltiVec/VMX ISA, which requires a supporting ISA for control
flow). All arithmetic and logical instructions operate on 128-bit wide vector registers and each SPU core contains 128 registers. The full set of arithmetic instructions is limited to 32-bit integer types and single- and double-precision floating point types, though there are individual instructions for common operations on other data types. The non-SIMD instructions all operate on a value stored in the *preferred slot* in the vector register. On the Cell BE, this is the first 32-bit element in the register.

The Cell BE synergistic processing element core consists of a two asynchronous processors, a SPU and a memory I/O unit, and 256 kb of user managed local store. The local store is similar to the L1 cache on the PowerPC, with very fast six-cycle loads and stores. Unlike an L1 cache, the local store is managed by the programmer. The programmer uses the memory unit to move data between main memory and the local store. Memory instructions exposed as a set of “channel” commands accessed through the read and write channel instructions, `rch` and `wrch`, respectively. Memory instructions use get/put semantics, must be aligned on natural (16-byte) boundaries, and can be issued in bulk using lists.

Because the PowerPC, AltiVec, and SPU 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.

2. Synthetic Components

To simplify the creation of synthetic programs, common instruction sequences, optimization strategies and code patterns can be abstracted into synthetic components in the same way that objects and functions abstract traditional code components. Techniques for developing synthetic components will necessarily depend on the features available in the host language. In Python, there are two primary approaches. The first, used for simple instruction sequences, is to simply provide a function that generates a common sequence of instructions.
For example, all supported architectures include a `util` module that contains simple code generation functions. The first synthetic function implemented for any platform is `load_word(code, target, value)`, a function that takes an `InstructionStream`, target register, and word sized immediate value and generates the instruction sequence to load the value into the register. The following code loads the constants `0xFFFFFFFF` and `0xFF` into two registers using `load_word`:

```
1 ra = code.acquire_register()
2 rb = code.acquire_register()
3 load_word(code, ra, 0xFFFFFFFF)
4 load_word(code, ra, 0xFF)
```

Section 1.1 describes how `load_word` is implemented to generate optimized code sequences for each case. By convention, the first argument to a synthetic 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.

Language features can be used to make synthetic components more readable. The `Variable` and `Iterators` libraries, covered in Chapter 5 use operator overloading and the Python `Iterator` protocol to provide powerful libraries that allow developers to generate code sequences using Python expression and loop syntax.

### 3. Managing Execution

The `Processor` provides a basic set of methods for executing synthetic programs asynchronously. Asynchronous execution enables 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., \texttt{lwzu} and \texttt{stwu} on PowerPC. On the SPU, all synthetic programs are executed asynchronously.

Management code typically contains high-level operations and manages workloads for computational kernels, allowing it to be implemented primarily in the host language. For example, the interactive particle system (Chapter 7) and BLAST (Chapter 8) applications are both managed by pure Python algorithms. In both cases, the management code dispatches work units to synthetic kernels by passing references to data blocks. In synthetic programs, and in the case of the particle system, additional native libraries perform the data and compute intensive operations.

4. Data Sharing

In order to perform non-trivial operations, developers need to share data between the host program and the synthetic program. Data can be shared using a combination of two techniques: parameter passing and memory sharing. On all supported platforms, the system ABIs specify rules for passing parameters and returning values through registers and the stack. The synthetic programming environment uses these mechanisms to allow developers to pass integer and floating point values between synthetic programs and the host program. Values passed as parameters or return values use pass-by-value semantics. For example, the CorePy parameter passing system uses an object to pass integer parameters into an SPU synthetic program:

```python
1  params = spu_exec.ExecParams()
2  params.p1  = 1
3  params.p2  = 2
4  r = proc.execute(code, params = params)
```

Memory sharing is a more general technique for sharing data using pass-by-reference semantics. Instead of passing data directly, memory addresses are passed as parameters or directly encoded into the instruction sequence. Passing addresses as parameters is conceptually identical to C’s notion of passing pointers to functions. However, because there are
no assumed type semantics, memory addresses are equivalent to `void*` and it is up to the developer to create proper instruction sequences for interpreting the data in memory.

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. When a data address will never change, this technique is useful for register management, allowing the address register to be needed only for a short sequence of instructions. The following sequence of Python code demonstrates this technique by creating an array and a sequence of instructions that loads its address and first element into a register:

```python
1 a = array(range(10))  # create an array
2 c.add_storage(a)      # inc ref count on a
3 rd = c.acquire_register() # address register
4 load_word(c, rd, addr(a)) # load the address
5 rv = c.acquire_register() # value register
6 c.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. The call to `add_storage()` saves a reference to the array, ensuring that the array is not destroyed by the garbage collector before the synthetic program is executed.

5. **Communication**

In addition to sharing data when the synthetic program is started, synthetic programs can communicate with host programs using standard techniques for passing messages between threads. This is especially useful for memory-resident synthetic programs that service short requests from the host program. The simplest method for sending messages is to use atomic read and write operations from the synthetic program. The PowerPC instruction set provides atomic versions of most load/store instructions that can be used for this purpose.

Applications developed for the Cell BE have a number of additional techniques available for passing messages between the host program and a synthetic program running on an SPU. In addition to memory-based communication techniques, the Cell BE supports mailboxes...
for sending short messages between SPUs and the PPU and a signal channel for sending
notifications to an SPU from the PowerPC processing unit (PPU). In the synthetic BLASTP
implementation, the two major kernels that remain resident on the SPU are implemented
using command loops. Commands are sent to the programs as signals and the parameters,
pointers to the next blocks of data, are passed using mail boxes.

The Python portion of an application can receive messages from an executing synthetic
program using data sharing techniques or through native functions designed for message
passing. The SPU Processor object, for instance, is extended with the entire library of
communication methods from libspe 1.2, enabling the Python code to communicate effi-
ciently with a synthetic program.

The following code demonstrates a simple SPU synthetic program that communicates
with the host application using mail boxes:

```python
1 def spu_mbox():
2     code = synspu.InstructionStream()
3
4     # Send a message to the PPU
5     spu_write_out_mbox(code, 0xDEADBEEFl)
6
7     # Get a message from the PPU
8     reg = spu_read_in_mbox(code)
9
10     # And send it back
11     code.add(spu.wrch(reg, SPU_WrOutMbox))
12
13     # Execute the synthetic program
14     proc = synspu.Processor()
15     spe_id = proc.execute(code, mode='async')
16
17     # Get the first message
18     while synspu.spu_exec.stat_out_mbox(spe_id) == 0: pass
19     print 'spe said: 0x%X' % (synspu.spu_exec.read_out_mbox(spe_id))
20
21     # Send a message
22     synspu.spu_exec.write_in_mbox(spe_id, 0x88CAFE)
23
24     # Get the reply
25     while synspu.spu_exec.stat_out_mbox(spe_id) == 0: pass
26     print 'spe said: 0x%X' % (synspu.spu_exec.read_out_mbox(spe_id))
```
4. SYNTHETIC PROGRAMMING

```python
proc.join(spe_id)
return
```

synspu is the Python model for the SPU synthetic programming environment. Lines 4-11 generate the synthetic program. The read/write synthetic functions generate the instruction sequences to send messages using the mail box and manage the temporary registers for the operations. The final `wrch` instruction (line 11) echoes the host program’s message back using the same register it was read into. Line 15 executes the synthetic program on the SPU.

The remainder of the example contains the host program that communicates with the synthetic program. Using `libspe` functions exposed through the SPU synthetic programming environment, the host program waits for the first message (lines 18-19), then sends a new message (line 22), waits for the response (line 26), and finally waits for the SPU thread to complete (line 28).

6. Continual Specialization

Throughout the lifetime of an application, multiple synthetic programs may be created that differ by a few instructions. When the synthetic programs are executed sequentially, it is possible to reuse the overall structure of the synthetic program and only replace the instructions that change between runs. Reusing instruction sequences with minor changes is called continual specialization. In both CorePy and the SPE, continual specialization is supported using the bracket operator on the `InstructionStream`. Any instruction in the stream can be replaced with another instruction, effectively specializing the stream for the run.

The most common use of continual specialization is to replace the address of a data stream as an alternative to passing the address as a parameter. It can also be used to update constant data sizes, branch conditions, or numeric instructions, or to 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:
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().

7. Working With Native Libraries

Synthetic programs are can be used with data from any source or library, provided there is a way to access it using load and store instructions. This makes it possible to use synthetic programs in conjunction with native libraries to optimize applications.

The simplest method for interacting with data from native libraries is to acquire the memory address of the data and write a synthetic program that works with the data format. The Python buffer interface provides this functionality for most Python data types that have native storage formats. For example, arrays created using array package are stored internally using C arrays. The following code demonstrates how to access and modify each element in a Python array:

```python
import array

def ppc_buffer():
    size = 10
    a = array.array('I', range(size))
    code = synppc.InstructionStream()
    r_value = code.acquire_register()
    r_addr = code.acquire_register()
    r_index = code.acquire_register()
    synppc.load_word(code, r_addr, a.buffer_info()[0])
    for index in range(0, size * WORD_SIZE, WORD_SIZE):
        code.add(ppc.lwzx(r_value, r_addr, r_index))
```
In this example, the address for the array \( a \) is hardcoded into the synthetic program after it is extracted using the buffer interface (line 13). Line 15 is the start of a Python loop that will execute 10 times, with the induction variable \( \text{index} \) containing the offset of the next integer in the array. The loop body (lines 16-18) loads each value, adds 10 to it, and stores it back in the array.

Arrays have a simple native format and make for a convenient native library to use with synthetic programs. Numeric Python also contains many array data types based on native C data structures. However, Numeric Python does not support the buffer interface. For this library, and libraries that do not provide a mechanism for acquiring buffer address, small native extensions are required. For example, the \texttt{synnumeric} library extracts the address of the data pointer from a Numeric array. Using Numeric Python, the above example would replace lines 1, 5, and 13 with:

1. # Line 1 replacement:
2. import Numeric
3. import synnumeric
4. 
5. # Line 5 replacement:
6. \( a = \text{Numeric.arange(size, typecode = Numeric.Int}) \)
7. 
8. # Line 13 replacement:
9. \( \text{synppc.load_word(code, r_addr, synnumeric.addr(a))} \)

In some cases, it may be necessary to transform the data into a format that is easier to process using synthetic programs. In these cases, a synthetic program can be developed for perform the transformation or a native library may be used. One area where the native library has been applied is in the \texttt{aligned\_memory} class. In cases where performance is
not an issue, the Python `struct` module can be used to convert binary data into different formats entirely from Python.

8. 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.

The SPE implemented a basic set of features for printing out instruction sequences as they were generated. As applications scaled and began to use complex synthetic programs, the simple debugging features in the SPE proved inadequate. As a result, CorePy implemented a much more flexible system for generating and collecting instructions. The primary improvement was the addition of a robust class system for modeling instructions. In the SPE, instructions were simply implemented as functions that generated binary coded instructions as they were called.

In CorePy, instructions are full featured classes and a generated instruction is an instance of the class. Methods provide full access to all operands and additional data about the instruction. Most importantly, the CorePy `Instruction` class supports a customizable “to string” method that can be used generate arbitrary representations of an instruction. For debugging purposes, the developer can use different renderers to view the instructions and instruction stream in different formats, including traditional assembly programs or binary-coded instruction sequences, or both. The most advanced renderer collections call stack information from the Python program and displays the instructions, their address in memory, and the code that generated them. The address information can be used to set breakpoints in `gdb`.

For example, the following synthetic program computes the sum of the even integers less than 100:

```python
7 def sum(code):
8
```

DRAFT: May 23, 2007
for i in syn_range(code, 0, 100, 2):
    a = ppcvar.UnsignedWord(0)
    a.v = a + i

return_var(a)

The debugging printout for the synthetic program is:

0x003894E0 0 addi(r30,0,0) [debug.py:sum: 9] a = ppcvar.UnsignedWord(0)
0x003894E4 1 addi(r29,0,0) [debug.py:sum: 11] for i in syn_range(code, 0, 100, 2):
0x003894E8 2 add(r28,0,100) *** ***
0x003894EC 3 ori(0,0,0) *** ***
0x003894F0 4 addx(r30,<r30>,<r29>) [debug.py:sum: 12] a.v = a + i
0x003894F4 5 addi(r29,r29,2) [debug.py:sum: 11] for i in syn_range(code, 0, 100, 2):
0x003894F8 6 cmp_(0,0,r29,r28) *** ***
0x003894FC 7 bcx(13,0,-12,0,0) *** ***
0x00389500 8 addi(r29,0,0) *** ***
0x00389504 9 addi(r3,<r30>,0) [debug.py:sum: 14] return_var(a)

The columns display the address of the instruction, the index of the instruction in the InstructionStream, the instruction itself, the file, function and line number that generated the instruction, and the text of the line that generated the instruction. This example uses the Variable and Iterator libraries introduced in Chapter 5 and shows the code generated by each operation.

Instructions 0-2 are generated from the write function in line 5 of the example, instruction 3 from line 8, and instruction 4 from line 11.

While instruction printouts are a useful first step in debugging, it is often necessary to inspect a running program to fully understand a bug. Synthetic programs can be debugged directly using a debugger such as gdb. Both CorePy and the SPE provide methods to display the addresses of synthetic programs and the offset of individual instructions within those programs, making it possible to set arbitrary breakpoints in a synthetic program. If a synthetic program crashes, the instruction sequence, memory locations, an even registers can be examined directly.

If the program does not crash, but is still malfunctioning (e.g., running an infinite loop), the program can be forced to crash by inserting the special instructions into the instruction stream that force the program to halt. These instructions are specific to the ISA and have
slightly different effects on the state of the program. In a synthetic program running on a PowerPC, the `Illegal()` mnemonic issues an illegal instruction that stops program execution, allowing the developer to examine registers and memory at any point in the program. For example, changing the loop in the buffer example from Section 7 to:

