

**Efficient Programming Techniques for the SACLIB Computer Algebra Library**

A Thesis

Submitted to the Faculty

of

Drexel University

by

David G. Richardson

in partial fulfillment of the

requirements for the degree

of

PhD in Computer Science

May 2009

© Copyright May 2009  
David G. Richardson. All Rights Reserved.

## Dedications

*To Jen. You make hard things worth doing.*

## Acknowledgements

No one ever finishes a PhD by themselves. I would like to thank all of the people that have helped me along the way. First, I would like to thank my wife, Jennifer Richardson. You helped me through all the times I thought about giving up. I never would have finished without you.

I would like to thank my adviser, Werner Krandick, for his help and guidance. You helped me to realize that despite all the existing scientific knowledge, there is no reason for me to expect I cannot add something. You helped nurture my research through its formative stages, and helped me build the confidence to tackle the unknown by myself.

I would like to thank David Breen, Jeremy Johnson, Brian Mitchell, and David Musser for serving on my committee. I would like thank Jeremy Johnson for introducing me to performance counters and making me take a computer architecture course. You introduced me to a part of computing I might have otherwise missed. I would like to thank Brian Mitchell for introducing me to the field of software engineering. Without this, it would have been too easy to miss the connection between design and management. I would like to thank David Musser for his contributions to generic programming. You created a field I have thoroughly enjoyed studying. Your way of looking at problems has allowed me to solve many software design problems with the confidence that I am always starting with the best techniques the last 40 years has to offer.

I would like to thank Jeff Abrahamson, John Granieri, Yelena Kushleyeva, Walt Mankowski, Adam O'Donnell, and David Pettey for their friendship, encouragement, and moral support. I would like to thank Cameron Abrams for teaching me that once I have data, its up to me to figure out how to analyze it. Not every problem has a standard analysis written up in a text book. I would like to thank Mike Kain for teaching me the fundamentals of networking. You've given me the foundation to keep teaching myself.

## Table of Contents

List of Figures .....	iv
List of Tables .....	vii
Abstract .....	viii
1. Introduction .....	1
2. Literature Review .....	5
2.1 Generic Programming .....	5
2.1.1 Introduction to Generic Programming.....	5
2.1.2 Adoption of Generic Programming .....	6
2.1.3 Open Problems in Generic Programming .....	7
2.1.4 Template Metaprogramming .....	8
2.2 Memory Safety.....	10
2.2.1 Garbage Collection .....	10
2.2.2 Static Analysis.....	11
2.3 High-Performance in Computer Algebra and Numerical Methods.....	13
3. Iterators for Compiler Enforced Memory Safety .....	16
3.1 Introduction.....	16
3.2 The Recursively Fixed Iterator and Structure Protecting Iterator Concepts .....	18
3.2.1 Definitions From the STL .....	18
3.2.2 Conditions for memory leaks and double deletes .....	19
3.2.3 Recursively Fixed Iterator Concept .....	23
3.2.4 Structure Protecting Iterator Concept.....	28
3.3 Implementation .....	31
3.3.1 Operator Overloading .....	31
3.3.2 Protecting Memory Structure .....	33
3.3.3 Detecting Memory Structure .....	34
3.3.4 The <code>rec_fixed_itr</code> and <code>struct_protect_itr</code> .....	47
3.3.5 Preventing Memory Leaks .....	50
3.4 Performance Testing.....	50
3.4.1 Writing Code for the Optimizer .....	54
3.4.2 Compilation Overhead .....	55

3.5	Comparison to existing Work .....	55
3.5.1	STL Iterator concepts .....	55
3.5.2	Multi-Dimensional Containers and Smart Pointers .....	56
3.5.3	The C++0x Standard .....	57
3.6	Conclusion .....	58
4.	Automatic Test Bed Generation .....	60
4.1	Introduction .....	60
4.2	Function Level Tracing .....	62
4.3	Function call identification .....	64
4.4	Recording input/output .....	66
4.5	Recording global state .....	75
4.6	Aspect based tracing .....	77
4.7	First order tracing .....	82
4.8	Test case filtering and execution .....	82
4.9	Test harness generation .....	83
4.10	Experimental Results .....	84
5.	Automatic Performance Tuning .....	86
5.1	Introduction .....	86
5.2	Auto-Tuning For the Taylor shift and de Casteljau's Algorithm .....	88
5.2.1	Tile Definitions .....	88
5.2.2	Tile Scheduling .....	92
5.2.3	Code Generation .....	94
5.2.4	Compiler Optimizations .....	96
5.3	Experimental Results and Discussion .....	97
5.3.1	Evaluated Descartes Method Implementations .....	101
5.3.2	Evaluated Processor Architectures .....	102
5.3.3	Hardware configuration and timing .....	103
5.3.4	Compilation protocol .....	104
5.3.5	Input Polynomials .....	104
5.3.6	Taylor shift Benchmarks .....	105
5.3.7	Root Isolation Benchmarks .....	107
5.4	Conclusions .....	109

6. Conclusions ..... 110  
Bibliography ..... 113

## List of Figures

3.1	Multi-dimensional memory. $P$ is a C++ variable with type <code>int**</code> . The gray memory cells contain pointers and are the structure of $P$ (see Definition 3.2.19). The white memory cells contain integers and are the contents of $P$ (see Definition 3.2.20). . . . .	22
3.2	An example of the dangers to memory safety when swapping structure elements. . . . .	27
3.3	An example of the dangers to memory safety caused by incrementing structure elements. . . . .	27
3.4	The SACLIB 3.0 <code>simple_ptr</code> implementation. . . . .	31
3.5	The SACLIB 3.0 implementation of <code>AADV</code> . . . . .	32
3.6	Implementation of the <code>Dim</code> metafunction. See Figure 3.7 for the implementation of <code>Dim_non_array</code> . . . . .	35
3.7	Implementation of the <code>Dim_non_array</code> metafunction used to implement <code>Dim</code> (Figure 3.6). . . . .	36
3.8	Improved implementation of the <code>Dim</code> metafunction using the g++ <code>typeof</code> extension. . . . .	37
3.9	Implementation of the <code>is_dereferenceable</code> metafunction. . . . .	38
3.10	Implementation of a short-circuit compile-time boolean <code>AND</code> metafunction. . . . .	39
3.11	Example usage of the <code>AND</code> metafunction. . . . .	40
3.12	Outline of the <code>struct_protect_itr</code> implementation techniques. . . . .	48
3.13	The SACLIB 2.1 version of <code>MMPDDF</code> contains a memory leak that results from incorrect manipulation of the memory structure of a matrix. These kinds of errors are not possible with the new SACLIB 3.0 memory management system. See Figure 3.14 for the SACLIB 3.0 version of this code. . . . .	51
3.14	The SACLIB 3.0 implementation of <code>MMPDDF</code> . The use of the SACLIB 3.0 memory management removes the leak shown in the SACLIB 2.1 implementation in Figure 3.13. . . . .	52
3.15	Probability of observing a given execution time (cycles) for writing to every element of a 500 element array via a C++ native pointer. . . . .	53
3.16	Example programs for reading memory through a C++ pointer and a <code>rec_fixed_itr</code> . The <code>use_memory</code> function writes the value to <code>/dev/null</code> . The <code>icc</code> compiler generates identical assembly code for the iterators to write to <code>/dev/null</code> . . . . .	54
3.17	Concept refinement hierarchy for the STL iterators, the Recursively Fixed Iterator, and the Structure Protecting Iterator. Boxes denote concepts. An arrow from box A to box B means that concept B is a refinement of concept A. Dashed boxes are concepts we present in Section 3.2. . . . .	56
4.1	Manually inserted tracing code. The inputs (lines 18-20) and the outputs (lines 45-47) are traced using functions obtained from the header file <code>trace_utils.h</code> (line 14). . . . .	63

4.2	Test cases produced by invoking the function in Figure 4.1 as <code>LIST2(LIST2(12,15),11)</code> . Each invocation of <code>LIST2</code> produces a record that starts with the signature of the traced function (Lines 1,8) and ends with <code>%%</code> (Lines 7,14). Lines 2-6, 9-13 trace inputs and outputs. ....	64
4.3	The implementation of <code>trace_signature</code> and <code>trace_return</code> must record the call stack in order to allow tracing of recursive functions.....	65
4.4	An overload of the <code>trace_input</code> function must be provided for each type to be traced. The first overload is for streamable types. The second overload is for <code>SACLIB</code> objects. ...	66
4.5	The <code>SACLIB</code> routines <code>FRAPGET</code> and <code>FRAPFREE</code> use their own bookkeeping data for managing heap allocated memory. ....	67
4.6	Implementation of <code>operator new</code> and <code>operator delete</code> to track the information about heap allocated memory required to allow the implementation of the <code>trace_input</code> overload in Figure 4.7. ....	68
4.7	Implementation of a <code>trace_input</code> overload that uses the information recorded from <code>operator new</code> and <code>operator delete</code> (Figure 4.6) to serialize heap allocated memory....	69
4.8	The <code>rec_fixed_itr&lt;BDigit*&gt;</code> argument to the <code>SACLIB</code> routine <code>AII</code> requires the serialization of a pointer to an array of <code>BDigits</code> . ....	70
4.9	A test case serialized from calling the <code>AII</code> routine in Figure 4.8 as “ <code>BDigit b[] = {1,1,0}; AII(b);</code> ” .....	71
4.10	An aspect to weave tracing code around execution joinpoints. ....	77
4.11	A <code>trace_input</code> overload to trace all of the arguments in an <code>AspectC++ JoinPoint</code> . ....	78
4.12	The <code>trace_input</code> overloads from Figure 4.4 are augmented to prevent stack overflow during tracing. ....	81
4.13	A test harness to execute test cases for the <code>SACLIB</code> routine <code>LIST2</code> . ....	83
4.14	Routines use to exercise the automatic test generation of <code>SACLIB 3.0</code> . ....	85
5.1	(a) The pattern of integer additions in Pascal’s triangle, $a_{i,j} = a_{i,j-1} + a_{i-1,j}$ , can be used to perform Taylor shift by 1. (b) In de Casteljou’s algorithm all dependencies are reversed, the intermediate results are computed according to the recursion $b_{j,i} = b_{j-1,i} + b_{j-1,i+1}$ . .	87
5.2	Register tiling can be applied to (a) Taylor shift and (b) de Casteljou’s algorithm. ....	89
5.3	Index numbering conventions of square tile. The nodes represent a computation. The edges represent data dependencies. Each node is labeled with its index. Triangles and pentagons are numbered according to the same indexing convention, but have fewer nodes.	90
5.4	For a given polynomial degree and square tile size, there is only a single size of corresponding pentagon and triangle that can be used for tiling. ....	91

5.5	Tile shapes are defined so only the following transitions may occur. Arrows represent a data dependency between tiles.....	92
5.6	Schematic processing schedule for a $S_{x,y,4}$ square tile. Tiles are scheduled in using three kinds of passes: 1) a single register load pass, 2) one or more computation passes, and 3) a register store pass. Arrows represent an input. Circles represent a computation involving all of the inputs to the circle. The circles in each pass are numbered in the order they are scheduled by the code generator. The $S_{x,y,4}$ tile is enclosed in the black square. The code to process this tile for the Taylor shift is in Figure 5.7 and the code to process this tile for de Casteljaou's algorithm is in Figure 5.8. ....	94
5.7	The code to process the schedule for a $S_{x,y,4}$ Taylor shift square. See Figure 5.6 for a schematic representation of the $S_{x,y,4}$ schedule. This code was produced by our code generator (Section 5.2.3). It was hand reformatted and commented to improve presentation. See Figure 5.8 for the corresponding de Casteljaou's tile. ....	98
5.8	The code to process the schedule for a $S_{x,y,4}$ de Casteljaou's square tile. See Figure 5.6 for a schematic representation of the $S_{x,y,4}$ schedule. This code was produced by our code generator (Section 5.2.3). It was hand reformatted and commented to improve presentation. See Figure 5.8 for the corresponding Taylor shift tile. ....	99
5.9	The x86 assembly produced by gcc for the C++ code to process a $S_{x,y,4}$ square Taylor shift tile (Figure 5.7). The assembly is shown for (a) the register load pass, (b) the first computation pass, (c) the register store pass. ....	100
5.10	Speedup obtained by the tiled version of the Taylor shift by 1 algorithm on the Pentium EE, Opteron, UltraSPARC III, and Pentium 4 for different register tile sizes. Speedup is calculated with respect to a straightforward GMP-based implementation of Taylor shift by 1. The speedup from the worst to best tile size is more than a factor of 2. ....	105
5.11	Classical Taylor shift with register tiling is faster than the fastest known asymptotically fast Taylor shift for a wide range of inputs. These speedups are obtained by using the optimal tiles size found by searching with our code generator. These speedups are obtained by using the optimal tiles size found by searching with our code generator. ....	106
5.12	Speedup with respect to the monomial SACLIB implementation for random polynomials on four architectures. These speedups are obtained by using the optimal tiles size found by searching with our code generator. ....	108

## List of Tables

3.1	Minimum and mode computing times (cycle counts) on a Pentium 4 for a native C++ pointer and a <code>rec_fixed_iterator</code> for construction, writing to a 500 element array, and reading from a 500 element array. ....	53
5.1	Lines of code generated for Taylor shift tiles of various sizes. The analysis shows that for a square of edge length $e$ , the number of generated lines of code is $O(e^3)$ . The above table only presents values of $e$ that were used for tuning. This dynamic range of $e$ is not large enough to clearly show the $O(e^3)$ scaling. This is expected, as the lower order terms have large constants. ....	96

**Abstract**

Efficient Programming Techniques for the SACLIB Computer Algebra Library

David G. Richardson

Advisor: Werner Krandick, PhD

This dissertation describes three contributions to the SACLIB computer algebra library. Using generic programming we specify the Recursively Fixed Iterator and the Structure Protecting Iterator concepts. We prove that models of these concepts cannot leak or double delete resources. Through template metaprogramming, we implement models of these concepts that provide this memory safety at compile-time and without any run-time overhead. Using aspect oriented programming, we allow unit and regression tests to be automatically obtained from an executing SACLIB program. This program instrumentation is transparent both to SACLIB maintainers and clients. This mechanism allows the automated creation of regression test suites during the execution of normal SACLIB programs. Using code generation and auto-tuning, we allow high performance implementations of the Taylor shift, de Casteljaou's algorithm, and the Descartes Method for polynomial root isolation to be automatically tuned for different architectures. This allows a speedup of 10-20 over existing methods to be obtained on modern CPU architectures. The key to this speedup is the creation of a tiling scheme that allows the irregular structure of the computation to be tiled using a small number of tiles that are easy for a code generator to produce code for.

The implementation of the Recursively Fixed Iterator and the Structure Protecting Iterator provide memory safety by using C++ template metaprograms that selectively disable certain classes of side effects in programs using the iterators. Other safety properties can be achieved using similar disabling of side effects. We believe that the most important continuation of the work presented here is the development of programming language techniques to allow programmers to control where their code should sit along the spectrum of the absence of side effects in pure functional languages and the unrestricted side effects of imperative languages.



## 1. Introduction

The field of Computer Algebra provides the ability for computers to represent and symbolically manipulate algebraic objects. Using Computer Algebra Systems such as Maple [135] and Mathematica [209], computers can be used to solve problems such as obtaining the exact solution to a system of equations, taking symbolic integrals and derivatives of functions, computing the greatest common divisor of two elements in a unique factorization domain, isolating polynomial roots, and simplifying algebraic expressions. The ability to solve these types of problems plays a central role in the advancement of science, engineering, computational finance, and operations research. Because of this, computer algebra is extensively used in these fields for its ability to provide exact and closed form solutions to critical problems.

One of the major factors limiting the wider use of computer algebra is performance. Although computer algebra can provide exact answers, this mathematical certainty comes with a price: obtaining an exact solution typically requires more execution time and memory than corresponding numerical solutions of the same problem. For exact computation, each additional arithmetic operation requires more memory to be used as the length of the number increases. As this intermediate expression swell occurs, both the storage and processing needs of the computation increase. This does not occur for numerical methods, where floating point numbers never grow and the cost to process them is constant regardless of the number of arithmetic operations that have been performed. Because of this performance penalty numerical methods are often used when exact methods would provide a better answer.

There are two strategies to improve the performance of computer algebra programs. The first is to improve the time or space complexity of the computer algebra algorithm. The second is to improve the implementation of the algorithm. In this dissertation, we are only concerned with obtaining speedups through the improvement of the implementation of existing algorithms.

There are two fundamental areas that must be addressed in any computer algebra library implementation. The first is memory management. Because of the large memory requirements imposed by intermediate expression swell, it is critical that memory be well managed. The second is efficient utilization of the hardware features of modern processors. The pursuit of improved system performance via better memory management and processor utilization almost always results in more complex and technically demanding implementations. This complexity increases the development,

verification, and maintenance cost of computer algebra systems. Because of this, any implementation techniques that improve the efficiency of computer algebra systems must provide a way to effectively verify the efficiency and correctness of the improved implementation.

This dissertation describes three advances in the the implementation of efficient computer algebra systems. In Chapter 3 we present iterator concepts that guarantee the absence of memory leaks and double deletes for memory used via the iterators. Using template metaprogramming, we allow a C++ compiler to enforce this memory safety at compile-time. This requires no extension of the C++ toolchain. We demonstrated that with careful implementation, these benefits may be enjoyed with no run-time overhead. These iterator concepts enforce memory safety by selectively disallowing dangerous side effects in code using the iterators. We call concepts that restrict side effects *safety concepts*. Although we only used safety concepts to provide memory safety, formulating safety concepts as restrictions on side effects allows them to be used to specify any kind of safety property. In previous work, we [157, 158] introduced an earlier version of the Recursively Fixed Iterator. The main contribution of the original formulation was using the C++ type system to designate that certain functions were not responsible for memory management of their arguments. The original formulation did not have the theoretical foundations for formal specification of the iterator, use with the STL, or application to other types of safety properties. In this dissertation, we provide the theoretical framework to overcome all of the original limitations.

C++ templates are a cumbersome mechanism for implementing models of safety concepts, and are not capable of enforcing arbitrary restrictions of side effects. Making safety concepts easy to specify and implement will require the development of programming language techniques to allow programmers to control where their code should sit along the spectrum of the absence of side effects in pure functional languages and the unrestricted side effects of imperative languages. We believe the most important continuation of the work we have presented is the research required to allow the scoped control of side effects to become a first class element of mainstream programming languages.

In Chapter 4 we present aspect-based tracing techniques to automatically collect unit tests from a running program. This is the first use of aspects for this kind of automated testing. This allows the execution of any SACLIB main program to be used to generate a regression test suite. When faults are introduced into a program, the regression test suite will identify the location of the faults. This program instrumentation is transparent both to SACLIB maintainers and clients. It is also robust in the face of changes to program execution flow.

These program instrumentation techniques allow legacy software without a test suite to be more

safely modified. This can lower the maintenance cost of the legacy system and extend its useful life. The mechanisms used to filter test cases are general and can be used for the arbitrary capture of function inputs and outputs. This provides a mechanism for running a computation and collecting only the inputs that are useful for a specific kind of benchmarking or testing.

In Chapter 5 we use auto-tuning techniques to improve the performance of the Taylor shift, de Casteljaou’s algorithm, and the Descartes root isolation method. In the original application of tiling to these three algorithms, it was clear from the complexity of the original implementation that code generation would be required to reap the full benefits of tiling [103, 161]. We have developed a formalism for defining tiles in a way that is amenable to code generation. Using code generation algorithms based on that formalism, we use automatic tuning to obtain a speedup of a factor roughly of 10-20 over existing implementations, including those that have been hand-tuned in assembly language. These speedups were obtained by using code generation to search for optimal tile sizes on four different hardware architectures. The optimal tile size differed among the four architectures. The search process required the generation of 40,000 lines of fairly complicated C++ code. Without the code generation algorithms, it would have been infeasible to perform the search for optimal tile size. For some architectures, the optimal tile size provided a speedup of a factor of 2 over the lowest performing tile size.

The code generation requires the ability to generate irregular tile shapes. Additionally, the code generation algorithms must deal with the fact that the shape and size of the different tiles influences the shape and size of the other generated tiles. We remove the need to iteratively account for this influence by defining the tiles in a way that allows for a single pass code generation algorithm.

The experimental performance measurements obtained with our code generator demonstrate that computer algebra algorithms can profitably be optimized using code generation in a high level language. This is in contrast to the traditional method of tuning computer algebra algorithms with hand written assembly language. This is a demonstration both that high-level optimizations and optimization techniques normally reserved for numeric computation are useful in computer algebra. We also compare the effectiveness of our generated code to asymptotically faster algorithms. The factor of 2 speedup provided by the code generation moves the cross-over point between the classical and asymptotically fast Taylor shift from inputs of degree 2,000 to inputs of degree 10,000. This demonstrates that any algorithm engineering efforts in computer algebra must take into account fully optimized classical implementations.

These techniques have all been applied to the SACLIB [36, 157, 158] computer algebra library.

When used in combination, these techniques address all of the factors needed to implement computer algebra libraries with improved performance. Auto-tuning provides improved run-time performance. By providing a correct foundation to build on, memory safety and automated test generation allow for aggressive optimizations to be safely applied. In Chapter 6, we discuss how our contributions apply outside of computer algebra.

## 2. Literature Review

### 2.1 Generic Programming

#### 2.1.1 Introduction to Generic Programming

The goal of generic programming is to produce software components that can be used in the largest possible range of applications. Additionally, generic programming aims to provide this flexibility without any sacrifice in run-time efficiency. Implementing software in this manner requires that the constraints a component places on its clients are required for the component to function.

In generic programming, these essential constraints are formulated using constraints on types which are called concepts [12, 139, 142]. The constraints may deal with the syntax and semantics of a type or the time/space complexity of operations supported by the type. The following definitions from the C++ Standard Template Library (STL) [12, 94, 191] are useful in discussing concepts:

**Definition 2.1.1. (from STL)** A C++ type  $X$  is a *model* of a concept when it meets all the constraints of the concept. Note that a concept specifies the minimal set of constraints a type must have to be a model of the concept. Therefore even when a type has more features than those described by a concept it is still considered a model of the concept.

**Definition 2.1.2. (from STL)** A concept  $C2$  is a *refinement* of the concept  $C1$  if and only if  $C2$  specifies all the constraints of  $C1$  plus additional constraints not specified by  $C1$ . When  $C2$  is a refinement of  $C1$  we say that  $C2$  is *stronger* concept than  $C1$  and  $C1$  is *weaker* concept than  $C2$ .

Using concepts to specify components is an outgrowth of research that has been done on abstract data types (ADT) and component specification [30, 80, 81, 82, 106, 124, 139, 202].

Once concepts have been formulated, the specifications of other components such as functions can then be made in terms of concepts. Concepts aid both in reasoning about software and in documenting it. When a component is specified in terms of the weakest required concept, this automatically results in the component being usable by the largest number of clients. Because of this, correctly defining concepts is a critical aspect of the successful application of generic programming. In Section 2.1.2 will detail how this method of component specification has allowed the STL to provide a large number of loosely-coupled and interchangeable components while maintaining run-time efficiency.

An important enabling technology for generic programming is component assembly technology. Certain assembly techniques, such as inheritance and virtual functions, can result in components being assembled with a run-time overhead that is only necessitated by the assembly technology. Limitations in assembly technologies can also result in interface constraints on the components. These interface constraints may force the components to have constraints that are not essential for the problem the component solves.

Because components are specified in terms of concepts, it is in principle quite easy to determine if the component is being used correctly. However, the ability to have this checked automatically relies on the abilities of the assembly technologies and the existence of automatic theorem provers.

### **2.1.2 Adoption of Generic Programming**

Generic programming has been successfully employed in the implementation of libraries in such diverse problem domains as basic data structures and algorithms [94, 191, 141, 12, 104, 134, 152, 143, 140], acyclic automata [128], graph algorithms [78, 174, 176], numerical computing [17, 18, 109, 107, 201, 200], parallel algorithms [13, 22, 78, 101, 118, 214, 110, 107], bioinformatics [118], computational chemistry [215, 147], memory management [16, 6, 146, 158, 38], pollution transport modeling [47], computer algebra [53, 105, 162, 49, 46, 168, 154, 139, 142, 158], computational geometry [56], and linear algebra [129, 49, 105, 125, 180, 46, 154, 44, 43].

### **A Generic Programming Case Study: The C++ Standard Template Library**

The C++ Standard Template Library [94, 191, 141, 12, 104, 134] is one of the most well known applications of generic programming. The most important kinds of components in the STL are

1. Containers - data structures that store data.
2. Iterators - data structures that allow access to data in containers.
3. Algorithms - algorithm implementations that operate on iterator ranges.

The STL has hierarchies of concepts to categorize the different kinds of containers and iterators it provides. Algorithm inputs and outputs are then specified using these concepts. C++ templates and inheritance are then used to assemble the containers, iterators, and algorithms into a functioning program.

This organization of components produce a tremendous benefit: data structures are decoupled from algorithms. The algorithms can be used on any container that provides appropriate iterators. Because the iterators provide time complexity guarantees on operations, this also allows a given algorithm component to chose the most suitable implementation to be used with a given iterator.

As an example, let us consider sorting with the STL

```

1: vector<int> v; //container with random access
2: deque<int> d; //container with random access
3:
4: sort(v.begin(), v.end());
5: sort(d.begin(), d.end());

```

On lines 1-2 two instances of random access containers are declared. On lines 4-5, the containers are sorted. The calls to `begin()` and `end()` produce a range of random access iterators that is passed to `sort`. Because both `vector` and `deque` provide random access iterators, the same implementation of `sort` is able to sort elements held in both kinds of containers.

### 2.1.3 Open Problems in Generic Programming

#### Fundamentals

Metrics for determining the quality of concepts are not firmly established. As recently as 2000 [45] there was an effort to provide a solid basis for such fundamental notions as assignment. Generally, such work on developing high quality concepts is done in the context of applying generic programming to a specific problem domain.

There has also been work on modifying existing software engineering metrics to try and assess the quality of software implementations that make heavy use of generic programming techniques [57].

#### C++ Library Support

One limitation in the C++ template mechanism is a lack of direct support for concepts in the language [77] or even the ability to easily validate template parameters with reasonable error messages [162]. Libraries have been developed to largely solve these problems [6, 172, 126].

Difficulties in the syntax of template metaprogramming has resulted in the creation of libraries to ease metaprogramming [6, 2, 43, 96].

After dissatisfaction with the binders in the STL [181], libraries to support functional programming in C++ were developed [95, 132, 133]. Type safe covariance support has also been made available as a library [195].

These libraries underscore two points: 1) additional tool support for generic programming is widely desired, and 2) the emphasis placed on making C++ a multi-paradigm language [189] has allowed it to support generic programming through library extensions.

## Generic Programming in other Languages

Due to its combination of support for both object-oriented and generic programming [189, 12], a significant amount of generic programming research has been conducted in C++. Prior to the STL a library of generic algorithms had been developed in ADA [143]. Other languages such as C# [65, 67, 97], Eiffel [65], G [175], Haskell [65], Java [48, 67, 97], and ML [66] have recently begun incorporating support for generic programming. There have been several papers [67, 65, 97] comparing the merits of the various approaches to language support. The C++ standards committee is also working on adding more language support for generic programming [192, 79, 173].

## Software Components

Improvements in the specification and use of software components is thought to be one of the technologies that has the chance to provide great benefit to software engineering [167, 63]. There have been efforts aimed at using concepts to organize catalogs of design components [98], assembling components while paying attention to the final systems performance [163, 182], improving the interface between components and formal analysis tools for components [188], and improving component interoperability while preserving efficiency [204]. We have used concepts to specify memory safety of software components [158, 156].

### 2.1.4 Template Metaprogramming

The application of generic programming in C++ relies heavily on the use of templates and template metaprogramming. Originally, the C++ template instantiation mechanism was intended to provide a type-safe alternative to macros for the generation of families of functions that differ only in their types. But then Erwin Unruh [199] discovered that the C++ template instantiation mechanism offered both branching and recursion, allowing arbitrary computation at compile-time. The C++ template instantiation mechanism offers a Turing-complete functional language that is

evaluated at compile-time. The evaluation mechanism supports the computation of either variable type or integral constants.

For readers not familiar with template metaprogramming, this section will provide a brief introduction. For a more comprehensive treatment, readers are referred to any of the available reference books [2, 6, 199, 43].

As an example of a type computation consider the following metaprogram to compute a pointer to a given type:

```

1: template <typename T>
2: struct pointerTo;
3:
4: template <typename T>
5: struct pointerTo{
6:     typedef T* type;
7: };

```

Lines 1-2 are a template class declaration. For a template metaprogram, this serves the same purpose a prototype serves for a function. Lines 4-6 are a template class definition. For a template metafunction this serves the same purpose as a function definition does for a function. Line 6 uses a typedef to provide the result of the type computation as the nested typedef type. This is a standard convention used in template metaprograms [2, 24]. The template parameters of a template metafunction serve as its arguments.

The `pointerTo` metaprogram would be used as follows:

```

pointerTo<int>::type p; //same as int* p;

```

In this example, it is clearly not advantageous to use the `pointerTo` metafunction when `int*` could just as easily have been written. However, when used inside another template metafunction, the argument to `pointerTo` may be another template parameter.

As an example of computing an integral constant, consider the computation of whether a type is a pointer:

```

1: template <typename T>
2: struct isPointer;
3:

```

```

4: template <typename T>
5: struct isPointer{
6:     enum {value=false};
7: };
8:
9: template <typename T>
10: struct isPointer<T*>{
11:     enum {value=true};
12: };

```

The `isPointer` metaprogram is used as follows

```

bool b1 = isPointer<int*>::value; //b1 == true
bool b2 = isPointer<int>::value; //b2 == false

```

## 2.2 Memory Safety

A program is memory safe if all of its memory management and access are performed correctly. The ability to provide programming languages and tools to allow programs to be automatically memory safe has been a topic of much research. With respect to memory management, the two main research results have been garbage collection and static verification via stronger typing.

### 2.2.1 Garbage Collection

Garbage collection has been a highly pursued strategy in helping programmers automatically achieve correct memory management. An ISI Web of Science [165] search for "garbage collection" from 1990-2005 returns 512 articles. The Open Directory Project [150] catalogs 90 programming languages as being garbage collected languages. Representative examples of such garbage collected languages are Java [70], Perl [203], Python [159], Ruby [130], and Lisp [72, 11].

These languages generally provide bounds checking in an attempt to prevent invalid memory access (for example, Java) or provide transparent dynamic memory management that ensures all memory access is to allocated memory (for example, Perl).

While these techniques can allow for tremendous gains in programmer productivity compared to languages with manual memory management, the price for run-time efficiency can be quite high. Necula [144] has conducted a series of benchmarks where C programs were converted to use garbage

collection and bound checking algorithms. For some work loads, the overhead introduced was almost a factor of 10 degradation of run-time performance. The vast majority of the slow down was due to the garbage collector. When using 5 times more memory than explicit memory management, current garbage collectors are currently capable of achieving comparable performance to explicit memory management. However, performance can be degraded by up to 70% when restricted to twice the memory needed by explicit memory allocation [85]. Compacting garbage collection is also capable of causing performance degradation when used on current multi-core processors [196].

In the domain of performance critical and real-time applications, the variability of garbage collections [23, 86] and its potentially high run-time overhead [144, 85, 196] prevent it from being a viable technique. It is not clear how long the performance gap between garbage collection and explicit memory management will persist. Current work on integrating application level garbage collection and the virtual memory component of the operating system shows promise for removing this overhead [86]. Until this gap is closed, garbage collection is not suitable for applications with demanding performance constraints.

### 2.2.2 Static Analysis

The CCured [145, 39, 84, 144] system consists of a set of language extensions to annotate pointer usage of C programs and a CCured to C source to source translator. The goal of CCured is to use stronger typing to statically verify the safety of memory operations. When the memory safety of an operation cannot be verified statically, CCured inserts code to perform run-time verification of memory safety.

Correct memory management is obtained by altering `malloc` to return memory that is managed by a garbage collector. Calls to `free` are set to do nothing.

The foundation of the CCured type system are the following pointer qualifiers: `SAFE`, `SEQ`, `WILD`. Note: CCured has additional refinements of these type qualifiers that will not be discussed here. The type qualifiers have the following meanings:

- `SAFE` - The pointer is only ever used for memory dereference. There is no pointer arithmetic or type casting performed on the pointer.
- `SEQ` - The pointer is involved in pointer arithmetic. Type casts are still not performed on the pointer.

- WILD - Due to type casts, the static type of the pointer may not provide an accurate description of the proper semantics to be ascribed to the memory it points to.

SAFE pointers must be checked at run-time to ensure they are not null before dereferences. SEQ pointers must have a null check and a bounds check performed at run-time. WILD pointers must have a null check, bounds check, and a semantic check of the memory they refer to at run-time.

CCured must store additional data about memory to allow such checks to be performed. The library uses two approaches:

1. Pointers only used in CCured applications have their metadata stored with the pointer. This results in CCured pointers having a different size than built in pointers. This has a performance advantage in that the metadata is stored with the pointer that needs it.
2. For pointers that must be used with external libraries that do not use the CCured types, the metadata is stored separately from the pointer. This representation of CCured pointers can be passed to external functions that expect a one word pointer.

In order to limit the programmer involvement as much as possible, CCured has a type inference algorithm that tries to deduce the correct pointer qualifiers to use. Obviously, the run-time overhead introduced by CCured is minimized when it can find the least expensive type qualifier that will guarantee memory safety for a pointer.

However the CCured system is not fully automatic. Necula [144] estimates that converting a 100 kloc program to use the CCured type system will require approximately 30 hours. Programs using a large number of C "dirty tricks" are expected to take longer. The steps to convert a C program to use CCured are as follows:

1. Alter the projects build process to use the CCured translator. Annotate program constructs CCured cannot process (for example sizeof and variable argument functions).
2. Review the type casts CCured cannot prove the safety of and manually annotate the safe casts.
3. Resolve linker errors with external libraries caused by incompatibilities between some CCured pointer types and native pointers.
4. Test and debug the CCured version of the applications.

Due to the metadata required for the CCured type inference algorithm to work, external libraries can be problematic. In the worst case, a programmer must make hand wrapper functions for all external

programs that are called from a CCured application. The CCured system provides such wrappers for the C standard library. The combination of run-time overhead and the difficulty integrating with external libraries prevents techniques like CCured from being used in all applications.

### 2.3 High-Performance in Computer Algebra and Numerical Methods

Computer algebra algorithms are typically optimized in two ways. The first is to improve the time or space complexity of an algorithm. We will not consider any optimizations of this kind. The second is to optimize the multi-precision arithmetic that forms the computational foundation of all computer algebra algorithms. This type of implementation with GMP [74] as the multi-precision library is used by current computer algebra systems such as Maple [135], NTL [168], and Pari [1] for implementing high performance computer algebra algorithms.

GMP consists of highly optimized data structures and algorithms for arithmetic with multi-precision integers, rational numbers, and floating point numbers. The majority of the optimization in GMP comes from implementing the algorithms in platform specific assembly language. The only motivation for implementing GMP in assembly is performance. If optimizing compilers were able to achieve similar performance as the hand-implemented assembly, there would be no need to implement GMP in anything other than a high-level language. Despite all of the abilities of optimizing compilers [14, 148], they still cannot gain the level of performance of the hand coded assembly. This is because the compiler lacks domain knowledge. The GMP developers know they are making a multi-precision arithmetic library. They can optimize it accordingly. The compiler developers have no way of knowing when the compiler is being given a multi-precision arithmetic program as an input. From the compiler's perspective, the code consists of a series of loads, stores, branches, and single word arithmetic. The compiler developers have even less domain knowledge than the GMP developers. Not only do they not know the context of the multi-precision arithmetic, but they don't even know that multi-precision arithmetic is occurring.

As an example of one of the optimization opportunities that compilers are unlikely to notice, consider carry propagation. GMP developers select different carry propagation schemes depending on the architecture they are targeting. On some platforms, GMP exploits instruction-level parallelism by performing digit additions in the full machine word and reconstructing the carry separately from the main digit additions. To a compiler, most high-level implementations of carry propagation will present a dependency that cannot be optimized away because this type of optimization can only be

safely applied when the dependency is known to be caused by carry propagation.

The inability of compilers and assembly implementations to compete with specialized domain knowledge is not unique to computer algebra. Numerical linear algebra algorithms are one of the most studied optimization problems. However, it is still common for assembly implementations [71] to provide superior performance compared to implementations in a high level language. This is despite the fact that linear algebra algorithms benefit from so many of the loop nest optimizations designed to improve the usage of the memory hierarchy [28, 207, 32, 131, 208, 99, 183, 100, 29].

General purpose optimizing compilers have another constraint compared to specialized implementations for a specific domain: the average compiler user will only tolerate a moderate amount of compilation time spent on optimization. This means that compiler optimizations have a time budget that cannot be exceeded. Given that compilers are used in a wide range of applications and by different kinds of users, this tends to result in optimization algorithms that spend less time than would be tolerated by people with more demanding applications.

Unlike computer algebra, for numerical methods these limitations are commonly addressed with auto-tuning. When done correctly, auto-tuning allows the limitations from the optimization time budget, the lack of domain knowledge in general purpose solutions, and the introduction of new platforms to be overcome. Auto-tuning consists of using a code generator to generate many potential implementations of a given algorithm, compiling each implementation with a general purpose optimizing compiler, and selecting the best implementation through benchmarking. The entire process is a more exhaustive search through the optimization space than can usually be made by a compiler or a developer.

The first auto-tuner was PHiPAC [20]. It used a code generator to generate ANSI C code for matrix multiplication. It produced code utilizing register and cache tiles and explicitly avoided memory aliasing by copying array values to scalar variables. The generated code also favored code constructs that were more likely to allow a compiler to access array elements with the index as an immediate constant in the generated assembly code. The types of optimizations performed by PHiPAC are representative of the benefits that can be obtained from auto-tuning. PHiPAC was eventually superseded by ATLAS [205, 206], the Automatically Tuned Linear Algebra System. Similar types of auto-tuning were also applied to Fourier Transforms (FFTW [59, 60, 61]) and Digital Signal Processing (SPIRAL [210, 151]). Auto-tuning has also been applied to MPI operations [54, 55, 197].

All of these uses of auto-tuning have some fundamental similarities. The first is that each auto-

tuner is targeting a set of specific algorithms. Because the author of the auto-tuner knows the target algorithms, it is possible to exploit domain knowledge in the production of the generated code. This knowledge can be used to apply code transformations that compilers either cannot perform or cannot perform safely.

The auto-tuner can generate code that is searching over different hardware parameters (e.g. the tile size to optimize memory accesses). This allows the auto-tuner to function on multiple hardware architectures (including ones the authors of the auto-tuner did not know about). The only portability constraint is that the parameters searched over must generally apply to a given platform. This is a much less severe constraint than it may first appear. Modern architectures share features such as multi-level memory hierarchies, pipelining, instruction-level parallelism, and vector instructions. One of the other benefits of this approach is that optimizations do not need to be tried in isolation. The auto-tuner is free to perform multiple optimizations at the same time and determine the net effect.

Auto-tuning allows optimizations to be performed in a fashion that is complementary to the current state of the art in general purpose optimizing compilers. As an example, consider register allocation. Despite the fact that register allocation is NP-complete [166], compilers will typically do a good job of using either graph coloring or integer linear programming to allocate registers for scalar variables [29]. However, they still do not do as well as auto-tuners for identifying which array elements can be placed into registers. Because of this, a common optimization performed by auto-tuners is to copy array elements into scalar variables [28, 207, 29, 102]. The auto-tuners are using domain knowledge to identify temporal locality that the compiler would not find in the array elements, but are delegating the register allocation to the compiler.

### 3. Iterators for Compiler Enforced Memory Safety

#### 3.1 Introduction

The generic programming paradigm has allowed the creation of libraries with loosely coupled and highly reusable components. Through the use of concepts [139, 142] these libraries have been able to both raise the level of abstraction used in the library interfaces while at the same time maintaining the same run-time efficiency as hand-made components optimized for specific data structures. The most successful libraries developed with these techniques are the C++ Standard Template Library (STL) [12, 141, 94, 191], the Boost Graph Library (BGL) [177, 174], the Matrix Template Library (MTL) [178, 179, 125], and Loki [6, 7].

One of the major enabling technologies used by these libraries is the iterator concept [64, 12]. Through careful consideration of data traversal patterns of many algorithms, the STL iterator concept hierarchy [12] specifies iterators in such a way that all algorithms can be implemented in terms of iterators. This allows algorithms to be decoupled from the implementation details of any specific data structure. At the same time, this generality is not at the cost of run-time efficiency. This is possible because the iterator concepts in the STL are arranged in a hierarchy of increasingly refined concepts. As the iterator concepts become more refined, the concepts become more restrictive of iterator behavior. These restrictions can be exploited by algorithm implementors to provide efficient algorithm implementations for each type of iterator.

One current limitation in generic programming technology is the difficulty in proving that a particular combination of generic software components will work correctly together. Several tools for mechanized proof and mathematical knowledge management show promise for automatically validating combinations of generic software components [164]. There is also interest in direct programming language support for this problem [65, 192, 78, 173].

To the best of our knowledge, the Recursively Fixed Iterator [158] and the Structure Protecting Iterator (Section 3.2.4) are the only concepts developed that enforce safety properties. We have successfully employed the Recursively Fixed Iterator concept in the SACLIB [36] computer algebra library to allow standards compliant [94, 191] C++ compilers to detect the absence of memory leaks and double deletes from static code properties during compilation [158]. SACLIB has been developed over the course of the last three decades. It now contains about 80,000 lines of C++ code and serves

as a reference implementation of numerous computer algebra algorithms. Relatively simple code transformations sufficed to replace the SACLIB memory management with our implementation of a model of the Recursively Fixed Iterator concept.

In this chapter, we provide a formal proof of the ability of the Recursively Fixed Iterator and the Structure Protecting Iterator concepts to prevent memory leaks or double deletes. To carry out the proofs we introduce new concepts dealing with safety that are fundamentally different from existing concepts; in particular, they are different from the current STL concepts. We also describe a general methodology for specifying safety in concepts. Finally, we present an implementation methodology that allows existing compilers to enforce concept-specified safety properties without requiring any run-time or space overhead in the generated object code.

In Section 3.2 we provide a formal definition of the Recursively Fixed Iterator and Structure Protecting Iterator concept and prove their ability to prevent memory leaks and double deletes. In Section 3.3 we demonstrate that our implementation does not have any run-time or space overhead and explain what features of modern compilers are exploited to avoid overhead. In Section 3.4 we present benchmarks of compilation time and execution time. In Section 3.5 we compare our techniques with existing work. Finally, we conclude in Section 3.6 with a description of the concept specification techniques that allowed the correctness proofs for the Recursively Fixed Iterator and Structure Protecting Iterator and outline how these techniques can be applied to other safety problems.

The definitions and theorems in this chapter fall into the following categories:

- standard definitions from reference sources (Definitions 3.2.1, 3.2.2, 3.2.3, 3.2.4),
- definitions and theorems previously presented [158, 157], but included for ease of reading (Definitions 3.2.6, 3.2.12, 3.2.25, 3.2.25; Theorem 3.2.24),
- definitions and theorems that have been improved from their original presentation [158, 157] to cover additional cases or to provide stronger semantics (3.2.9, 3.2.11, 3.2.14, 3.2.16, 3.2.17, 3.2.18, 3.2.19, 3.2.20, 3.2.21, 3.2.22, 3.2.23, 3.2.26, 3.2.27, 3.2.34; Theorem 3.2.36),
- new definitions that have not been previously presented (Definitions 3.2.7, 3.2.13, 3.2.15, 3.2.28, 3.2.29, 3.2.30, 3.2.31, 3.2.32, 3.2.33, 3.2.40, 3.2.41, 3.2.38; Theorem 3.2.39).

### 3.2 The Recursively Fixed Iterator and Structure Protecting Iterator Concepts

In C++, memory leaks and double deletes are notoriously difficult software defects to avoid, find, and repair. In the general case, absence of memory leaks and double deletes is a run-time property. This prevents static analysis tools from proving their absence. Preventing these defects is further complicated by the fact that at the language level, access to dynamic memory via pointers is decoupled from the ownership and management of the memory. As a result, programmers must constantly guard against the introduction of memory leaks and double deletes.

In this section, we will introduce definitions and concepts to allow memory management and ownership to be clearly discussed. Using these definitions we will specify the Recursively Fixed Iterator and Structure Protecting Iterator concepts and then prove the concepts can be used to prevent memory leaks and double deletes at compile-time. The specifications of the Recursively Fixed Iterator and Structure Protecting Iterator were carefully designed to allow them to be easily used in existing code implemented with pointers and to allow information about memory semantics to be embedded into the C++ type system.

#### 3.2.1 Definitions From the STL

In the development of the theoretical framework used to describe our memory management system and theorems about various aspects of its correctness we will use the following definitions from the STL [12]:

**Definition 3.2.1. (from STL)** A concept [12, 139, 142] is a collection of constraints on a type. Constraints may deal with the syntax and semantics of a type or the time/space complexity of operations supported by the type.

The following definitions from the C++ STL [12, 94, 191] regarding concepts will be used:

**Definition 3.2.2. (from STL)** A C++ type  $T$  is a *model* of a concept when it meets all the constraints of the concept.

Note that a concept specifies the minimal set of constraints a type must have to be a model of the concept. Therefore even when a type has more features than those described by a concept it is still considered a model of the concept.

**Definition 3.2.3. (from STL)** A concept  $C_2$  is a *refinement* of the concept  $C_1$  if  $C_2$  specifies all the constraints of  $C_1$  plus additional constraints not specified by  $C_1$ . When  $C_2$  is a refinement of  $C_1$  we say that  $C_2$  is a *stronger* concept than  $C_1$  and  $C_1$  is a *weaker* concept than  $C_2$ .

We use the following concept from the STL:

**Definition 3.2.4. (from STL)** A type  $T$  is a model of the *Default Constructible* concept if it has a default constructor.

*Remark 3.2.5.* A default constructor is defined by the C++ standard [94, 191] as a constructor that may be called without any user supplied arguments.

### 3.2.2 Conditions for memory leaks and double deletes

In order to discuss the usage of multi-dimensional memory (see Figure 3.1 for an example) and prove which usages will not result in a memory leak or double delete we provide the following definitions and theorems.

**Definition 3.2.6.** Let  $T$  be a C++ type. If  $T$  is a user-defined type then  $T::operator*$ ,  $T::operator->$ , and  $T::operator[]$  have the usual C++ meaning of referring to  $T$ 's member functions  $operator*$ ,  $operator->$ , and  $operator[]$ . If  $T$  is a built-in pointer type then  $T::operator*$ ,  $T::operator->$ , and  $T::operator[]$  refer to the functionality offered by the compiler when  $*$ ,  $->$ , and  $[]$  are applied to instances of type  $T$ .

**Definition 3.2.7.** Let  $t_1$  and  $t_2$  be C++ variables of type  $T$ .  $T$  is a model of the *Copyable* concept if for all  $t_1$  and  $t_2$

1.  $T(t_1)$  returns a variable of type  $T$  that is an equivalent copy of  $t_1$ , and
2.  $T t_1(t_2)$  constructs  $t_1$  to be an equivalent copy of  $t_2$ .

*Remark 3.2.8.* This definition of Copyable makes the Assignable [12] concept a refinement of Copyable.

**Definition 3.2.9.** A C++ type  $T$  is a model of the *Pointer-like* concept if

1.  $T$  is a model of the Copyable concept, and

2.  $T::\text{operator}^*$  has pointer semantics.

**Definition 3.2.10.** A C++ type  $T$  is a model of the *Offsettable Pointer* concept if

1.  $T$  is a model of the Pointer-like concept and,
2. `difference_type` is a signed integral type capable of representing the distance between two variables of type  $T$ , and
3.  $T::\text{operator}+(\text{difference\_type})$  and  $T::\text{operator}-(\text{difference\_type})$  are provided with pointer semantics.

**Definition 3.2.11.** A C++ type  $T$  is a model of the *Fully Pointer-like* concept if

1.  $T$  is a model of the Offsettable Pointer concept,
2.  $T::\text{operator->}$ , and  $T::\text{operator}[]$  are valid function calls with pointer semantics and,
3.  $T$  supports all pointer arithmetic.

**Definition 3.2.12.** Let  $F$  be a C++ function. If  $F$  never uses `const_cast` to remove `const` from a cv-qualified type,  $F$  is a *const-respecting function*.

**Definition 3.2.13.** Let  $F$  be a C++ function. We denote the return type of  $F$  by  $\text{ret}(F)$ .

**Definition 3.2.14.** Let  $F$  be a C++ function and  $T$  be a C++ type. If  $T$  is not a model of the Pointer-like concept, then the *dimension* of the type  $T$  is  $\text{dim}(T) = 0$ . If  $T$  is a model of the Pointer-like concept, then  $\text{dim}(T) = 1 + \text{dim}(\text{ret}(T::\text{operator}^*))$ . For all instances  $t$  of type  $T$ ,  $\text{dim}(t) = \text{dim}(T)$ .

**Definition 3.2.15.** A *valid memory dereference* occurs in the following cases:

1. Let  $c$  be a C++ variable of type  $C$  where  $C$  is a model of the STL Container concept. Let  $i$  be a C++ variable of type  $I = C::\text{iterator}$ . The C++ expression  $*i$  is a *valid memory dereference* if there is a  $c$  in scope such that  $i \in [c.\text{begin}(), c.\text{end}())$ .
2. Let  $p$  be a C++ variable of type  $P$  where  $P$  is a built-in C++ pointer. Let  $a$  be a C++ array of  $n$  elements. Let the elements of  $a$  be allocated with type  $\text{iterator\_traits}\langle P \rangle::\text{value\_type}$ . Let  $p \in [a, a + n)$ . Then  $*p$  is a *valid memory dereference* if  $a$  has not been deallocated.
3. Let  $P$  be a built-in C++ pointer type. Let  $i$  be a C++ variable of type  $\text{iterator\_traits}\langle P \rangle::\text{value\_type}$ . Let  $i$  not be an element of a built-in C++ array. Let  $p$  be a C++ variable of type  $P$  and let  $p = \&i$ . Then  $*p$  is a *valid memory dereference* if  $i$  is stack allocated and in scope or heap allocated and not deleted.
4. Let  $p$  be a C++ variable of type  $P$ . Let  $P$  be a model of the Pointer-like concept. Let  $P$  not be a built-in C++ pointer. Let  $p$  not be an iterator into a container. Then  $*p$  is a *valid memory dereference* if all memory dereferences caused by  $p.\text{operator}*(\ )$  are valid memory dereferences.

**Definition 3.2.16.** Let  $t$  be a C++ variable of type  $T$ . Then the *index span* of  $t$  is defined as

$$\text{span}(t) = \begin{cases} \emptyset & T \text{ does not model the Offsettable Pointer concept;} \\ \{n \in \mathbb{Z} \mid * (t + n) \\ \text{is a valid memory dereference}\} & T \text{ models the Offsettable Pointer concept.} \end{cases}$$

**Definition 3.2.17.** Let  $t$  be a C++ variable of type  $T$ . Then the set of *directly dereferenceable elements* of  $t$  is defined as

$$\text{DD}(t) = \begin{cases} \emptyset & T \text{ does not model the Pointer-like concept;} \\ \{*(t + n) \mid n \in \text{span}(t)\} & T \text{ models the Offsettable Pointer concept;} \\ \{*t\} & \text{otherwise.} \end{cases}$$

**Definition 3.2.18.** Let  $t$  be a C++ variable of type  $T$ . Then the *dereferenceable closure* of  $t$  is defined as

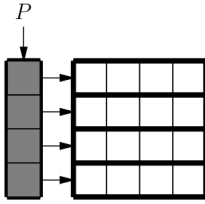


Figure 3.1: Multi-dimensional memory.  $P$  is a C++ variable with type `int**`. The gray memory cells contain pointers and are the structure of  $P$  (see Definition 3.2.19). The white memory cells contain integers and are the contents of  $P$  (see Definition 3.2.20).

$$\text{DC}(t) = \begin{cases} \emptyset & T \text{ does not model the Pointer-like concept;} \\ \text{DD}(t) \cup_{i \in \text{DD}(t)} \text{DC}(i) & \text{otherwise.} \end{cases}$$

**Definition 3.2.19.** Let  $t$  be a C++ variable of type  $T$ . Then the *structure* of  $t$  is defined as

$$\text{struct}(t) = \{i \in \text{DC}(t) \mid \dim(i) > 0\}.$$

**Definition 3.2.20.** Let  $T$  be a C++ type and let  $t$  be an instance of  $T$ . Then the *contents* of  $T$  are defined as

$$\text{cont}(t) = \text{DC}(t) - \text{struct}(t).$$

Using these definitions, we may classify C++ functions in the following three ways.

**Definition 3.2.21.** Let  $F$  be a C++ function. Let  $F$  take an argument  $p$  of type  $P$ . Let  $P$  be a model of the Pointer-like concept. Then,  $F$  is a *memory manager* of  $p$  if  $F$  writes to  $\text{struct}(p)$  or deallocates memory referred to by any of the elements in  $\text{struct}(p)$  in at least one of its execution paths.

**Definition 3.2.22.** Let  $F$  be a C++ function. Let  $F$  take an argument  $p$  of type  $P$ . Let  $P$  be a

model of the Pointer-like concept. Then,  $F$  is a *contents user* of  $p$  if  $F$  reads or writes any elements of  $\text{cont}(p)$  in at least one of its execution paths.

**Definition 3.2.23.** Let  $F$  be a C++ function. Let  $F$  take an argument  $p$  of type  $P$ . Let  $P$  be a model of the Pointer-like concept. Then,  $F$  is a *contents-only user* of  $p$  if  $F$  is a contents user of  $p$  and  $F$  is not a memory manager of  $p$ .

**Theorem 3.2.24.** *Let  $F$  be a C++ function. Let  $F$  be a contents-only user of all arguments that model the Pointer-like concept. Then  $F$  does not leak memory or double delete memory referred to by these arguments.*

*Proof.*  $F$  does not leak memory because as a contents-only user of its Pointer-like arguments it cannot write to their structure. Because the structure is not written to, none of the existing references to allocated memory can be lost and therefore there cannot be a memory leak.  $F$  cannot cause a double deletion because as a contents-only user of its fully pointer like arguments it will never delete any of the memory referred to by the structure of its Pointer-like arguments.  $\square$

Theorem 3.2.24 offers a condition for verifying that a function does not leak or double delete any memory referred to by its Pointer-like arguments. However, without a tool to identify which functions are memory managers and which functions are contents-only users developers must manually determine the memory correctness of each function they are debugging for memory leaks or double deletes. This is a tedious process because when memory errors occur they are generally not confined to a single function. Furthermore, maintenance programming results in the need for maintenance programmers to constantly guard against introducing memory leaks and double deletes.

### 3.2.3 Recursively Fixed Iterator Concept

Motivated by the desire to bring the ability to declare a function to be a contents-only user of a Pointer-like argument into the C++ type system we gave a first definition of the Recursively Fixed Iterator concept [158]. However, the original definition had several shortcomings. It specified some constraints at the level of implementation details, was not sufficiently rigorous to allow a formal proof that models of the Recursively Fixed Iterator do not leak or double delete memory, and it did not specify how models of the type can be constructed.

To facilitate the discussion of object construction and resource management, we specify the following concepts:

**Definition 3.2.25.** Let  $T$  be a C++ type. *Implicitly acquired resources of  $T$*  are any resources for whose allocation and deallocation  $T$  is responsible. Resources not documented as the client's responsibility are assumed to be implicitly acquired.  $T$  is a model of the *Leak Free Destroyable* concept if for all instantiations  $t$  of objects of type  $T$  the method  $T::\sim T$  correctly releases all resources that were implicitly acquired by  $t$  and it is statically verifiable that there is no execution sequence of  $t$ 's methods that can produce a resource leak.

**Definition 3.2.26.** Let  $T$  be a C++ type.  $T$  is a model of the *Double Delete Free Destroyable* concept if for all instantiations  $t$  of objects of type  $T$  the method  $T::\sim T$  does not cause multiple deletions of resources implicitly managed by  $T$  and it is statically verifiable that there is no execution sequence of  $t$ 's methods that can produce a double deletion of a resource implicitly managed by  $T$ .

**Definition 3.2.27.** Let  $T$  be a C++ type.  $T$  is a model of the *Safely Manageable* concept if it is a model of the Leak Free Destroyable and Double Delete Free Destroyable concepts.

We introduce the following definitions and notations to simplify discussing pointers.

**Definition 3.2.28.** Let  $F$  be a C++ function. Let  $T$  be a C++ type. We define the *reference free return type* of  $F$  as

$$\text{ref\_free\_ret}(F) = \begin{cases} T & \text{if } \text{ret}(F) \text{ is the reference } T\&; \\ \text{ret}(F) & \text{otherwise.} \end{cases}$$

**Definition 3.2.29.** Let  $P$  be a C++ type that models the Pointer-like concept. The *base type* of  $P$ , is defined as

$$\text{base}(P) = \begin{cases} \text{ref\_free\_ret}(P::\text{operator}^*) & \text{if } \text{dim}(P) = 1; \\ \text{base}(\text{ref\_free\_ret}(P::\text{operator}^*)) & \text{if } \text{dim}(P) > 1. \end{cases}$$

**Definition 3.2.30.** We denote a built-in C++ pointer type of dimensionality  $d$  and base type  $b$  as  $\text{pointer}(d, b)$ .

We introduce the following definitions to simplify discussions of memory structure.

**Definition 3.2.31.** Let  $t_1, t_2$  be C++ variables of type  $T$ ,  $d = \dim(T)$ ,  $d > 0$ , and  $B = \text{base}(T)$ . Let  $P$  be a fully-pointer like type with  $\dim(T) = \dim(P)$  and  $\text{base}(T) = \text{base}(P)$ . Let  $p$  be a C++ variable of type  $P$ .  $T$  is a model of the Structure Alias concept for variables of type  $P$  if

1.  $T$  is a model of the Assignable concept. Let  $T::v$  be a variable of type  $P$ .  $T$  provides the constructor  $T::T(P)$ . For all  $p$ , the C++ expressions  $T :: T(p)$  and  $T t(p)$  construct a  $t$  such that  $t.v = p$ . The expression  $t_1=t_2$  has the side effect  $t_1.v = t_2.v$ . For the lifetime of  $t$   $\text{struct}(t) = \text{struct}(t.v)$  The value of  $p$  when  $t$  is constructed is referred to as  $p_0$ .
2. If  $d > 1$  let  $R$  be a C++ type that models the Structure Alias concept for variables of type  $\text{ref\_free\_ret}(P::\text{operator}^*)$ . Otherwise, let  $R = B$ .  $T$  provides  $T::\text{operator}^*$ ,  $T::\text{operator}->$ , and  $T::\text{operator}[]$  with the following semantics

```
R& T::operator*(){
    return *v;
} //T::operator*
```

```
P T::operator->(){
    return v;
} //T::operator->
```

```
R& T::operator[](std::iterator_traits<P>::size_type n){
    return v[n];
} //T::operator[]
```

3.  $T$  provides all pointer arithmetic operators that are not self-modifying ( $+$ ,  $-$ ,  $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $==$ ,  $!=$ ). Let  $\cdot$  represent one of these operators. For all such operators  $t_1 \cdot t_2 \Leftrightarrow t_1.v \cdot t_2.v$ .
4.  $T$  is not a memory manager of  $p_0$  (see Definition 3.2.21).