```python
1  for index in range(0, size * WORD_SIZE, WORD_SIZE):
2       if index == 3 * WORD_SIZE:
3           code.add(ppc.Illegal())
4 16     code.add(ppc.lwzx(r_value, r_addr, r_index))
5 17     code.add(ppc.addi(r_value, r_value, 10))
6 18     code.add(ppc.stwx(r_value, r_addr, r_index))
```

will force execution to stop at the third element in the array. On the SPU, the `stop()` instruction performs a similar goal, stopping the synthetic program and allowing examination of memory and registers through a debugger.

In addition to low-level debugger-based techniques, Chapter 9 demonstrates how to use synthetic programming to build a full debugging environment for synthetic programs on the SPU.

9. 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 system. 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.

10. Conclusion

Beyond the low level processor libraries, a number of techniques have been developed to make synthetic programming a useful approach to generating high-performance code. All of these methods were developed in conjunction with the synthetic programming environment.
and the example applications, allowing them to be refined to the form they are presented in here. While this chapter introduces the basic approaches to synthetic programming, it only scratches the surface of what is possible. The next chapter delves into a number of powerful applications of these methods that allow synthetic programming to scale beyond a replacement for assembly languages and into the realm of domain-specific code synthesis.
Library-Defined Semantics and Optimizations

The synthetic programming environment introduces a new approach for developing machine and assembly level code. Using it as the basis for more complex code synthesis abstractions allows developers to start to replace many other elements in the compilation toolchain. This chapter introduces a number of synthetic programming techniques for lifting common code generation and optimization techniques out of the compiler stack and directly into user-level libraries. These techniques enable library writers to develop highly customizable libraries for common and novel data types while giving the end user fine-grained control over the final semantics of code generation.
1. Synthetic Components

The design and implementation of the SPE, CorePy, and the applications built on both environments led to the development of a number of synthetic components that demonstrate different techniques for code synthesis and optimization. Some of these components are included in the standard utility library in CorePy and others serve as examples for domain-specific optimizations.

1.1. Constant Formation. One of the more common tasks in used synthetic programs is loading fixed values into registers. Fixed values can take the form of bit-masks, constant values in arithmetic expressions, or memory addresses. For vector-based algorithms, the same fixed value may be loaded into each slot in a vector register or a register may be initialized with multiple values taken from a data array. In all cases, each instruction set supports multiple methods for loading fixed values into arrays without requiring loads or stores. Generally referred to as constant formation instructions, these instructions encode all or part of a fixed value in an immediate operand field on the instruction. The choice of instructions depends not only on the instruction set, but also the size and sign of the value. Small values may fit in the immediate operand of a single instruction, whereas large values and vector register may require multiple instructions.

Each platform supported by both the SPE and CorePy included a `load_word()` function that abstracted constant formation. In each implementation, `load_word()` takes a code object, target register, and a value or array of values to load into the register. The value is examined to determine which sequence of instructions is appropriate and those instructions are added to the instruction stream.

For example, the PowerPC implementation of `load_word()` inspects the value to see if it can be loaded using a single 16-bit immediate operand:

```python
1   def load_word(code, r_target, word):
2       start = code.add(ppc.addi(r_target, 0, word))
3
4       if (word & 0xFFFF) != word:
5           code.add(ppc.addis(r_target, r_target, ((word + 32768) >> 16)))
6       return start
```

DRAFT: May 23, 2007
The function tests the size by masking the value and comparing the result against itself. If the values are identical, only one instruction is required. Otherwise, a second instruction shifts the immediate operand and adds it to the value in the current register. The constants from the example in Section 2, 0xFFFFFFFF and 0xFF, demonstrate the different sequences generated by load_word. The first requires two instructions while the second only needs one.

The load_word() family of synthetic components is designed to load the value supplied at synthesis time into the register. Thus, for initializing vector registers with values stored in arrays, the values in the array when the function is called are encoded into immediate operands of constant formation instructions. Loading constants from arrays using this technique can generate a large number of instructions. For instance, the SPU implementation of load_word() for values supplied in arrays makes four calls to the underlying load_word() function for scalar values, uses three rotate instructions, and requires temporary registers. On the SPU, where vector loads are completed in six cycles, it is faster to load the value from the local store than form it in place. However, for architectural reasons, it may not be practical to store the value in local store, requiring the use of the constant formation instructions.

Constant formation illustrates an important feature of library-defined semantics. In a compiled language, the user has no control over how constants are formed. Often, they are simply stored in the data segment of the compiled code and loaded from memory as needed. For fine tuning high-performance kernels, allowing the user to make subtle changes in the semantics at the library level, such as selecting how constants are formed or loaded, can have a positive impact on performance.

1.2. Extended Instructions. Machine instruction sets rarely “complete” from the end-user’s perspective. That is, while all operations on a certain type of value, say integers, may be possible, the machine instruction set may only support a limited number of instructions that are in used to build support for the full set of operations. For example, the
SPU’s instruction set is limited to two comparison primitives, cgt and ceq, compare greater-than and compare equal, respectively. All other comparison operations must be built from these primitives. To standardize the generation of short instruction sequences that correspond to “missing” instructions, CorePy introduced ExtendedInstruction, a subclass of Instruction that behaves like a single instruction. InstructionStream is also aware of ExtendedInstruction classes, a feature that can be used when generating debugging information for synthetic programs.

Subclasses of ExtendedInstruction implement a single method, block, that provides the instruction synthesis code for the instruction. The call signature for extended instructions is intended to match those of instructions, with the values being instances of Register or an immediate value. Operands should be ordered as if the instruction was an assembly instruction, with the destination register first, followed by the register operands, and concluding with immediate values.

Two examples of SPU extended instructions are the lt and lti, less-than and less-than immediate, operations:

class lt(SPUExt):
    """
    Word compare less-than
    """
    def block(self, d, a, b):
        spu.cgt(d, b, a)
        return

class lti(SPUExt):
    """
    Word compare less-than immediate
    """
    def block(self, d, a, b):
        temp = code.acquire_register()
        spu.cgti(temp, a, b)
        spu.ceqi(d, a, b)
        spu.nor(d, d, temp)
        code.release_register(temp)
Note that for \texttt{lt}, it is sufficient to simply reverse the operands on the greater-than instruction. \texttt{lti}, however, requires three instructions and a temporary register to properly compute the less-than relationship between a value stored in a register and an immediate value.

1.3. Parameter Passing. Parameters are passed to synthetic programs following the ABI conventions of the execution environment. For instance, the first seven PowerPC integer parameters are passed in registers \texttt{gp4} to \texttt{gp10}. While a user familiar with these conventions can simply retrieve the values from the registers, both the SPE and CorePy provide synthetic functions for copying parameters to allocated registers.

In a similar manner, each supported platform includes a \texttt{syn\_return(reg)} function that places the value in \texttt{reg} in the appropriate register for returning a value from a function. For example, the following program creates a synthetic program that adds two parameters and returns the result:

```python
1 a = code.acquire_register()
2 b = code.acquire_register()
3
4 synspu.copy_param(code, 1, a)
5 synspu.copy_param(code, 2, b)
6
7 ppc.addx(a, a, b)
8 syn_return(a)
9
10 p = synppc.ExecParams()
11 p.p1 = 11
12 p.p2 = 31
13
14 r = proc.execute(code, params = p)
15 --> r == 42
```

Lines 1-8 create the synthetic program, with lines 4 and 5 copying the input parameters to user registers and line 8 storing the value of a user register in the return register. Lines 10-12 set up the parameter data structure and initialize the parameters to 11 and 31. Line 14 executes the synthetic program and retrieves the return value.
2. Variables

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

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

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

Most programming languages abstract values stored in registers and operations on performed on them using language or user-defined types. Types encapsulate the storage and operational semantics for a class of data. Most programming languages have primitive types that are the building blocks for all other types. The rules for storing and operating on values represented as primitive types are fixed, and compilers transform expressions involving primitive types into the appropriate machine instructions. More complex types are built using primitive types.

By using primitive types as the building blocks for all other types, developers are limited to the instructions used by primitive types for optimizing code. For instance, a struct built using four C-style int variables with SIMD semantics may never actually use SIMD instructions, and instead use sequences of general purpose instructions to perform arithmetic.

Synthetic programming allows developers to avoid these limitations by enabling the development of new types that have fully customizable, user-defined machine-level semantics. The Variable library was developed to study the use of user-defined type semantics.
The Variable library contains a collection of base classes that implement the interpreter design pattern using Python classes and operators overloading. It allows developers to create arbitrary types and have explicit control over how types are mapped to registers and what instructions are used to evaluate operators and other operations.

Like all synthetic components, components built using the Variable library are generative components that create, rather than evaluate, instruction sequences. Unlike components based around functions and methods on objects, the Variable library allows synthetic components to borrow syntax and certain semantic properties from Python. For example, operator precedence is handled at the Python level, freeing Variable classes from implementing their own precedence rules. This symbiotic relationship between the synthetic components and the host language is one of the more useful techniques available for developing synthetic programs.

The main objects used to implement the Variable library in the SPE are illustrated in Figure 1.

The base classes, Variable, Literal, and Expression implement the Interpreter design pattern and manage register allocation, initialization, and expression evaluation. Python's
underlying expression evaluation engine handles operator precedence. The base classes are
type-less and type-specific subclasses generate instructions as the expressions are evaluated.
In the diagram, the floating point class hierarchy shows how the subclasses share operator
implementations, ensuring that floating point expressions are evaluated consistently.

Callable objects or functions that return Expression instances allow developers to ex-
pose operations that do not map directly to operators. These functions can be used directly
in larger expression statements and may abstract sequences of instructions. This is use-
ful for binary operators such as fmin, which is actually a three instruction sequence, and
various ternary vector instructions.

To illustrate how the Interpreter classes work, consider the example in Figure 2 for eval-
uating the expression \(a = a \ast b + a \ast c\). \(a, b,\) and \(c\) are instances of Variable with the \(\ast\) and \(+\)
operators overloaded. When an operator is evaluated by Python, the overloaded implement-
tion delays evaluation and returns an instance of Expression that contains references to
the operator and operands. In this example, the operands for \(\ast\) are Variables and the
operands for \(\ast\) are Variables. The root of the parse tree is the \(+\) Expression.

Up to this point, no code has been generated and registers have not been allocated.
All Interpreter classes have an eval method. Subclasses override eval to generate the
appropriate instructions for their data type and registers. When the Expression tree is
assigned \(^1\) to a Variable, the Variable triggers the expression evaluation, which in turn
generates the instruction sequence to evaluate the expression using a depth-first traversal of
the tree. Each Expression has a register associated with its results that lives for the lifetime
of the expression evaluation. After the assignment, the tree is traversed again to free up
temporary resources. Note that the final Expression uses the register from the assigned
Variable. The Variable passes this to the Expression’s eval method.

Expression evaluation is the same for Variables of all types. Expressions can contain
variables with different types, as long as the operator applied to the variables supports the
types. As the expression tree is created, the variables are checked against each other and

\(^1\) The assignment operator cannot be overloaded in Python for simple assignment, e.g. \(a = b\). Instead, a special
attribute, variable.v, intercepts the assignment using a Python property attribute. Property attributes attach
methods to attribute accesses, replacing a.set.v(1) with the syntactically simpler a.v=1.
other expressions in the tree. An exception is thrown if there is a type error. While the SPE uses an explicit class for literals, CorePy treats literals as simply variables with a different type. As long as they are used in an expression with a Variable subclass that can utilize them, they are legal. Handling literals as variables of a different type made it simple for Variable classes to insert special immediate mode instructions when appropriate, instead of always converting a literal value to a register value.

In CorePy, Instructions can take Variable instances as arguments. In addition, Instructions also have a special static method, `ex(...)`, that takes the instructions standard argument list with the destination operand omitted and returns an Expression instance that evaluates to the instruction with the operands applied. This allows any instruction defined in an ISA to be used in an expression. For instance, the following example uses the extended instruction `fmin` and the regular instruction `fmadd` in an expression:

```plaintext
1  r.v = fmadd.ex(a, b, fmin.ex(a, b)) + c
```

### 2.1. Processor Types

A matrix of types implied by instructions was developed to identify the types that could be supported directly by short sequences of instructions with the Cell BE SPU's. The original goal was to use the matrix to build types that closely matched the processor semantics. However, the matrix revealed an interesting aspect of the instructions supported by the SPU that is worth reporting. Rather than a small set of types, such
5. Library-Defined Semantics and Optimizations

As 32-bit integers and single-precision floating point numbers, the SPU instructions implied over 30 distinct types, all with unique semantics. Some types were limited to a few instructions, but many were restricted versions of common types, such as 10-bit integers. Thus, the goal of a clean, optimized type system built directly from processor instructions appears to be infeasible. Any type system with sufficiently expressive types will have to implement many basic operations as composite operations, requiring a tradeoff between performance and expressiveness.

3. The Iterator Library

Along with expression evaluation, methods for expressing iteration are some of the most common features in programming languages. All major imperative languages have at least one form of abstraction for iteration and loops, and many have multiple forms. Iteration is an important abstraction for high-performance computing and can have the largest impact on performance of any language feature. The compiler has many choices for expressing iteration in machine code and many optimizations are based on methods for optimizing how iteration occurs on the hardware. For example, a simple loop optimization, loop unrolling, duplicates the body of a loop multiple times, reducing the number of branch instructions by the unrolling factor. This allows the processor to avoid costly branches and keep its internal instruction pipeline full.

Iteration abstractions hide the complexity of managing iteration at the assembly level and provide the opportunity for unrolling and other optimizations. Just as classes and operator overloading make it possible to abstract code generation for expressions, the Python iterator protocol enables iteration abstractions. The synthetic Iterator library allows developers to use Python syntax to express algorithms that use complex iteration and optimization strategies using a natural syntax.

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

3.2. Synthetic Iterators. Using the basic modules in the synthetic programming environment, code for iteration is difficult to express and maintain. Consider the following code that sums the values in the array data:

```
# r_* are processor registers, e.g. r_sum = 3
# Load a pointer to the array
# Set the counter
# Zero the sum
# Set a loop label and load the next value
start = c.add(ppc.lwz(r_current, r_addr, 0))
# Update the sum
# Increment the pointer
# Decrement the counter and loop
next = code.size() + 1
c.add(ppc.bdnz(-(next - start) * 4))
result = proc.execute(c)
```

Hand-coding even a simple loop is tedious and error prone. The same basic loop expressed using a Python iterator is much easier to read and maintain:

```
sum = 0
for current in data:
  sum += current
```
Synthetic Variables can be used to implement the syntax in lines 1 and 3. The next step is to provide an abstraction with the expressiveness of line 2 that can generate the code similar to the low-level example above.

Synthetic iterators use the Python iterator protocol to provide a concise representation for the generation of instruction sequences for implementing machine-level loops. The sum example, implemented using a synthetic iterators from the SPE, is:

```python
1  sum = var(code, 0)
2  for current in var_iter(code, data, mode = 'ctr'):
3    sum.v += current
```

This example generates the code for the sum example using the ctr register to manage iteration. Every line in this example adds code to the instruction stream. The Python loop iterates exactly once, generating the loop prologue before allowing the body to generate its code, and then generating the loop epilogue before finishing the iteration.

All synthetic iterators work in the same basic way. A side-by-side view of the sum loop and the calls into the Python and synthetic iterators shows their operation:

<table>
<thead>
<tr>
<th>Loop code:</th>
<th>Call sequence:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sum = var(c, 0)</code></td>
<td><code>__iter__(): start()</code></td>
</tr>
<tr>
<td><code>for i in syn_iter(c, data):</code></td>
<td><code>1st next(): setup()</code></td>
</tr>
<tr>
<td><code>    sum.v += current</code></td>
<td><code>... body ...</code></td>
</tr>
<tr>
<td></td>
<td><code>2nd next(): cleanup()</code></td>
</tr>
<tr>
<td></td>
<td><code>    end()</code></td>
</tr>
</tbody>
</table>
| | `raise StopIteration`

When the iterator is created by the `for` loop, `start()` acquires the registers, generates initialization code and returns a reference to the iterator. On the first iteration of the `for` loop, i.e. when `next()` is called the first time, the `setup()` generates the loop prologue, initializing the loop counter and branch label, and returns a variable for the current value. Then the loop body is executed, and the expression code is added to the instruction stream. Note that any instructions from the ISA can be used here, not just expressions. On the second iteration of the `for` loop, `cleanup()` generates the loop epilogue with the branch instruction.
FIGURE 3. The Iterator Hierarchy. The synthetic iterator protocol extends Python’s iterator protocol to provide additional hooks into loop evaluation. *syn_iter* generates code to manage loops using the counter register or general purpose registers, as selected by the user. The subclasses override methods to implement iterator specific functionality. *unroll* and *parallel* are Proxies that restructure synthetic iterators for high-performance code generation.

and any loop cleanup code. It also resets the counters, in case the loop is nested. Finally, *end()* frees loop registers and other resources. While still at the beginning of the second iteration, the iterator raises *StopIteration*, ending the Python loop.

Synthetic iterators make it possible to implement a number of traditional optimizations at the library-level. The general optimizations supported by the SPE and CorePy are traditional, well-known high-performance optimizations. [10] provides a comprehensive survey of these optimizations in the context of compiler transformations.

A number of different iterators have been developed using Python iterator protocol in conjunction with the synthetic programming environment. Many of these iterators are part of a larger iterator hierarchy that allows common code generation components to be shared across iterators. The synthetic iterator hierarchy is shown in Figure 3. The design of the iterator hierarchy is platform independent, but the implementation is platform dependent. Two versions of the core hierarchy are implemented in CorePy, one for PowerPC/AltiVec/VMX programs and another for SPU programs. The next few sections describe the synthetic iterators developed for both platforms.

3.3. Basic Iteration: *syn_iter*. *syn_iter* handles the mechanics of sequential loop generation and is the base class for most other iterators. On the PowerPC, it supports three
different modes of iteration: counter based (CTR), register decrement (DEC), and register increment (INC). The SPU does not have a count register and \texttt{syn.iter} on the SPU does not support CTR mode. \texttt{syn.iter}'s constructor takes the iteration count, step size, and mode. The generated loop performs \((\text{count} \div \text{step})\) iterations of the loop body. In addition to constant values for count and step size, \texttt{syn.iter} can be initialized with registers that will contain the values at run time.

In PowerPC programs, CTR iterators generate the most efficient loops. CTR loops use the \texttt{ctr} register to hold the loop count and the \texttt{bdnz} (decrement counter, branch if non-zero) instruction to branch. CTR iterators do not require any general purpose registers, but only one CTR loop can be active at any time. Reading \texttt{ctr} is an expensive operation and as a result, the induction variable (e.g., \texttt{i} in the above example), is set to \texttt{None} for CTR loops. Loops that need access to the count should use INC or DEC iterators instead.

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

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

In all modes, \texttt{syn.iter} can be directed to excluded the branch instruction, allowing more complicated iterators to have fine-grained control over loop generation.

3.4. Counting: \texttt{syn.range}. \texttt{syn.range} is the simplest subclass of \texttt{syn.iter}. Its arguments follow the semantics of Python's \texttt{range} and support \texttt{start}, \texttt{stop}, and \texttt{step} keyword parameters. \texttt{syn.range} uses the INC mode and overloads the constructor to pass the correct count value to \texttt{syn.iter}. The induction variable contains the current value from the generated sequence.

The following code computes to the sum of all the even integers between zero and one hundred using \texttt{syn.range}:
3.5. Scalar and Vector Iterators. `var_iter` and `vector_iter` iterate over arrays of integers or floating point values, supplying the current value as a scalar or vector variable of the appropriate type. The arrays can be native Python or Numeric Python arrays. For example, the following code implements the sum operation on an array:

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

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

```python
1 code = synppc.InstructionStream()
2 proc = synppc.Processor()
3 data = array.array('I', range(100))
4 result = array.array('I', [0, 0, 0, 0])
5 sum = vector(code, result)
6 for value in vec_iter(code, data):
7    sum.v = sum + value
8    sum.save()
9 proc.execute(code)
10 final_sum = reduce(lambda a,b: a+b, result)
```

`sum` is a vector initialized with the `result` array. In line 8, the contents of `sum` are saved back to `result`. Because vectors contain four integers, to compute the final sum, the four values stored back to `result` need to be added together (line 14).
The variable and vector iterators also track changes to the underlying data and insert store instructions if the induction variable was modified. Consider the following two loops:

```python
sum = vector(code, 0)
for value in vec_iter(code, data):
    sum.v = sum + value

for value in vec_iter(code, data):
    value.v = value + value
```

In the first loop, sum is a variable declared outside the loop and no changes are made to the data array. In second loop, the induction variable `value` is modified. When the epilogue is generated, if the induction variable was modified, the result is written back to the memory storage for the array.

### 3.6. Iterator Composition: zip.iter

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

```python
code = synppc.InstructionStream()

x = array.array('f', range(100))
y = array.array('f', range(100))
z = array.array('f', range(100))
r = array.array('f', range(100))

X = vec_iter(code, X)
Y = vec_iter(code, Y)
Z = vec_iter(code, Z)
R = vec_iter(code, Z)

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

`zip.iter` works by disabling the branch operations for its wrapped iterators and generating its own loop using the smallest count value from the iterators. For each step of
code generation, it calls the appropriate methods on each iterator in the order they are supplied. Note that other than the disabled branch instruction, the wrapped iterators operate normally. In the above example, \( R \) will insert the store instruction save the the modified \( r \) induction variable.

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

`unroll` is a Proxy object that unrolls synthetic loops by allowing the Python iterator to iterate over the body multiple times, generating a new set of body instructions each time. Between iterations, the the loop maintenance methods are called with the flag to exclude branches instructions. On the final iteration, a single branch instruction is generated along with the loop epilogue. The following is a simple loop and pseudo-code for the generated code:

Loop code: Generated code:

```python
ger = syn_range(c, N) start()
for i in unroll(rng, 3):
    sum.v = sum + 2
    setup(); body; cleanup();
```

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

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

achieves near peak integer performance on a PowerPC 970.

### 3.8. Parallel Loops.

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

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

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

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

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

if MPI:
    t1 = proc.execute(c, mode='async', params=(mpi.rank, mpi.size, 0))
else:
    t1 = proc.execute(c, mode='async', params=(0, 2, 0))
t2 = proc.execute(c, mode='async', params=(1, 2, 0))
proc.join(t1); proc.join(t2)
```
In this example, each thread adds its rank to the value in the array. Two threads are created with rank 0 and 1, respectively. The first 50 elements in `data` remain the same, while the second 50 elements are increased by 1. The `join` method blocks until the thread is complete.

3.9. Pipelining and Double Buffering. The final synthetic iterator example demonstrates how synthetic iterators can be used to pipeline loops to overlap computation and communication in SPU programs.

Pipelining is a general optimization strategy that decomposes an operation into multiple stages and interleaves stages from successive iterations of the algorithm for improved resource utilization. For example, consider an algorithm that loads data, modifies the data, and saves the modified data. Each step (LOAD, MOD, and STORE, respectively) depends on the completion of the previous step, preventing any parallelism. If the algorithm executes once, then there is little hope of improving how the stages are executed. But, if the algorithm executes many times on an array of data, subsequent iterations can be interleaved with prior iterators to overlap memory load and stores. This application of pipelining is called double buffer.

Without double buffer, the LOAD/MOD/STORE algorithm for \( N \) elements looks like:

\[
\begin{align*}
&\text{LOAD}[1] \quad \text{MOD}[1] \quad \text{STORE}[1] \\
&\text{LOAD}[2] \quad \text{MOD}[2] \quad \text{STORE}[2] \\
&\ldots \\
&\text{LOAD}[N] \quad \text{MOD}[N] \quad \text{STORE}[N]
\end{align*}
\]

In this example, each iteration occurs independently of the next and there is no overlap. But, what if the load for each iteration was issued before the previous iteration, allowing the load to occur while the previous iteration is executing?

A double buffer pipeline that operates on \( N \) elements of an array is implemented using the following sequence of stages:

\[
\begin{align*}
&\text{LOAD}[1\rightarrow A] \\
&\text{LOAD}[2\rightarrow B] \quad \text{MOD}[1] \quad \text{STORE}[1] \\
&\text{LOAD}[3\rightarrow A] \quad \text{MOD}[2] \quad \text{STORE}[2]
\end{align*}
\]
The -> notation shows that the data for each load is going to different buffers. Note that in this example, two loads are issued at the beginning to get the pipeline started and then it proceeds until the end, where the final iteration skips the load operation.

The SPU stream_buffer iterator implements a double buffered pipeline for streaming data between main memory and the SPU local store. From the user's perspective, stream_buffer looks like any other iterator, with a few extra parameters:

```python
n = 1024
a_array = array.array('I', range(n))
buffer_size = 16
lsa = 0x0
stream = stream_buffer(code, a_array.buffer_info()[0], n, buffer_size, lsa,
                       buffer_mode = 'double', save = True)
```

Lines 5 and 6 initialize the stream_buffer iterator with data from an array in main memory and set the local store buffers to start at address 0x0 and hold 16 bytes. The keyword arguments tell it to double buffer the data and save the buffers back to main memory. The induction variable for stream_buffer is the address of the current buffer (line 8). The SPU vector iterator can use this as the source for iterating over a data array (line 9).

Internally, stream_buffer works by breaking the streaming operation into the stages outlined above. It generates two instances of the loop body, one for the first buffer and one for the second using two internal induction variables, and generates the code to initialize the pipeline before executing the first loop iteration.
4. Conclusion

Using synthetic components to abstract both code generation and optimization is a powerful technique that enables clear syntax without placing hard restrictions on how machine-code is generated. Simple components make it possible to abstract mundane operations while supporting simple optimizations, whereas complex components, such as the iterator library, allow developers to have an unprecedented amount of control over the generation and optimization of large code segments. The components presented here demonstrate that common and exotic optimizations are not only possible using synthetic programming, but also practical to use and straightforward to implement.
Progressive Specialization: Case Studies

To balance productivity and performance, algorithms developed using synthetic programming mix high-performance synthetic programs with computational and data flow code developed in the host language. Progressive specialization allows developers to systematically adjust the granularity of synthetic programs and find the appropriate balance between synthetic code and host language code.

For any algorithm, there is a point where high-performance optimizations cease to have a significant impact on overall application performance. To achieve a high level of productivity, synthetic code should be limited to the kernel operations that impact performance the most. The remainder of the algorithm should be implemented using features in the host language.
This chapter explores the granularity of synthetic programs using three different case studies to understand the relationship between code implemented using synthetic programs and code developed in the host language. Using progressive specialization, each application is implemented as a pure Python application and the kernel operations are moved into synthetic programs until performance stops increasing significantly. The applications are also compared against implementations from other languages to provide context for the performance numbers generated by the synthetic programs.

The first, and simplest, algorithm is an implementation of the array min operation. Given an unordered array, it finds the smallest value in the array. Array min is an example of the basic type of kernel operations synthetic programming is designed to handle. It contains a few basic operations that map directly to processor instructions and can be implemented concisely a number of different ways. In this study, synthetic programming produces the fastest implementation, out-performing even a compiler-optimized C++ version.

The next algorithm is an implementation of matrix-matrix multiplication based on the blocking techniques used by Goto-BLAS. The goal of this case study is to identify the level of granularity required to achieve near peak-performance using a synthetic kernel. The result is rather surprising: of the three blocking levels used by the algorithm, only the inner-most loop needs to be implemented synthetically to compete with the Goto-BLAS implementation. For the synthetic kernels, Python language features simplify the fine tuning of the instruction stream, making it possible to easily explore different operation orderings that impact performance.

The final algorithm is a chemical database search application that generates a similarity matrix for a collection of compounds based on a bit-vector operation. Starting from a pure Python implementation, the element-wise operations and loops are moved into the synthetic kernel to identify the point at which further optimizations yield minimal improvements. As with the matrix-matrix multiplication, this point turns out to be one of the inner loops rather than the entire algorithm.
1. Array Min

The \texttt{min} function iterates over an array and returns the smallest value in the array. Four synthetic versions were developed, a sequential and a parallel version for both scalar and vector arrays, and two native versions, one in Python and the other in C++. For the each example, the kernel was executed 10 times and the average time was recorded. All kernels operated on the same 10 million element array from the same Python instance.

1.1. Details. The timing results were obtained on a Dual 2.5 GHz PowerPC G5 with 3.5 GB RAM running Mac OS X 10.4.6. The Python version was 2.3.5 (build 1809) with Numeric Python 23.8 and PyOpenGL 2. The native extensions were built using gcc 4.0.1 (build 5250) with -03 and SWIG 1.3.25. Native versions were also tested with IBM’s XLC 6.0 compiler. The results did not differ significantly between the compilers and the results for gcc are reported. Times were acquired using the Python \texttt{time} module’s \texttt{time()} function.

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

```python
1  def var_min(c, data):
2      m = var(c, _max_float)
3      for x in var_iter(c, data):
4          m.v = fmin(m, x)
5      m.store()
6      return m

7  def vec_min(c, data):
8      m = vector(c, _max_float)
9      for x in vector_iter(c, data):
10     m.v = vmin(m, x)
11    m.store()
12    return m
```

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

\textsuperscript{1}In all cases, any extra computation in Python was included in the execution time.
The parallel versions of the vector and scalar examples extend the sequential implementations by including code to store results based on the thread’s rank. The code for the parallel scalar version is:

```python
1 def par_min(c, data, result):
2     min = var(c, _max_float)
3     rank = int_var(c, reg=c.r_rank)
4     offset = var(c, 0)
5
6     for x in parallel(var_iter(c, data)):
7         min.v = fmin(min, x)
8
9     offset.v = rank * synppc.WORD_SIZE
10     min.store(address(result), r_offset=offset.reg)
11
12     return
13
14     mn = min(result) # final reduction
```

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

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

The C++ implementation is:

```cpp
1 float fmin(unsigned int addr, unsigned int size) {
2   float *data = (float*)addr, min = 268435456.0;
3
4   for(int i = 0; i < size; ++i) {
5     min = std::min(min, data[i]);
6   }
7
8   return min;
```
### Table 1. Performance results for different array min implementations on a 10 million element array. The parallel speedup for vec.min is relative to var.min.

<table>
<thead>
<tr>
<th></th>
<th>Time (sec)</th>
<th>Speedup</th>
<th>Parallel Speedup</th>
<th>M Compares/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>1.801</td>
<td>1.0</td>
<td>–</td>
<td>5.6</td>
</tr>
<tr>
<td>var.min</td>
<td>0.033</td>
<td>55.2</td>
<td>–</td>
<td>306.6</td>
</tr>
<tr>
<td>par.min</td>
<td>0.017</td>
<td>106.4</td>
<td>1.93</td>
<td>590.9</td>
</tr>
<tr>
<td>vec.min</td>
<td>0.014</td>
<td>125.0</td>
<td>2.26</td>
<td>694.2</td>
</tr>
<tr>
<td>par_vec</td>
<td>0.010</td>
<td>177.5</td>
<td>1.42</td>
<td>986.0</td>
</tr>
<tr>
<td>C++</td>
<td>0.068</td>
<td>26.4</td>
<td>–</td>
<td>147.8</td>
</tr>
</tbody>
</table>

#### 1.3. Results.

The results of the timing experiments are listed in Table 2. In every case, the synthetic versions outperformed the native versions. The parallel implementations achieved good speedups over their sequential counterparts. The parallel speedup listed for vec.min is the speedup compared to the scalar implementation.

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

### 2. Matrix-Matrix Multiplication

As discussed in the introduction, matrix-matrix multiplication is one of the most studied problems in high-performance computing and capable of executing at near-peak performance on any given system. However, implementing a fast matrix-matrix multiplication is challenging, due primarily to the number of hardware factors that affect data-flow during its execution. [26] provides detailed overviews of the different strategies for optimizing
the algorithm, with a focus on different strategies for blocking the matrices to best utilize the cache hierarchy. Matrices are blocked by recursively sub-dividing the input matrices into blocks that can be used to compute some or all of the result for a given block in the result matrix. Blocking can occur on multiple levels and strategies exist for blocking based on row and column partitions, rectangular sub-matrices, and registers. Once a strategy is identified, the sizes of the blocks can affect overall performance.

To study the impact of mixing Python and synthetic programs, a collection of synthetic implementations of matrix-matrix multiplication were developed. The goal of the study was to identify the blocking level at which the Python code no longer impacted the overall performance of the implementation. A secondary goal was to use synthetic programming to develop an algorithm that achieved near-peak performance on a system.

2.1. A Blocked and Partitioned GEMM. The algorithm used here is the (GEPP, GEPB) implementation of GEMM as described in [26]. In this implementation, matrix-matrix multiplication (GEMM) is built using a kernel that operates on row/column partitions (GEPP). GEPP in turn is built on GEPB, which operates on a partition and a block, or rectangular sub-matrix. Finally, GEPB is optimized using one additional level of blocking on the registers. In the released version of Goto-BLAS for the PowerPC, GEPB also uses a very careful ordering of loads, stores, and fused multiply-add instructions not described in the paper.

The outer-most level of the algorithm, GEMM, computes the matrix product of $A = (M \times K)$ and $B = (K \times N)$ and stores the result in $C = (M \times N)$: The first blocking level, $GEPP$, computes the matrix product between a column partition of $A$, $A' = (M \times kc)$, and a row partition of $B$, $B' = (kc \times N)$: The partition-partition product is sub-divided into

---

$^{2}$The original paper actually is based on a column major ordering of the matrices and uses GEBP. This work used row major matrices and replaced GEBP with GEPB.
a partition-block product, GEPB, with the row partition from $B$ broken into blocks of size $kc \times nc$: The results of GEPB are stored in the corresponding column of $C$, $C' = (M \times nc)$.

To optimize the matrices for sequential data loads, both $A'$ and $B'$ are packed into temporary matrices, $tA$ and $tB$, respectively, before they are used. Different packing strategies are possible but in all approaches the goal is to have elements stored close to each other in based on the order they are accessed by the processor. Assuming row-major storage, for $A$, this simply means extracting the $|kc|$ rows from $A$ and storing them in $tA$. For $B$, it is sufficient to transpose the matrix block and store it in $tB$.

One final level of register blocking is used inside GEPB. The next elements from $mr$ rows in $tA$ and the next $nr$ elements from rows in $tB$ are read into registers and the resulting $(mr \times nr)$ sub-matrix of $C$ is directly computed in registers: This optimization limits the number of times values from $C$ are updated in memory. Finally, the values computed for $C$ are written to a temporary matrix, $C_{aux} = (mr, nc)$, which is added to $C$ at the end of each GEPB operation.

The pseudo-code for the algorithm is:

```python
1 def gemm(A, B, C, mc, kc, nc, mr, nr):
2     # A = (M x K), B = (K x N) are the input matrices
3     # C = (M x K) is the output matrix
4     # kc, nc are the block sizes for each dimension
```
# mr, nr are the block sizes for each register sub-matrix

# Temporary matrices
# tA = (M x kc), tB = (nc x kc), C_aux = (mr x nc)

# GEMM
for kk in range(0, K, kc):
    tA[:, :] = A[:, kk:kk+nc]

# GEPP
for jj in range(0, N, nc):
    tB[:, :] = B[kk:kk+kc, jj:jj+nc]

# GEPB
for i in range(0, M, mr):
    # Register Block
    for j in range(0, nc, nr):
        for k in range(0, kc):
            for ai in range(mr):
                a[ai].load(tA, ai * A_row_stride)

            for bj in range(nr):
                b[bj].load(tB, bj * B_col_stride)

            for ci in range(mr):
                for cj in range(nr):
                    c[ci][cj].v = fmadd(a[ci], b[cj], c[ci][cj])

            store(c, C_aux[:, j:j_nr])  # Save c registers to C_aux
            # /end for k
        # /Register Block

    C[jj:j+nc] += C_aux  # Update C
    # /GEPB
# /GEMM
return

The partition and block sizes – \( kc, nc, mr \), and \( nr \) – affect how efficiently data is moved between cache hierarchies and L1 and the processor. The best values for these parameters are determined experimentally by performing a parameter sweep across candidate values.
2.2. Synthetic GEMM. The sub-division of GEMM into GEPP and GEPB provides an opportunity to explore where the line between a synthetic kernel and the Python management code can be drawn. Performing progressive specialization on the algorithm will potentially step through three different synthetic implementations of GEMM, one with only GEPB implemented synthetically, one with synthetic GEPP, and finally a full synthetic GEMM. However, if there is no performance difference between the synthetic GEPB and GEPP as the matrices grow, then the Python management loops are not impacting the performance and a synthetic GEMM is not necessary.

The first synthetic GEMM, $\text{gemm\_syn\_gepb}$, implements GEPB synthetically and uses the Numeric Python $\text{transpose()}$ function to pack $B$ into $tB$:

```python
def gemm_syn_gepb(A, B, C, mc, kc, nc, mr, nr):
    gebp = SynGEPB(...); pm = ExecParams()
    
    for kk in range(0, K, kc):
        tA[:, :] = A[:, kk:kk+nc]
        pm.p3 = addr(C)

    for jj in range(0, N, nc):
        # Pack B into tB
        tB[:, :] = Numeric.transpose(B[k:k+kc, j:j+nc])
        proc.execute(gebp)
        pm.p3 += nc8

    return
```

In this version, the synthetic GEPB is synthesized with the addresses for $tA$ and $tB$ as constants and passed the current address of $C$ as a parameter.

Initial tests with synthetic GEPB showed that Numeric's $\text{transpose}$ was inflicting a noticeable penalty on small and medium sizes matrices. To adjust, a second synthetic program was added to $\text{gemm\_syn\_gepb}$ to pack $tB$. The new synthetic program takes the current address of $B$. The updated lines for $\text{gemm\_syn\_gepb}$ are:
pack_pm.p1 = B_addr + k * N * 8
for jj in range(0, N, nc):
    # Pack B into tB
    proc.execute(packb, params = pack_pm)
    proc.execute(gepb, params = pm)

pack_pms.p1 += nc8
pm.p3 += nc8

This version was used for the remainder of the study.

To implement the synthetic GEPP, gemm_syn_gepp, the synthetic programs for GEPB and PACKB were reused as the inner loop for GEPP. The Python manager for gemm_syn_gepp is:

def gemm_syn_gepp(A, B, C, mc, kc, nc, mr, nr):
    gepb = SynGEPB(...); pm = ExecParams()
    for k in range(0, K, kc):
        # Pack A into tA
        tA[:,:] = A[:,k:k+kc]
        pm.p3 = C_addr
        pm.p5 = B_addr + k * N * 8
        proc.execute(cgepp, params = pm)

The inner-most loop in all programs operates on a block of $C$ stored in a register. The values streamed from $tA$ and $tB$ represent the lowest level of data access. Assuming cache utilization is optimized, stalls created by the order of the instructions in the inner-most loop will have a profound impact on performance. With only 26 instructions in this loop when a $4 \times 4$ register block is used (8 loads, 2 pointer updates, 16 fmadds), a one-cycle in loads can impact performance by up to 30%.

Three different orderings were used by synthetic GEPP and GEPB for the inner-most loop. The default ordering simply loads the $A$ values, loads the $B$ values, computes the $C$ block, updates the pointers and repeats. The proximity of the loads to the instructions that use them suggests that unrolling the loop once and prefetch the $A$ and $B$ values may impact the results. The algorithm used for the prefetch versions of the synthetic programs is:

LOAD(A1)
LOAD(B1)
for k in range(kc / 2):
    LOAD(A2)
    LOAD(B2)
    C += A1 * B1
    LOAD(A1)
    LOAD(B1)
    C += A2 * B2

As the results will show, this prefetching algorithm had almost no impact on performance. To attempt to find a better ordering, the loads and computations were manually reordered. The hand reordering reported here is:

# Load order
load(a1[0], b1[1], b1[0], a1[1], b1[2], b1[3], a1[2], a1[3])
for k in range(kc / 2):
    load(a2[0], b2[1], b2[0], a2[1], b2[2], b2[3], a2[2], a2[3])
# Compute using a1 and b1
    c[0][0], c[0][1], c[1][0], c[1][1]
    c[1][2], c[0][2], c[2][0], c[2][1]
    c[2][2], c[2][3], c[0][3], c[1][3]
    c[3][0], c[3][1], c[3][2], c[3][3]
load(a1[0], b1[1], b1[0], a1[1], b1[2], b1[3], a1[2], a1[3])
# Compute using a1 and b1
    c[0][0], c[0][1], c[1][0], c[1][1]
    c[1][2], c[0][2], c[2][0], c[2][1]
    c[2][2], c[2][3], c[0][3], c[1][3]
    c[3][0], c[3][1], c[3][2], c[3][3]

2.3. Results. The synthetic programs were executed on square matrices with 128, 256, 512, 1024, 2048, and 4096 elements. A parameter sweep revealed that $4 \times 4$ register blocks, $kc = 256$ and $nc = 128$ gave the best results on a 2.5 GHz PowerPC 970 running Mac OS X 1.4.9. The PowerPC version of gemm from Goto-BLAS 1.07 was used to provide a control for achievable peak-performance.
The results are shown in figure 1. At large numbers, synthetic implementations with comparable inner-most loops converge to the same throughput rate. For smaller numbers, Numeric’s transpose has a detrimental impact that disappears when using the synthetic packing routine. While no synthetic version reached the performance attained by Goto-BLAS, hand-ordered inner-loop version demonstrated that inner-loop optimizations are affecting the overall performance of the synthetic programs. Examinations of Goto-BLAS gemm suggest that a custom packing strategy and inner-loop instruction ordering may provide additional performance increases. Fixing address in the synthetic code (i.e., maximizing cache use) generates numbers comparable to Goto-BLAS, suggesting that better packing strategies may lead to a peak-performance synthetic implementation.
3. Chemical Fingerprint Comparison

Chemical fingerprints are used in chemical informatics applications for comparing data collections and performing chemical database searches. They are implemented as bit vectors that encode properties of compounds in each bit. For example, one bit in the vector may encode the presence of a benzene ring while another bit may tell if the compound is hydrophilic or hydrophobic. In addition to being a compact method for storing large amounts of information, the bit vector representation also makes it simple to compare two compounds based on their properties.

While various methods for generating fingerprints exist, the Tanimoto/Jaccard metric for bit vectors is the most commonly used metric for comparing chemical fingerprints [29]. A typical database search compares a query compound against the subjects in the database and reports the subjects that have the highest similarity score. An extension to the single query database search is a full pair-wise comparison between all subjects in the database, with each compound compared against all the other compounds and the results used as input a data clustering algorithm.

A single database search is a linear operations, but a full pair-wise comparison generates a $N \times N$ matrix, where $N$ is the size of the database, and is a quadratic operation. A naive 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.

The chemical fingerprint application was developed to study progressive optimization and explore the design space for loop abstractions. It was implemented using the SPE and the results from this study were applied to the design and implementation of the Iterator library.

3.1. The Chemical Fingerprint Application. The chemical fingerprint application takes a database of compounds and generates the pair-wise similarity matrix of the database using the Tanimoto metric.
Figure 2. 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 run-time 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.

The pseudo-code for the entire algorithm is:

```python
1  data # collection of bit vectors
2
3  # Compute the upper-triangular matrix
4  for x in data:
5      for y in data[index(x):]:
6          tanimoto(x, y)
7
8  def tanimoto(x, y):
9      ab = popc(x XOR y)
10     c = popc(x AND y)
11     return c / (ab + c)
```

The standard Tanimoto metric computes $\text{similarity} = c/(a + b + c)$ 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 \texttt{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. Each fingerprint is a 166-bit vector, 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 \texttt{array} with a typecode of ‘I’.

3.2. Implementations. The chemical fingerprint application was first implemented in pure Python. This implementation established a performance baseline and provided the framework for working with chemical data-sets. Working from the Python code base, successive optimization iterations converted more portions of the core comparison algorithm to synthetic code. Finally, a C++ version was developed to compare the synthetic versions against a natively compiled version.

3.2.1. 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. \texttt{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 [42].

3.2.2. 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 \texttt{x} and \texttt{y} vectors. The result is returned as a bit vector for AND and XOR and an integer for \texttt{popc}. The remaining math for the comparison is implemented in Python. The \texttt{popc} implementation is able to take advantage of a special operation in the AltiVec. The \texttt{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 [6]. \texttt{vperm} can perform 16 look-ups in parallel.

3.2.3. Synthetic Tanimoto. The next synthetic implementation uses the previous bitwise 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 Writer’s Guide [30].

3.2.4. Continual Tanimoto. The previous two implementations fully specialize the instruction stream for $x$ and $y$ each time a comparison is performed. However, only the addresses of the $x$ and $y$ vectors change between calls 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 continuous specialization to update the existing stream when the addresses change.

3.2.5. 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.

3.2.6. Synthetic Matrix. The final synthetic implementation uses a second synthetic loop for the outer loop to 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 2 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 data sharing avoids the creation of temporary variables to pass values between components, removing one of the so-called “abstraction penalties” common in object-oriented systems.

3.2.7. 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.

3.3. Results. 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 5,000 compounds through 50,000 compounds.

The results for the single row comparisons are shown in Figure 2. 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 Continual 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>
<thead>
<tr>
<th></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>Cont. 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>

Table 2. 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.
The results for the full matrix comparisons are in Figure 3. 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, the synthetic algorithm uses operations for multi-word bit vectors on the AltiVec processor. There is not enough information available in the C++ code for the compiler to safely determine that the algorithm processes bit vectors and, as such, it can only optimize the code so much.

**3.4. Related Work.** 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 pilot 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. [48].

**4. Conclusion**

The case studies in this chapter demonstrate the utility of progressive specialization and provide examples of the performance gains possible with simple synthetic programs for
targeted kernels in an algorithm. The array minimum and chemical fingerprint examples show how small synthetic programs can compete directly with compiler optimized code. In both cases, synthetic programming gains a significant advantage over the compiled version by encouraging the developer to use vector instructions, an implementation decision a programmer working in C is discouraged from making. While the synthetic matrix-matrix multiply operation did not achieve the same performance as Goto-BLAS, it shows that, even for high-performance operations, entire algorithms do not have to be fully implemented at the machine level.

In each example, progressive specialization and constant testing was essential to guide the optimization. The bottom-up approach to optimization from a top-down implementation avoided unnecessary work and allowed systematic development and testing of each algorithm.
Synthetic programming is designed to be used in the context of larger application architectures. A number of applications were developed, both to research specific topics and provide input into the design of the synthetic programming environment. The first full-scale application developed was the interactive particle system. The particle system application presents the user with a window and a mouse cursor. As the user moves the mouse around the window, the mouse emits particles that bounce around the window and are affected by gravity and air resistance, updated using Euler integration.

Throughout the development of the synthetic programming environment, the particle system application was used to study different aspects of synthetic programming. Initially,
it provided a full application for developing progressive specialization techniques and integrating synthetic programs with third party libraries. As the synthetic programming environment evolved, the particle system was used to study library-defined semantics and usability by porting the application to the Variable and Expression libraries.

This chapter describes the design iterations of the particle systems application and how it was used to study synthetic programming.

1. The Particle System

![Figure 1. A “time-lapse” image of the particle system application.](image)

The particle system application (Figure 1) is an interactive application that models and displays a user-generated particle system. Particles are emitted as the mouse moves in the application window. The 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 particle system application contains three main subsystems: an OpenGL-based renderer and GUI, an event-handler for capturing mouse events and emitting new particles, and the physics loop for updating the particle positions. Particles and other simulation state are all stored in Numeric arrays. The particle data arrays, one each for position, velocity,
and color, are preallocated with a preset number of particles. The particle renderer uses GL_POINTS to draw each particle and renders all particles once per frame. The array containing the \((x, y)\) coordinates for the particles is passed as a pointer to the OpenGL runtime, minimizing the overhead of calling OpenGL from Python. The window is created using the GLUT library and event handling is managed using GLUT callbacks. In addition to the mouse events, the user can adjust simulation parameters such as gravity and air resistance while the application is running.

The physics loop uses Euler integration to update the particle positions. 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 physics and rendering systems are the most computationally intensive sections of the application. The rendering code is offloaded to the graphics hardware, leaving the physics loop as the primary code section to optimize.

The physics loop was implemented using three different techniques. Each technique is described in detail the next few sections and the approaches are compared against each other.

2. Numeric Particle System

The Numeric Python version, numparticle, implements the physics loop using Numeric Python operations. Numeric Python is a high-level array library that provides optimized implementations of many array operations that form the building blocks of more complex operations. Numeric arrays use standard Python list operators to access elements stored internally in C arrays. Numeric Python’s logical and arithmetic operations operate element-wise on whole arrays using a vector-like API. As an example, consider two arrays of integers, \(A\) and \(B\). The following code adds the elements of both arrays and store the results in \(B\):
B = A + B

Numeric Python makes use of operator overloading where possible to provide a more natural syntax but also includes functions for each operations that can be used to optimize operations:

```python
Numeric.add(A, B, B)
```

This example performs the same operation but has slightly different semantics. In the first example, a temporary array holds the result before it is copied to B. In the second example, the result is stored directly in B, avoiding the temporary array.

`numparticle` is implemented with Numeric Python array operations and avoids temporary arrays whenever possible. The source for `numparticle` is:

```python
def physics_numeric():
    # Gravity
    Numeric.add(v[:,1], GRAVITY, v[:,1])

    # Some air resistance
    air = Numeric.where(Numeric.greater(v, 0), 1.0, -1.0)
    Numeric.multiply(air, AIR, air)
    Numeric.add(v, air, v)

    # Numeric.subtract(points[:,1], 1, points[:,1])
    Numeric.add(points[:,0], v[:,0], points[:,0])
    Numeric.add(points[:,1], v[:,1], points[:,1])

    # 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])

    # Bounce off the ceiling and floor (damping at the floor)
    lt = Numeric.where(Numeric.less(points[:,1], 1), -.5, 1)
    Numeric.multiply(lt, Numeric.where(Numeric.greater(points[:,1], height), -1, 1), lt)
    Numeric.multiply(v[:,1], lt, v[:,1])

    # Add a 'floor'
```

DRAFT: May 23, 2007
7. APPLICATION: INTERACTIVE PARTICLE SYSTEM

28   Numeric.putmask(points, Numeric.where(Numeric.less(points, 1),
29     1, 0), [1.0])
30
31 return

Line 4 updates the $y$ velocity component of each particle with the coefficient of gravity. Lines 7-9 apply air resistance to both coordinates and lines 12-13 update the particle position. Lines 15-29 perform the bounds checks and update the particle positions accordingly. `numparticle` scales to 20,000 particles at 30 frames per second.

3. Synthetic Particle System, Version 1

The first synthetic implementation, `synparticle1`, replaces the Numeric physics loop with a synthetic program that performs the same sequence of operations using 4-element floating-point vectors and the AltiVec/VMX instruction set. The vector operations provided by AltiVec/VMX are similar to the full array operations in Numeric Python, but operate on 4-element vectors instead of full arrays. `synparticle1` iterates over the data arrays, brings in two particles for each iteration, and performs a full update on the particles before proceeding to the next pair of particles. The full source for `synparticle1` is:

1 def physics_altivec(self, code = None):
2   if code is None: code = self.code
3
4   # GP registers
5   r_points = code.acquire_register()
6   r_vels = code.acquire_register()
7   r_size = code.acquire_register()
8   r_index = code.acquire_register()
9   r_temp = code.acquire_register()
10
11   # Vec registers
12   v_point = code.acquire_register('vector')
13   v_vel = code.acquire_register('vector')
14   v_gravity = code.acquire_register('vector')
15   v_air = code.acquire_register('vector')
16   v_extents = code.acquire_register('vector')
17   v_zero = code.acquire_register('vector')
18   v_one = code.acquire_register('vector')
19   v_negone = code.acquire_register('vector')
20   v_negpt5 = code.acquire_register('vector')

DRAFT: May 23, 2007
v_mask = code.acquire_register('vector')
v_floor = code.acquire_register('vector')

# Load the constants
vec_float_splat(code, r_temp, array_address(self._fzero), v_zero)
vec_float_splat(code, r_temp, array_address(self._fone), v_one)
vec_float_splat(code, r_temp, array_address(self._fnegpt5), v_negpt5)
code.add(av.vsubfp(v_negone, v_zero, v_one))

# Load the addresses
synppc.load_word(code, r_points, array_address(self._points))
synppc.load_word(code, r_vels, array_address(self._vels))

# Load the current values for gravity (y word only) and air into vectors
vec_float_splat(code, r_temp, array_address(self._gravity), v_gravity)
code.add(av.vmrghw(v_gravity, v_zero, v_gravity))  # v_gravity = [-0, G, -0, G]
vec_float_splat(code, r_temp, array_address(self._air), v_air)

# Load the extents (use v_point/v_vel as temps)
vec_float_splat(code, r_temp, array_address(self._width), v_point)
vec_float_splat(code, r_temp, array_address(self._height), v_vel)
code.add(av.vmrghw(v_extents, v_point, v_vel))

# Create a vector with the floor damping factor
code.add(av.vmrghw(v_floor, v_negone, v_negpt5))

# Setup the index and counter
size = self._points.shape[0]
if (size % 4) != 0:
    print 'Warning: the number of points is not a multiple of 4', size,
    size = size - size % 4
    print '->', size

# Load the size (points * elts_per_point * bytes_per_elt)
synppc.load_word(code, r_size, size * 2 * 4)
synppc.load_word(code, r_index, 0)

# Pointer to the start of the loop
loop_start = code.size() + 1

# The main loop
# Operate on two points at a time
for l in [0]:  # indent :)

DRAFT: May 23, 2007
# Load the current vector
code.add(av.lvx(v_vel, r_index, r_vels))
code.add(av.lvx(v_point, r_index, r_points))

# Updated the velocity with gravity
code.add(av.vaddfp(v_vel, v_vel, v_gravity))

# Add some air resistance
code.add(av.vcmpgtfpx(v_mask, v_zero, v_vel, 0))
code.add(av.vsel(v_mask, v_one, v_negone, v_mask))
code.add(av.vmaddfp(v_mask, v_mask, v_zero, v_air))
code.add(av.vaddfp(v_vel, v_vel, v_mask))

# Update the point
code.add(av.vaddfp(v_point, v_point, v_vel))

# Bounce off the zero extents (floor and left wall)
code.add(av.vcmpgtfpx(v_mask, v_zero, v_point, 0))
code.add(av.vsel(v_mask, v_one, v_floor, v_mask))
code.add(av.vmaddfp(v_vel, v_vel, v_zero, v_mask))

# Bounce off the positive extents (ceiling and right wall)
code.add(av.vcmpgtfpx(v_mask, v_point, v_extents, 0))
code.add(av.vsel(v_mask, v_one, v_negone, v_mask))
code.add(av.vmaddfp(v_vel, v_vel, v_zero, v_mask))

# Add a 'floor' at y = 1.0 so the points don't disappear
code.add(av.vcmpgtfpx(v_mask, v_one, v_point, 0))
code.add(av.vsel(v_point, v_point, v_one, v_mask))

# Save the new velocity and position
code.add(av.stvx(v_vel, r_index, r_vels))
code.add(av.stvx(v_point, r_index, r_points))

# Increment the index
code.add(inst.addi(r_index, r_index, 16))

# Compare index against size and branch
code.add(inst.cmpw(0, r_index, r_size))

next = code.size() + 1
code.add(inst.bne(-(next - loop_start) * synppc.WORD_SIZE))

# /end loop

# debug

DRAFT: May 23, 2007
7. APPLICATION: INTERACTIVE PARTICLE SYSTEM

This listing illustrates key features in all synthetic programs and shows how to implement common operations using the ISA module directly. The function starts by allocating all the general-purpose and vector registers used in the synthetic program (lines 4-22) and creating five constants: 0 (line 25), 1 (line 26), −0.5 (line 27), and −1 (line 28). Next, the run-time parameters, including the coefficients of gravity and air resistance, along with the display extents, are loaded into vector registers from memory buffers (lines 24-46).

The mechanics of the physics loop (lines 48-110) are handled by testing the offset into the data array against the total size of the array (line 105). When the values are not equal, a branch instruction (line 108) loops back to the start, computed from the index of the first loop body instruction (calculated in line 60) and the index of the branch instruction. The loop iterates \( n/4 \) times and is initialized in lines 48-60.

The body of the loop spans lines 68-105. At each iteration, it loads the current four element position and velocity vectors (lines 68-69), 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 (lines 72, 81, resp.). 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 (e.g., line 84). The new vector is used as an argument to the vse1 instruction (line 85), 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 (line 86). The final synthetic instruction sets an integer return value (line 113).

While four times as long as the numparticle, the actual physics code in synparticle1 is very similar to the Numeric physics code. For example, consider the air resistance code in both implementations. Line 7 in the Numeric version contains two operations, a greater-than comparison and a selection using where. These two operations are mirrored in lines 75 and 76, respectively, in the synthetic code. The next two lines in each version, lines 8 and 9 in the Numeric code and lines 77 and 78 in the synthetic code, update the velocity component with the result. The remainder of the physics code has similar correspondences between the two versions.

The synthetic physics loop scales 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.

The performance increase is due to two main factors. First, synparticle1 uses vector instructions for the updates, allowing four updates per instruction, compared to only one update per instruction in numparticle. There is no guarantee that Numeric Python uses vector operations for arithmetic and logical operations, leaving an important optimization decision out of the control of the developer.

At a more fundamental level, numparticle's performance is limited by the design of Numeric Python's array operations. synparticle1 performs a full update on each point before proceeding to the next point. Each point is loaded into a register, updated, and stored. numparticle, on the other hand, performs each operation in the update on the entire particle array before proceeding to the next operation. Each point is moved between
registers and memory once per operation. Because loads and stores are expensive compared to operations, doing multiple loads for the same data significantly degrades performance. By focusing on developing algorithms on the hardware, the synthetic programming model encourages careful evaluation of the effects of memory accesses and as a result can lead to code designed to optimize memory bandwidth utilization.

4. Synthetic Particle System, Version 2

The second synthetic implementation, synparticle2, uses the Variable and Iterator libraries to implement the physics loop. The libraries hide the complexity of moving data into and out of registers and provide a natural syntax for expressions. The source code for synparticle2 is:

```python
1 def physics_altivec(self, code = None):
2     if code is None: code = self.code
3
4     # Data
5     vels = vector_iter(code, self._vels, length = self._vels.shape[0] * 2)
6     points = vector_iter(code, self._points, length = self._points.shape[0] * 2)
7
8     # Constants
9     zero = vector(code, 0.0)
10    one = vector(code, 1.0)
11    negone = vector(code, -1.0)
12
13     # Parameters
14    gravity = vector(code, self._gravity)
15    air = vector(code, self._air)
16    extents = vector(code, self._extents)
17    floor = vector(code, self._floor)
18
19     for vel, point in zip_iter(code, vels, points):
20         # Forces - Gravity and Air resistance
21         vel.v = vel + gravity
22         vel.v = vel + vmadd(vsel(one, negone, (zero > vel)), air, zero)
23         point.v = point + vel
24
25         # Bounce off the zero extents (floor and left wall)
26         # and positive extents (ceiling and right wall)
27         vel.v = vmadd(vel, vsel(one, poszero, (zero > point)), zero)
28         vel.v = vmadd(vel, vsel(one, negone, (point > extents)), zero)
```

DRAFT: May 23, 2007
7. APPLICATION: INTERACTIVE PARTICLE SYSTEM

<table>
<thead>
<tr>
<th></th>
<th>Parameters</th>
<th>Loop/Iters</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>–</td>
<td>–</td>
<td>13</td>
</tr>
<tr>
<td>Syn. AltiVec</td>
<td>43</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>Syn. Expr/Iter</td>
<td>8</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1. Lines of code allocated to parameter allocation, loop and iterator management, and the update algorithm for the three different implementations of the particle system update function.