**Definition 3.2.32.** Let  $t$  be a C++ variable of type  $T$ .  $T$  is a model of the Immutable Structure Alias concept if

1.  $T$  is a model of the Structure Alias concept for variables of type  $P$ ,
2. for all  $t$ ,  $\text{struct}(t) = \text{struct}(p_0)$  for the life-time of  $t$ , and
3. for all  $t$   $\text{struct}(t)$  is immutable.

**Definition 3.2.33.** Let  $t$  be a C++ variable of type  $T$ .  $T$  is a model of the Constant Value concept if

1. for all  $t$ ,  $t$  is immutable after construction.

Using these definitions, we now define the Recursively Fixed Iterator concept.

**Definition 3.2.34.** Let  $t$  be a C++ variable of type  $T$  with  $d = \text{dim}(T)$ ,  $d > 0$ , and  $b = \text{base}(T)$ . Let  $P$  be a fully-pointer like type with  $\text{dim}(T) = \text{dim}(P)$  and  $\text{base}(T) = \text{base}(P)$ .  $T$  is a model of the *Recursively Fixed Iterator* concept if

1.  $T$  is a model of the Immutable Structure Alias for variables of type  $P$ ,
2.  $T$  is a model of the Constant Value concept, and
3.  $T$  is a model of the Safely Manageable concept.

Our prior work [158] only presented an intuitive argument for the ability of a memory management system using models of the Recursively Fixed Iterator concept for memory access to prevent memory leaks and double deletes. We now present a formal proof using our revised definition of the concept.

**Definition 3.2.35.** Let  $C$  be a C++ type that models one of the STL Container concepts. Note that, for all instances  $c$  of type  $C$ , the elements contained by  $c$  are implicitly managed resources. If  $C$  is also a model of the Safely Manageable concept, and it contains elements of a type that models the Safely Manageable concept then  $C$  is a model of the *Resource Safe Container* concept.

**Theorem 3.2.36.** *Let  $C$  be a C++ type that is a model of the Resource Safe Container concept. Then all instances  $c$  of type  $C$  will not leak or double delete resources.*

*Proof.* Immediate from Definition 3.2.35. □

```

1: //initialize r1 to point to "responsibly" managed memory
2:   array2d<int>          memory_manager;
3:   rec_fixed_itr<int**> r1(memory_manager.begin());
4:
5: //initialize r2 to "irresponsibly" managed memory
6:   int* a = new int[10];
7:   rec_fixed_iter<int*> r2(a);
8:
9: //what if this were allowed?
10:  std::swap(r1[0],r2); //memory_manager loses track of r1[0]

```

Figure 3.2: An example of the dangers to memory safety when swapping structure elements.

```

1: //initialize r1 to point to "responsibly" managed memory
2:   array2d<int>          memory_manager;
3:   rec_fixed_itr<int**> r(memory_manager.begin());
4:
5: //what if this were allowed?
6:   r[0]++; //memory_manager loses track of r[0]

```

Figure 3.3: An example of the dangers to memory safety caused by incrementing structure elements.

**Theorem 3.2.37.** *Let  $P$  be a C++ program. Let  $P$  use Resource Safe Containers for all resource management. Let all functions in  $P$  be const-respecting. Let all Pointer-like function arguments outside of Resource Safe Containers in  $P$  be passed by types that are models of the Recursively Fixed Iterator concept. Then,  $P$  never leaks or double deletes resources.*

*Proof.* Resource Safe Containers guarantee the resources managed by the container will delete exactly once, so the Resource Safe Containers will not leak or double delete memory.

The Recursively Fixed Iterator prevents both writes and deletes of memory structure referred to by a model of the Recursively Fixed Iterator concept. This means that a function taking a Pointer-like argument that models the Recursively Fixed Iterator concept is a contents-only user of that argument. Because all function arguments outside of the Resource Safe Containers are Recursively Fixed Iterators, all functions outside of the Resource Safe Containers are contents-only users of all memory passed to them. Theorem 3.2.24 guarantees that these functions will not leak or double delete resources referred to by their Pointer-like arguments.  $\square$

### 3.2.4 Structure Protecting Iterator Concept

Although the constraints imposed by the Recursively Fixed Iterator provide memory safety, there are two desirable use cases that they restrict. The first use case is the swapping of two models of the Recursively Fixed Iterator. Consider the example in Figure 3.2 where an attempt is made to swap two structure elements. The example uses two SACLIB 3.0 types. `array2d<int>` is a container that manages the memory for a two dimensional array. The `rec_fixed_itr` is a model of the Recursively Fixed Iterator concept (see Section 3.3 for implementation details). On lines 1-3 the variable `memory_manager` is declared and the variable `r1` is initialized to alias the structure managed by `memory_manager`. On lines 5-7 an array is allocated on the heap and `r2` is initialized to alias the heap allocated array. The call to `std::swap` will not compile. `std::swap` requires its arguments to be models of the Assignable concept, and models of the Recursively Fixed Iterator are not Assignable because they are refinements of the Constant Value concept. Preventing this call to `std::swap` protects the structure that `memory_manager` is managing. The call to `std::swap` is not safe to allow to compile because it removes `r1[0]` from `struct(memory_manager)` and replaces it with `r2`. In general, a given memory manager is not capable of managing structure whose allocation it does not know about.

A similar problem occurs with self-modifying pointer arithmetic on models of the Recursively Fixed Iterator concept. Consider the example in Figure 3.3. Lines 1-3 instantiate a `memory_manager` and make `r` alias its structure. On line 6, an attempt is made to increment `r[0]`. This also does not compile because the Recursively Fixed Iterator concept is a refinement of the Constant Value concept. Allowing `r[0]` to be incremented on line 6 would also result in `struct(memory_manager)` being modified without the awareness of `memory_manager`. This causes the same memory safety problems as allowing the swap.

However, swapping and incrementing are not always detrimental to memory safety. The danger of allowing these operations comes from a memory manager having the structure of the memory it manages replaced with structure that was not allocated according to invariants the memory manager requires in order to maintain memory safety. There are two conditions under which structure can be modified without violating the invariants. The first is when a model of the Structure Alias concept is known to be a copy of managed structure. In this case, there is an original maintained by a memory manager and it is safe to modify the copy via assignment or pointer arithmetic. The second is when two structure elements that maintain the same memory safety producing invariants are swapped.

We define the Structure Protecting Iterator concept to allow self-modification while maintaining the memory safety properties of the Recursively Fixed Iterator.

**Definition 3.2.38.** Let  $t$  be a C++ variable of type  $T$  with  $d = \dim(T)$ ,  $d > 0$ , and  $b = \text{base}(T)$ . Let  $P$  be a fully-pointer like type with  $\dim(T) = \dim(P)$  and  $\text{base}(T) = \text{base}(P)$ . Let  $R$  be the return type of  $T::\text{operator}^*$  and  $T::\text{operator}[]$ . Then  $T$  is a model of the *Structure Protecting Iterator* concept if

1.  $T$  is a model of the Structure Alias concept for variables of type  $T$  and  $P$ ,
2.  $T$  provides the self-modifying pointer arithmetic operators ( $++$ ,  $+=$ ,  $--$ ,  $-=$ ) with the following semantics (only addition is shown).

```

T& operator+=(const std::iterator_traits<P>::difference_type i){
    return *this;
} //operator+=

T& T::operator++(){
    *this += 1;
    return *this;
} //operator++()

const T T::operator++(int){
    T old = *this;
    ++(*this);
    return old;
} //operator++(int)

```

3.  $R = b$  when  $d = 1$ . Otherwise,  $R$  is a model of the Recursively Fixed Iterator concept.
4.  $T$  is a model of the Safely Manageable concept.

**Theorem 3.2.39.** Let  $P$  be a C++ program. Let  $P$  use Resource Safe Containers for all resource management. Let all functions in  $P$  be const-respecting. Let all Pointer-like function arguments out-

side of Resource Safe Containers in  $P$  be passed by types that are models of the Structure Protecting Iterator or Recursively Fixed Iterator concepts. Then,  $P$  never leaks or double deletes resources.

*Proof.* Resource Safe Containers guarantee the resources managed by the container will be deleted exactly once, so the Resource Safe Containers will not leak or double delete memory.

All function arguments that model the Recursively Fixed Iterator will not be leaked or double deleted due to Theorem 3.2.37.

Let  $s$  be a C++ variable of type  $S$ . Let  $S$  be a model of the Structure Protecting Iterator concept. The Structure Protecting Iterator concept guarantees that  $\text{struct}(s)$  cannot be written to or deleted. Changes to the value of  $s$  do not alter  $\text{struct}(s)$ , they simply cause  $\text{struct}(s)$  to alias a different memory structure. Accesses to  $\text{struct}(s)$  via  $S::\text{operator}^*$  or  $S::\text{operator}[]$  return a type that models the Recursively Fixed Iterator concept. Therefore, functions that take arguments that are models of the Structure Protecting Iterator concept are contents-only users of those arguments, and the memory accessible from those arguments is not leaked or double deleted by the same reasoning used to prove Theorem 3.2.37.  $\square$

We define the Swappable Structure concept to allow the exchange of structure elements.

**Definition 3.2.40.** Let  $t$  be a C++ variable of type  $T$  with  $d = \dim(T)$  and  $d > 1$ . An *index vector* for  $t$  is a sequence  $I = (i_0, i_1, i_2, \dots, i_{d-2})$  and the expression  $\text{element}(t, I)$  denotes  $t[i_0][i_1][i_2] \dots [i_{d-2}]$ .

**Definition 3.2.41.** Let  $t$  be a C++ variable of type  $T$  with  $d = \dim(T)$  and  $d > 1$ . Let  $P$  be a fully pointer-like type with  $\dim(T) = \dim(P)$  and  $\text{base}(T) = \text{base}(P)$ . Let  $v_1$  and  $v_2$  be index vectors for  $t$ . Then  $T$  is a model of the Swappable Structure Alias concept for variables of type  $P$  if

1.  $T$  is a model of the Structure Alias concept for variables of type  $P$ ,
2.  $t.\text{swap}(\text{element}(t, v_1), \text{element}(t, v_2))$  has the side effect  $\text{std}::\text{swap}(\text{element}(t.v, v_1), \text{element}(t.v, v_2))$ ,  
and
3. For all  $t_1$  and  $t_2$ ,  $t_1.v$  and  $t_2.v$  are interchangeable with respect to the invariants their memory managers require to ensure memory safety.

We do not implement a model of the Swappable Structure Alias concept, as there is no need for a model of it in the current SACLIB 3.0 implementation. We expect that static verification of

```

1: template<typename T>
2: class simple_ptr{
3:
4:     public:
5:         simple_ptr(T *p):pointer_(p){}
6:
7:         T& operator*() { return *pointer_; }
8:         T* operator->(){ return pointer_; }
9:
10:        const T& operator*() const{
11:            return *pointer_;
12:        }
13:        const T* operator->() const{
14:            return pointer_;
15:        }
16:
17:     private:
18:         T* pointer_;
19:
20: }//simple_ptr

```

Figure 3.4: The SACLIB 3.0 `simple_ptr` implementation.

condition (3) will require the implementation of  $P$  to be coordinated with all memory managers that can manage objects of type  $P$ . This excludes  $P$  being a built-in pointer when condition (3) is verified statically.

### 3.3 Implementation

#### 3.3.1 Operator Overloading

Implementing `simple_ptr`, `rec_fixed_itr`, and `struct_protect_itr` as models of the Simple Pointer (Section 3.2) , Recursively Fixed Iterator (Section 3.2) and Structure Protecting Iterator (Section 3.2) concepts depend on the use of operator overloading to allow users of the iterators to enjoy the same syntax as built-in pointers. Because of the prevalence of user-defined iterators and smart pointers, this type of operator overloading has been employed in the design and implementation of C++ libraries [12, 94, 191, 38, 3, 7].

The Trivial Iterator Concept [12, 94, 191] is an instructive example to consider. Dereferencing is the only pointer operation required of types that model the Trivial Iterator Concept. As a result, models of the concept have short and straightforward implementations compared to models of more complex iterators. The `simple_ptr` (Figure 3.4) distributed with SACLIB 3.0 is a model of the Trivial

```

1: /*=====
2:           AADV(L; a,Lp)
3: Arithmetic advance.
4:
5: Inputs
6:   L : list.
7:
8: Outputs
9:   a : if L = () then a = 0,   otherwise a = FIRST(L).
10:  Lp : if L = () then Lp = (), otherwise Lp = RED(L).
11:
12: =====*/
13:
14: #include "saclib.h"
15:
16: void AADV(Word L, simple_ptr<Word> a_, simple_ptr<Word> Lp_){
17:     Word Lp,a;           /* hide algorithm */
18:
19: Step1: /* Advance. */
20:     if (L != NIL)
21:         ADV(L,&a,&Lp);
22:     else
23:     {
24:         a = 0;
25:         Lp = NIL;
26:     }
27:
28: Return: /* Prepare for return. */
29:     *a_ = a;
30:     *Lp_ = Lp;
31:     return;
32: }

```

Figure 3.5: The SACLIB 3.0 implementation of AADV.

Iterator Concept. The usage of `simple_ptr` is demonstrated in the SACLIB 3.0 routine `AADV` (see Figure 3.5). `AADV` takes two `simple_ptr<Word>` arguments `a_` and `Lp_` (line 17) and dereferences them (lines 29-30) to return its outputs. When the `simple_ptr<Word>` arguments are dereferenced (lines 29,30), it is with the same syntax that would be used if `a_` and `Lp_` were of the built-in pointer type `Word*`.

The `simple_ptr` implementation (Figure 3.4) uses a private member variable (line 18) to store the state required to implement pointer operations. Public member functions (Lines 5-15) are provided to expose the functionality required by the Trivial Iterator Concept. Operator overloads provide pointer dereferencing for both mutable (Lines 7-8) and const (lines 10-15) `simple_ptr`s. The `simple_ptr` class is templated (line 1) to allow `simple_ptr`s to point to arbitrary types.

The usage and implementation of `simple_ptr` is representative of user-defined iterators and smart-pointers. Operator overloading is used to provide the same syntax enjoyed by built-in pointers. However, the implementation provides different functionality than a built-in type. Combining the built-in pointer syntax with different functionality is the key way that user-defined smart pointers and iterators provide benefits to their users. In the case of `simple_ptr`, it prevents all pointer arithmetic so programmers can be sure there are no unintended uses of pointer arithmetic on `simple_ptr` variables.

For `simple_ptr`, `rec_fixed_itr`, and `struct_protect_itr`, selecting which operators to overload and the scope to overload them at is straightforward. This is generally the case when implementing user-defined iterators and smart pointers. There are two major reasons for this. The first is that C++ restricts the operators that can be overloaded and the scope the overload can take place in [190, 94, 191]. This rigidity makes overloading easier to work with by narrowing the design space. The second reason is that the existing body of iterator and smart pointer implementations provides guidance on both the operators to overload and the scope to define them in.

### 3.3.2 Protecting Memory Structure

Instances of `rec_fixed_itr` and `struct_protect_itr` guarantee memory accessible from their structure will not be leaked or double deleted. This is accomplished with no run-time overhead by making the memory structure immutable at compile-time. Despite the increased implementation complexity of imposing immutability at compile-time, the structure must be made immutable at

compile-time because all run-time mechanisms of structure protection either incur run-time overhead or cannot meet the iterator requirements.

Solutions that detect run-time modification of structure access will not be able to provide the static guarantee provided by the iterators. In the general case, run-time properties of software are input dependent. Because of this, a run-time solution that produces an exception or program termination when memory structure is modified will require testing to validate the absence of structure modification. As with all run-time testing, there is the inability to ensure the absence of memory leaks or double deletes for inputs other than the test cases. The only benefit that such a scheme would provide over traditional memory profilers is the ability to perform the tests without the need for an external profiler. In addition to removing the static guarantee, the generation of exceptions or program termination will introduce run-time overhead in space and time.

Preventing memory leaks at run-time is possible. The shallow copy of the iterator's memory structure could be replaced with a deep-copy or copy-on-write. This would ensure that modifications of memory structure made through an iterator would result in the responsible memory manager retaining a copy of the original state of the structure it required to reclaim memory. Either of these changes would make them different from built-in pointers. The result is that they can no longer be used as direct replacements for built-in pointers. This makes applying them to existing code more labor intensive. The change in semantics places a burden on programmers by requiring them to understand yet another type of semantics for memory access. In addition to the difficulties programmers would have applying such a solution, there is also a run-time performance penalty to be paid. A shallow copy can be completed with  $O(1)$  time and storage. A deep copy requires  $O(n)$  time and storage and copy-on-write requires  $O(n)$  time and storage if a write occurs.

The limitations of run-time detection and altered copy semantics are fundamental limitations of a run-time solution. Any run-time solution will have a combination of the limitations of the two schemes outlined above. As a result, obtaining a fully functional and overhead free solution to protecting memory structure forces the use of a compile-time solution.

### 3.3.3 Detecting Memory Structure

Before we can protect memory structure, we must first be able to identify it. Recall that the dimension of a variable is the number of times it can be dereferenced and that a variable is part of a memory structure if its dimension is greater than 0 (see Section 3.2). Due to the C++ requirements on function return types, for any C++ type  $T$  all variables of type  $T$  will have the same dimension.

```

1: //Metafunction: Dim<T>
2: //Description: Compute the dimension of T.
3: //Invariants:
4: //
5: //   Preconditions:
6: //   (
7: //       is_dereferenceable<T>::value == true and
8: //       std::iterator_traits<T> is a valid specialization
9: //   ) or
10: //   is_dereferenceable<T>::value == false
11: //
12: //   Postconditions:
13: //       1) Dim<T>::value is the dimension of T.
14: //
15:   template <typename T> struct Dim;
16:
17:   template <typename T>
18:   struct Dim{
19:       static const size_t value = Dim_non_array<T>::value;
20:       typedef Dim<T> type;
21:   };//Dim<T>
22:
23:   template<typename T, size_t N>
24:   struct Dim<T[N]>{
25:       static const size_t value = 1 + Dim<T>::value;
26:       typedef Dim<T[N]> type;
27:   };//Dim<T[N]>

```

Figure 3.6: Implementation of the Dim metafunction. See Figure 3.7 for the implementation of Dim\_non\_array.

```

1: //Metafunction: Dim_non_array<T,true|false>
2: //Description: Compute the dimension of T
3: //Invariants:
4: //
5: //   For Dim_non_array<T,true>:
6: //
7: //     Preconditions:
8: //       1) is_dereferenceable<T>::value == true
9: //       2) std::iterator_traits<T> is a valid specialization
10: //       3) boost::is_array<T>::value == false
11: //
12: //     Postconditions:
13: //       1) Dim_non_array<T,true>::value is the dimension of T.
14: //
15: //   For Dim_non_array<T,false>:
16: //
17: //     Preconditions:
18: //       1) is_dereferenceable<T>::value == false
19: //       2) boost::is_array<T>::value == false
20: //
21: //     Postconditions:
22: //       1) Dim_non_array<T,false>::value == 0
23: //
24:   template <
25:     typename T,
26:     bool dereferenceable = is_dereferenceable<T>::value
27:   >
28:   struct Dim_non_array;
29:
30:   template <typename T>
31:   struct Dim_non_array<T,true>{
32:     BOOST_STATIC_ASSERT(
33:       true == is_dereferenceable<T>::value &&
34:       false == boost::is_array<T>::value
35:     );
36:
37:     const static size_t value =
38:       1 +
39:       Dim<
40:         typename std::iterator_traits<T>::value_type
41:       >::value
42:     ;
43:
44:   };//Dim_non_array<T,true>
45:
46:   template <typename T>
47:   struct Dim_non_array<T,false>{
48:     BOOST_STATIC_ASSERT(
49:       false == is_dereferenceable<T>::value &&
50:       false == boost::is_array<T>::value
51:     );
52:
53:     const static size_t value = 0;
54:   };//Dim_non_array<T,false>

```

Figure 3.7: Implementation of the `Dim_non_array` metafunction used to implement `Dim` (Figure 3.6).

```

1: //Metafunction: iterator_value_type<typename T>
2: //Description: Compute the value_type of T.
3: //Invariants:
4: //   Preconditions:
5: //       is_dereferenceable<T>::value == true
6: //   Postconditions:
7: //       iterator_value_type<T>::type is the value_type of T.
8: template <typename T>
9: struct iterator_value_type{
10:     BOOST_STATIC_ASSERT(is_dereferenceable<T>::value);
11:
12:     static const T t;
13:     typedef typeof(*t) type; //typeof is a g++ extension
14: };
15:
16: //Metafunction: Dim<typename T>
17: //Description: Compute the dimension of T.
18: //Invariants:
19: //   Preconditions:
20: //       None
21: //   Postconditions:
22: //       Dim<T>::value is the dimension of T.
23: template <typename T> struct Dim;
24:
25:     template <typename T, bool dereferenceable> struct Dim_impl;
26:         template <typename T> struct Dim_impl<true>{
27:             const static size_t value =
28:                 1 + Dim<typename iterator_value_type<T>::type>
29:         };
30:         template <typename T> struct Dim_impl<false>{
31:             const static size_t value = 0;
32:         };
33:
34:     template <typename T> struct Dim{
35:         const static size_t value = Dim_impl<T,is_dereferenceable<T>::value>;
36:     };

```

Figure 3.8: Improved implementation of the Dim metafunction using the g++ typeof extension.

```

1: namespace guard{//use ADL to protect operator*s defined at namespace scope
2:
3:     //types to represent presence of operator*
4:     struct no_operator_star{};
5:     struct provides_star{char b[1];};
6:     struct used_our_star{char b[2];};
7:     BOOST_STATIC_ASSERT(sizeof(no_operator_star) != sizeof(provides_star) );
8:     BOOST_STATIC_ASSERT(sizeof(no_opeartor_star) != sizeof(used_our_star) );
9:
10:    //Allow any type to be converted to a conversion_from_T.
11:    struct conversion_from_T{
12:        template <typename T> conversion_from_T(T){}
13:    };
14:
15:    //operator* for types that don't provide one
16:    no_operator_star operator*(conversion_from_T);
17:
18:    //Given the variable T t, function overloads to determine the return type *t
19:    used_our_star provides_operator_star(no_operator_star);
20:    provides_star provides_operator_star(conversion_from_T);
21:
22:    template <typename T>
23:    struct provides_user_defined_operator_star_impl{
24:        BOOST_STATIC_ASSERT((false == boost::is_fundamental<T>::value));
25:
26:        typedef typename boost::remove_cv<T>::type no_cv_type;
27:        const static no_cv_type t;
28:        const static bool value =
29:            sizeof(provides_star) == sizeof(provides_operator_star(*t))
30:        ;
31:    };
32: }//namespace guard
33:
34: template<typename T>
35: struct provides_user_defined_operator_star : public
36:     boost::mpl::and_<
37:         boost::mpl::not_<boost::is_fundamental<T> >,
38:         guard::provides_user_defined_operator_star_impl<T>
39:     >
40: {};
41:
42: template<typename T>
43: struct is_dereferenceable : public
44:     boost::mpl::or_<
45:         boost::is_pointer<T>,
46:         boost::is_array<T>,
47:         provides_user_defined_operator_star<T>
48:     >
49: {};

```

Figure 3.9: Implementation of the `is_dereferenceable` metafunction.

```

1: //TRUE and FALSE metafunction
2:     struct TRUE {
3:         typedef TRUE type;
4:         const static bool value = true;
5:     };
6:     struct FALSE {
7:         typedef FALSE type;
8:         const static bool value = false;
9:     };
10:
11: //AND metafunction with short-circuit evaluation
12: //forward declaration of helper classes
13:     template <bool b1, typename b2> struct AND2_impl;
14:     template <bool b>                 struct AND1_impl;
15:
16: //short-circuit AND
17:     template <typename b1, typename b2>
18:     struct AND : AND2_impl<b1::value, b2>{};
19:
20: //implementation metafunctions
21: //metafunction to evaluate first AND argument
22:     template <typename b2>
23:     struct AND2_impl<false, b2>{
24:         const static bool value = false;
25:     };
26:
27:     template <typename b2>
28:     struct AND2_impl<true, b2> : AND1_impl<b2::value>{};
29:
30: //metafunction to evaluate second AND argument
31:
32:     template <>
33:     struct AND1_impl<true>{
34:         const static bool value = true;
35:     };
36:
37:     template <>
38:     struct AND1_impl<false>{
39:         const static bool value = false;
40:     };
41:
42:
43:

```

Figure 3.10: Implementation of a short-circuit compile-time boolean AND metafunction.

```

1: //testing AND's truth table
2:   BOOST_STATIC_ASSERT( AND< TRUE, TRUE >::value == true ); //compiles
3:   BOOST_STATIC_ASSERT( AND< FALSE, FALSE >::value == false ); //compiles
4:   BOOST_STATIC_ASSERT( AND< TRUE, FALSE >::value == false ); //compiles
5:   BOOST_STATIC_ASSERT( AND< FALSE, TRUE >::value == false ); //compiles
6:
7: //metafunction to force a default construction of its argument
8:   template <typename T>
9:   struct default_construct{
10:       const static T value = T();
11:   };
12:
13: //class that cannot be default constructed
14:   class cannot_default_construct{
15:       private:
16:           cannot_default_construct();
17:   };
18:
19: //trying to default construct
20:   default_construct<int> d1; //compiles
21:   default_construct<cannot_default_construct> d2; //does not compile
22:
23: //short-circuit evaluation in action
24: bool b1 = AND< FALSE, default_construct<cannot_default_construct> >::value;
25: //compiles, b1 == false
26: bool b2 = AND< TRUE, default_construct<cannot_default_construct> >::value;
27: //does not compile

```

Figure 3.11: Example usage of the AND metafunction.

This allows the problem of memory structure detection to be reduced to computing the dimension of an arbitrary C++ type.

Unfortunately, there is no direct language support for computing the dimension of a C++ type. Given the constraint of performing the dimension computation at compile-time, we have no choice but to use template metaprogramming. Figure 3.6 shows the implementation of the `Dim` metafunction used in SACLIB 3.0 to compute the dimension of a type. Let  $t$  be a C++ variable of type  $T$ . If  $*t$  is a valid expression, let  $T_d$  be its return type. The `Dim` templates (lines 17-21, 23-27) compute the dimension of  $T$  directly from the recursive definition of dimension: if  $*t$  is a valid expression, the dimension of  $T$  is 1 + the dimension of  $T_d$ ; otherwise the dimension of  $T$  is 0. The `is_dereferenceable` metafunction (see Figure 3.9) returns true if  $*t$  is a valid expression and false otherwise. Its implementation will be discussed after the rest of the implementation of `Dim` has been considered.

Unfortunately, there is no way to reliably compute  $T_d$  for an arbitrary type. For a user-defined type, the expression  $*t$  will invoke the function `T::operator*` or `operator*(T)`. C++ does not provide a mechanism for obtaining the return type of a function invocation. Because of this limitation, the best standards conforming way to compute  $T_d$  is to use the `std::iterator_traits` template [12, 94, 191]. The C++ standard requires specializations of `std::iterator_traits` for all pointers and STL iterators. For each of the required `std::iterator_traits<T>` specializations, the nested type `std::iterator_traits<T>::value_type` is required to be the type we have defined as  $T_d$ . Because user-defined iterators can only be used with the STL if there is a `std::iterator_traits` specialization, this allows computation of  $T_d$  for all iterators usable with the STL.

There are no `std::iterator_traits` specializations for arrays. However, type conversion makes arrays dereferenceable. Handling arrays requires the two `Dim` specializations (lines 17-21, 23-27). The template on lines 23-27 is specialized to handle array types. Let  $a$  be a C++ variable of an array type  $A$  with elements of type  $E$ . The expression  $*a$  is valid, but only because  $a$  will decay to a pointer of type  $E^*$  through implicit type conversion. There is a specialization defined for `std::iterator_traits<E*>`, but there is no specialization for `std::iterator_traits<A>`. The pointer decay that occurs for  $*a$  is not considered when matching template specializations. As a result, the dimension of array types must be computed as a special case. Note that line 25 calls `Dim<E>` rather than attempting to compute the dimension directly from  $A$ . This is necessary because it is possible that  $E$  could be an array.

The template on lines 17-21 handles the general case. The only purpose of this template is

to use the template matching rules to ensure that its argument is not an array before forwarding the argument to `Dim_non_array` (see Figure 3.7) on line 19. The `Dim_non_array` template is declared on lines 24-28. There are two template arguments: the type `T` to compute the dimension of, and a boolean to identify if `T` is dereferenceable. The dereferenceable argument is given a default value of `is_dereferenceable<T>::value`. The implementation of `Dim_non_array` depends on users not changing the default value of dereferenceable. It also depends on `T` not being a built-in array type. These pre-conditions are enforced by using the `BOOST_STATIC_ASSERT` [126] macro, `is_dereferenceable`, and `boost::is_array` [4] on lines 32-35 and 48-55. The `BOOST_STATIC_ASSERT` macro forces a compile-time error if its argument is not convertible to true. `boost::is_array<T>::value` is true if `T` is a built-in array and false otherwise.

The base case is handled in the template defined on lines 46-54. The partial specialization on lines 46-47 ensures that the template will only be selected when variables of type `T` cannot be dereferenced. By definition, the dimension of such a type is zero (line 53). All other cases are handled by the template definition on lines 37-44. It computes the dimension as 1 plus the dimension of the type returned by dereferencing a variable of type `T`.

The `g++` `typeof` extension can be used to significantly improve the implementation of `Dim`. Figure 3.8 shows an implementation that works for any `T` without the need for an `std::iterator_traits<T>` specialization. On lines 8-14, the `iterator_value_type` metafunction is defined. The `g++` `typeof` extension evaluates to the type of its argument and can be used any place a type is required by the C++ standard. On line 13, `typeof(*t)` evaluates to the type of the object returned by dereferencing `t`. The only restriction on `T` is that it must be dereferenceable for `typeof(*t)` to compile. This precondition is enforced on line 10 using `BOOST_STATIC_ASSERT` and `is_dereferenceable`. Using `iterator_value_type`, `Dim` is implemented on lines 23-36 directly from the definition of the dimension of a type. There is one template specialization on lines 30-32 that sets value to 0 when `T` is not dereferenceable and another on lines 26-29 that sets value to `1+Dim<T>::value` when `T` is dereferenceable.

In addition to removing the dependence on a `std::iterator_traits` specialization, using `typeof` significantly simplifies the implementation of the `Dim` compared to the `std::iterator_traits` based implementation in Figures 3.6 and 3.7. There are no template specializations to handle the case where `T` is an array because `iterator_value_type` now handles the implicit decay of arrays to pointers. All of the preconditions of the `std::iterator_traits` are no longer required. This makes `Dim` easier to use for clients, simplifies the documentation, and removes the need for the checking

of preconditions with `BOOST_STATIC_ASSERT`. Although these benefits require the use of the `typeof` extension, there are plans to extend the next C++ standard to provide functionality similar to the `typeof` extension provided by g++.

The `is_dereferenceable` metafunction implementation (see Figure 3.9) is much more demanding than the metafunctions we have considered so far. There was some discussion of a similar metafunction on the Boost mailing list. However, it was never added to `Boost.TypeTraits`, and we needed to implement our own. The general idea of the implementation strategy is straightforward. A programmer can easily test if a given type is dereferenceable by simply compiling `T t; *t`. If `T` is dereferenceable, `*t` is a valid expression and the code will compile. If `T` is not dereferenceable, `*t` will produce a compiler error. Unfortunately, the compiler error will simply halt the compilation at the end of the translation unit. The compilation error does not provide any way to detect `*t` is invalid and continue compilation.

Implementing `is_dereferenceable` hinges on replacing the compiler error produced when `*t` is invalid to a compile-time diagnostic that can both indicate `*t` is invalid and allow compilation to continue. Fortunately, `operator*` can be overloaded at namespace scope. On line 16 an `operator*(conversion_from_T)` is declared. On line 12, a templated copy constructor allows `conversion_from_T` to be constructed from any type. This accomplishes two things. First, `*t` becomes a valid expression for any user-defined type. This is because for any user-defined type `T`, either `T` already provides support for `*t` or `t` can be converted to a `conversion_from_T` and then dereferenced. Second, it is possible to tell at compile-time if `*t` would have compiled without the declaration of `operator*(conversion_from_T)` because of the use of `no_operator_star` as the return type.

There are several subtleties to consider. The metafunction `provides_user_defined_operator_star_impl` (lines 22-32) uses the presence of a compile-time diagnostic to determine if `*t` is a valid expression for a user-defined type. The metafunction will not work for non-dereferenceable built in types (this will be discussed after user-defined types are fully considered), and uses `BOOST_STATIC_ASSERT` and `boost::is_fundamental` (line 24) to prevent the metafunction from being used with non-dereferenceable built-in types. The `boost::is_fundamental` metafunction returns true if its argument is a built in integral, floating point, or void type. It returns false otherwise. Lines 26 and 27 use the metafunction `boost::remove_cv` to remove any cv qualifiers of the argument to `provides_user_defined_operator_star_impl`. This removes the need to consider cv qualifiers as separate cases. Line 28 declares a variable of type `no_cv_type`. Line 30 calls `provides_operator_star(*t)` to determine if the type is dereferenceable. If `T` is a type that does provide an `operator*`, the

return type of `*t` will be a type other than `guard::no_operator_star`, and during overload resolution it will get converted to a `conversion_from_T` via the constructor on line 12 and the `provides_operator_star(conversion_from_T)` overload on line 20 will be selected. As this overload has a return type of `provides_operator_star`, line 30 will evaluate to true. If `T` is a type that does not provide an `operator*`, the return type of `*t` will be converted to a `conversion_from_T` (also via the constructor on line 12) and the `operator*` on line 16 will be selected. As this has a return type of `no_operator_star`, the `provides_operator_star(no_operator_star)` overload on line 19 will be selected. Since it has a return type of `used_our_operator_star`, the condition on line 30 will evaluate to false. Because the condition on line 30 relies on `sizeof` returning different sizes for `no_operator_star`, `provides_star`, and `used_our_star`, a `BOOST_STATIC_ASSERT` is used on lines 7-8 to ensure the compiler has not added any padding to these three types. If padding has been added, the types will need to have explicit filler added to ensure their sizes are different. The absence of a definition for the variable `t` declared on line 27 is not an error because the type of `t` is all that is needed by `provides_user_defined_operator_star`.

Providing `operator*(conversion_from_T)` means `*t` is a valid expression for all variables in a translation unit containing this declaration of `operator*`. This means that code like `int i; *i;` will be syntactically valid. Obviously this cannot be allowed to compile. This is accomplished by not defining `operator*(conversion_from_T)`. In the previous example, `*i` will be syntactically valid, but will not compile. Unfortunately, the error message from the compiler will not be particularly informative because it will indicate that a definition for `operator*(conversion_from_T)` is not available. This can be mitigated by placing comments by the declaration explaining what the error means.

When `T` already allows `*t`, care has to be taken to ensure that `operator*(conversion_from_T)` does not become a better match for `*t` than an existing `operator*` defined for `T`. When `T` is a user-defined type that defines `T::operator*`, there is no danger that `operator*(conversion_from_T)` will be selected because the member function will always be selected before an operator at namespace scope. When `operator*(T)` is defined at namespace scope, it will always be selected over `operator*(conversion_from_T)` because `operator*(T)` does not require type conversion. Note that this is true regardless of the namespace that `operator*(T)` is defined in. Let `X` be a C++ type and let `X` be implicitly constructible from `T`. When `operator*(X)` is defined at namespace scope, both `operator*(X)` and `operator*(conversion_from_T)` are possible matches for `*t`. Argument dependent lookup requires the compiler to select the `operator*` from the same names-

pace that `T` is defined in. If `operator*(X)` was intended to provide `operator*` for `T`, then it is part of the public interface of `T`. Because of this, `operator*(X)` would be placed in the same namespace as `T`. By placing `operator*(conversion_from_T)` in its own namespace (line 1), the compiler will select `operator*(X)` over `operator*(conversion_from_T)`. The end result is that `operator*(conversion_from_T)` will never be called when a user-defined `T` that provides an `operator*` as part of its public interface.

Unfortunately, the matching rules are different for built-in types. For a built-in type `T`, the compiler will never consider a function when evaluating `*t`. This means that `*t` will only ever be valid for a built-in `T` if `T` is a pointer or an array. For all other types, `*t` will produce a compiler error. The `provides_user_defined_operator_star` metafunction (lines 35-41) uses the short-circuit metafunction `boost::mpl::and_ [2]` (described below) to solve this problem. The public inheritance used on line 36 is known as metafunction forwarding [2]. It works because `boost::mpl::and_` is itself a metafunction that will provide the required `type` typedef and `value` member. The `boost::mpl::and_` metafunction returns true if all metafunctions passed to it return true. However, it evaluates arguments left to right and stops argument processing as soon as it encounters an argument that evaluates to false. This short circuit behavior allows processing to stop when `T` is a fundamental type (because the first argument `boost::mpl::not_<boost::is_fundamental<T> >` will be false). This means that `provides_user_defined_star_impl` will never be evaluated when `T` is a fundamental type. This avoids the problem of the `*t` on line 30 ever being compiled when `T` is a built-in type that is not dereferenceable.

Compile-time short-circuit logical operators, `boost::mpl::and_` and `boost::pl::or_ [2]`, are a critical piece of the `is_dereferenceable` implementation, and are worth considering in more detail. Figure 3.10 has an implementation of a two argument short-circuit AND metafunction. Lines 2-10 define the metafunctions `TRUE` and `FALSE`. These metafunctions are used to represent `true` and `false` at compile-time. They will be used in Figure 3.11. Lines 17-18 define a short-circuit AND that takes two arguments. The arguments `b1` and `b2` are types. This allows `AND` to operate on metafunctions. `AND` is implemented using metafunction forwarding to `AND2_impl`. `AND2_impl` is called with a first argument of `b1::value` and a second argument of `b2`. Using `b1::value` requires access to the value member of `b1`. This access forces the compiler to instantiate `b1` if it is a template. However, the second argument of `b2` simply names a type. If `b1` is a template, the compiler does not need to compile `b2` as a result of the call to `AND2_impl`.

Lines 22-25 and 27-28 define two partial specializations for `AND2_impl`. The first argument of both

AND specializations is an integral constant and the second is a type. This is because AND evaluates `b1::value`, which produces an integral constant. The specialization on lines 22-25 handles the case where the first argument is `false`. In this case, line 24 sets the value of `AND2_impl` to `false`, which causes AND to evaluate to `false`. Because `b2::value` is never referenced, when `b2` is a template the compiler does not instantiate `b2`. The specialization on lines 27-28 handles the case where the first argument to `AND2_impl` is `true`. In this case, `b2::value` must be evaluated. When `b2` is a template the compiler is required to instantiate `b2` in order to obtain `b2::value`. On line 28, `AND1_impl` is called via metafunction forwarding with an argument of `b2::value`. `AND1_impl` takes a single integral constant argument, and the templates on lines 32-35 and 38-40 simply set the value of `AND1_impl` to the value of the argument.

Implementing AND and its supporting templates to defer argument evaluation for as long as possible has two benefits. The first benefit is efficiency. If `b2::value` is not needed to determine `AND::value`, the compiler will not perform the computations needed to evaluate `b2::value`. The second benefit occurs when AND is called with a template argument that would make the instantiation of `b2` a compile error. In this case, the first argument of AND can be used to prevent the instantiation of `b2`.

In Figure 3.11 the usage of AND is demonstrated. Lines 2-5 test that AND produces the correct truth table for the logical and of two boolean values using the metafunctions `TRUE` and `FALSE` (defined in Figure 3.10). Lines 7-27 demonstrate how AND can be used to prevent the instantiation of templates that cannot be compiled. On lines 8-11 the metafunction `default_construct` is defined. Line 4 forces the default construction of an instance of type `T` with the assignment `value = T()`. Lines 13-17 define the class `cannot_default_construct`. Because line 16 makes the default constructor private and `cannot_default_construct` has no friends, it is a compile error to attempt to default construct an instance of `cannot_default_construct`. Line 20 instantiates `default_construct` with an `int`. This compiles successfully, because `int` has a default constructor. Line 21 attempts to instantiate `default_construct` with `cannot_default_construct`. This will not compile because of the definition of `cannot_default_construct`. Lines 24 and 26 both call AND with a second argument of `default_construct<cannot_default_construct>`. When AND is called on line 24, the first argument of AND is `FALSE`. Because of the short-circuit evaluation used in AND, line 24 compiles because `default_construct<cannot_default_construct>` is never instantiated. Line 26 does not compile, because the first argument of `TRUE` forces the instantiation of `default_construct<cannot_default_construct>`, which produces a compiler error.

The implementation of `boost::mpl::and_` and `boost::mpl::or_` follow a similar implementation strategy to `AND`, however they use preprocessor metaprogramming to allow arities of greater than 2.

With all of these pieces, it is now possible to implement the general `is_dereferenceable` metafunction (Figure 3.9, lines 42-49). It employs metafunction forwarding using `boost::mpl::or_`. `boost::mpl::or_` provides short-circuit logical evaluation similar to the `||` boolean operator. The `is_dereferenceable` metafunction returns `true` if its argument is a pointer (line 45, `boost::is_pointer` [4] returns true), an array (line 46, `boost::is_array` returns true), or there is a user-defined `operator*` (line 47, `provides_user_defined_operator_star` returns true). Notice that at this point, the code to implement `is_dereferenceable` has become relatively straightforward to read. This is typical of metafunctions. Lower level metafunctions providing functionality like type selection and conditional logic typically have implementations that are difficult to follow. This is because lower level metafunctions such as `provides_user_defined_operator_star` and `AND` can only be implemented using language features such as template specialization, overload resolution, and argument dependent lookup (ADL). These language features are among the most complicated and least used aspects of C++. However, once these language features are used to provide metafunctions that implement higher level abstractions such as `boost::mpl::or_` and `is_dereferenceable`, it becomes relatively straightforward to implement and understand metafunctions.

### 3.3.4 The `rec_fixed_itr` and `struct_protect_itr`

Using the metafunctions discussed earlier in the chapter, we have implemented the `rec_fixed_itr` as a model of the Recursively Fixed Iterator concept and the `struct_protect_itr` as a model of the Structure Protecting Iterator concept. Because of the similarity between the concepts they model, `rec_fixed_itr` and `struct_protect_itr` have similar implementations. `struct_protect_itr` is the more demanding implementation, and we will discuss it here to demonstrate the implementation techniques used for the iterators. The full implementation contains 350 lines of metafunctions, 600 lines of iterator code, and 1100 lines of unit tests.

Figure 3.12 has a listing of a few representative methods to illustrate the implementation techniques required to implement `struct_protect_itr`. Lines 1-5 declare the `struct_protect_itr` class. It is templated on the `AliasType` (line 2). This is the type that the `struct_protect_itr`

```

1: template<
2:     typename AliasType,
3:     bool     nested_spi=false
4: >
5: class struct_protect_itr{
6:     BOOST_STATIC_ASSERT( Dim<AliasType>::value >= 1);
7:
8:     const static unsigned int dim = Dim<AliasType>::value;
9:
10:    //iterator typedefs
11:    typedef
12:        typename std::iterator_traits<AliasType>::value_type
13:        value_type;
14:
15:    typedef
16:        typename if_type<
17:            1 == dim,
18:            typename std::iterator_traits<AliasType>::reference,
19:            struct_protect_itr<value_type,true>
20:        >::type
21:        reference
22:    ;
23:
24:    //ctors
25:    struct_protect_itr(AliasType p):itr_(p){}
26:
27:    //example operators
28:    inline reference operator*(){
29:        return *itr_;
30:    }//operator*
31:
32:    struct_protect_iter& operator+=(const difference_type i){
33:        BOOST_STATIC_ASSERT(false==nested_spi);
34:        itr_ += i;
35:        return *this;
36:    }//operator+=
37:
38:    private:
39:        AliasType itr_;
40:
41: };//struct_protect_itr

```

Figure 3.12: Outline of the `struct_protect_itr` implementation techniques.

will be a Structure Alias for. The `AliasType` is used to declare the private member `itr_` (line 39). All member functions referencing the aliased structure are implemented using `itr_`.

The boolean template argument `nested_spi` (line 3) determines what concept the `struct_protect_itr` instantiation must model. When `nested_spi` is false, it indicates that the type `struct_protect_itr<AliasType, false>` is not used as the return type for the methods of any `struct_protect_itr` instantiation and `struct_protect_itr<AliasType, false>` must model the Structure Protecting Iterator concept. When `nested_spi` is true it indicates that `struct_protect_itr<AliasType, true>` is used as a return type for the methods of another `struct_protect_itr` instantiation and `struct_protect_itr<AliasType, true>` must model the Recursively Fixed Iterator concept.

There are two key implementation techniques worth discussing. The first is the return type computation for `operator*` (lines 12-23, 28). The Structure Protecting Iterator requires it be a model of the Recursively Fixed Iterator when the dimension of `itr_` is greater than 1 and a reference to `base(itr_)` when the dimension of `itr_` is 1. On lines 11-12 `std::iterator_traits` is used to obtain the `value_type` of `AliasType`. The selection of the proper return type is accomplished using the `if_type` metafunction to compute the reference type (lines 15-22). When the first argument to `if_type` is true, `if_type` returns its second argument. Otherwise `if_type` returns its third argument. When `1 == dim`, `std::iterator_traits<AliasType>::reference` is used to compute a reference to `base(itr_)` (line 18). When `1 != dim`, `struct_protect_itr<value_type, true>`, a model of the Recursively fixed Iterator concept, is returned.

The second implementation technique is the use of `nested_spi` to cause the `struct_protect_itr` implementation to model a Recursively Fixed Iterator. This requires disabling methods required by the Structure Protecting Iterator concept when `nested_spi` is true. `operator+=` (lines 32-36) is an example of a method that needs to be disabled. `BOOST_STATIC_ASSERT` is used on line 33 to prevent compilation of the method when `nested_spi` is true. Because `struct_protect_itr` is a template, its methods are only instantiated when used by a client. When clients use the `struct_protect_itr` correctly, when `nested_spi` is true they will never call `operator+=` and the `BOOST_STATIC_ASSERT` will not be evaluated. However, if a client incorrectly calls `operator+=` when `nested_spi` is true, the method will be instantiated and the `BOOST_STATIC_ASSERT` will fail. This technique is used to disable all of the self-modifying pointer operations required by the Structure Protecting Iterator concept.

### 3.3.5 Preventing Memory Leaks

SACLIB 2.1 contained several memory leaks that prevented long running computations [116]. These memory leaks are prevented by the SACLIB 3.0 memory management system. In Figure 3.13 we show the SACLIB 2.1 routine for medium modulus polynomial distinct-degree factorization, `MMPDDF`. This routine contains a memory leak. We only show lines relevant to the memory leak. On line 16 `Q` is set to the return value of `MMAPBM`. The `MMAPBM` routine allocates memory for a two dimensional array and returns a `Word**` pointer to the allocated memory. The allocated memory contains `MAPDEG(A)` entries. On line 17, `n` is computed as `MAPDEG(A)-1`. The value of `n` is supposed to be the number of elements pointed to by `Q` and has been computed incorrectly. On line 42, `FREEMATRIX(Q,n)` is called. Because `n` is one less than it should be, the last element of `Q` is leaked. This leak is caused because `MMPDDF` is directly manipulating the memory structure of `Q` via the call to `FREEMATRIX`.

In Figure 3.14 we show the SACLIB 3.0 implementation of `MMPDDF`. This version has no memory leaks. On lines 4-7 memory is explicitly managed by instances of the `array` and `array2d` classes. The memory managed by the classes is referred to using a `rec_fixed_itr` on lines 10, 11, 13, 14, and 21. This prevents `MMPDDF` from manipulating any of the memory structure and guarantees that the memory will be deleted when `MMPDDF` returns.

## 3.4 Performance Testing

In order to obtain the most accurate timing possible, hardware performance counters were used to measure the number of CPU cycles required for different methods of memory management. All measurements were obtained on a 3.0 GHz Pentium 4 (1024 KB L2 cache) running Fedora Core 2. The Performance Counter API (PAPI) [90] version 3.0.7 was used to collect all measurements. The single processor 2.6.5 kernel shipped with Fedora Core 2 was patched with the `perfctr-2.6.x` patch distributed with PAPI. Tests were compiled with `icc` [92] version 9.0 and `gcc` [68] version 3.3.3. All tests were compiled with the flags “-O3 -DNDEBUG”.

Our experiments compared the performance of a native C++ pointer to our `rec_fixed_itr` and `struct_protect_itr` implementations when they were used as iterators into a native C++ array allocated on the heap. We measured both the time to construct the two iterators and the time it took to read and write every element of a 500-element array through the iterators. Each experiment was conducted 10 million times. Cycle counts were obtained with the function `PAPI_get_virt_cyc`.

```

1: /*=====
2:                               L <- MMPDDF(p,A)
3:
4: Medium modulus polynomial distinct-degree factorization.
5:
6: =====*/
7: #include "saclib.h"
8:
9: Word MMPDDF(p,Ap)
10:     Word p,Ap;
11: {
12:     Word *A,**Q,n,*B,*Bp,*W,i,j,*C,*Cp,*D,L,Lp,L1,A1,b,d,e,k,w1;
13:
14: Step1: /* Initialize. */
15:     A = MAPFMUP(Ap);
16:     Q = MMAPBM(p,A);
17:     n = MAPDEG(A)-1;
18:     B = MAPGET(n);
19:     Bp = GETARRAY(n+1);
20:     W = MAPGET(n);
21:     for (i = 0; i <= n; i++)
22:         MAPCF(B,i) = MATELT(Q,1,i);
23:     d = n;
24:     while (MAPCF(B,d) == 0)
25:         d--;
26:     MAPDEG(B) = d;
27:     C = MAPCOPY(A);
28:     L = NIL;
29:     k = 1;
30:
31: Step2: /* Compute A_k. */
32:     /* Code for Step2 not shown */
33:
34: Step3: /* Convert to lists. */
35:     /* Code for Step3 not shown */
36:
37: Step4: /* Free arrays. */
38:     MAPFREE(A);
39:     MAPFREE(B);
40:     FREEARRAY(Bp);
41:     MAPFREE(W);
42:     FREEMATRIX(Q,n);
43:
44: Return: /* Prepare for return. */
45:     return(L);
46: }

```

Figure 3.13: The SACLIB 2.1 version of MMPDDF contains a memory leak that results from incorrect manipulation of the memory structure of a matrix. These kinds of errors are not possible with the new SACLIB 3.0 memory management system. See Figure 3.14 for the SACLIB 3.0 version of this code.

```

1: Word MMPDDF(Word p, Word Ap)
2: {
3:     Word n,i,j,L,Lp,L1,A1,b,d,e,k,w1;
4:     array<Word>   A_storage, B_storage,
5:                 C_storage, Cp_storage,
6:                 D_storage, W_Storage;
7:     array2d<Word> Q_storage;
8:
9: Step1: /* Initialize. */
10:    rec_fixed_itr<Word*> A = MAPFMUP(Ap, A_storage);
11:    rec_fixed_itr<Word**> Q = MMAPBM(p,A,Q_storage);
12:    n = MAPDEG(A)-1;
13:    rec_fixed_itr<Word*> B = MAPGET(n, B_storage);
14:    rec_fixed_itr<Word*> W = MAPGET(n, W_storage);
15:    for (i = 0; i <= n; i++)
16:        MAPCF(B,i) = MATELT(Q,1,i);
17:    d = n;
18:    while (MAPCF(B,d) == 0)
19:        d--;
20:    MAPDEG(B) = d;
21:    rec_fixed_itr<Word*> C = MAPCOPY(A, C_storage);
22:    L = NIL;
23:    k = 1;
24:
25: Step2: /* Compute A_k. */
26:    /* Code for Step2 not shown */
27:
28: Step3: /* Convert to lists. */
29:    /* Code for Step3 not shown */
30:
31: /* There is no Step4, as memory is not directly managed */
32:
33: Return: /* Prepare for return. */
34:    return(L);
35: }

```

Figure 3.14: The SACLIB 3.0 implementation of MMPDDF. The use of the SACLIB 3.0 memory management removes the leak shown in the SACLIB 2.1 implementation in Figure 3.13.

	C++ Pointer		rec_fixed_itr	
	Cycles (min)	Cycles (Mode)	Cycles (Min)	Cycles (Mode)
construction	4,719	25,953	4,577	25,174
read	3,835,910	4,362,820	3,763,730	4,070,560
write	3,837,420	4,969,780	3,839,130	4,967,000

Table 3.1: Minimum and mode computing times (cycle counts) on a Pentium 4 for a native C++ pointer and a `rec_fixed_itr` for construction, writing to a 500 element array, and reading from a 500 element array.

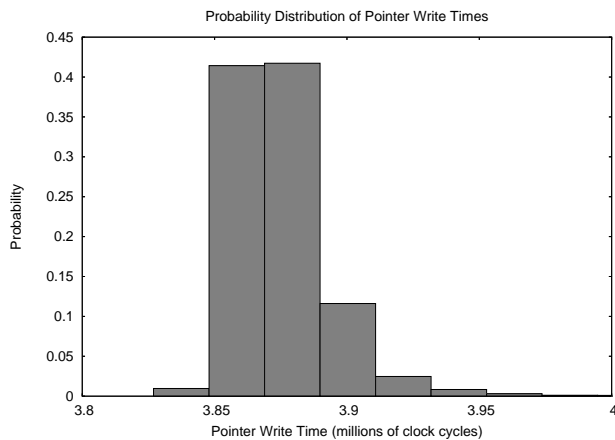


Figure 3.15: Probability of observing a given execution time (cycles) for writing to every element of a 500 element array via a C++ native pointer.

Histograms were constructed using 500 bins. Mins and modes were approximated from the mid-points of the histogram bins.

For all three tests on both iterators, the timing distributions were tightly peaked near the minimum computing time; Figure 3.15 shows a representative distribution. Data is only shown for the `rec_fixed_itr` compiled with `icc`; both the `gcc` and `struct_protect_itr` results are similar. Inspection of the assembly produced by the `icc` compiler shows that identical memory operations are being produced for all three test cases regardless of which iterator is used; Figure 3.16 shows a representative listing. Table 3.1 shows a summary of the computing times. Both the shape of the timing distributions and the observed times are consistent with the fact that the `icc` compiler scheduled identical memory operations for both iterators.

	C++ Pointer	rec_fixed_itr
C++ code	<pre>int main(int argc, char** argv){     int* i = new int;     use_memory(*i); } //main</pre>	<pre>int main(int argc, char** argv){     int* i = new int;     rec_fixed_itr&lt;int*&gt; rfi(i);     use_memory(*rfi); } //main</pre>
Assembly code produced for call to use_memory	<pre>pushl (%eax) push \$dev_null call _ZNSolsEi</pre>	<pre>pushl (%eax) push \$dev_null call _ZNSolsEi</pre>

Figure 3.16: Example programs for reading memory through a C++ pointer and a `rec_fixed_itr`. The `use_memory` function writes the value to `/dev/null`. The `icc` compiler generates identical assembly code for the iterators to write to `/dev/null`.

### 3.4.1 Writing Code for the Optimizer

At the machine code level, programmers desire the same functionality from the `rec_fixed_itr` and `struct_protect_itr` classes as they do from built-in pointers. This expectation is based on the fact that, at its core, the `rec_fixed_itr` and `struct_protect_itr` must perform the same pointer operations required of a pointer. Internally, all of the pointer operators of the `iterators` are implemented using pointers.

Because of this, any performance overhead using the `rec_fixed_itr` or `struct_protect_itr` class must be a result of the compiler's inability to determine that the machine code required for a `rec_fixed_itr` and `struct_protect_itr` is identical to that required for a pointer. The `rec_fixed_itr` and `struct_protect_itr` were carefully designed and implemented to give the compiler every opportunity to determine this equivalence.

There are three critical aspects to the iterator implementations that allow for optimizing compilers to produce efficient machine code:

1. Template metaprogramming is only used for return type computations,
2. All operators are implemented as inline forwarding functions, and
3. Wherever possible, operator arguments are taken as `const` qualified references.

By restricting template metaprogramming to return type computations, there will be no code generated by the template metaprograms. Because the type computation must be completed before the compiler knows the type, code generation cannot begin until the type computation has completed. This precludes the situation of code being passed to the optimizer from a code generating template metaprogram. While optimizers frequently can remove the overhead from template generated code,

some generators [27] produce code from which optimizers cannot remove all unessential overhead. Type-only metafunctions preclude this problem.

Implementing the operators as inline and using `const` qualified references work together to help the optimizer. By making all functions inline, the compiler is allowed to see more of the calling context of the operator. By making the reference `const` qualified, the compiler is assured writes will not occur to the argument. This saves the compiler from the task of determining this information from context. Giving the compiler access to this information makes it easier for the compiler to be able to prevent extraneous overhead that could otherwise be introduced through the use of operator overloading.

This design makes performance hinge upon optimizations that modern compilers are reasonably competent at performing. The performance measurements reported in Section 3.4 show that modern compilers are indeed capable of removing the unessential overhead.

### 3.4.2 Compilation Overhead

The implementation of the `rec_fixed_itr` and `struct_protect_itr` avoid run-time overhead by moving all computations relating to memory safety to template metaprograms. Although this does not result in run-time overhead, it does require many more template instantiations during compilation. We compiled both SACLIB 2.1, the C implementation, and SACLIB 3.0, the C++ implementation with our iterators, on an Ubuntu Linux 8 system with two 1.7 GHz AMD Opteron 244 processors and 4 GB of memory. On this system SACLIB 2.1 compiles in three minutes. SACLIB 3.0 requires 10 minutes. These compile-times are to build both a debug and optimized build of SACLIB. Although this is about a factor of three increasing in compilation time when using the iterators, the wall time is low enough that this is not a problem.