```c
29 # Add a 'floor' at y = 1.0 so the points don't disappear
30    point.v = vsel(point, one, (one > point))
31
32    release_all(code, zero, one, negone, gravity, air, extents, floor)
33
34    return
```

The physics code generated by synparticle2 is nearly identical to the code generated by synparticle1. The loop code varies slightly and includes a few additional instructions in the loop initialization code, but the overall element-wise nature of the loop is preserved. This version also scales to over 200,000 particles at 30 frames per second.

5. Particle System Comparison

While the performance evaluations provide a good demonstration of the benefits of programming at a low level, the different implementations of the particle system also make it possible to compare the general expressiveness of the different approaches.

To compare the expressiveness of the synthetic and Numeric versions, comments and blank lines were removed and each line was assigned a code marking it as parameter allocation, loop and iterator management, or algorithm implementation. The results are listed in Table 1. All three versions use the same Numeric arrays to store parameters and iterators, and the Numeric version did not require any additional parameter, loop, or iterator code.

The line counts demonstrate the utility of synthetic expressions and iterators and also show the similarity between the numeric and AltiVec/VMX physics loops. The original synthetic kernel contained 77 lines of code, 63 of which were used for register management and manual loop maintenance. In contrast, the new synthetic kernel uses only 11 lines to
manage the same operations, all of which use a clear syntax. Both the `numparticle` and `synparticle1` used similar amounts of code to implement the update algorithm. Because the AltiVec ISA contains many instructions that have direct Numeric counterparts the code for both versions is similar.

`synparticle2`, on the other hand, uses only six lines of code to implement the update. While `numparticle` could have been implemented using similar syntax, Numeric’s aliasing rules lead to multiple, unnecessary temporary arrays. Because the expression implementation works on registers, the cost of temporary values is kept to a minimum, allowing a more concise syntax.

6. Conclusion

The different implementations of the particle system demonstrate different aspects of synthetic programming and make it possible to compare and contrast different techniques. The overall performance gains generated using synthetic programming provide a strong case for progressive optimization using the synthetic programming environment for applications with small, well-contained computational kernels.
Application: BLASTP on the Cell BE

Synthetic programming is an effective technique for optimizing small, performance critical sections of applications on commodity processors, as demonstrated in the previous chapters. To study how well synthetic programming scales to larger applications and how effective it is for managing multi-core execution, an implementation of the BLASTP algorithm was developed for the Cell BE using the synthetic programming environment. BLASTP is an important bioinformatics application that allows scientists to quickly search databases of DNA or protein sequences. BLAST is a multi-stage algorithm that lends itself to a multi-core implementation, making it an ideal candidate for testing synthetic programming on the Cell BE.
Previous parallel and high-performance implementations of BLAST have focused on coarse-grained parallel decomposition of the database and small, targeted “hot-spot” optimizations. The synthetic implementation of BLAST, SYN-BLAST, takes a different approach to implementing the original BLASTP algorithm, with a focus on SIMD optimizations and multi-core decomposition. This development strategy allowed synthetic programming to be studied for practical use as a rapid development tool for SIMD and multi-core development. Many insights gained from this exercise were integrated back into the synthetic programming environment, the most important being the techniques for modeling and handling memory.

For this study, the BLASTP implementation was designed and developed from scratch for the Cell BE using the synthetic programming environment. Developing from scratch made it possible to radically redesign the data flow and computational kernels used in the search process. The new design led to a number of innovative techniques for using synthetic programs with Python in inherently parallel environments. For example, the manager/worker model used to control the SPU sub-programs uses a novel application of Python generators for load balancing and task assignment.

This chapter describes the design and implementation of SYN-BLAST. The first sections provide an overview of BLAST and related strategies for parallel implementations of BLAST. The next sections detail the SIMD kernels and management code and cover how synthetic programming was used to rapidly proceed through design iterations.

1. BLAST Overview

BLAST [3] is one of the most important tools to emerge from the bioinformatics community. BLAST is a fast heuristic algorithm for searching a database of DNA (genomic) or protein sequences for matches against a query sequence. The heuristic optimizations employed by BLAST reduce the search process from a quadratic comparison problem to a multi-stage linear database scan, making database searches a practical tool for working scientists. To handle the different sequence types and matching strategies, many versions of BLAST have
been developed. These include versions for scanning DNA and protein databases and pattern matching algorithms that can incorporate additional position-specific information to provide more sensitive searches [PSI-BLAST] [4, 25].

Genomic and proteomic sequences from different species may be related through evolutionary paths. When a new sequence is identified, one of the first tasks is to find known sequences that are similar to the new sequence. If similar sequences are found from different species, their function may provide clues to the function of the new sequences. Due to the mutations that occur throughout the evolutionary process, related sequences are never identical. Individual coding elements may be replaced by different elements, or sections of a sequence may be lost or inserted without having significant impact on the function of the gene generated by the sequence. Sequence search algorithms have to take such changes into account, making the general problem of finding similar sequences a difficult one.

BLAST was developed to provide a fast heuristic method for searching large genomic databases. Rather than compare every sequence fully, BLAST utilizes a two stage filtering process, looking for short, exact matches in the first stage, and extending the matches in the second stage. In each stage, significance scores determine which sequences are forwarded to the next stage. A final sorting step orders the results and presents the top results to the user.

BLAST is built around a statistical foundation that uses small, exact matches, or hits, between regions in the query and current subject as seeds for determining the overall similarity score. BLAST starts by building a list of fixed-length words based on words that occur in the query sequence that generate a high score when compared to any possible word in the database. Scores are determined using scoring matrices that assign a score to a direct comparison between two characters in the sequence alphabet. Word match scores are simply the sum of the scores of individual comparisons. If the word score is above a certain threshold, it is considered significant and used to scan the database.

For example, consider the string “thisisaquery”. With a word size of 3, the words that are tested for high-scores are:
query_words = [thi, his, isi, ..., uer, ery]

The algorithm to filter this list into a list of high-scoring words is:

```python
for word in query_words:
    for x in alphabet:
        for y in alphabet:
            for z in alphabet:
                if check_score(word, x|y|z):
                    save_hsw(x|y|z)
```

Once the set of high-scoring words is determined, the database is scanned for matches to the words. If a match is found, it is extended in both directions until the extended score starts to decrease from its maximum score. The extended word is called a high-scoring pair (HSP) and its score contributes the overall similarity score between the query and subject sequences. A variation on the hit extension algorithm, called two-hit extension, waits until two sufficiently close words have been found and then extends each word in one direction. This optimization reduces the number of hits that are extended while increasing the quality of the resulting HSPs.

Assuming the word “his” is a high-scoring word from the previous query, a database extension might be:

```
iitistruethathisisnottheone
    thisisaquery
<|||>>
```

After the HSPs are found, the original BLAST algorithm used the results to report the most significant hits in the database. However, it was noted that these so-called ungapped alignments often missed important matches between sequences [REF]. To address this, an alternative hit-extension algorithm was introduced that performed a gapped extension. The gapped extension algorithm is based on the Smith-Waterman local alignment algorithm [63] and is used to extend the original hit in both directions.

SYN-BLAST is an implementation of the ungapped BLAST algorithm for protein database searches. Where the literature was ambiguous on algorithmic details, the canonical
BLAST implementation supplied by the NCBI was used to determine the proper interpretation.

2. Related Work

Due to the number of whole genomes available as well as the development of high-throughput DNA microarrays [13, 17, 31], 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 [36, 60] by matching mapped regions in one species against unmapped regions in another species [22].

Another study performed by the author 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. This 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 were applied to the synthetic BLAST implementation.

In addition to the dot plot algorithm, a few other SIMD algorithms have been for bioinformatics applications. A version of BLAST developed by Apple and Genentech, AGBLAST [28], 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 [40] 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 [57]. Using the MMX vector unit in Intel processors [32], a 6x speedup was achieved.

---

1Here, SIMD refers to the vector processors found in modern workstations, not special purpose SIMD machines such as those from MasPar and Thinking Machines.
Parallel implementations of BLAST have also been developed. The main implementation is mpiBLAST [18], which distributes the database across the available processing nodes. ScalaBLAST [49] 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 a key bottleneck in mpiBLAST. BLAST has also been ported to more exotic architectures, including the Blue Gene/L [55].

3. Synthetic BLAST

The synthetic implementation of BLAST for the Cell BE decomposes the original BLAST algorithm into three separate sub-programs: the manager, the hit finder, and the ungapped extension kernel. The manager is a Python program that runs on the Cell’s PPU, while the hit finder and ungapped extension kernel are synthetic programs that execute on the SPU. The two SPU programs are SIMD implementations of the respective algorithms. SYN-BLAST is the first complete multi-core and SIMD implementation of the BLAST algorithm.

3.1. Cell BE Overview. The Cell BE processor is a heterogeneous multi-core processor with nine dedicated processing cores and a multi-tiered, user programable memory hierarchy. The Cell BE is available in a few hardware platforms, including Blades from IBM and the PlayStation 3 from Sony. SYN-BLAST was developed primarily for use on IBM QS21 Cell Blades and has also been tested on a PlayStation 3 running Linux.

A Cell BE Blade consists of two Cell BE cores, each with a coherent view of physical memory. Each Cell BE core contains a dual core PowerPC processor and eight synergistic processing element (SPE) cores. The PowerPC and SPE cores are connected to the main memory via a high-speed bus and the SPE cores are further interconnected using a local ring network. On the blade, the physical memory is divided into two equal sized segments connected by a high-speed bus that provides the coherent view of memory. While current implementations of Linux for the Cell BE and the SPE run-time libraries do not expose routines for exploiting processor affinity or locality, it is important to be aware of the underlying

2The original synthetic programming environment, abbreviated as SPE, shares an acronym with the Cell BE synergistic processing element. In this chapter, all references to SPE refer to the Cell BE SPE.
physical properties to help inform design decisions. The production QS21 Blades used for this work run at 3.2 GHz and have 1 GB main memory (512 MB/core).

The SPEs are vector processors with 256 KB of local memory for both data and instructions. The SPEs are composed of two processors, a synergistic processing unit (SPU) and a memory (DMA) controller. The SPU and DMA controller execute in parallel, allowing the SPE to overlap computation and communication. Under ideal operating conditions, an SPE can transfer data between its local store and main memory at around 20 GB/s [37]. For data-bound applications, however, throughput is limited by the amount of data that can be transferred into SPU registers, which our experiments suggest is around 2.5 GB/sec for small data processing kernels.

The Cell BE has two design elements that require special attention for developing high-throughput algorithms. First, the PPU on the Cell is a relatively low-performance processor. Its responsibility is to act as a program manager, rather than a dedicated data processor. It is important keep the amount of performance-critical code executing on the PPU to a minimum. The second element that affects performance is the overall cache hierarchy. Main memory is cache-coherent for the PPU and SPUs. While the SPUs do not have hardware managed caches, they can still cause cache related stalls by accessing dirty memory and forcing the PPU to flush its cache. Rather than move data through the PPU, it is important to have the PPU communicate memory locations to the SPUs via memory references rather than copied blocks of memory.

3.2. Mapping BLAST to the Cell BE. The architecture of the Cell BE is a radical departure from the processors that typically execute BLAST. The lack of an auto-parallelizing and auto-vectorizing compiler for the Cell BE makes it difficult to automatically generate an optimized version of BLAST for the Cell BE from existing code bases. Rather than optimizing hotspots in the NCBI implementation of BLAST, SYN-BLAST takes a holistic approach to remapping the general BLAST algorithm to the Cell BE's unique execution environment. SYN-BLAST breaks the two main filtering stages, the hit finder and the ungapped extension
SYN-BLAST Application Architecture

SYN-BLAST is based on a manager/worker decomposition with the manager executing on the PPU and the workers on the SPUs. The manager maps the sequence databases to main memory and partitions the workloads out to the worker programs by passing the addresses of the next working sets. The hit finder performs the first level of filtering, searching for high-scoring exact matches. Hits are stored back to main memory and a pointer to the hits is passed to the ungapped extension code, which proceeds to extend each hit. High-scoring extensions are written to main memory for final processing by the manager. The workers, executing in a limited memory environment on the SPUs, process the database and sequences in fixed-sized blocks (red sections) and store their results back to main memory.
8. APPLICATION: BLASTP ON THE CELL BE

kernel, into separate sub-programs that execute independently on the SPUs. Each sub-
program was redesigned from the ground up to utilize the SIMD instructions available on
the SPU. The sub-programs are controlled by a manager program executing on the PPU.
The manager, developed in Python, creates the synthetic sub-programs, manages the data-
base, and coordinates data flow throughout SYN-BLAST. The manager also generates the
high-scoring word table from the query sequence. Figure 1 shows the overall architecture
of SYN-BLAST.

The hit finder and extension kernel are implemented as memory resident programs
on the SPU. Each synthetic program has a command loop that receives the next unit of
work or the terminate signal from the manager. This pattern helped avoid the restart costs
associated with allocating a new SPU for each work unit.

3.3. The Hit Finder. The hit finding kernel in BLAST looks up each $k$-element word
in a sequence in the high-scoring word table and reports a hit if the score for the word is
non-null. Significant hits are determined based on the statistics of the sequence family. As a
result, the hit finder is able to filter out almost 95% of the sequences in any given database.
$k$ varies depending on the type of sequences used, but for protein databases is almost always
kept at the NCBI default value of 3. The hit finding algorithm is inherently a data streaming
algorithm, with each element in the database read only once. The SYN-BLAST hit finder
takes advantage of the common word size and data access characteristics to implement an
extremely compact data-parallel version of the algorithm.

Rather than operating on a single sequence, the SYN-BLAST hit finder processes sixteen
sequences in parallel. The input sequences are stored in main memory in a pre-computed,
strided database, with the next element in the sequence separated from the previous one by
sixteen characters (Figure 2). Storing the data this way makes it possible to use the SIMD
instructions directly, without having to reformat a non-strided database at run-time.

Using the strided data as input, the pseudo-code for the SIMD hit finder is:

```
1  def spu_hit_finder():
2
3   while next_cmd() != DONE:
```
8. APPLICATION: BLASTP ON THE CELL BE

```python
for vec in stream:
    chars = expand_chars(vec)
    words = update_current(chars, words)
    keys = update_keys(words, keys)
    hits = lookup_hits(keys)

    if hits != null_hit:
        save_hits(hits)
        report_hits()

return
```

The outer loop shows the command loop, which reads the location of the next sequence stream block to process. Once the stream is loaded into the local store, a 16-character vector is loaded into a register. Because the database is strided, the characters in the vector are from sixteen independent streams. The input vector is then separated into four 4-element vector registers (line 5 and Figure 2). The SPU lacks instructions for arithmetic on characters, so the algorithm must operate on sixteen 32-bit values instead of sixteen 8-bit values. The large number of registers available on the SPU makes this practical.

Operating on four registers at a time also helps provide a natural element of instruction level parallelism. Because the operations on the input registers are independent, issuing four identical instructions in a row (one for each input register) allows the issued instructions to proceed without dependency stalls. This allows the instructions for the composite 4-vector operation to proceed in parallel through the pipeline.

The current word in each of the sixteen streams is encoded using radix-24 numbers (there are 24 elements in the protein alphabet). Line 6 updates the registers that store the current word and line 7 uses the results to generate the radix-24 encoding of the current word for each stream.

Pattern matching using radix encoded strings is defined by the Rabin-Karp algorithm [16]. The main idea is to store each word as a single number that is the sum of \( k \) scaled radix-\( d \) numbers. For instance, if \( n = 10, k = 3 \) and the alphabet is the decimal digits, the encoding \( s_1 \) of 345 is formed by:

\[
s_1 = 3 \times 100 + 4 \times 10 + 5 = 345
\]
The hit finder reads in 16-element vectors and expands them into 4 4-element registers (left). Each element contains a character from one of 16 sequence streams. The current word in each stream is encoded as a radix-24 integer, using Rabin-Karp algorithm to compute the integer value and use it as a lookup key to the high-scoring word table.

For longer sequences, the encoding of the next word can be computed in constant time. For example, if the next digit in the string is 6, the next encoding is computed with:

$$s_2 = (s_1 - 3 \times 100) \times 10 + 6 = 456$$

More generally, given a sequence $S$ encoded in radix-$d$ with element $i$ denoted by $S[i]$, the first length $m$ word in the sequence is computed by:

$$S_0 = S[m] + d(S[m - 1] + d(S[m - 2] + ... + d(S[2] + dS[1])...))$$

and the subsequent encodings are computed by:

$$S_{t+1} = d(S_t - d^{m-1}S[t + 1]) + S[t + m + 1]$$

The encoded word forms a lookup key into the high scoring word table. Because the hit finder limits the word size to the common protein size of 3, and the scores never exceed the range of a 2-byte integer, the entire HSW table can be stored in 27,648 bytes in the local store. By locating the HSW table at address 0, the radix-24 key for each word multiplied by 2 corresponds to the address of the score for that word.

Once the keys for each of the sixteen streams have been computed, hit finder switches to a sequential lookup stage. The SPU does not have a parallel load instruction, so each key
must be looked up individually. The scores for all sixteen words are loaded and stored in 4 new registers. Once all scores are loaded, the scores registers are examined for hits (line 9). If a hit is found, the hit is saved in the results block (lines 10-11). Once the current block is processed, any hit results are stored back to main memory for further processing by the ungapped extension kernel (line 13).

The hit finder divides local store into three sections plus one section for the program code (Figure 1). As noted above, the HSW table is stored at the beginning of the local store to provide a close mapping between word keys and table addresses. The size of the subject database block is fixed at synthesis time and the block follows the HSW table in memory. The remainder of local store is available for storing results.

3.3.1. Performance. The vector processing loop in the hit finder is implemented using 226 instructions. Assuming no hits are found, and thus the branch to save results does not occur, and an instruction is issued every cycle, hit finder can process a theoretical peak of 14.1 M vectors/second, or 226 M-bases/second.

Figure 3 shows the measured performance numbers for two experiments.
3.4. The Ungapped Extension Kernel. After hits have been identified by the hit finder, they are extended using the original BLAST ungapped extension algorithm. Unlike the hit finder, ungapped extension does not lend itself to a data-parallel implementation. However, various instructions in the SPU instruction set were still useful for optimizing the sequential implementation of the algorithm.

The pseudo-code for the SPU ungapped extension kernel is:

```python
1   def spu_ungapped_extend():
2     load_hit_block()
3     for stream in [0,1,...15]:
4       load_stream(stream)
5       for hit in hit_block:
6         if hit[stream] != null:
7           right_extend()
8           left_extend()
9           if score > cutoff:
10              save_extension()
11     return
```

The extension kernel operates on blocks of data streamed in from main memory (line 3). The data blocks are the result blocks produced by the hit finder. The results are stored in four 4-element vectors, matching the sixteen streams processed by the hit finder. Because the extension kernel operates on one stream at a time, it first processes the first element in all result vectors before proceeding to the next element (line 4). This allows the kernel to load the current sequence once for the whole result set, rather than once for each result vector (line 5).

Once a stream is loaded, the element in each result vector corresponding to the stream is examined for a hit (lines 7-8). Because results are stored in the hit finder if any of the sixteen streams contains a hit, and because hits are relatively low probability events, most of the results in any given vector are null and no extension occurs. When a hit is encountered, the hit is extended in both directions until the score starts decreasing (lines 9-10). If the combined extension scores are greater than the acceptance cutoff, the extension is saved (line 12).
Stream Shift

Stream shifting is a technique for loading vectors that are not aligned on natural boundaries. The two vectors that straddle the natural boundary are loaded and a shuffle mask is used to extract the unaligned vector into a register. It is possible to process both forward and reverse streams using the same code by simply providing a different shuffle masks for forward and reverse iteration.

The algorithm for extending a hit to the right and left is identical, except for the direction of iteration. For right extension, the query and subject sequence indices increase, and for left extension, they decrease. In a purely sequential implementation, the extension code can be easily reused by simply changing the sign on a pointer update:

```c
int direction = -1; // left extension, use 1 for right
char *query = query_hit;
char *subject = subject_hit;
while(continuing) {
    query += direction;
    subject += direction;
    // compute extension
}
```

Vector operations complicate the implementation algorithm. Instead of loading a single character for each iteration, sixteen characters are loaded into a vector register. Those sixteen characters must be aligned on a 16-byte boundary, which may not match the location of the hit. To handle unaligned loads, a stream shift operation is used. A stream shift loads two vectors from the stream (32 bytes total) and uses the shuffle instruction along with a shuffle control vector to extract the unaligned stream into a single 16-byte register (Figure 4).
Stream shifting is essential for processing arbitrary data streams, and its implementation using shuffle has an additional benefit for the extension kernel. The shuffle operation can be used to reverse the order of a sequence, allowing the forward iteration code to be used for both forward and reverse iteration (Figure 4). Once the vector is loaded, the kernel is not aware of the direction, only that it must compute a running score. The synthetic code for extension is implemented once, and a different vector and index incremener are used to switch between forward and reverse iteration.

The local store for the ungapped extension kernel contains two constant data sections, two blocked sections, and one variable length section for the results (Figure 1). The two constant sections, initialized at startup, are the scoring matrix and the query sequence. The scoring matrix, stored starting at address zero, is used to compute the pairwise score between characters in the query and subject sequences. Following the query are the hit block that contains the results from the hit finder and the current subject database block. Unlike the strided database used by the hit finder, the database used in the extension kernel contains the sequences sequentially, with the sequences loaded in fixed size blocks. The remainder of the local store is reserved for storing extension results.

3.4.1. Performance. The ungapped extension kernel executes 1702 instructions per hit if the extension terminates after the the first character. If no hits are extended, the peak theoretical peak performance is 1.8M extensions/second. The observed performance on a sample database is 1.1M extensions/second.

3.5. Python Manager. The hit finder and ungapped extension kernel are synthesized and managed in a Python program executing on the PPU. The main method in the manager is:

```python
def run(self):
    self.process_query()
    self.load_database()
    self.init_hit_finder()
    self.init_ungapped_extender()
    self.start_hit_spus()
```
The SYN-BLAST Python manager coordinates data flow using two processors that manage the hit finder and extension kernel. The processors issue jobs to the synthetic programs running on the SPUs and communicate results using Python lists and queues. Here, Python objects and data structures are green and red, respectively, while synthetic programs are blue.

```python
9   self.start_extend_spus()
10  hit_tasks = self.hit_processor()
11  extend_tasks = self.extension_processor()
12
13  while self.hit_processor_running or self.extension_processor_running:
14      if self.hit_processor_running:
15          hit_tasks.next()
16      extend_tasks.next()
17
18  report_results()
```

At startup, the manager pre-processes the query and forms the high scoring word table (line 3). Next, it opens the databases as memory mapped files and collects statistics on the databases (line 4). The query size, HSW table, and database statistics are used to synthesize the hit finder and ungapped extension kernel (lines 5-6). Once the synthetic programs are created, they are started on available SPUs. At this point, the synthetic programs terminate and stay resident, waiting for the restart command from the manager. The actual management of the hit finder and extension kernel is handled by two special purpose generators, the `hit_processor` and `extension_processor`. Each generator is a coarse-grained job manager that monitors the running SPUs and assigns work to them as they complete.
their previous tasks (lines 14-17). Once the generators have no more work to process, the results are tallied, sorted, and reported to the user (line 9).

Python generators [58] are functions that yield execution at specific points. Python provides support for generators using the `yield` keyword. For example, the following function is a generator that generates the integers:

```python
def integers():
    current = 1
    while True:
        yield current
        current += 1

ints = integers()
one = ints.next()
two = ints.next()
```

The yield statement suspends the function and returns the value of `current` to the caller when the next value is requested.

Job assignment is managed by the `hit_processor` and `extension_processor` generators (Figure 5). Each generator contains a loop that iterates over the work units for the particular tasks. At each iteration, the loop reaps idle SPUs and assigns new work to them. At the end of the loop, it yields execution back to the main manager, which in turn iterates the other generator one cycle. This approach gives both the manager and the generators a predictable level of control over execution scheduling not possible with Python user-level threads.

The `hit_processor` assigns blocks of the database to the hit finder for processing. It keeps a list of assigned blocks and SPUs and, as results arrive, stores a pointer to the results and the source blocks in a queue use as input to the extension kernel. Once the hit finder has finished processing the database, the `hit_processor` releases its SPUs and shuts down.

The `extension_processor` pulls results off the `hit_processor`'s result queue and passes them onto the extension kernel. An early version of the `extension_processor` performed a filtering step and removed the null hits from the results. Performing the filtering in Python limited the extension throughput to 10K results/second. Moving the filtering into
the extension processor allowed the throughput to increase to 1.1M results/second, an improvement of two orders of magnitude. When the hit.processor releases its SPUs, the extension.processor launches new extension kernels on them to process the remaining hit results. Extension results are saved back to main memory and stored in a list by the extension.processor.

3.6. Data Management. Data management is important for achieving a good level of performance in data-bound applications on the Cell BE. SYN-BLAST is designed to minimize the amount of data that moves through the cache on the PPU. SYN-BLAST has two sources of streaming data that must be carefully managed. The first consists of the databases, the strided and linear versions of the genomic database, that are used by the hit finder and extension kernel, respectively. The second data stream contains the hit results generated by the hit finder. To minimize the contact the PPU has with both streams, the hit and extension processors refer to the actual data blocks by passing addresses to the synthetic programs executing on the SPUs. At no point do they touch the memory directly.

The strided database is the only database that is streamed in its entirety. The database itself is pre-computed from the linear database and is stored on disk. Both copies of the database are memory mapped, leaving it up to the operating system to decide how best to page the data into main memory as data is requested by the SPUs. An alternative would be to create buffers and use PPU code (either Python or synthetic) to read in data elements, a process that would needlessly waste PPU cycles. Because of the very low performance of the disks, no additional efforts were made to increase the performance of the data streams. This should be revisited once a Cell BE platform is developed that is better suited towards large data streams.

The hit results buffers are resident in main memory and, like the databases, only ever directly accessed by the SPU programs. The hit results queue stores meta-data about the buffers, but never touches them directly. The extension results are processed at the end by the manager, but only after the extension processor has completed. This limits PPU access to the buffer to an execution stage where there is no contention for the memory regions.
4. Discussion

SYN-BLAST was developed primarily to explore the utility of synthetic programming for developing full applications on emerging hardware with limited development tool support. Synthetic programming proved to be a useful technique for developing low-level SIMD algorithms on the Cell BE. With support for the SIMD instructions in C coming from intrinsics, the ISA functions exposed by the synthetic programming environment provided a similar level of abstraction as C does for the SPUs. By providing direct control over the loop semantics, the synthetic programming environment iterators were actually more useful than C loops, where proper stream shifting and vector optimizations are taken as a matter of faith. The fast round-trip time between edits and execution made it possible to very quickly explore different approaches for implementing the SPU algorithms.

The design process did reveal a few challenges in developing large synthetic programs. Prior to SYN-BLAST, synthetic programs were limited to small kernels that were easy to express in a few instructions. The most involved form of control flow consisted of well-defined loops, which were easily abstracted by synthetic iterators. The hit finder and extension kernel are complicated by the fact that the SPUs do not have a Python runtime, requiring the complete algorithms to be implemented synthetically, rather than just the performance-critical sections.

The biggest challenge was finding an appropriate abstraction for conditional branches. Python iterators provided a syntactically concise abstraction for loops, but no similar language feature was available for condition branches. Compounding the design challenge is the distance between branches and branch targets. In the Python code, the target may not be synthesized when the branch is added, requiring additional code to fill in the branch instructions. Branching is handled by inserting a placeholder at the point of the branch and adding the real instruction at the end of synthesis, when the offsets can be properly computed.

The hit finder and extension kernel are both fairly complex synthetic programs. To manage synthesis, they were implemented as objects, with methods acting as subroutines.
for generating specific code sections. This approach made it easy to organize the algorithms, share variables and registers across methods as attributes, and reuse synthetic code.

An novel design pattern to emerge from the design of SYN-BLAST was the use of Python generators for multi-core task scheduling. Encapsulating each collection of jobs in a dedicated processor object that manages the SPUs simplified the main manager code. Python generator syntax made it possible to implement each processor as if they were simple loops. This encapsulated the job control login in well-defined code sections while still allowing the manager to have control over the overall execution patterns.

5. Conclusion

SYN-BLAST provided an important test of synthetic programming. It demonstrated that it is possible to use synthetic programming as a method for developing more complex applications. At the same time, it led to the development of novel approaches for managing multi-core execution on the Cell BE. This study also demonstrated the viability of synthetic programming as a rapid development tool. SYN-BLAST was developed over the period of one month, an aggressive schedule made manageable by the rapid development capabilities of Python and the capabilities of the synthetic programming environment on the Cell BE.
The final two applications presented in this thesis demonstrate techniques that are made possible by the combination of the synthetic programming environment and the unique multi-core execution environment of the Cell BE processor. Both applications utilize live, interactive sessions with an SPU to give the user direct control over the processor. The first application, the Interactive SPE, lets a user execute arbitrary commands directly on a running SPU using a dynamically-modified synthetic program. Building on this idea, the second application demonstrates a lightweight, interactive debugger for synthetic programs running on the SPU.
**Figure 1.** Interactive SPU The Interactive SPU program lets the user interact directly with a live SPU in real time. The display shows the current state of the registers, highlighting any registers changed by the previous instruction, along with a selectable history of user commands. In this example, the user used SPU instructions to generate hex poetry, populating register 117 with the phrase “A faded adobe cafe”.

Both applications are based on an unusual property of the Cell BE. SPUs can be stopped and restarted with minimal impact on their runtime state. While registers remain unchanged, external processes can change the contents of the local store while the SPU is asleep. This can be used to update not only data, but also code. Using the synthetic programming environment, the user can easily modify the instruction stream and use PPU-initiated DMA commands to update it on the SPU.

Explicit runtime control of not only the processor but also the code executing on the processor makes it possible to develop new techniques for executing and debugging code.

### 1. Interactive SPE

The Interactive SPU (ISPU) contains both a library and a user interface for executing single instructions on a running SPU and examining the results in the SPU’s registers (Figure 1). The library can be used from the Python interpreter for a text-based session while the GUI has a simple interface for entering commands and displaying the current contents of the register file.
ISPU is built around a small synthetic program, SYN-ISPU, that stays resident on the SPU, and the ISPU program, running in Python. SYN-ISPU is specifically designed to execute in an interactive manner. On startup, ISPU creates SYN-ISPU and launches it on an SPU. SYN-ISPU stores the contents of the registers and suspends execution. ISPU then reads the register contents from the SPU's local store and waits for the user to execute another command. When a command is executed, ISPU refreshes the SPU's local store with a new version of SYN-ISPU containing the next user command and sends the resume signal to SPU to execute the command and save the registers.

The ISPU method that generates SYN-ISPU is:

```python
    def synthesize(self):
        code = spu.get_active_code()

        # Reload the instructions
        spu.sync(1)

        # Next instruction to execute
        lbl_op = code.size()
        spu.nop(0)

        # Placeholders for register store instructions
        lbl_store = code.size()
        for i in range(128):
            spu.nop(0)

        # Stop for next command
        lbl_stop = code.size()
        spu.stop(0xC)

        # Loop back to the beginning - this is triggered by SIGCONT
        lbl_br = code.size()
        spu.br(- (lbl_br + 1))

        # Error stop guard - this should never execute
        spu.stop(0xE)

        # Storage space for saved registers
        # Align to a 16-byte boundary
        while (code.size() % 4) != 0:
            spu.nop(0)
```

DRAFT: May 23, 2007
lbl_regs = code.size()

# Create space for the saved registers
for i in range(128):
    # 16 bytes/register
    for j in range(4):
        spu.nop(0)

# Insert the store commands
for i in range(128):
    st_offset = ((lbl_regs - lbl_store) - i) + i * 4
    code[lbl_store + i] = spu.stqr(i, st_offset)

# Constants used to update the running program
code_size = len(code._prologue._code) * 4
self.xfer_size = code_size + (16 - (code_size) % 16);
self.code_lsa = (0x3FFFF - code_size) & 0xFFF80;
self.reg_lsa = self.code_lsa + lbl_regs * 4

self.lbl_op = lbl_op

return

Lines 4-9 generate the two instructions required to execute arbitrary user instructions. 
sync in line 5 forces the SPU to flush its instruction cache and reload the instruction stream
from the local store. At synthesis time, line 9 inserts a no-op into the instruction stream.
When SYN-ISPU is running, the no-op is replaced by the next instruction to execute and is
referenced by the label created in line 8.

Lines 12-14 and 41-43 synthesize the instructions to save the registers to the data cache.
The data cache itself is created in lines 35-38 and initially consists of a stream of nop
instructions, with 16 bytes (4 nops) reserved for each register. The nops inserted in line 14
are replaced with store instructions, one per register, created in line 43, once the offsets
into the data cache are known.

The final instructions in the synthetic program are created in lines 17-22. Line 18 inserts
the stop instruction that suspends execution after the user instruction has been executed
and the register contents saved. The next instruction, created in line 22, branches back
to the beginning of SYN-ISPU and repeats the process. The branch instruction is the first instruction executed after SPU execution is resumed by the ISPU execution manager.

When used from the command line, ISPU lets the user issue instructions created from the ISA module and access the contents of the registers in read-only tuples:

```python
>>> import corepy.arch.spu.isa as spu
>>> import ispu

>>> cli = ispu.ISPU()

>>> cli.start()

>>> cli.execute(spu.iohl(127, 0xB8CA))

>>> cli.execute(spu.iohl(126, 0x1234))

>>> cli.execute(spu.a(125, 126, 127))

>>> regs = cli.get_regs()

>>> print '%X' % regs[125][0]

---> 'CAFE'
```

In this example, the user executed three commands to populate register 125 with the hex value 0xCAFE. The graphical user interface built on top of ISPU works similarly and highlights registers changed by the previous instruction (Figure 1).

2. SPE Debugger

Debugging synthetic programs on the Cell BE SPU is a difficult process. As with other development tools, debuggers for the SPU are still in their infancy. Attaching gdb to a running SPU program is a non-trivial task and a number of additional steps must be taken to properly debug synthetic programs executing on an SPU. Debugging in this manner adversely affects productivity and limits the overall usefulness of the standard debugging tools for synthetic programming on the SPU.

The ISPU program introduced above provides clues as to how the debugging process can be improved by utilizing features of the synthetic programming environment and the Cell BE SPU. Dynamic access to the instruction stream at runtime makes it possible to inspect and modify synthetic programs at run time. The ability to start and stop a running SPU makes it possible to exert a fine level of control over the execution of a synthetic
Figure 2. SPU Debugger The SPU debugger modifies the InstructionStream at run time to execute one instruction at a time. The leftmost instruction listing shows the original synthetic program and the four extra instructions added to support debugging. The next listing shows the first instruction executing (green). The debug branch and stop instructions are blue. When a conditional branch is encountered (step (4)), both the next instruction and the target instruction are modified to branch to the debug section. The stop code tells the debugger which branch was taken.

Program. These two features were used to implement a very compact interactive debugger for SPU synthetic programs.

As with ISPU, the debugger allows the instruction stream to progress one instruction at a time, using stop instructions inserted into the synthetic program to halt execution. The debugger works by examining the current instruction in the InstructionStream and replacing the potential next instructions with branch instructions. In most cases, the next instruction is simply the next instruction in the sequence. But, branch instructions have two potential next instructions, the next sequential instruction and the branch target. In this case, both instructions are replaced with a branch instruction. The replacement branch targets jump to one of two stop instructions inserted at the end of the InstructionStream, depending on which branch was taken. The stop instructions return different values to the debugger, indicating if the source of the stop was the next sequential instruction or the branch target. Immediately following the stop instructions are branch instructions that jump back to the next instruction in the program. These are generated and inserted by the debugger at each step. When the restart signal is sent to the SPU, the program branches to the next instruction and the process is repeated.
The SPU debugger is implemented as a subclass of Processor and called DebugProcessor. From the user's perspective, DebugProcessor works just like Processor until the execute() method is called. execute() modifies the instruction stream to support debugging, executes the first instruction, and returns control to the user. Progress is controlled by the user using the nexti() method. Each call to nexti() advances execution by one step using the above procedure. Using this procedure, an entire SPU program can be executed stepwise from the Python interpreter.

Between instructions, the user can examine the contents of the registers using the get_regs() method on DebugProcessor. This method returns a 128-element list containing 4-word tuples that contain the current values in the registers.

To minimize the amount of instructions added to the original synthetic program, acquiring the register values is handled by an auxiliary synthetic program that is loaded into the local store when the values are requested. The register acquisition program uses a different approach than ISPU to transfer the contents of the registers to the PPU. Instead of copying the values to memory, it uses the SPU signal interface to send the values directly to the PPU. The program contains 128 signal instructions, each followed by a stop instruction. The payload on each signal is the contents of a register. The debugger waits for the signal, saves the value, and restarts the synthetic program. This process is repeated until all values are transferred. The transfer program is stored at address zero in the local store. Branch instructions are used to branch to and from the original synthetic program. The contents of local store used by the register transfer program are saved to main memory and restored following completion of the transfer.

Separating the register transfer program from the original program limits the amount of local storage needed to support stepwise debugging. The stop and branch instructions appended to the original InstructionStream occupy four instruction slots, or the same amount of space as one vector register.

The following listing shows a sample debugging session:

```python
1 # Create the synthetic program
2 code = InstructionStream()
```
3 spu.set_active_code(code)
4
5 spu.ai(127, 0, 1)
6 spu.ai(126, 0, 2)
7 spu.ai(125, 0, 3)
8 spu.brnz(125, 2)
9 spu.ai(124, 0, 4)
10 spu.ai(123, 0, 5)
11
12 # Execute it with a debug processor
13 proc = DebugProcessor()
14 r = proc.execute(code)
15
16 while r is not None:
17    regs = proc.get_regs()
18    print [reg[0] for reg in regs[123:]]
19    r = proc.nexti()
20
21 # Output
22 --> [1,0,0,0,0]
23 --> [1,2,0,0,0]
24 --> [1,2,3,0,0]
25 --> [1,2,3,0,5]

Lines 1-10 create a simple synthetic program that copies values to registers, conditionally skipping one instruction. The loop in lines 16-19 executes the instructions one at a time, printing the first element in each affected register. Lines 22-25 show the output.

InstructionStream can store additional information useful for debugging. As each instruction is added, it can save the current stack frame, making it possible to tie an instruction back to the code that generated it.

The approach to debugging used by DebugProcessor is useful for general, stepwise debugging of SPU programs. While not cycle accurate, it makes it possible to easily debug a synthetic program and spot logic errors or bad values stored in registers. Using the DMA commands, the user can also examine the local store between instructions.
3. Conclusion

This chapter presented two applications that lay the groundwork for building tools that allow developers to interact directly with running synthetic programs. ISPU provides a convenient interface for learning the SPU instruction set and experimenting with instruction-level algorithms. The SPU debugger goes a step further by enabling interactive control over the execution of synthetic programs on the SPU. Both applications demonstrate how, with minimal effort, synthetic programming can be used to create new types of applications and replicate the functionality of traditional development tools.
Conclusion

This thesis presented synthetic programming and the synthetic programming environment. Synthetic programming mixes direct access to the hardware environment with the dynamic run-time provided by Python to give developers a high-productivity environment for developing high-performance applications. Using the base libraries of the synthetic programming environment to develop components and applications, synthetic programming was demonstrated as both feasible and practical for application development on existing and emerging hardware platforms.

Synthetic programming is an alternative to traditional approaches for developing high-performance applications. It has the potential to replace not only assembly languages for
low-level programming, but also compiled languages for generating high-performance libraries. The synthetic components presented here demonstrate straightforward techniques for using the synthetic programming environment to perform targeted optimizations on domain-specific representations of data and algorithms. By taking advantage of run-time information, developers can create and utilize libraries that automatically optimize themselves based on the characteristics of the data and processing environment. The flexibility available to developers using the synthetic programming environment makes it possible to develop additional components that generate very high-performance code in ways not possible using the compiled languages.

In addition to providing support for instruction-level optimizations, the organization of the synthetic programming environment encourages developers to architect applications for multi-core environments. As SYN-BLAST demonstrates, when used to control the execution of synthetic programs, the high-level language features of Python can lead to concise designs for multi-core applications.

A key premise behind synthetic programming is that by making low-level programming more accessible, developers will explore more approaches to generating high-performance code. Traditional compiled and assembly languages carry too much historical baggage and, more often than not, are rejected in favor of high-productivity languages, regardless of the impact on performance. By presenting the processor from a fresh perspective, synthetic programming encourages developers to experiment with low-level and multi-core programming techniques. While studying the effect of synthetic programming on a larger developer community was beyond the scope of this thesis, there is recent historical precedent to suggest that when given ready access to the lowest levels of a system, the development community will enthusiastically search the problem space for novel applications and solutions.

In the early 1990s, two low-level specifications were published that led to the development on innumerable programming languages, development tools, and programming paradigms: HTTP and HTML. These specifications formed the machine language of the Web. In the early days, developing applications using these protocols was an ad-hoc process performed primarily by computer scientists using compiled languages. Without pre-defined
standards and best practices, developers were able to explore many different strategies for creating applications. Due to the number of techniques developed at the same time, no one paradigm emerged dominant and instead a number of paradigms took hold in different application domains. Fifteen years later, new languages and techniques are still evolving and developers are still working at all layers in the development stack to create new solutions.

In stark contrast, machine-level programming has stagnated and is becoming a lost art in all but a few small areas of computer science. Unfortunately, this is occurring at a time when commodity processors are undergoing their most radical changes since the introduction of the microprocessor. Compiler developers are struggling to develop the technologies to make it possible to utilize multi-core and SIMD processors. Without a larger community looking at the problem, high-performance computing risks becoming a lost art.

By removing the legacy tools from the high-performance toolchain, the approach taken by synthetic programming democratizes high-performance computing in the same way HTTP and HTML made distributed computing practical for a large class of developers. Of course, synthetic programming will necessarily have to achieve this with a smaller community of developers. But, with processor architects making bold design decisions with the current generation of multi-core processors, there is an opportunity to start to develop new ways of looking at software development that help developers create applications for these processors.

As processors and programming paradigms go through their next evolutionary stage, new approaches to programming are bound to emerge. Synthetic programming demonstrates one approach that is both feasible and practical across a range of computer systems and applications. The synthetic programming environment provides tools for both instruction level programming and multi-core program management from high-level scripting languages, making it possible for developers to rapidly develop high-performance applications and develop entirely new approaches to optimizing applications. Synthetic programming programming will ultimately be successful not if it is the final solution, but if it encourages developers to look at these problems from new perspectives and develop truly revolutionary programming paradigms.
Bibliography