## 3.5 Comparison to existing Work

### 3.5.1 STL Iterator concepts

The STL defines [12] the following iterator concepts: Trivial Iterator, Input Iterator, Output Iterator, Bidirectional Iterator, and Random Access Iterator. All algorithms in the STL are specified in terms of these iterator concepts. The iterator concepts are arranged in the refinement hierarchy depicted in Figure 3.17.

As can be seen in Figure 3.17, the Recursively Fixed Iterator does not fit anywhere in the STL

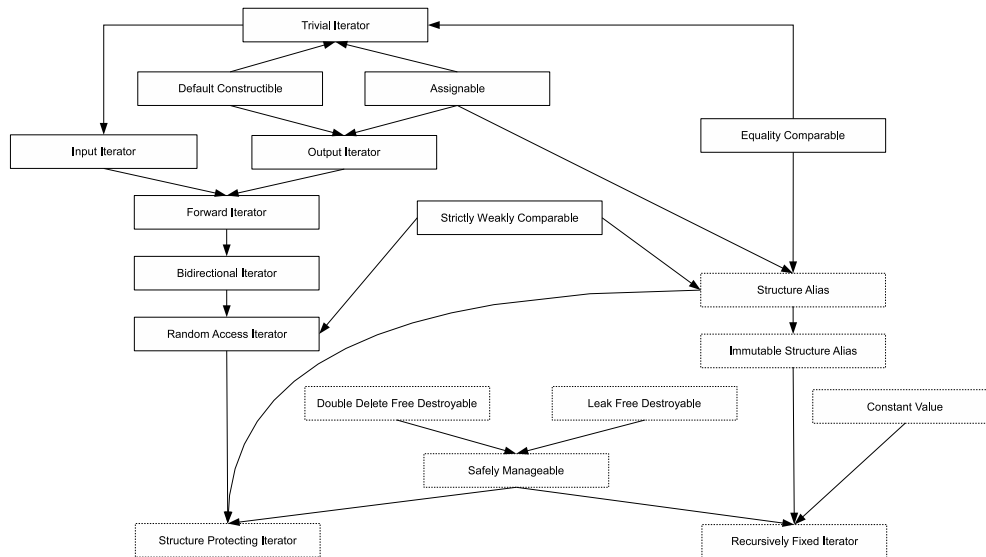


Figure 3.17: Concept refinement hierarchy for the STL iterators, the Recursively Fixed Iterator, and the Structure Protecting Iterator. Boxes denote concepts. An arrow from box A to box B means that concept B is a refinement of concept A. Dashed boxes are concepts we present in Section 3.2.

iterator concept hierarchy. This is a direct consequence of the fact that models of the Recursively Fixed Iterator concept must keep their structure constant. Keeping the structure constant requires that the Recursively Fixed Iterator concept is not a model of the Assignable concept. Without being a model of Assignable, the Recursively Fixed Iterator also cannot be a model of Default Constructible because a default-constructed model of the Recursively Fixed Iterator could never be assigned a meaningful value. However, Structure Protecting Iterators can be used when Random Access Iterators are required.

### 3.5.2 Multi-Dimensional Containers and Smart Pointers

Existing multi-dimensional containers [66, 200, 107, `std::vector`] and smart pointers [6, 38, 7, 104] generally have several of the following limitations when being used in code not explicitly designed to use them for memory management: 1) they impose storage requirements that degrade program efficiency, 2) they impose ownership semantics that are incompatible with other methods of ownership, 3) they do not support the same dereferencing syntax as multi-dimensional pointers, 4) they do not support the same element access syntax as multi-dimensional pointers, and 5) they impose a

run-time abstraction penalty.

These limitations are not, per se, design or implementation deficiencies in the multi-dimensional containers and smart pointers. They stem from the fact that the designs of these components require specific programming idioms to gain their benefits. In new development, these constraints do not pose much of a burden and are well worth the benefits gained from using the components. However, in existing systems with firmly established notions of memory ownership and heavy use of pointers, these components require extensive modifications to be used. SACLIB is a perfect example of a library where such invasive changes would be difficult. It utilizes irregularly shaped and sparse multi-dimensional memory, frequently exploits these memory layouts for run-time efficiency, uses pointers, and has its own clearly defined memory ownership policies. For libraries like this, the Recursively Fixed Iterator provides a much smoother upgrade path to modern C++ memory management benefits.

### 3.5.3 The C++0x Standard

The next version of the C++ standard, referred to as C++0x until it is finalized, will include direct language support for concepts [79] and static assertions [114]. The new static assertions will allow for a much cleaner implementation of the `rec_fixed_itr` and `struct_protect_itr`. Both iterator implementation currently make extensive use of the `BOOST_STATIC_ASSERT` [126] macro to check that templates are instantiated with valid arguments. Although able to check assertions at compile-time, the `BOOST_STATIC_ASSERT` error messages are difficult for non-expert programmers to understand. They generally have error messages about the application of `sizeof` to an incomplete type. If a programmer checks the line of code referenced in the error message it will be on a line that contains a `BOOST_STATIC_ASSERT`. Because the new static assertions will be a direct part of the language, it will be possible for the compiler to give much clearer error messages.

ConceptGCC [75] is an extension to the the gcc compiler [68] that provides a prototype of the concept support that will be found in C++0x. Using these features it is possible to directly define a concept in C++ and require that a template argument model the concept. It would be possible to formulate many of the existing `BOOST_STATIC_ASSERT` checks as concepts and use this mechanism to validate template arguments. We have not attempted to use the current ConceptGCC implementation to try redoing template argument validation on our iterator implementations. As of the writing of this dissertation, the authors of ConceptGCC consider their implementation to be “far from being production-quality, and is only really good for getting a basic feel for how concepts

will work. ” [76] The implementation is not to the point of being able to be used for tests about compile-times using concepts. The authors believe, “concepts will probably slow down complex [*sic*] in experimental and early implementations, but there is no fundamental reason for the cost of checking concepts to be particularly expensive.” [76]

### 3.6 Conclusion

Existing generic programming concepts are focused on providing building blocks for additional functionality. They require the presence of methods and associated types. They prescribe the semantics of the methods and sometimes impose time/space complexity requirements on them. This style of specification has been very successfully used for the specification of concepts that provide client-accessible functionality. The specification of invariants ensures that methods perform the tasks they are intended to. Time/space complexity bounds ensure that these tasks are done efficiently. The emphasis is on what tasks are to be done. There is no emphasis on what things are *not* to be done. A type is a model of a concept if a subset of the type’s functionality meets the concept requirements. As a result, it is frequently the case that a model of a concept will provide more functionality than the concept specifies. This additional functionality may not be desired by all clients.

In both the specification of the Recursively Fixed Iterator and the Structure Protecting Iterator, and the proof of their ability to prevent memory leaks and double deletes, we were required to use concepts to disallow operations. These restrictions were required to ensure memory safety.

The Recursively Fixed Iterator concept is a refinement of the Constant Value and Safely Manageable concepts. The Safely Manageable concept is a refinement of the Leak Free Destroyable and Double Delete Free destroyable concepts. These three concepts are *negative*. Without these negative constraints, no positive statements can be made about memory safety.

While the Recursively Fixed Iterator and Structure Protecting Iterator are useful in their own right, we hope the insights that can be gained from the nature of their specification and the proof of its correctness will help others specify concepts. Careful metaprogramming can allow such concepts to be implemented without run-time overhead or the need to expand the C++ tool chain.

We call concepts that restrict side effects *safety concepts*. We expect that safety concepts can be specified for additional safety properties such as thread safety, additional elements of memory safety (in-bounds access, proper typing, etc.). This has the potential to allow software engineers

to assemble software components into programs that are guaranteed not to cause dangerous side effects.

## 4. Automatic Test Bed Generation

### 4.1 Introduction

Regression testing is a software engineering technique that retests previously tested segments of a software system to ensure that they still function properly after a change has been made [21, 149]. Functional regression testing involves executing unit tests and verifying that the output agrees with the output of an earlier version of the software. Regression tests are usually automated and performed in regular intervals during software development. During software maintenance automated regression tests are performed with modifications to the software.

Developers of computer algebra software use a variety of execution-based testing methods. In many cases published test suites such as mathematical tables are used. Some test suites involve mathematically conceived test cases designed to exercise certain features of an algorithm. Other testing techniques involve round-trip computations, comparisons with results computed by other computer algebra systems, or comparisons with results computed by reference implementations within the same computer algebra system. Those testing techniques typically test high-level functionalities and thus tend to be of limited value for the localization of program defects.

In earlier work we ported SACLIB [36, 87] from C to C++ so as to be able to use iterator concepts to refactor the SACLIB memory management subsystem. In the resulting library, SACLIB 3.0, the absence of memory leaks and double deletes is proved during compilation ([158, 157], chapter 3). The work presented in this chapter allows us to perform a systematic regression test of SACLIB 3.0 with respect to the original SACLIB—the last step before a release of SACLIB 3.0.

For SACLIB, regression testing has traditionally been done by performing a large quantifier elimination using QEPCAD [37] or QEPCAD B [25, 26]. The hope was that the quantifier elimination would exercise enough of SACLIB that any defects introduced to SACLIB would be manifest as an incorrect result for the quantifier elimination. This approach suffers from several limitations. There is no guarantee that the quantifier elimination will exercise all of SACLIB. If a given quantifier elimination does not provide sufficient test coverage, it is difficult to increase the test coverage. In general, there is no guarantee that it is possible to construct a set of quantifier elimination problems that will test all of SACLIB. Additionally, increasing coverage through more quantifier elimination problems is difficult. Because quantifier elimination problems are posed at such a high level relative

to SACLIB implementation details, it is difficult to identify a quantifier elimination problem that exercises a particular routine or basic block of SACLIB. Finally, when a defect is detected from a quantifier elimination, there is no indication of where in SACLIB the defect has been introduced.

We address this final limitation with a technique for the automated generation of unit tests for the SACLIB library of computer algebra programs. While running a high-level computation we automatically collect the input–output pairs of each function that is called. We then automatically generate, for each function, a test environment that takes the collected inputs, runs the function, and checks whether the obtained outputs agree with the collected outputs.

Our technique does not verify whether functions conform to specifications, nor does it provide more code coverage than the high-level computation we run. However, the unit tests we generate help localize errors and provide a framework that can be easily augmented with additional test cases.

We use AspectC++ [184, 119, 121, 186, 187] to weave tracing code into SACLIB functions. Aspect-Oriented Programming (AOP) [112, 111] is a programming methodology designed to facilitate the encapsulation of program requirements that cannot be implemented in a single component using traditional software development methods. AspectJ [113, 58] is an extension to the Java [70] programming language that provides direct language support for AOP. AspectJ was the first language to support AOP, and the majority of AOP research and development has been conducted in AspectJ. However, performance critical software such as operating systems and embedded software is typically written in a language like C [93, 91] or C++ [94, 191]. Motivated by the desire to use AOP in performance critical domains, AspectC [31, 69] and AspectC++ [184, 119, 121, 186, 187] provide AOP extensions for C and C++. The extensions provided by AspectC++ are intentionally designed to be as similar as possible to those provided by AspectJ. However, C++ development practices that place an emphasis on templates and static type checking required some alternative design decisions [119, 121].

AspectC++ has been applied to operating systems (BOSS [5], CiAO [123], eCos [120], PURE [127, 185]), embedded micro-controllers [19, 122], message passing [15], network simulation [171], and middleware [62, 108, 136]. AspectC has been applied to the FreeBSD operating system [31]. These efforts have been primarily focused on using AOP to provide refactorings and implementations with improved modularity and configurability without compromising run-time efficiency in memory footprint or execution speed. There has also been some use of aspects for generating trace information useful in debugging and profiling [127]. The testing research on aspects has been focused on adapting existing testing algorithms to handle aspects [153], improving test selection in the presence of

aspects [212, 211, 213, 216], and providing unit test facilities for aspects [155]. We are not aware of any literature on the use of aspects for automated test bed generation.

There are 1100 SACLIB routines; of these, 900 take only SACLIB lists, SACLIB atoms, C++ built-in types, and `std::strings` as arguments. The remaining 200 take pointers, simple pointers, recursively fixed iterators, or structure protecting iterators. We have implemented aspects that are capable of serializing all of these argument types.

## 4.2 Function Level Tracing

Automatically recording function level test cases during the execution of a program requires that function inputs and outputs are collected as each function is executed. Figure 4.1 shows the SACLIB function for composing two SACLIB objects into a list, `LIST2`. The function has been hand instrumented to trace its inputs (lines 18-20) and outputs (lines 45-47) using functions obtained from our `trace_utils.h` header (line 14). Figure 4.2 shows the test cases that are produced from invoking `LIST2(LIST2(12,15),11)`. Each invocation produces a record that starts with the signature of the traced function (lines 1,8) and ends with `%%` (lines 7,14). Lines 2-6 and 9-13 contain the input/output trace for invoking `LIST2(LIST2(12,15),11)`.

In order for the trace functions to record enough information to use as a test case, they must serialize enough execution context of the instrumented function to allow it to be invoked solely from the information recorded in the trace. This is only possible if the `trace_*` functions have knowledge of the execution context of the instrumented function. However, the full execution context is the entire address space (stack, heap, and registers) of the process running the instrumented function, the full state of the operating system, the full state of any process that can be interacted with via IPC, and the full state of any peripherals accessible to the process running the instrumented function. It is clearly infeasible to serialize such an extensive amount of execution context.

The key to building a test harness for SACLIB is to determine both what is necessary for the serialization of a test case and how the needed state can be efficiently computed on a time scale that makes testing worthwhile. The design and implementation of the tracing functions is driven by striving for the proper balance between these two competing objectives. For SACLIB, correct tracing requires the ability to identify the function currently executing, handle recursive calls of the traced function, trace input/outputs of arbitrary type, allow SACLIB functions to be used inside of the `trace_*` functions without being traced, trace relevant global state used by the traced function,

```

1: /*=====
2:                               L <- LIST2(a,b)
3:
4: List, 2 elements.
5:
6: Inputs
7:   a,b : objects.
8:
9: Outputs
10:  L  : the list (a,b).
11: =====*/
12:
13: #include "saclib.h"
14: #include "trace_utils.h"
15:
16: Word LIST2(Word a, Word b)
17: {
18:   trace::trace_signature("Word LIST2(Word,Word)");
19:   trace::trace_input("argument 0", a);
20:   trace::trace_input("argument 1", b);
21:
22:   Word L,M;
23:
24: Step1: /* Store a. */
25:       L = AVAIL;
26:       if (L == NIL) {
27:         GC();
28:         goto Step1; }
29:       SFIRST(L,a);
30:
31: Step2: /* Store b. */
32:       M = RED(L);
33:       if (M == NIL) {
34:         GC();
35:         goto Step1; }
36:       SFIRST(M,b);
37:
38: Step3: /* Set AVAIL to reductum of M. */
39:       AVAIL = RED(M);
40:
41: Step4: /* Set reductum of M to NIL. */
42:       SRED(M,NIL);
43:
44: Return: /* Prepare for return. */
45:        trace::trace_output("argument 0", a);
46:        trace::trace_output("argument 1", b);
47:        trace::trace_return(L);
48:        return(L);
49: }

```

Figure 4.1: Manually inserted tracing code. The inputs (lines 18-20) and the outputs (lines 45-47) are traced using functions obtained from the header file `trace_utils.h` (line 14).

```

1: Word LIST2(Word,Word)
2:   return: (12,15)
3:   argument 0 input: 12
4:   argument 0 output: 12
5:   argument 1 input: 15
6:   argument 1 output: 15
7: %%
8: Word LIST2(Word,Word)
9:   return: ((12,15),11)
10:  argument 0 input: (12,15)
11:  argument 0 output: (12,15)
12:  argument 1 input: 11
13:  argument 1 output: 11
14: %%

```

Figure 4.2: Test cases produced by invoking the function in Figure 4.1 as `LIST2(LIST2(12,15),11)`. Each invocation of `LIST2` produces a record that starts with the signature of the traced function (Lines 1,8) and ends with `%%` (Lines 7,14). Lines 2-6, 9-13 trace inputs and outputs.

and trace functions with multiple exit points. Performing this tracing automatically requires the ability to insert tracing code into all `SACLIB` functions without the need for hand instrumentation. Care must also be taken in the storage and reuse of test cases. A test case is only worth storing for use in later testing if it provides fault detection power beyond the test cases that have been previously stored. After the test cases are stored, a test harness must be provided to execute the test cases.

### 4.3 Function call identification

The `trace_signature` function (Figure 4.3) is responsible for identifying the function being traced and allowing the tracing of recursive function invocations. Each stack frame is described by a `stack_frame_record` (lines 4-14). When `trace_signature` is invoked, it adds a `stack_frame_record` to the map `stack_frame` and stores the signature for the function being traced (lines 22-23). The signature of the traced function is supplied by the caller of `trace_signature` (line 20). This argument must match the function being traced. Because the other tracing functions require knowledge of the stack frame they are tracing data for, `trace_signature` must be called inside the traced function before any other tracing functions are called.

Immediately before the traced function returns, `trace_return` (lines 27-36) must be called with the return value of the traced function. The return value is stored in `stack_frame_record` (line 30-31) and the trace information for the stack frame is serialized to `trace_stream` (line 32). The

```

1:
2: namespace trace{
3:
4:     struct stack_frame_record{
5:         stack_frame_record():has_return(false){}
6:
7:         std::string      signature;
8:         bool             has_return;
9:         std::string      return_value;
10:        std::vector<std::string> input;
11:        std::vector<std::string> output;
12:
13:        void clear(); //reset all fields
14:    };
15:    std::ostream& operator<<(std::ostream& out, const stack_frame_record& r);
16:
17:    extern int                stack_frame_id;
18:    extern std::map<int, stack_frame_record> stack_frame;
19:
20:    void trace_signature(const std::string& signature){
21:
22:        ++stack_frame_id;
23:        stack_frame[stack_frame_id].signature = signature;
24:
25:    } //trace_signature
26:
27:    template <typename T>
28:    void trace_return(T t){
29:
30:        stack_frame[stack_frame_id].has_return=true;
31:        stack_frame[stack_frame_id].return_value = to_string(t);
32:        trace_stream << stack_frame[stack_frame_id];
33:        stack_frame[stack_frame_id].clear();
34:        --stack_frame_id;
35:
36:    } //trace_return
37:
38: } //namespace

```

Figure 4.3: The implementation of `trace_signature` and `trace_return` must record the call stack in order to allow tracing of recursive functions.

```

1: #include <string>
2:
3: namespace trace{
4:
5:     extern std::ostream& trace_stream;
6:
7:     template <typename T>
8:     void trace_input(std::string& argument arg, const T& value){
9:         trace_stream << arg << " input: " << value << "\n";
10:    }//trace_input
11:
12:    void trace_input(std::string& argument arg, const Word& value){
13:        trace_stream << arg << " input: ";
14:        OWRITE(trace_stream, value);
15:    }//trace_input
16:
17: }//namespace

```

Figure 4.4: An overload of the `trace_input` function must be provided for each type to be traced. The first overload is for streamable types. The second overload is for SACLIB objects.

`stack_frame_record` is then made available for reuse (lines 33-34). Because `trace_return` removes the most recently added `stack_frame_record`, it can only be called after all `trace_input` and `trace_output` calls have completed.

All tracing information for a stack frame is aggregated into a `stack_frame_record` and is not serialized until all trace information for a single frame is available. This is needed to trace recursive function applications and functions that call other traced functions. If this aggregation were not performed, the serialized tracing output from different functions would be interspersed in `trace_stream`. Although the implementation of the `trace_*` functions all properly aggregate information to `trace::stack_record`, all subsequent code examples in this chapter will be shown with direct serialization to `trace_stream` in order to simplify the presentation.

#### 4.4 Recording input/output

Generally, input/output tracing of a C++ variable `t` of type `T` requires the ability to serialize objects of type `T`. Because C++ does not provide a uniform way to serialize arbitrary types, each type must be handled differently. Tracing SACLIB requires the ability to serialize C++ fundamental types, C++ pointers to fundamental types, `std::strings`, SACLIB atoms, SACLIB lists, recursively fixed iterators, structure protecting iterators, and simple pointers [158].

Tracing of the inputs is accomplished by calls to a `trace_input` function overload. Figure 4.4

```

1: /*=====
2: A <- FRAPGET(d,n)
3: Finite ring array polynomial get memory.
4:
5: Inputs
6: d, n: positive BETA-digits.
7: Outputs
8: A : a pointer to an array large enough to hold an element of
9: (Z/(m)[x])/(M)[y] having degree d, where M has degree n.
10: =====*/
11: Word **FRAPGET(Word d, Word n){
12:
13: Step1: /* Allocate memory for polynomial. */
14:     Word** A = new Word*[d+2];
15:     A = A + 1;
16:     A[-1] = GETARRAY(2);
17:     FRAPSIZE(A) = d;
18:     for(Word i = 0; i <= d; i++)
19:         FRAPCF(A,i) = MAPGET(n);
20:
21: return(A);
22: }
23:
24: /*=====
25: FRAPFREE(A)
26: Finite ring array polynomial free memory.
27:
28: Inputs
29: A : a pointer to an array, memory for which was allocated using FRAPGET.
30: Side effects
31: The memory allocated to A is freed.
32: =====*/
33: void FRAPFREE(Word **A){
34:
35: Step1: /* Determine how many coefficients. */
36:     Word d = FRAPSIZE(A);
37:     FREEARRAY(A[-1]);
38:
39: Step2: /* Free memory of each coefficient. */
40:     for (Word i = 0; i <= d; i++)
41:         MAPFREE(FRAPCF(A,i));
42:
43: Step3: /* Free main array. */
44:     A = A - 1;
45:     FREEARRAY(A);
46:
47: return;
48: }

```

Figure 4.5: The SACLIB routines FRAPGET and FRAPFREE use their own bookkeeping data for managing heap allocated memory.

```

1: #include "trace_utils.h"
2: #include <boost/static_assert.hpp>
3:
4: void* operator new(size_t s){
5:     BOOST_STATIC_ASSERT(sizeof(char)==1);
6:
7:     void* storage = malloc(s);
8:     trace::allocated_memory_push_back(
9:         trace::memory_range(storage, static_cast<char*>(storage) + s)
10:    );
11:
12:     return storage;
13: }//operator new
14:
15: void operator delete(void* p){
16:
17:     trace::memory_range* i;
18:     for(
19:         i = trace::allocated_memory_begin();
20:         i != trace::allocated_memory_end();
21:         ++i
22:     ){
23:
24:         if(i->in_range(p)){
25:             trace::allocated_memory_erase(i);
26:             break;
27:         }//if
28:
29:     }//for
30:
31:     free(p);
32: }//operator delete
33:

```

Figure 4.6: Implementation of `operator new` and `operator delete` to track the information about heap allocated memory required to allow the implementation of the `trace_input` overload in Figure 4.7.

```

1: #include <string>
2: #include <vector>
3:
4: namespace trace{
5:
6:     extern std::ostream& trace_stream;
7:
8:
9:     struct memory_range{
10:         memory_range(void* b, void* e):begin(b),end(e){}
11:
12:         void* begin;
13:         void* end;
14:
15:         bool in_range(void*p){return (begin <= p) && (p < end); }
16:     };
17:
18:     memory_range allocation_range(void* p){
19:         for(int i=0; i < allocated_memory_size(); ++i){
20:             if(allocated_memory_at(i).in_range(p)){
21:                 return allocated_memory_at(i);
22:             }//if
23:         }//for
24:     }//allocation_range
25:
26:     template <typename T>
27:     void trace_input(std::string& argument arg, T* value){
28:         trace_stream << arg << " input: ";
29:
30:         memory_range m = allocation_range(value);
31:
32:         int offset = m.begin() - value;
33:         trace_stream << "offset=" << offset;
34:
35:         int i=0;
36:         for(T* p = (T*)m.begin; p < (T*)m.end; ++p){
37:             trace_stream << ":[ " << i << "]" << *p;
38:             ++i;
39:         }//for
40:
41:     }//trace_input
42:
43: }//namespace

```

Figure 4.7: Implementation of a `trace_input` overload that uses the information recorded from operator `new` and operator `delete` (Figure 4.6) to serialize heap allocated memory.

```

1: /*=====
2:                               B <- AII(A)
3:
4: Array integer to integer.
5:
6: Input
7:   A : an integer in array representation.
8:
9: Output
10:  B : the integer in list representation.
11:
12: =====*/
13:
14: #include "saclib.h"
15:
16: Word AII(rec_fixed_itr<BDigit*> A)
17: {
18:   Word B;
19:   BDigit b,i,n,s;
20:
21: Step1: /* Single precision. */
22:   s = A[0];
23:   n = A[1];
24:   if (s == 0) {
25:     B = 0;
26:     return(B); }
27:   if (n == 1) {
28:     B = A[2];
29:     if (s < 0)
30:       B = - B;
31:     return(B); }
32:
33: Step2: /* Multiple precision. */
34:   rec_fixed_itr<Word*> Ap(A + 2);
35:   B = NIL;
36:   for (i = n - 1; i >= 0; i--) {
37:     b = Ap[i];
38:     if (s < 0)
39:       b = - b;
40:     B = COMP(b,B); }
41:
42:   return(B);
43: }
44:

```

Figure 4.8: The `rec_fixed_itr<BDigit*>` argument to the SACLIB routine `AII` requires the serialization of a pointer to an array of `BDigits`.

```

1: int AII(rec_fixed_itr< int * >)
2:   return: 0
3:   argument 0 input:  shift=0:[0]=1:[1]=1:[2]=0
4:   argument 0 output: shift=0:[0]=1:[1]=1:[2]=0
5: %%

```

Figure 4.9: A test case serialized from calling the AII routine in Figure 4.8 as “BDigit b[] = {1,1,0}; AII(b);”

contains a `trace_input` overload using `operator<<` on lines 7-10. As all C++ built in types are streamable, this overload provides support for C++ built-in types. Support for SACLIB objects is provided on lines 12-15. The SACLIB routine `OWRITE` performs an object write of the SACLIB object contained in `value` to the stream `trace_stream`. Calls to `trace_input` function overloads must occur once for each argument to the trace function. The `trace_input` calls must occur after the call to `test_signature`, once for each argument of the traced function, before any calls to `trace_output` or `trace_return`, and with an indication of which input is being traced. Tracing of the outputs is handled similarly by `trace_output`. All `trace_output` calls must occur after all `trace_input` calls and before the `trace_return` call. Note the string literals used as the arguments to the `trace_*` functions correspond to the strings in the output file.

The most demanding type of arguments to trace are C++ pointers. Figure 4.5 contains the SACLIB routines for allocating (`FRAPGET`, lines 1-22) and deallocating (`FRAPFREE`, lines 24-48) heap memory used to represent finite ring polynomials. On line 14 memory is allocated using `new` and the pointer `A` is set to the beginning of the allocated memory. The pointer `A` is immediately incremented (line 15) and `GETARRAY` is used to store a pointer to a 2 element array at `A[-1]` (line 16). The macro `FRAPSIZE(A)`, which expands to `A[-1][1]`, is used to set the size of `A` to `d` (line 17). The remaining elements of `A` are initialized (lines 18-19) and `A` is returned (line 21). At the point that `A` is returned, it does not point to the beginning of the memory region allocated by `new` on line 14. Instead, it points into the interior of the memory region.

The purpose of returning a pointer to the interior of the allocated region is to make space for bookkeeping data. Had the finite ring polynomial been implemented as a structure, it would have been possible to store the bookkeeping data as members of the structure. However, this would have made the finite ring polynomial incompatible with any other SACLIB routines that both expected to operate on a pointer of type `Word**` and did not have a signature that explicitly passed the bookkeeping data.

An example use of the bookkeeping data occurs in `FRAPFREE`. `FRAPFREE` begins by using the `FRAPSIZE` macro to obtain the size (line 36) and then frees the bookkeeping data (line 37). The rest of the function frees the remaining memory (lines 38-48). The size is used as the loop bound to ensure the correct number of coefficients are freed (line 40). The signature of `FRAPFREE` illustrates advantages of this style of bookkeeping: clients of `FRAPGET` and `FRAPFREE` are able to use pointers to finite ring polynomials without any knowledge of the bookkeeping data.

The usage of heap memory in `FRAPGET/FRAPFREE` illustrates a fundamental problem in the serialization of data from the heap via a pointer. When serializing memory from the heap for a unit test, all memory accessed through the pointer must be serialized. In the general case, the correct amount of data to serialize in each direction from the pointer is a run-time property. For the case of `FRAPGET/FRAPFREE` it can be statically determined that one element before the pointer must be serialized, but the number of elements after the pointer depends on the degree of the finite ring polynomial.

In order to serialize the proper amount of data, we impose the following constraint on programs we trace: serialization of heap memory only has defined behavior if pointer arithmetic never causes a pointer to address memory from a different allocated block. This is not a significant limitation because performing such pointer arithmetic will result in undefined program behavior. When heap memory is allocated there is no guarantee about what memory is adjacent to the block of allocated memory. The pointer arithmetic we cannot trace is a defect in the traced program that should be repaired. Memory profiling tools such as Purify [89] and Valgrind [198] will diagnose these defects as “array out of bounds” errors.

With this constraint, a pointer to heap memory can be serialized by serializing the entire allocated block the pointer points into and the offset of the pointer from the beginning of the block. Because the entire allocated block is serialized, we are guaranteed to store all elements that could possibly be referenced via valid pointer arithmetic. Pointers to the stack can be handled in a similar fashion: the entire stack can be serialized.

The decision to serialize the entire block consciously accepts the potential of serializing more memory than is accessed by the trace function. However, it significantly simplifies the implementation of the serialization because run-time memory accesses do not need to be monitored during the execution of the trace function to determine the minimal set of elements to serialize. It also makes tests cases serialized with the entire allocated block more robust because the test case can be used on functions that have been altered to access different memory elements but still produce the same

output.

Figure 4.6 and Figure 4.7 contain the implementation for the serialization of heap allocated memory. Without user intervention (by overloading `operator new` or using placement `new`), C++ creates new heap allocated objects in two steps. The first step is to allocate the heap memory via `malloc`. The second step is to use the object's constructor to initialize the memory. In order to be able to serialize the memory, we need to record the location and size of the blocks allocated on the heap. Recording this bookkeeping information about the allocated blocks also requires that the information be discarded when blocks are deallocated.

When implementing `operator new` and `operator delete`, it is not desirable to call functions that in turn call `operator new` or `operator delete`. The chances of causing an infinite recursion or corrupting memory are too high. This rules out using STL containers. To allow the functionality of a `std::vector<trace::memory_range>`, the following functions are defined in the `trace` namespace: `allocated_memory_begin`, `allocated_memory_end`, `allocated_memory_size`, `allocated_memory_at`, `allocated_memory_erase`. They provide the same functionality as the `std::vector` member functions of the same name. They always operate on the same storage.

Recording the bookkeeping data for the allocated blocks is accomplished by overloading `operator new`. Overloads of `operator new` allow users to control what actions are taken when a user requests memory allocation via `new`. Figure 4.6 contains an `operator new` (lines 4-10). The `BOOST_STATIC_ASSERT` (line 5) is there to indicate that the implementation relies on the fact that characters have a size of one byte. On line 7 `malloc` is used to allocate the memory. The beginning (`storage`) and end (`static_cast<char*>(storage) + s`) of the block allocated by `malloc` are stored into `trace::allocation_range` (lines 8-10). The `trace::memory_range` represents a memory range by storing a void pointer to the beginning of the range and void pointer to the byte immediately beyond the range. After storing the `memory_range`, the pointer to the allocated memory is returned (line 12). This implementation allows us to record information about the location of allocated blocks as memory is allocated.

Discarding the bookkeeping data is accomplished by providing an `operator delete`. Figure 4.6 contains a `operator delete` (lines 17-32). Unlike the `operator new` implementation there is no `BOOST_STATIC_ASSERT`. This is because the implementation of `operator delete` does not perform pointer arithmetic and has no dependence on the size of a character. The implementation loops over all memory ranges that have been recorded by `operator new` (lines 17-29) and erases the `memory_range` that `i` points into (lines 24-27). This implementation of `operator delete` is de-

signed to be robust in the face of common memory defects in the traced program. The test `i->in_range(p)` returns true when `p` points into the `memory_range *i`. This protects against two common programming errors that may be present in the traced program. The first error this protects against is the traced program calling `delete` on a pointer that does not point to the beginning of an allocated block. This gives the traced program some leeway with respect to pointer arithmetic errors that cause pointers to the interior of an allocated block to be deleted. The second error is a double delete in the traced program. The bookkeeping information will be erased the first time the for loop (lines 18-29) is traversed. Any subsequent traversals caused by double deletes in the traced program will not find anything to delete. Ultimately, the bookkeeping data maintained by this `operator new` and `operator delete` cannot compensate for all memory defects in the traced program. This protection against common errors is added as a service for the user. Programs with substantial memory defects will not be traced correctly. However, these memory defects will cause other problems for the traced program and will need to be repaired.

When `operator new` and `delete` are provided in this form in the global namespace, they must be defined in the same translation unit as `main`. SACLIB requires that all client programs provide a routine called `sacMain`. The SACLIB library is responsible for providing a `main` routine that properly initializes the SACLIB library and then executes the `sacMain` provided by the client. This allows the implementation of `operator new` and `operator delete` to be defined with the SACLIB `main` routine. Because this definition of `operator new` and `operator delete` serve no purpose other than to enable tracing, conditional compilation can be used to only provide the operators when a SACLIB program is compiled with tracing. If all SACLIB programs did not share the same `main` function, each SACLIB program would have to provide an `operator new` and `operator delete` for tracing. Although these could be provided in a header, this would make the tracing harder to use (each `main` program would need to include the header) and would cause problems with tracing SACLIB programs that had already replaced `operator new` and `operator delete`.

Once pointers can be traced, recursively fixed iterators and structure protecting iterators can be traced using similar techniques. Figure 4.8 contains the SACLIB routine `AII`. `AII` takes a single function argument, `A`, of type `rec_fixed_itr<BDigit*>`. The `rec_fixed_itr` class is implemented in terms of a `BDigit*`. Figure 4.9 contains a test case serialized when the `BDigit*` pointer in `A` points to `{1,1,0}`. The pointer is serialized on line 3. The `shift` is the number of bytes the serialized pointer is past the beginning of the block it was allocated in. Each element of the array is serialized as `[n]=v` where `n` is the offset from the beginning of the array and `v` is the value of the

element. This representation was selected for two reasons. The first is that it provides an easy to parse representation. The second is that the use of the index allows the selective serialization of array elements.

We also modified `new` so that all memory is zeroed before being returned to the caller. This was necessary because some SACLIB routines allocated more memory than they ever used. In a single run of SACLIB, this did not cause a problem. However, it caused spurious failures when running regression tests. The reason is that the unused memory contained uninitialized values. It was possible for the uninitialized memory to contain different values during tracing and execution of the test in the harness. This also required the test harness to have a modified `new` that zeroed memory. The result is that uninitialized memory has the same value during the tracing and the testing.

We had considered identifying the blocks of heap allocated memory by placing a “magic” byte pattern before and after the allocated memory. This would have allowed tracing of the heap memory by simply writing all data between the magic values, and would have resulted in a simpler implementation of the heap tracing. The use of “magic” values was abandoned for several reasons. In general, it is not possible to pick a “magic” value that is guaranteed to not occur in normal SACLIB data. Additionally, it causes problems with alignment, because it is never clear if a given pointer reference memory with the same alignment as the “magic” value. A similar implementation could have been done using virtual memory in a similar fashion to Electric Fence [51]. However, this would have been more difficult to implement than recording information on each `new/delete` call and not provided any additional benefits.

#### 4.5 Recording global state

The output of SACLIB routines depends on a relatively limited part of the external state. Most routines are only influenced by the state of the heap and the space array. The space array is where all garbage collected SACLIB lists and integers are stored. The `trace_input`, `trace_output`, and `trace_return` functions described in Section 4.4 automatically take care of the heap and space array. This is because they serialize the value of the variables stored on the heap or in the space array. The stored values may then be deserialized into storage provided by the test harness. All computations on SACLIB objects are performed on the values of the BDigits in the objects. Because of this, SACLIB objects can be serialized and deserialized solely in terms of values. SACLIB will not

notice any difference in the memory location of the BDigits that may be caused by the serialization process.

A few SACLIB routines are influenced by the state of the SACLIB random number generator (the global variables `RINC`, `RTERM`, and `RMULT`) and the floating point error status (the global variable `FPHAND`). These global variables are all SACLIB objects and are handled by adding code to the `trace` aspect that uses `trace_input` and `trace_output` to serialize these four variables from all SACLIB routines. Because each of these variables is a C++ `int`, the space overhead of serializing these for all functions is small. Although it would be possible to serialize these variables only when tracing functions that depend on them, this is undesirable. Determining which functions depend on these variables requires construction of either a static or dynamic call graph for use in determining which functions require these variables to be serialized. The call graph will obviously change as maintenance is done on SACLIB. Because of the small size of these variables, it is not worthwhile to implement the logic needed to compute the call graph and perform serialization based on it.

It is not feasible to serialize state for the SACLIB routines `CREAD`, `CWRITE`, `SWRITE`, and `BKSP`. No attempt was made to serialize state for these functions. The reason these functions are so difficult to trace is that they produce i/o side-effects. Regression testing of these routines using our tracing would require the serialization of the full state of the streams used for i/o. This would also require the ability to serialize any data that had been written to storage by the streams. Not tracing these functions is not a limitation in assuring the correctness of SACLIB. The SACLIB library is primarily designed for computation and does not perform very much i/o. Additionally, many uses of i/o are only for debugging. Not only is i/o not a significant part of SACLIB, but constructing unit tests for the i/o routines requires less effort than devising a method for serializing the state of the i/o streams. The i/o routines in SACLIB are not traced.

The remaining global state does not need to be serialized for testing. There are several read-only lookup tables that are populated by SACLIB when a SACLIB program is started. They contain precomputed reference data such as a list of the first primes. These tables will be populated with identical values each time a SACLIB program is initialized with `BEGINSACLIB`. All other SACLIB routines cannot be used before `BEGINSACLIB` has been called. Because of this, these SACLIB global variables are effectively serialized in the code used to implement `BEGINSACLIB`. This allows SACLIB functions called from a test harness to safely access this global state without any danger of missing global state required by the function.

```

1: #ifndef TRACE_AH
2: #define TRACE_AH
3:
4: #include "trace_utils.h"
5:
6: aspect Trace{
7:
8:     pointcut exclude_set() = execution(
9:         "% trace::%(...)"
10:    );
11:
12:    pointcut trace_set() = !exclude_set() && execution("% %(...)");
13:
14:    advice trace_set(): around() {
15:        trace::trace_signature(tjp->signature());
16:        trace::trace_input(tjp);
17:
18:        tjp->proceed();
19:
20:        trace::trace_output(tjp);
21:        trace::trace_return(tjp->result());
22:    }
23:
24: };
25:
26: #endif

```

Figure 4.10: An aspect to weave tracing code around execution joinpoints.

#### 4.6 Aspect based tracing

SACLIB contains over 1,000 functions. Adding hand tracing to all of these functions is clearly a tedious and undesirable task. The requirement to call the tracing functions in the correct order with the correct arguments is an error prone process. The requirement to call the `trace_*` functions to match the structure of the trace function results in duplicating information about the trace functions arguments in the code. This poses a maintenance hazard: any updates to the traced function require `trace_*` calls to be updated. Given that adding the `trace_*` calls is tedious, error-prone, and the source of the traced function provides the information of the only correct way to call the `trace_*` functions, automated insertion of the `trace_*` functions is a natural solution.

We remove the limitations of hand instrumentation by using AspectC++ [184, 119, 121, 186, 187]. AspectC++ provides a convenient mechanism to automatically add tracing code to all SACLIB functions. AspectC++ extends C++ with three significant extensions: aspects, pointcuts, and

```

1: #include <boost/static_assert.hpp>
2: #include <cassert>
3: #include <iostream>
4: #include <map>
5: #include <vector>
6:
7: namespace trace{
8:
9:     extern std::ostream& trace_stream;
10:    extern bool          tracing_enabled;
11:
12:    struct more_args{};
13:    struct done_args{};
14:    template<size_t count> struct have_more_args;
15:        template<size_t count> struct have_more_args    {typedef more_args type;};
16:        template<>          struct have_more_args<0>{typedef done_args type;};
17:
18:    template <size_t count, typename TJP>
19:    void stream_args(TJP* tjp, done_args){
20:        BOOST_STATIC_ASSERT(0 == count);
21:        assert(tracing_enabled);
22:    }//stream_args<count>(TJP*, done_args)
23:
24:    template <size_t count, typename TJP>
25:    void stream_args(TJP* tjp, more_args){
26:        assert(tracing_enabled);
27:
28:        trace_stream
29:            << "argument " << TJP::ARGS-count << "input: "
30:            << *(tjp->template arg<TJP::ARGS-count>())
31:            ;
32:
33:        stream_args<count-1>(
34:            tjp,
35:            typename have_more_args<count-1>::type()
36:        );
37:
38:    }//stream_args<count>(TJP*, more_args)
39:
40:    template <typename TJP>
41:    void trace_input(const TJP& tjp){
42:
43:        if(tracing_enabled){
44:            stream_args<TJP::ARGS>(
45:                &tjp,
46:                typename have_more_args<TJP::ARGS>::type()
47:            );
48:        }//if
49:
50:    }//trace_input
51:
52: }//namespace

```

Figure 4.11: A `trace_input` overload to trace all of the arguments in an AspectC++ JoinPoint.

joinpoints. The purpose of these extensions is to allow code that implements *cross-cutting concerns* to be stored in an aspect as *advice* and then *woven* into existing source code using an *aspect weaver*. Joinpoints are points in the code where the aspect weaver may place the code contained in an aspect's advice. A pointcut is a set of joinpoints.

Figure 4.10 contains an aspect to weave tracing into all SACLIB functions. Lines 6-24 define the aspect. Lines 8-10 define a pointcut for all of the functions used to implement the tracing. The pointcut `exclude_set()` defines a pointcut name `exclude_set`, and the `execution("% trace::%(...)")` provides the pointcut that contains the execution joinpoint for each function matching the expression `"% trace::%(...)"`. The first `%` is a wild card that matches any return type, the second `%` is a wild card that matches any function name, and the `...` is a wild card that matches any argument list. The execution joinpoint for a function is the function invocation. The net result is that `"% trace::%(...)"` matches all functions in the namespace `trace` and `exclude_set` contains all execution joinpoints for these functions.

On line 12 the pointcut `trace_set` is created. It is created from all joinpoints in `execution("% %(...)"`) that are not in the pointcut `exclude_set`. Because `"% %(...)"` matches any function, `trace_set` will contain all execution joinpoints except those that implement tracing logic. This is exactly the set of joinpoints that tracing should be added to. Lines 14-22 provide the advice needed to trace a function. We use `around` advice because it allows advice to be woven both before and after each joinpoint. Lines 15-16 weave signature and input tracing to the beginning of the joinpoint, line 18 executes the code contained in the joinpoint, and lines 20-21 weave output and return value tracing after the joinpoint. Notice that the signatures `trace_input` and `trace_output` have been modified to take `tjp`, which stands for "the joinpoint", as an argument. This is possible because in the advice, `tjp` is aware of its arguments.

In addition to removing all the limitations of hand instrumentation, using AspectC++ also automatically handles traced functions with multiple return statements. On line 21 the return value is obtained from `tjp->result`. This provides the return value of the joinpoint after it has executed. Correctly weaving the advice into functions with multiple return statements is handled by the weaver.

AspectC++ assigns a static type to the joinpoint based on the number, type, and order of the arguments for the joinpoint. This design has two benefits. The first is that it allows all statically known attributes about the arguments to be encoded in the type system. The second is that it relieves the need for the AspectC++ implementation to deal with the fact that C++ only allows a

function to have a single return type. Because arguments can have different types, this would need to be dealt with if there were a single AspectC++ function that allowed arguments to be accessed at run-time. Because of this limitation, any code that wants to process all of the arguments in a joinpoint must use template metaprogramming.

Figure 4.11 contains a `trace_input` (lines 41-48) implementation to trace a joinpoint. The template argument `TJP` is used to deduce the type of the joinpoint being traced. If tracing is enabled, `tjp` is passed to the `stream_args` function (lines 43-48). The `stream_args` function has to keep track of two things. The first is the index of the argument being streamed. The second is if there are additional arguments to stream. Because of how AspectC++ types joinpoints, these two things must be kept track of separately – we will explain why after discussing the implementation. On line 44 `stream_args` is passed `TJP::ARGS` (line 44) as the template argument `count` (lines 19 and 24). This is used to compute the correct indexing (lines 29-30) that is used to serialize the argument to the stream.

Before discussing the implementation, it is helpful to review to the compilation model for AspectC++. The AspectC++ compiler is implemented using a source-to-source translator. It takes AspectC++ source as input and outputs C++ source code corresponding to the AspectC++ program. The generated C++ code is then compiled by a standard C++ compiler. This means that by the time the C++ compiler is executed, all AspectC++ extensions have been transformed into C++ code. The C++ compiler has no direct knowledge of the AspectC++ keywords or weaving used in the original AspectC++ program. This means that after the AspectC++ source-to-source translator has run, joinpoints have been replaced with C++ variables and functions. One of the things encoded by the translation is the number and type of the arguments associated with each joinpoint. Because of this, all processing of the joinpoint arguments must be done in C++.

The code in Figure 4.11 determines if there are more arguments to process through a combination of `stream_args` overloading and the `have_more_args` metafunction. The `have_more_args` metafunction (lines 14-16) returns the type `more_args` when `count` is not zero (line 15) and `done_args` when `count` is 0 (line 16). When `stream_args` is called (lines 33-36, 44-47) `have_more_args` is always used to create a temporary variable for the second argument. When `have_more_args` is called with an argument of zero, this will select the overload on line 19 and no arguments will be accessed or printed. Otherwise, the overload on line 25 will be selected, an argument will be printed (lines 28-31), and `stream_args` will be called again (lines 24-36).

Indexing and argument access need to be handled separately because argument access occurs at

```

1: #include <string>
2:
3: namespace trace{
4:
5:     extern std::ostream& trace_stream;
6:     extern bool         tracing_enabled;
7:
8:     template <typename T>
9:     void trace_input(std::string& argument arg, const T& value){
10:
11:         if(tracing_enabled){
12:             trace_stream << arg << " input: " << value << "\n";
13:         }//if
14:
15:     }//trace_input
16:
17:     void trace_input(std::string& argument arg, const Word& value){
18:
19:         if(tracing_enabled){
20:             trace_stream << arg << " input: ";
21:             tracing_enabled=false;
22:             OWRITE(trace_stream, value);
23:             tracing_enabled=true;
24:         }//if
25:
26:     }//trace_input
27:
28: }//namespace

```

Figure 4.12: The `trace_input` overloads from Figure 4.4 are augmented to prevent stack overflow during tracing.

compile-time. In place of the argument streaming on line 30, consider attempting to use a run-time index of the form:

```

a: if(0 != count) {
b:   trace_stream << tjp->template arg<TJP::ARGS-count>();
c: }//if

```

This will result in a compile-time error. When `count` is zero, the `if` will not execute. However, valid indexes to the `arg` member function are in `[0, TJP::ARGS)`. When `count` is zero, `arg<TJP::ARGS-count>` will be compiled with an out-of-range index.

#### 4.7 First order tracing

Because the SACLIB routines `OREAD` and `OWRITE` are used to serialize SACLIB atoms and lists, they will be called by routines such as `trace::trace_input`. If the invocation of `OWRITE` made from `trace::trace_input` were also traced, each call to `OWRITE` from `trace::trace_input` would result in a call to `trace::trace_input`, which would result in another call to `OWRITE`, and ultimately result in a stack overflow. This is dealt with by disabling tracing inside of the `trace_*` functions. Once tracing has been disabled, SACLIB routines can safely be used for serialization.

The implementation in Figure 4.12 augments the implementation from Figure 4.4 with the code required to prevent the stack overflow. The global variable (line 6) `tracing_enabled` is used to record if tracing should be performed. Both trace input overloads now have an if statement to only perform tracing when `tracing_enabled` is `true` (line 12, 19). The `trace_input` overload for SACLIB objects (lines 17-26) sets `tracing_enabled` to `false` (line 21), calls `OWRITE` (line 22), and resets `tracing_enabled` to `true` (line 23). This ensures that the call to `OWRITE` will not be traced. There is a potential maintenance hazard associated with this method of disabling tracing: `tracing_enabled` must be manually set for each call to a SACLIB function from within a tracing function. In practice, this is an acceptable method for disabling tracing because there are few calls to SACLIB functions within the tracing functions, the tracing functions are rarely modified, and the convention for setting `tracing_enabled` only needs to be adhered to in the single module contained in the trace namespace. The other `trace_input` overloads are handled similarly.

#### 4.8 Test case filtering and execution

During the execution of a SACLIB executable that has been woven with the Trace aspect, it is possible that a single SACLIB routine will be called multiple times (possibly with the same arguments). This is particularly true for routines such as the list processing functions that are used in the implementation of most SACLIB routines. Because each execution of a traced routine will serialize a test case, it is possible that disk space will be used inefficiently. This can occur in two ways. The first is when a test case does not add to the fault-detecting power of the already collected test cases. In this case, the test could be discarded. The second source of inefficiency arises if the run-time required to execute the collected test cases exceeds the time a user is willing to devote to running the test cases. Currently, we address only the second problem. When the trace aspect is woven into SACLIB, it can be directed to stop test collection for each routine after a certain number

```

1: using namespace std;
2:
3: int sacMain(int argc, char **argv){
4:
5:     ifstream test_cases("test_input");
6:
7:     while(!test_cases.eof()){
8:         string signature; read_signature(test_cases,signature);
9:         verify_signature("Word LIST2(Word,Word)", signature);
10:
11:         Word return_value, expected_return;
12:         read_expected_return(test_cases, expected_return);
13:
14:         Word a0_in; read_input (test_cases, a0_in,  "0");
15:         Word a0_out; read_output(test_cases, a0_out, "0");
16:         Word a1_in; read_input (test_cases, a1_in,  "1");
17:         Word a1_out; read_output(test_cases, a1_out, "1");
18:
19:         return_value = LIST2(a0_in, a1_in);
20:         check_equal(return_value, expected_return);
21:         check_equal(a0_in,      a0_out      );
22:         check_equal(a1_in,      a1_out      );
23:
24:         string s;
25:         getline(test_cases,s);
26:         if("%%"!=s){cerr << "terminator='" << s << "'\n"; exit(1);}
27:
28:     }//while
29:
30: }//main

```

Figure 4.13: A test harness to execute test cases for the SACLIB routine LIST2.

of test cases have been collected. In the future, we will address the first problem by using code coverage tools. We currently use the tests with the Retest-All strategy.

#### 4.9 Test harness generation

Once the test cases have been obtained from the `trace` aspect, they must be played through a test harness. Figure 4.13 contains a test harness for LIST2. This test harness was produced automatically from a Python code generator that constructs a test harness capable of executing test cases for any SACLIB routine. The output in Figure 4.13 was restricted to only the code needed to test LIST2 and was slightly modified for readability.

The test harness is a standard SACLIB program that begins from the `sacMain` routine (line 3).

This means that the SACLIB run-time environment is available for use in the test harness. The test harness opens the file containing the test cases (line 5) and looping over all of the test cases (line 7). For each test case the function for the the test is read from the file and checked to make sure it is SACLIB function (lines 8-9). The return value and input/output pairs are read (lines 11-17). Then LIST2 is then invoked with the arguments from the test case (line 19) and the outputs of the LIST2 invocation are checked against the results expected by the test case (lines 19-22). The test harness completes the test case by verifying that a test case terminator is found in the input file (lines 24-26).

#### 4.10 Experimental Results

In Figure 4.14 we show the three high level test routines used on SACLIB 3.0 with our trace aspect. Although these routines were developed to exercise a wide range of SACLIB functionality with a small amount of code, the primary purpose of the routines was to demonstrate the effectiveness of the automatic code generation. Executing these routines collects 78,000 test cases. Before being stored, the tests are filtered to store only tests that execute a previously unexecuted basic block. This filtering reduces the number of stored test cases to 433 tests. These tests cover 283 of the 1100 SACLIB routines. Of the covered routines, 96 have 100% basic block coverage. An additional 47 have more than 90% coverage and an additional 87 have more that 75% coverage.

We collected these tests on an Ubuntu Linux 8 system with two 1.7 GHz AMD Opteron 244 CPUs and 4 GB of memory. When the functions in Figure 4.14 are compiled with tracing, it takes less than 15 seconds to execute them. Using all of the 433 tests as a regression suite takes less than 50 seconds. The filtering of the 78,000 test cases takes 4 days. It is possible to reduce the filtering cost, as the filtering is currently done with an unoptimized Python script. However, a large fraction of the filtering time is from running the coverage analysis tool as each test case is checked for a coverage improvement. It is not clear how much faster the filtering can be made without filtering on the coverage improvements of batches of test cases processed in a single run of the coverage tool.

There were no errors detected from the conversion from SACLIB 2.1 to SACLIB 3.0. The only “errors” detected were from differing values of uninitialized memory. Because no errors were detected, faults were explicitly added to a few routines to ensure the test harness was able to detect them. The absence of conversion errors is not unexpected. The majority of the conversion was automated [158, 157] and we have successfully used SACLIB 3.0 with the QEPCAD B [25, 26] quantifier elimination library.

```

1: void bernoulli(){
2:
3:     //Compute a Bernoulli polynomial
4:     BDigit n=10;
5:     Word A,L;
6:
7:     L = BERNOULLINUM(n);
8:     A = BERNOULLIPOL(n,L);
9:
10: }//bernoulli
11:
12: void polynomial_factoring(){
13:
14:     Word A,B,L,q;
15:     BDigit c,f,i,k,n,r,s;
16:
17:     r = 2;           //number of variables
18:     k = 2;           //bit-length of coefficients
19:     q = RNRED(1,1); //probability of nonzero term
20:     n = 2;           //total degree
21:     f = 2;           //number of factors
22:
23:     //make a polynomial from multiplication
24:     A = PFBRE(r,1);
25:     for (i=0; i<f; i++) {
26:         B = IPGTDRAN(r,k,q,n);
27:         A = IPPROD(r,A,B);
28:     }
29:
30:     //factor the polynomial
31:     IPFAC(r,A,&s,&c,&L);
32:
33: }//polynomial_factoring
34:
35: void resultant(int argc,char **argv){
36:     Word A,B,C,q;
37:     BDigit k,n,r;
38:
39:     r = 2;           //number of variables
40:     k = 2;           //bit-length of coefficients
41:     q = RNRED(1,1); //probability of nonzero term
42:     n = 2;           //total degree
43:
44:
45:     //generate two random polynomials
46:     A = IPGTDRAN(r,k,q,n);
47:     B = IPGTDRAN(r,k,q,n);
48:
49:     //Compute the resultant.
50:     C = IPRES(r,A,B);
51:
52: }//resultant

```

Figure 4.14: Routines use to exercise the automatic test generation of SACLIB 3.0.

## 5. Automatic Performance Tuning

### 5.1 Introduction

The Descartes Method [34, 117] can be used to isolate the roots of polynomials represented in both the monomial and Bernstein bases. The method was first proposed for the monomial basis [33] and later applied to the Bernstein basis [117]. Once the method is given an input polynomial, it maps the roots of the polynomial into the interval  $[0, 1)$  and recursively divides the interval until each root is in its own isolating interval. The computation can be thought of as a recursion tree, where each node represents an interval. The nodes, regardless of basis, are tested using the Descartes rule of signs to determine when they contain a single root. For a given integer polynomial, both bases have the same asymptotic computing time and produce the same recursion tree [50, 116]. To the best of our knowledge, we [102] were the first to compare the empirical computing times of the two methods [138, 102].

For both variants, the cost to process a node is primarily determined by a single sub-algorithm. For the monomial basis this sub-algorithm is the Taylor shift; for the Bernstein basis, it is de Casteljaou’s algorithm. Each of these algorithms perform a series of integer additions involving the coefficients of the input polynomial. The patterns of integer additions are given by the following recurrences:  $a_{i,j} = a_{i,j-1} + a_{i-1,j}$  (Taylor shift) and  $b_{j,i} = b_{j-1,i} + b_{j-1,i+1}$  (de Casteljaou’s). The base cases are initialized with coefficients of the input polynomial and zeros (see Figure 5.1). By using these recurrence relations as an addition schedule it is possible to directly implement both algorithms with an arbitrary precision integer arithmetic library such as GMP [74]. These implementations have cubic time complexity and depend on the quality of the multi-precision library for their performance. We call this kind of implementation the “straightforward classical method.”

This type of implementation with GMP as the multi-precision library is used by current computer algebra systems such as Maple [135], NTL [168], and Pari [1] for implementing high performance computer algebra algorithms. Using GMP is popular for two reasons. The first is that its interface allows the algorithms to be expressed solely in terms of arithmetic operations. This allows the algorithm implementor to focus on the specifics of their algorithm and delegate optimization of the multi-precision arithmetic to GMP. The second is that the GMP implementation is highly tuned. For most common platforms it is hand-implemented in assembly language. These assembly imple-

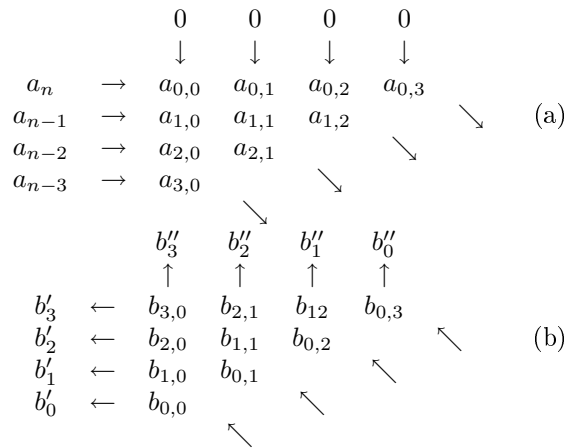


Figure 5.1: (a) The pattern of integer additions in Pascal's triangle,  $a_{i,j} = a_{i,j-1} + a_{i-1,j}$ , can be used to perform Taylor shift by 1. (b) In de Casteljau's algorithm all dependencies are reversed, the intermediate results are computed according to the recursion  $b_{j,i} = b_{j-1,i} + b_{j-1,i+1}$ .

mentations achieve better performance than compilers can obtain from high-level implementations of the same arithmetic algorithms.

Despite the extensive tuning of the GMP routines, it is still possible to obtain better performance. There are two opportunities that GMP cannot exploit. The first opportunity for optimization is the pattern of additions in an algorithm. GMP optimizes the implementation of single arithmetic operations regardless of the context of the operations. This is because the implementors of GMP only know that single multi-precision arithmetic operations are being performed. They have no knowledge of the combinations of the multi-precision arithmetic operations, or if the combination presents an additional optimization opportunity. The second opportunity for optimization GMP cannot exploit is a new platform. Because GMP obtains its performance from hand-coded assembly, clients of GMP are restricted to the set of platforms where GMP has been optimized. As new platforms appear, GMP clients must either wait for new a release or extend GMP.

Both the Taylor shift and de Casteljau's algorithm have optimization opportunities that cannot be exploited by GMP or current optimizing compilers. It is possible to use delayed carry propagation, radix reduction, and an interlaced coefficient representation to implement the Taylor shift with register tiling [103]. This implementation retains the "classical" cubic time complexity. However, the use of a multi-precision integer library is replaced with the register tiling and the performance of the implementation depends on the quality of the tiling. In the original presentation of the tiled Taylor shift method [103] speedups of 2-7 over a GMP implementation (depending on the degree of

the input polynomial) were obtained on the UltraSPARC III. We [102] later applied a similar tiling to de Casteljou's algorithm.

The original implementation of the tiled Taylor shift [103] was hand-implemented for a single tile size in 275 lines of C++ code. Some basic loop unrolling was applied, producing an implementation of 850 lines of C++ code. The tile size was selected to be a reasonable size based on the number of registers in the UltraSPARC III and the register allocation used by the C++ compiler. Although this tile size did provide a speedup, it was unclear whether this was the optimal tile size for the platform. Other tile sizes were not considered because the complexity of implementing them made experimentation with different tile sizes infeasible.

In the remainder of this chapter, we give a detailed description and analysis of code generation algorithms we developed for an experimental study of the Descartes Method [102]. We tile any given addition schedule using only square, pentagonal, and triangular tiles. By exploiting the geometry of the addition schedule, we require only a single size of each tile. This property simplifies the code generation algorithm. Our code generation algorithms allow code to be generated for arbitrary tile sizes and automatically tuned [20] to different hardware platforms. We discuss how optimization tasks can be divided between the compiler and the auto-tuner, how tiles must be represented to allow the tiling, and why it would be infeasible for a programmer to hand-implement multiple tile sizes. We then (Section 5.3) discuss the results of searching for optimal tile sizes on four different processors, and the impact the tiling has on the Descartes Method.

Although auto-tuning [20, 205, 206, 59, 60, 61, 210, 54, 55, 197] has been widely used in other areas of computer science, we believe the work presented in this chapter is the first application of auto-tuning in computer algebra. The closest related work we are aware of is the computation of crossover points for various arithmetic algorithms in GMP [73, 74]. These crossover points only depend on the quality of the implementation of the different algorithms and do not result in any code generation or hardware adaptation. The tiles we consider are more irregular than those found in most numeric applications.

## 5.2 Auto-Tuning For the Taylor shift and de Casteljou's Algorithm.

### 5.2.1 Tile Definitions

We [102] have implemented a code generator that allows different tile sizes to be automatically explored. The code generator exploits the fact that the Taylor shift and de Casteljou's algorithm

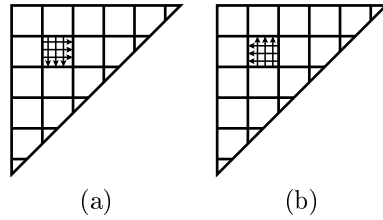


Figure 5.2: Register tiling can be applied to (a) Taylor shift and (b) de Casteljau's algorithm.

share similar computational structures. This allows them to be tiled with the same shapes, as the only difference between the two is that their dependencies are reversed (see Figure 5.2). In this section, we will define the tiles needed to implement a tiled version of Taylor shift and de Casteljau's algorithm.

The Taylor shift is computed using the recurrence  $a_{i,j} = a_{i,j-1} + a_{i-1,j}$ . We use the subscripts of the recurrence to index the elements of the corresponding Pascal's triangle used in the computation of the Taylor shift (Figure 5.1). We represent the elements of Pascal's triangle needed to compute the Taylor shift for an input polynomial  $I_n$  of degree  $n - 1$  as the set

$$A_n = \{(i, j) \in \mathbb{N} \times \mathbb{N} \mid i + j < n\}.$$

$A_n$  can be tiled using three shapes. Note that a tiling of  $A_n$  amounts to a partition of  $A_n$ . We will consider the shapes in isolation by numbering all indexes beginning with 0. Figure 5.3 gives a graphical representation of this indexing convention. Figure 5.4 gives a graphical representation of the tile shapes.

The first tile shape is a square with edge length  $e \in \mathbb{N}$ , where  $\mathbb{N}$  is the set of non-negative integers:

$$S_e = \{(i, j) \in \mathbb{N} \times \mathbb{N} \mid 0 \leq i < e \wedge 0 \leq j < e\}. \quad (5.1)$$

The second shape is a triangle:

$$T_{e,d} = \{(i, j) \in \mathbb{N} \times \mathbb{N} \mid 0 \leq i + j < d\}, \quad 0 \leq d < e. \quad (5.2)$$

For triangles  $d$  represents the number of diagonals contained in the triangle. The meaning of the bound  $e$  will be explained shortly. Note that  $T_{e,0}$  is empty. Allowing the empty triangle simplifies code generation.

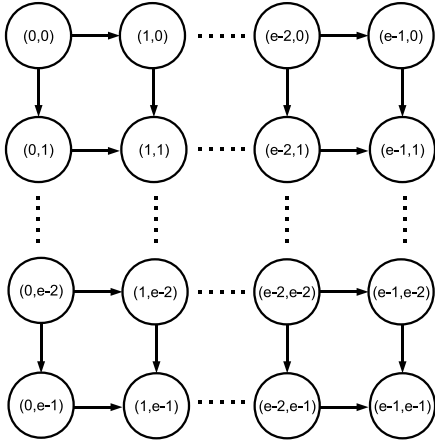


Figure 5.3: Index numbering conventions of square tile. The nodes represent a computation. The edges represent data dependencies. Each node is labeled with its index. Triangles and pentagons are numbered according to the same indexing convention, but have fewer nodes.

The third shape is a pentagon:

$$P_{e,d} = \{(i, j) \in \mathbb{N} \times \mathbb{N} \mid 0 \leq i + j < e + d \wedge i < e \wedge j < e\}, \quad 0 \leq d < e. \quad (5.3)$$

For pentagons,  $d$  represents the number of diagonals that need to be added to  $T_{e,e}$  to create the pentagon  $P_{e,d}$ . Note that we call  $P_{e,0}$  and  $P_{e,e}$  pentagons despite the fact that  $P_{e,0}$  is geometrically a triangle and  $P_{e,e}$  is geometrically a square. Calling these “pentagons” also simplifies the code generation.

For a set  $s$  and a vector  $(x, y) \in \mathbb{N} \times \mathbb{N}$  we define the following translation

$$s + (x, y) = \{i + (x, y) \mid i \in s\}.$$

Using this translation, we can place the tiles at indexes starting at multiples of  $e$  by defining

$$S_{x,y,e} = S_e + (x, y) e, \quad (5.4)$$

$$T_{x,y,e,d} = T_{e,d} + (x, y) e, \text{ and} \quad (5.5)$$

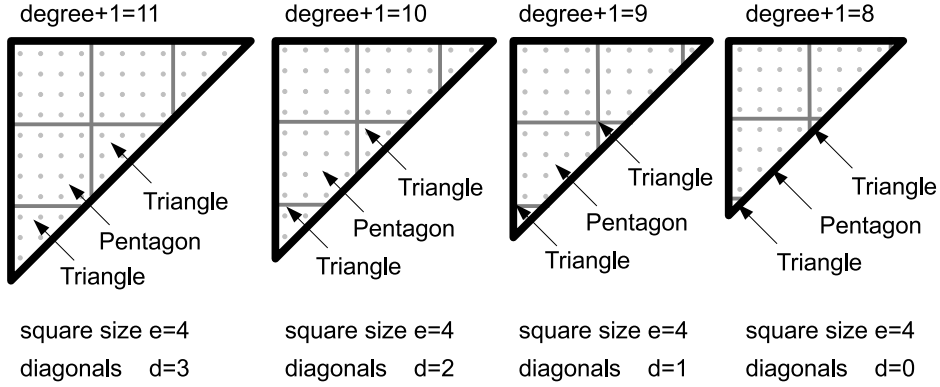


Figure 5.4: For a given polynomial degree and square tile size, there is only a single size of corresponding pentagon and triangle that can be used for tiling.

$$P_{x,y,e,d} = P_{e,d} + (x,y)e \quad (5.6)$$

For an input polynomial  $I$  of degree  $n-1$ ,  $e < \lceil \frac{n}{2} \rceil$ ,  $d = n - \lfloor \frac{n}{e} \rfloor e$ ,  $t = \lfloor \frac{n}{e} \rfloor + 1$ ,  $(x,y) \in \mathbb{N} \times \mathbb{N}$ , and  $x+y < t$ , we can use these three shapes to define a tile as

$$A_{x,y} = \begin{cases} S_{x,y,e} & \text{if } x+y < t-2 \\ P_{x,y,e,d} & \text{if } x+y = t-2 \\ T_{x,y,e,d} & \text{if } x+y = t-1 \end{cases} .$$

We restrict  $e$  to  $e < \lceil \frac{n}{2} \rceil$  because this is the largest square tile that can be used for  $A_{0,0}$ . The number of tiles on both input edges of  $A_n$  (see Figure 5.1) is given by  $t$ . The quantity  $x+y$  identifies the diagonal  $A_{x,y}$  lies on. This is why the value of  $x+y$  can be used to determine the shape of  $A_{x,y}$ . Triangles only fit on the final diagonal,  $x+y = t-1$ . Pentagons only fit on the second to last diagonal,  $x+y = t-2$ . Squares are the only shape that fit on the remaining diagonals,  $x+y < t-2$ . The reason for defining pentagons and triangles as we did is to ensure that all triangles are on the last diagonal and all pentagons are on the second to last diagonal. These geometric motivations for the tile definitions can be seen in Figure 5.4. An important consequence of this definition of the tiles is that any given Taylor shift may be tiled using only a single size of square, triangle, and pentagon.

This allows us to tile  $A_n$  as

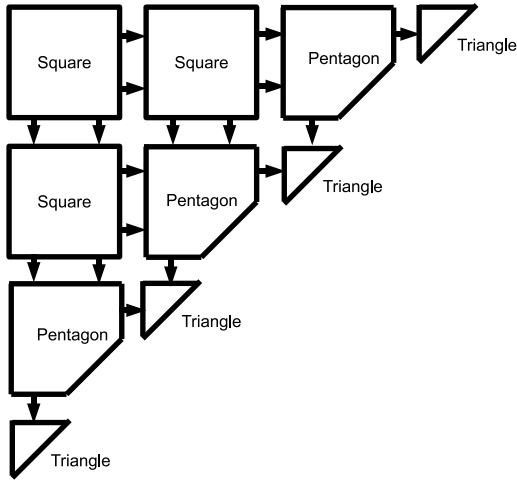


Figure 5.5: Tile shapes are defined so only the following transitions may occur. Arrows represent a data dependency between tiles.

$$A_n = \{A_{x,y} \mid x + y < t\}.$$

These tile definitions are similar to the original presentation of the tile method [103]. However, we have changed them to explicitly allow tiles shapes to be determined by the diagonal they are on. This allows an auto-tuner to only need to address the tile dependencies in Figure 5.5. We [102] also applied this tiling scheme to de Casteljau’s algorithm. It required altering the indexing presented above to match the recurrence (Figure 5.1) for the de Casteljau’s algorithm addition schedule. The techniques for defining these tiles are identical to the those we employed for the Taylor shift.

### 5.2.2 Tile Scheduling

Tiles are scheduled using a register load pass, one or more computation passes, and a register store pass. These passes are used for squares, pentagons, and triangles. The only difference between the three kinds of tiles is that relative to the square, some of the passes in a pentagon or triangle may be shorter. Because of this, we will only describe the processing of a square tile. For the initial discussion of the tile schedule we will assume there is no register spill and the compiler and CPU schedule the computation as we describe in this section. These assumptions will be removed in

### Section 5.2.3.

Figure 5.6 shows the schedule for a square  $S_{x,y,4}$  Taylor shift tile. The first pass is the register load pass. This pass is responsible for retrieving one edge of the inputs and loading them into registers. Each input loaded into a register corresponds to a single machine word from a different multi-precision coefficient of the input polynomial  $I_n$ . Because the coefficients of  $I_n$  are stored using an interlaced representation [103], they are stored in contiguous memory. The register load pass loads each coefficient digit in the order it is stored in memory. This allows the loading of the registers to benefit from any hardware prefetching that occurs as the coefficients are brought from main memory to cache. At the end of the register load pass, four registers are being used by the tile to store the loaded coefficient digits. In general, this pass will end with  $e$  registers having been loaded with values from memory.

After the register load pass, the computation passes can begin. Each computation pass is broken into three stages. The first is the compute/load stage. During this stage, each computation still requires an input coefficient digit from main memory. These inputs are also loaded in the order they are stored in memory. Once the compute/load stage is complete, the pass enters the compute stage. All nodes in the compute stage can be processed without any loads from memory. After the compute stage is completed, we enter the compute/store stage. This stage is the end of the pass. It processes the final nodes of the pass and stores the results to memory.

The entire computation pass requires only a single additional register over those used in the register load pass. The reason is that the first load from the compute/load phase requires an additional register, but once that load is completed, the loaded value can immediately be added to the other input for node 1 of the compute pass. One of the registers storing an input of node 1 is used to store the result of node 1. The other node 1 input register becomes free for loading the next input to node 3. Processing of the nodes is scheduled to allow this pattern of register usage to be applied to all nodes in the computation pass. This is why the compute stage (node 2) is started before the compute/load stage (node 3) is completed. This has the benefit that at the end of the computation pass, all of the “left edge” inputs of the next pass are in registers. Computation passes continue until all of the nodes of the tile have been computed.

After the computation passes complete, the “right edge” outputs of the tile are stored in the register store pass. This pass requires no additional registers. This means that a tile may be computed using  $e + 1$  registers. After each tile is completed, a carry propagation must be performed. This requires adding any carries produced by the tile to the inputs of subsequent tiles.

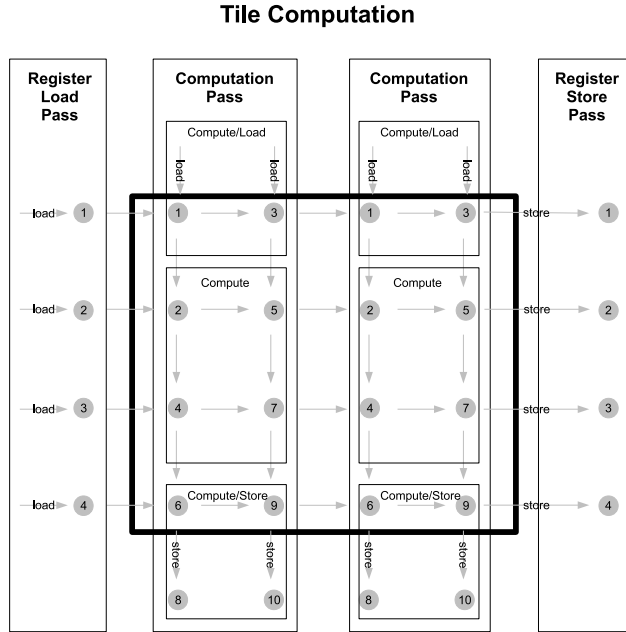


Figure 5.6: Schematic processing schedule for a  $S_{x,y,4}$  square tile. Tiles are scheduled in using three kinds of passes: 1) a single register load pass, 2) one or more computation passes, and 3) a register store pass. Arrows represent an input. Circles represent a computation involving all of the inputs to the circle. The circles in each pass are numbered in the order they are scheduled by the code generator. The  $S_{x,y,4}$  tile is enclosed in the black square. The code to process this tile for the Taylor shift is in Figure 5.7 and the code to process this tile for de Casteljou’s algorithm is in Figure 5.8.

The tiles are shaped this way to allow maximum computation for the smallest number of loads and stores. The number of additions in a tile is directly proportional to its area, while the number of loads and stores are directly proportional to its perimeter. A square is the shape that provides the best ratio of area to perimeter.

This is also why the register load pass loads an entire edge of inputs in one pass, but leaves the other loads to the compute/load stage of the computation passes. If the tile were scheduled to load all inputs before performing computation, it would require  $2e$  registers to store the inputs. But this would mean that for a machine with  $r$  registers, the largest tile that could be processed without register spill would be  $S_{x,y,\frac{r}{2}}$ . This gives an area:perimeter ratio of  $r^2 : 8r = r : 8$ . With the same number of registers, the passes described above allow a  $S_{x,y,r-1}$  tile to be processed without register spill. This gives an area:perimeter ratio of  $(r-1)^2 : 4(r-1) = (r-1) : 4$ , which is better by a factor of  $2\frac{r-1}{r}$ .

### 5.2.3 Code Generation

It is possible to implement each tile shape using loops. Current compiler technology is unable to produce good run-time performance for tiles implemented using loops. Although the performance is poor, the loop implementations are straightforward. They only require  $O(1)$  lines of code regardless of the tile sizes considered, and it is easy to verify their correctness. This makes them ideal to use in testing the correctness of more heavily optimized implementations.

In order to obtain a good speed-up, it is necessary to generate code that has explicitly unrolled the “concise” loop implementation. We accomplish this with a 1,000 line Perl script that generates fully unrolled C++ code for each tile. This full unrolling results in generating C++ code to explicitly process each node of the tile. The generator is given as input the value of  $e$  to generate tiles for. It begins by determining the related pentagons and triangles required for using a square tile with edge length  $e$ . For each of the required tiles, it generates the code for the passes in Figure 5.6. It also uses the shape and size of the pentagons and triangles to determine which phases have fewer elements. In addition to producing the code for processing the tiles, the generator produces a top-level routine that is given an input polynomial to perform the Taylor shift on. This routine is responsible for using the degree of the input polynomial to select the correct tiles to use, converting the input to an interlaced representation, and calling the tile processing routines. Calling the tile processing routines requires mapping the inputs to the indexing convention used by the tiles. Because of the interlaced coefficient representation, this consists only of adding values to the indexes of a built-in array. Because the tiles are implemented in terms of scalar variables, no additional indexing computations are needed once the code to process a tile is called.

Hand implementation of the fully unrolled tile code is infeasible because of the number of operations. Processing each tile requires  $O(e)$  memory operations and  $O(e^2)$  additions. Implementing these computations requires the proper indexing of input and output arrays and the implementation must respect the data dependencies needed for the computations. These kinds of implementations are notoriously error prone and difficult to debug. The  $O(e^2)$  scaling creates the opportunity for error greater as  $e$  increases. The use of passes provides some help here, as only  $O(e)$  passes are needed for a single tile.

For each implementation of a square  $S_{x,y,e}$ , there must be  $O(e)$  implementations of triangles and pentagons. This is because every  $S_{x,y,e}$  must be used with the pentagon  $P_{x,y,d}$  and the triangle  $T_{x,y,d}$ . For an input polynomial of arbitrary degree,  $d \in 0, 1, 2, \dots, e - 1$ . This requires  $O(e^2)$  passes

$e$ (square tile edge length)	lines of generated code
4	1,124
6	1,876
8	3,044
10	4,724
12	7,012
14	10,004
16	13,796
<b>total</b>	<b>41,580</b>

Table 5.1: Lines of code generated for Taylor shift tiles of various sizes. The analysis shows that for a square of edge length  $e$ , the number of generated lines of code is  $O(e^3)$ . The above table only presents values of  $e$  that were used for tuning. This dynamic range of  $e$  is not large enough to clearly show the  $O(e^3)$  scaling. This is expected, as the lower order terms have large constants.

to be generated for the full set of tiles needed to use a single size square. Because each pass performs  $O(e)$  operations, a single value of  $e$  requires code for  $O(e^3)$  operations to be generated. The optimal value of  $e$  is obtained from search, which means code must be generated for  $O(e^4)$  operations while searching for the optimal tile size. Our code generator emits one line for each operation. Table 5.1 shows the number lines generated code for values of  $e = 4, 6, 8, \dots, 12, 14, 16$ .

#### 5.2.4 Compiler Optimizations

One of the main goals of the code generator is to delegate as much of the optimization as possible to the compiler. Figure 5.7 has a listing of the code generated to process a  $S_{x,y,4}$  square. Lines 1-3 include files needed by the tile. The files `generated/types.h` and `generated/constants.h` are both created by the code generator. The typedef `baseint` is in `types.h`, and is used to select between 32-bit and 64-bit digits. The header `constants.h` contains a number of integer constants that are used for carry propagation. It also contains the constant `TILE_WIDTH`, which corresponds to  $e$ . This is used in the static assert (line 6) to ensure that all tiles were generated for the same value of  $e$ .

The register load pass is performed on lines 11-14. The pointer `Pz` corresponds to the right-edge inputs in Figure 5.6. The first compute pass is performed on lines 17-23 and the second compute pass is performed on lines 35-38. The pointer `Zz` corresponds to the top-edge inputs in Figure 5.6.

The generated code does not contain any assembly controlling the placement of inputs in registers. It does not even contain any suggestions (via the `register` keyword) about which values to place in registers. Memory is simply accessed from a pointer and written to the scalar variables `z1`, `z2`, `z3`, and `z4` (lines 11-14). Because the C++ code is written this way, the compiler determines where the

inputs are stored. Figure 5.9(a) has the assembly gcc produces for the register load stage for an x86 CPU. Gcc does produce code that loads each input into a register. Pz is stored in the register `%eax`, and lines 1-4 move the input integers to the registers `%edx`, `%ecx`, `%ebx`, and `%esi`. The assembly for the register store pass is in Figure 5.9(c). The outputs of the computation passes were all kept in registers, so this pass just stores the values in the registers to memory. The register `%eax` contains Zz and lines 1-4 store the values in the `%edx`, `%ecx`, `%ebx`, and `%esi` registers. Because compilers do not always detect opportunities to place array values in registers, this kind of explicit copying of array values to scalar variables is common practice [28, 207, 29, 102]. This allows the compiler to select the number and type of registers used for a given CPU.

The compiler is also allowed to schedule the order of the additions. Although the C++ code does provide an order for processing the nodes of the stages, the compiler is free to reorder the computations in the stages any way that respects the data dependencies. The assembly for the first computation stage is in Figure 5.9(b). None of the Zz inputs are placed in registers; they are used from memory during the additions on lines 1 and 4. The compiler has also reordered the additions. Although lines 1-2 and 9-10 preform the operations in the same order as the C++ code, the other operations (lines 3-8) have been reordered relative to the order in the C++ code. Despite the fact that we emitted code for a CPU with two integer execution units, this reordering has placed three addition instructions in a row (lines 7-9). Because the compiler can reorder additions, it is possible for the compiler to schedule compute stages for CPUs with more than two integer execution units.

In addition to providing CPU portability with respect to number of registers and integer execution units, generating the C++ code in this fashion frees us from worrying about the interaction of number of registers and integer execution units on the final schedule for the tile. The search over the different tile sizes allows us to select the best tile for a given CPU/compiler combination without understanding exactly how those two architectural features interact with each other.

### 5.3 Experimental Results and Discussion

This section details the experiments performed to validate the performance and portability of the tile method. The beginning of this section contains background material about

- the methods we compare against the tile method (Section 5.3.1),
- the hardware platforms experiments were conducted on (Section 5.3.2),
- the compilation and timing procedures used to collect data (Section 5.3.3 and 5.3.4), and

```

1: #include <boost/static_assert.hpp>
2: #include <generated/types.h>
3: #include <generated/constants.h>
4:
5: inline void regtile(baseint *Zz, baseint *Pz){
6:     BOOST_STATIC_ASSERT(4 == TILE_WIDTH);
7:
8:     baseint z1, z2, z3, z4;
9:
10:    //register load pass
11:        z1 = Pz[0];
12:        z2 = Pz[1];
13:        z3 = Pz[2];
14:        z4 = Pz[3];
15:
16:    //computation pass 1
17:        z1+=Zz[0]; //begin compute/load stage
18:        z2+=z1; //begin compute stage
19:        z1+=Zz[1]; //complete compute/load stage
20:        z3+=z2; z2+=z1; //continue compute stage
21:        z4+=z3; //begin compute/store stage
22:        z3+=z2; //complete compute stage
23:        Zz[0]=z4; z4+=z3; Zz[1]=z4; //complete compute/store stage
24:
25:    //computation pass 2
26:        z1+=Zz[2]; //begin compute/load stage
27:        z2+=z1; //begin compute stage
28:        z1+=Zz[3]; //complete compute/load stage
29:        z3+=z2; z2+=z1; //continue compute stage
30:        z4+=z3; //begin compute/store stage
31:        z3+=z2; //complete compute stage
32:        Zz[2]=z4; z4+=z3; Zz[3]=z4; //complete compute/store stage
33:
34:    //register store pass
35:        Pz[0] = z1;
36:        Pz[1] = z2;
37:        Pz[2] = z3;
38:        Pz[3] = z4;
39:
40: }//regtile

```

Figure 5.7: The code to process the schedule for a  $S_{x,y,4}$  Taylor shift square. See Figure 5.6 for a schematic representation of the  $S_{x,y,4}$  schedule. This code was produced by our code generator (Section 5.2.3). It was hand reformatted and commented to improve presentation. See Figure 5.8 for the corresponding de Casteljau’s tile.

```

1: #ifndef regtiledc_h
2: #define regtiledc_h
3:
4: #include <boost/static_assert.hpp>
5: #include <generated/types.h>
6: #include <generated/constants.h>
7:
8: inline void regtiledc(baseint *Zz, baseint *Pz){
9:     BOOST_STATIC_ASSERT(4 == TILE_WIDTH);
10:
11:     baseint z1, z2, z3, z4;
12:
13:     //register load pass
14:     z1 = Pz[0];
15:     z2 = Pz[1];
16:     z3 = Pz[2];
17:     z4 = Pz[3];
18:
19:     //computation pass 1
20:     z1+=Zz[0]; //begin compute/load stage
21:     z2+=z1; //begin compute stage
22:     z1+=Zz[-1]; //complete compute/load stage
23:     z3+=z2; z2+=z1; //continue compute stage
24:     z4+=z3; //begin compute/store stage
25:     z3+=z2; //complete compute stage
26:     Zz[0]=z4; z4+=z3; Zz[-1]=z4; //complete compute/store stage
27:
28:     //computation pass 2
29:     z1+=Zz[-2]; //begin compute/load stage
30:     z2+=z1; //begin compute stage
31:     z1+=Zz[-3]; //complete compute/load stage
32:     z3+=z2; z2+=z1; //continue compute stage
33:     z4+=z3; //begin compute/store stage
34:     z3+=z2; //complete compute stage
35:     Zz[-2]=z4; z4+=z3; Zz[3]=z4; //complete compute/store stage
36:
37:     //register store pass
38:     Pz[0] = z1;
39:     Pz[1] = z2;
40:     Pz[2] = z3;
41:     Pz[3] = z4;
42:
43: }//regtiledc
44:
45: #endif

```

Figure 5.8: The code to process the schedule for a  $S_{x,y,4}$  de Casteljau’s square tile. See Figure 5.6 for a schematic representation of the  $S_{x,y,4}$  schedule. This code was produced by our code generator (Section 5.2.3). It was hand reformatted and commented to improve presentation. See Figure 5.8 for the corresponding Taylor shift tile.

(a) register load pass

```

1: movl  (%eax), %edx    ; z1 = Pz[0];
2: movl  4(%eax), %ecx   ; z2 = Pz[1];
3: movl  8(%eax), %ebx   ; z3 = Pz[2];
4: movl 12(%eax), %esi   ; z4 = Pz[3];

```

(b) compute pass

```

1: addl (%edi), %edx    ; z1+=Zz[0];
2: addl %edx, %ecx     ; z2+=z1;
3: addl %ecx, %ebx     ; z3+=z2;
4: addl 4(%edi), %edx  ; z1+=Zz[1];
5: addl %ebx, %esi     ; z4+=z3;
6: movl %esi, (%edi)   ; Zz[0]=z4;
7: addl %edx, %ecx     ; z2+=z1;
8: addl %ecx, %ebx     ; z3+=z2;
9: addl %ebx, %esi     ; z4+=z3;
10: movl %esi, 4(%edi) ; Zz[1]=z4;

```

(c) register store pass

```

1: movl %edx,  (%eax)   ; Pz[0] = z1;
2: movl %ecx,  4(%eax)  ; Pz[1] = z2;
3: movl %ebx,  8(%eax)  ; Pz[2] = z3;
4: movl %esi, 12(%eax)  ; Pz[3] = z4;

```

Figure 5.9: The x86 assembly produced by gcc for the C++ code to process a  $S_{x,y,4}$  square Taylor shift tile (Figure 5.7). The assembly is shown for (a) the register load pass, (b) the first computation pass, (c) the register store pass.

- the classes of input polynomials used for the experiments (Section 5.3.5).

After presenting this background material, we present

- the impact of tile size on the performance of the tile method (Section 5.3.6) and
- the speedup to root isolation provided by the tile method (Section 5.3.7).

### 5.3.1 Evaluated Descartes Method Implementations

#### The monomial SACLIB method, IPRRID

The program IPRRID in the SACLIB library [35] processes the recursion tree in breadth-first order [115, 160]. When processing nodes in the recursion tree, IPRRID tries to avoid the complete application of a Taylor shift by stopping the Taylor shift as soon as it is clear the node is not an isolating interval. The program IUPTR1 that implements Taylor shift by 1 operates on a single array containing the coefficients of the input polynomial. IUPTR1 avoids the overhead of calling integer addition routines and of normalizing after each integer addition [103].

#### The Bernstein SACLIB method, IPRRIDB

The program IPRRIDB in the SACLIB library [35] converts the input polynomial from its monomial representation into a fraction-free Bernstein-basis representation. IPRRIDB processes the bisection tree in the same way as the program IPRRID (Section 5.3.1). It also uses the same techniques IPRRID uses to avoid the overhead of calling integer addition routines and normalizing after each integer addition.

#### The method by Hanrot et al.

Hanrot et al [83]. provide an efficient implementation of the monomial version of the Descartes method that incorporates the memory-saving technique of Rouillier and Zimmermann [160]. Their implementation uses GMP [74] for the integer additions required by Taylor shift operations. They apply many optimizations that either remove the need to apply the Taylor shift to some of the nodes in the recursion tree or to allow the Taylor shift to be partially applied. A pre-processing step is applied to determine the greatest  $k$  such that the input polynomial  $A(x)$  is a polynomial in  $x^k$ , which allows  $A(x)$  to be replaced by  $A(\sqrt[k]{x})$ . If  $k$  is even, the method isolates only the positive roots. These optimizations are effective heuristics that result in a speedup when the input polynomial produces

a recursion tree where checking for the ability to skip a Taylor shift or apply a partial Taylor shift provides a reduction in processing time that exceeds the time for the heuristic checks.

### **The SYNAPS method**

The SYNAPS [137] implementation `IslBzInteger<QQ>` [52] of the Descartes method uses GMP [74] for the integer additions required by the de Casteljaou’s operations. Otherwise, the method is a straightforward implementation of the Bernstein-bases variant. A hard-coded limitation of the recursion depth to 96 prevents the method from isolating the roots of Mignotte polynomials of degrees greater than 80.

### **The Tiled Bernstein method**

The method described in 5.2, applied to the Bernstein basis.

### **Asymptotically fast Taylor shift**

Von zur Gathen and Gerhard experimentally compared six methods for computing Taylor shifts. They implemented all methods on top of the NTL library [170, 169] and identified a divide-and-conquer method as the fastest method. We performed experiments using their implementation and confirmed that, on the Pentium EE, the Opteron, and the UltraSPARC III, the divide-and-conquer method, for degrees  $\geq 255$  and “large” coefficients, is indeed faster than the other implementations of asymptotically fast Taylor shift. We used the fastest of these implementations for comparison with the classical Taylor shift implementations.

### **5.3.2 Evaluated Processor Architectures**

The tiled method primarily achieves its speedup from delaying carries and using register tiling to improve locality of reference. The computation schedule for the register tiles allows multiple integer execution units to be used simultaneously in the processing of a register tile. When implementing the tiled method for a given processor the number of general purpose integer registers and the integer execution units determines the maximum speedup that can be obtained by the method; the speedup will be larger with a larger number of general purpose integer registers and integer execution units. We consider the following CPUs.

### 64-bit processors

Current processors such as the Pentium EE [42], Opteron [9, 8], and UltraSPARC III [88, 194] support native 64-bit integer operations, have at least 16 64-bit general purpose integer registers, and at least 2 integer execution units. These are the kind of processors for which the tiled method was developed. Below, we briefly summarize the features of these three processors.

**Pentium EE.** The Intel Pentium Extreme Edition is based on the NetBurst microarchitecture [40, 42]. The Pentium EE dual-core processor supports both the 32-bit x86 and 64-bit EM64T instruction sets. Each core of the Pentium EE provides 16 64-bit general purpose integer registers and has an 8-way set-associative 16 kilobyte L1 data cache and an 8-way set-associative 1 megabyte L2 cache. The Pentium EE is capable of executing 2 integer instructions per cycle.

**Opteron.** The Opteron processor is based on Athlon's QuantiSpeed architecture [8, 9, 10, pp. 250–252]. The Opteron supports the 32-bit x86 and 64-bit AMD64 instruction sets. The Opteron provides 16 64-bit general purpose integer registers and has a 2-way set-associative 64 kilobyte L1 data cache and a 4-way set-associative 1 megabyte L2 cache. The Opteron can execute 3 integer instructions per cycle.

**UltraSPARC III.** The Sun UltraSPARC III processor [88, 194] supports the SPARC V9 instruction set. The UltraSPARC III provides 32 64-bit general purpose integer registers and has a 64 kilobyte 4-way set-associative L1 data cache and an 8 megabyte 2-way set-associative L2 cache. The UltraSPARC III can execute 2 integer instructions per cycle.

### 32-bit processors

The Pentium 4 [41] is included for comparison only and is not expected to perform well with the tiled Bernstein method due to the small number of general purpose integer registers.

**Pentium 4.** The Intel Pentium 4 processor is based on the NetBurst microarchitecture [40, 41]. The Pentium 4 processor supports the 32-bit x86 instruction set. The Pentium 4 provides 8 32-bit general purpose integer registers and has a 16 kilobyte 8-way set-associative L1 data cache and a 1 megabyte 8-way set-associative L2 cache. The Pentium 4 can execute 2 integer instructions per cycle.

### 5.3.3 Hardware configuration and timing

The `getrusage` system call is used on all platforms to obtain timings. The hardware platforms used in this study are configured as follows.

**Pentium EE.** We use a Pentium Extreme Edition 840 Dual-Core CPU with a clock speed of 3.2 GHz and 1 GB of main memory. The Gentoo Linux distribution with the 2.6.14-gentoo-r2 kernel is installed. Hyper-Threading is disabled in the BIOS.

**Opteron.** We use an Opteron 244 with a clock speed of 1.8 GHz and 2 GB of main memory. The Gentoo Linux distribution with the 2.6.14-gentoo-r2 kernel is installed.

**UltraSPARC III.** We use a Sun Blade 2000 with two 900 MHz UltraSPARC III processors and 2 GB of main memory. The Solaris 9 operating system is installed.

**Pentium 4.** We use a Pentium 4 with a clock speed of 3.0 GHz and 1 GB of main memory. The Fedora Core 2 Linux distribution is installed.

### 5.3.4 Compilation protocol

**SACLIB monomial and SACLIB Bernstein.** The SACLIB [35] programs `IPRRID` and `IPRRIDB` are compiled using `gcc 3.4.4` with the flags “`-O3 -march=nocona -m64`” on the Pentium EE, `gcc 3.4.4` with the flags “`-O3 -march=opteron -m64`” on the Opteron, Sun Studio 9 compilers [193] with the flags “`-xO3`” on the UltraSPARC III, and `gcc 3.3.3` with the flags “`-O3 -march=pentium4`” on the Pentium 4.

**Tiled Bernstein.** The tiled Bernstein method is compiled with the same compilers and flags as the SACLIB methods. For the Opteron and Pentium EE we use a tile size of  $e = 12$ . For the UltraSPARC III we use  $e = 8$ . For the Pentium 4 we use  $e = 6$ . See section 5.3.6 for how these tile sizes were selected.

**NTL.** On the Pentium EE, Opteron, and Pentium 4, NTL 5.4 [169] is compiled with the same compilers as the SACLIB method. Compiler flags are the defaults set by NTL. On the UltraSPARC III, NTL 5.4 is compiled with the Sun Studio 9 compiler [193] with the flags “`-xO3 -xarch=v9b`”. Because of the way it performs multiplication, NTL is limited to 32-bit integer arithmetic; however, for compatibility, NTL is compiled to use the 64-bit ABI.

**GMP.** On the Pentium EE, Opteron, UltraSPARC III, and Pentium 4, GMP 4.2 [74] is compiled using the same compilers as the SACLIB method. Compiler flags are the defaults set by GMP.

**SYNAPS.** On the Pentium EE, Opteron, and Pentium 4, SYNAPS 2.4 [154, 137] is compiled

with the same compilers and flags as the SACLIB method. On the UltraSPARC III, SYNAPS 2.4 is compiled with the Sun Studio 9 C++ compiler with the flags “-x03 -xarch=v9b”. SYNAPS required minor porting before it could be compiled with the Sun Studio 9 C++ compiler.

**Hanrot et al.** The code of Hanrot et al [83]. is compiled with the same compilers and flags as SYNAPS.

### 5.3.5 Input Polynomials

In the original study of the the Descartes method, we examined three classes of input polynomials [102]. These polynomials were selected to generate recursion trees with differing shapes. Here we only present data for random polynomials with integer coefficients of absolute value less than  $2^{20}$ ; the coefficients are pseudo-randomly generated from a uniform distribution. For each degree we generate 50 random polynomials and report average computing times for degrees 100, 200, . . . , 1000. These random polynomials are sufficient to demonstrate the effectiveness of the tiling. Readers interested in the application of the Descartes method to different classes of input polynomials should refer to our previous work [102].

### 5.3.6 Taylor shift Benchmarks

In order to find the proper tile size  $e$  for the 4 CPUs examined we used our code generator to produce tiles for  $e = 2, 4, 6, 8, 10, 12, 14, 16$ . Figure 5.10 contains the results of this search. This range of tile sizes was selected because there is no processor for which the largest tile size had the best performance. Because we expect tiles to become too large to be processed without register spill, it is reasonable to expect that this range of tile sizes has allowed us to find the optimal tile size for each combination of compiler and processor.

For the Pentium EE and the Opteron the best performing tile size is  $e = 12$ . For the UltraSPARC III it is  $e = 8$  and for the Pentium 4 it is  $e = 6$ . We define best as the speedup curve having the largest integral over all input polynomial degrees. These sizes make sense. The Opteron, Pentium EE, and UltraSPARC III have a larger number of general purpose integer registers and of integer execution units. These are the two hardware features that most allow a large tile size to be effective.

Using the optimal value of  $e$  for each platform, we compared the best Tiled Taylor shift to the best asymptotically fast Taylor shift. The results are in Figure 5.11. The optimizations of the Tiled Taylor shift make it a better choice for Taylor shift until the degree of the input polynomial starts

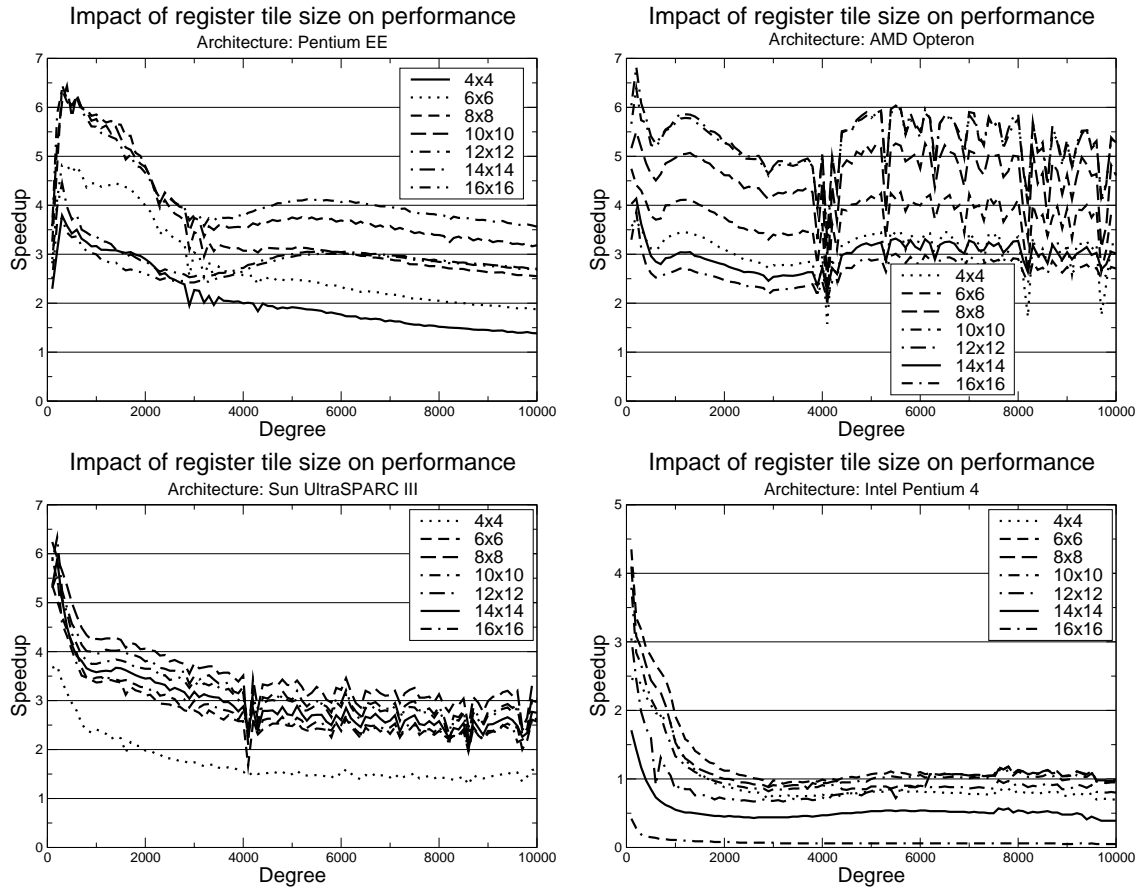


Figure 5.10: Speedup obtained by the tiled version of the Taylor shift by 1 algorithm on the Pentium EE, Opteron, UltraSPARC III, and Pentium 4 for different register tile sizes. Speedup is calculated with respect to a straightforward GMP-based implementation of Taylor shift by 1. The speedup from the worst to best tile size is more than a factor of 2.

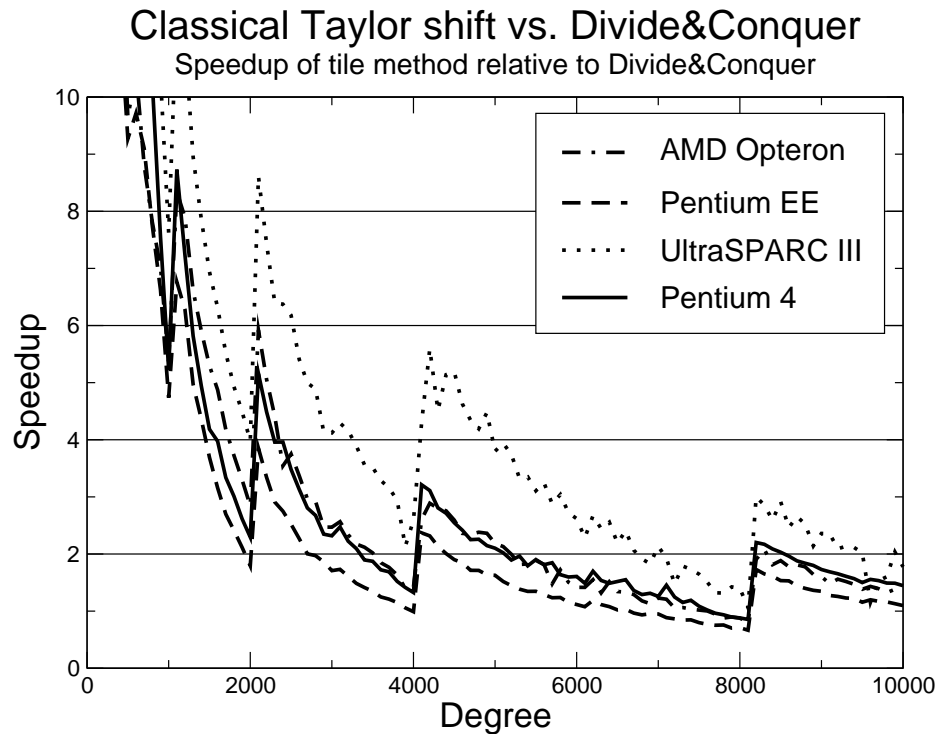


Figure 5.11: Classical Taylor shift with register tiling is faster than the fastest known asymptotically fast Taylor shift for a wide range of inputs. These speedups are obtained by using the optimal tiles size found by searching with our code generator. These speedups are obtained by using the optimal tiles size found by searching with our code generator.

to exceed 10,000.

### 5.3.7 Root Isolation Benchmarks

In the previous section, we compared the performance of the tile method relative to a straightforward implementation using GMP. In this section, we will examine the performance of the tile method when used as a sub-algorithm of the Descartes method. Because the SACLIB Bernstein method (Section 5.3.1) provides better performance than the SACLIB Monomial method (Section 5.3.1), the Tiled Bernstein method (Section 5.3.1) is the only tile method we report results for. We compare the Tiled Bernstein method to the SACLIB Bernstein method (Section 5.3.1), Hanrot's method (Section 5.3.1), and SYNAPS (Section 5.3.1). When plotting results, all execution times are normalized by dividing by the execution time of the SACLIB Monomial implementation (Section 5.3.1). This allows all results to be reported as speedups.

In Figure 5.12, results are presented for the random polynomials described in Section 5.3.5. The

Tiled Bernstein is never the best method on the Pentium 4. This is not surprising. The Pentium 4 has only 8 registers and this results in the smallest optimal tile size,  $e = 6$ , among all of the CPUs. There are not enough registers for the tile method to outperform GMP. For most degrees, tiling does not provide a significant speedup over GMP (Figure 5.10). The SYNAPS and Hanrot methods use optimizations that are not very sensitive to the number of registers in the CPU, which allows them to outperform the Tiled Bernstein method on the Pentium 4.

On the UltraSPARC III, the Tiled Bernstein method is frequently outperformed by the Hanrot and SYNAPS methods. This appears to be because of the inability of the Sun Studio compiler to utilize all of the registers on the UltraSPARC III. Despite the fact that there are 32 general purpose integer registers, the optimal tile size is only  $e = 8$ . It is believed [103] that the Sun Studio compiler will not place more than 8 inputs into registers. This causes larger tiles to require register spills while they are being processed. It seems likely that if a larger number of registers could be used to process a single tile, the Tiled Bernstein method would perform better on the UltraSPARC III. The auto-tuner may be over-relying on the compiler for register allocation.

On the Opteron and Pentium EE, the Tiled Bernstein method outperforms all of the other methods. Both of these CPUs have 16 registers, and the optimal tile size is  $e = 12$ . This number of registers is enough for the speedup from the tiling to outperform the optimizations of the other methods.

## 5.4 Conclusions

Using the tile method is an effective way to improve the speed of Taylor shift, de Casteljou's algorithm, and the Descartes Method. The tiling provides good speedup on modern CPU architectures. The key to being able to obtain this speedup on multi architectures is the use of auto-tuning. Because of the complexity of the tile implementation and the inability to determine the optimal tile size without search, it is infeasible to perform these kinds of optimizations without the ability to perform code generation. This is the first demonstration that auto-tuning may successfully be applied to computer algebra.

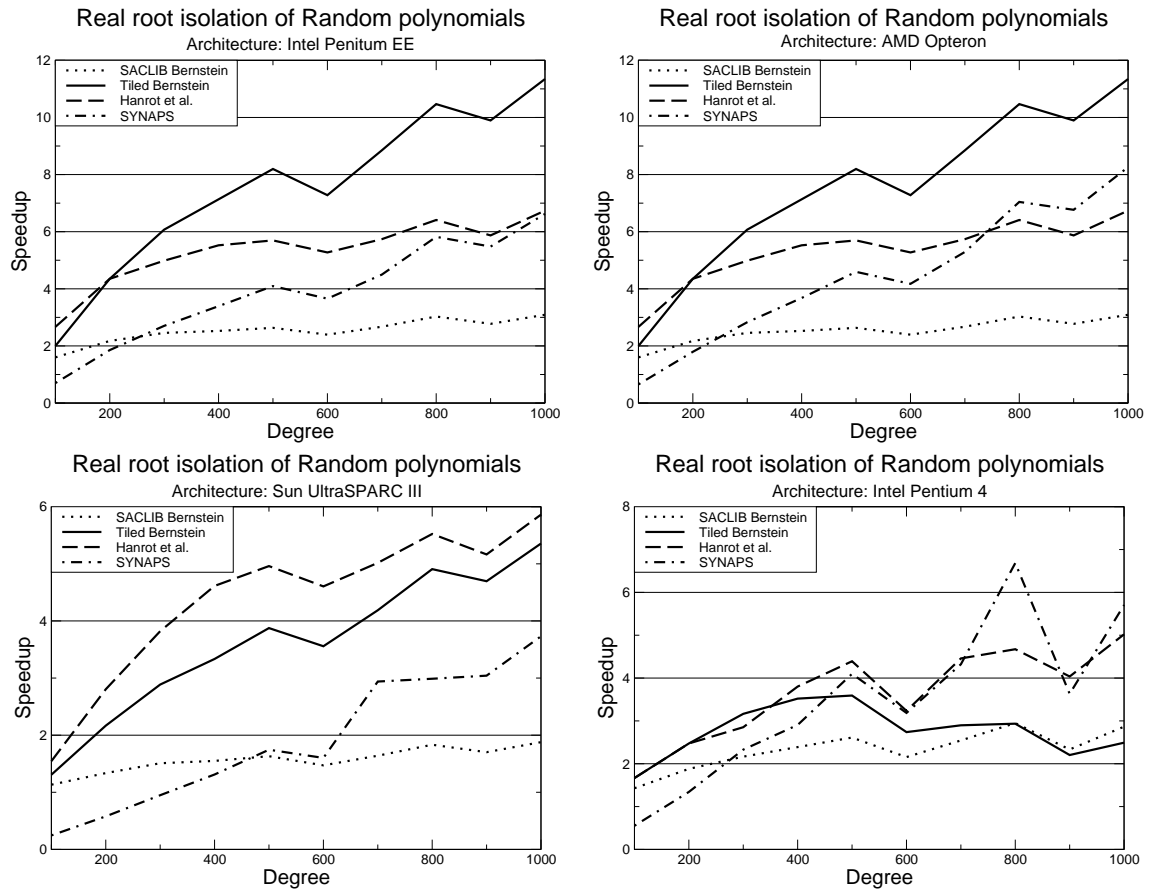


Figure 5.12: Speedup with respect to the monomial SACLIB implementation for random polynomials on four architectures. These speedups are obtained by using the optimal tiles size found by searching with our code generator.

## 6. Conclusions

In many ways, the research presented here is an investigation of the ways metaprogramming can be applied to the software development process. All of the auto-tuning and template metaprogramming is performed in an *ad hoc* manner because there is no direct language support for these these activities. This results in three major limitations for using these techniques in software development.

The first is that any implementation using these techniques is technically demanding. Template metaprogramming requires significant expertise in C++ and generic programming. Auto-tuning requires a similar level of expertise in existing compiler optimizations and the performance characteristics of modern CPU architectures. This level of expertise is required to even attempt applying these techniques. Due to the time required to build this level of specialized knowledge, the majority of programmers will not be able to employ either of these techniques.

The second limitation is that the techniques are very time consuming to apply. Both template metaprograms and the code generated for auto-tuning are difficult to implement and debug. They require the use of at least two programming languages at one time. There are no tools like debuggers for template metaprograms or the interaction between an auto-tuner and its generated code.

The most significant limitation is that there is very little in current programming languages or development tools that suggests the use of metaprogramming for problems where it is an appropriate solution. Without direct tool or language support it is extremely unlikely that these techniques will ever be used by those unfamiliar with the computer science, and in particular scientific computing, literature. This is unfortunate, because there are many programmers who would benefit from the use of these types of programming techniques.

For automated test generation, our use of aspects addresses all of these limitations. Formulating the automatic collection of unit tests as an aspect oriented programming problem allows a direct expression of the needed code generation and program instrumentation in the AspectC++ language. This allows a concise implementation and allows programmers to enjoy automated test generation without knowledge of grammars, language parsing, tree rewriting, or source-to-source translation. This ease of use is only possible because of the direct language support provided by AspectC++.

The memory safety provided by models of the Recursively Fixed Iterator and Structure Protecting Iterator Concepts is only possible because the template metaprograms could be used to prevent memory structure from being modified in unsafe ways. It was necessary to allow memory structure

to be modified during allocation and deallocation. Only preventing the memory structure from being modified in dangerous parts of the code was one of the most difficult parts of the implementation. Because the default behavior of C++ was to allow all memory structure to be modified, the metaprograms needed to selectively remove the ability for any side effects that could modify the memory structure.

It is clear that there are other kinds of safety properties that could be enforced through the selective disabling of side effects. Consider being able to call a function knowing from its declaration that it was incapable of accessing global variables, accessing variables visible in another thread, or allocating memory. Being able to express these kinds of constraints directly in a programming language with a compiler that enforces the constraints would allow many classes of programming defects to be entirely eliminated at compile-time. Such a language would allow programmers to control where their code should sit along the spectrum between the absence of side effects in pure functional languages and the unrestricted side effects of imperative languages. Allowing this directly in a programming language would require the development of a useful categorization of different types of side effects and exposing the categories through a useful syntax for controlling the side effects allowed in a given scope. We believe the most important continuation of the work we have presented is the research required to allow the scoped control of side effects to become a first class element of mainstream programming languages.



## Bibliography

- [1] *PARI/GP*. Bordeaux, 2003. <http://pari.math.u-bordeaux.fr/>.
- [2] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming*. Addison-Wesley, 2005.
- [3] David Abrahams, Jeremy Siek, and Thomas Witt. Boost.IteratorAdaptor. [http://www.boost.org/doc/libs/1\\_35\\_0/libs/iterator/doc/iterator\\_adaptor.html](http://www.boost.org/doc/libs/1_35_0/libs/iterator/doc/iterator_adaptor.html).
- [4] Adobe Systems, Inc., David Abrahams, Steve Cleary, Beman Dawes, Aleksey Gurtovoy, Howard Hinnant, Jesse Jones, Mat Marcusand, Itay Maman, John Maddock, Alexander Nasonov, Thorsten Ottosen, Robert Ramey, and Jeremy Siek. Boost.TypeTraits. [http://boost.org/doc/libs/1\\_35\\_0/libs/type\\_traits/doc/html/index.html](http://boost.org/doc/libs/1_35_0/libs/type_traits/doc/html/index.html).
- [5] Francisco Afonso, Carlos Silva, Sergio Montenegro, and Adriano Tavares. Applying aspects to a real-time embedded operating system. In *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, page 1, New York, NY, USA, 2007. ACM Press.
- [6] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.
- [7] Andrei Alexandrescu, Richard Sposato, and Peter Kuemmel. Loki. <http://sourceforge.net/projects/loki-lib>.
- [8] AMD. AMD Eighth-Generation Processor Architecture. [http://www.amd.com/us-en/assets/content\\_type/white\\_papers\\_and\\_tech\\_docs/Hammer\\_architecture\\_WP\\_2.pdf](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/Hammer_architecture_WP_2.pdf), October 2001.
- [9] AMD. Processor Reference. [http://www.amd.com/us-en/assets/content\\_type/white\\_papers\\_and\\_tech\\_docs/23932.pdf](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23932.pdf), June 2004.
- [10] AMD. *Software Optimization Guide for AMD64 Processors*, September 2005.
- [11] American National Standards Institute, <http://ansi.org/>. *Information Technology - Programming Language - Common Lisp (formerly ANSI X3.226-1994 (R1999))*, 1999.
- [12] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1998.
- [13] Matthew H. Austern, Ross A. Towle, and Alexander A. Stepanov. Range partition adaptors: a mechanism for parallelizing STL. *SIGAPP Applied Computing Review*, 4(1):5–6, 1996.
- [14] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
- [15] Purushotham V. Bangalore. Generating parallel applications for distributed memory systems using aspects, components, and patterns. In *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, page 3, New York, NY, USA, 2007. ACM Press.
- [16] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 114–124, New York, NY, USA, 2001. ACM Press.

- [17] G. Berti. GrAL - The Grid Algorithms Library. In P.M.A. Sloot, C.J. Kenneth Tan, J.J. Dongarra, and A.G. Hoekstra, editors, *Computational Science - ICCS 2002: International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part III*, volume 2331 of *Lecture Notes in Computer Science*, pages 745–754. Springer-Verlag, 2002.
- [18] Enrico Bertolazzi and Gianmarco Manzini. Algorithm 817 P2MESH: generic object-oriented interface between 2-D unstructured meshes and FEM/FVM-based PDE solvers. *ACM Transactions on Mathematical Software*, 28(1):101–132, 2002.
- [19] Danilo Beuche and Olaf Spinczyk. Variant management for embedded software product lines with pure::consul and AspectC++. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 108–109, New York, NY, USA, 2003. ACM Press.
- [20] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM.
- [21] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [22] Holger Bischof, Sergei Gorlatch, and Roman Leshchinskiy. Generic Parallel Programming Using C++ Templates and Skeletons. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 107–126. Springer-Verlag, 2004.
- [23] Hans-J. Boehm. The space cost of lazy reference counting. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 210–219. ACM Press, 2004.
- [24] Boost. Boost Libraries and Documentation. <http://www.boost.org>.
- [25] Christopher W. Brown. QEPCAD B: A program for computing with semi-algebraic sets using CADs. *SIGSAM Bulletintin*, 37(4):97–108, 2003.
- [26] Christopher W. Brown. QEPCAD B: A system for computing with semi-algebraic sets via cylindrical algebraic decomposition. *SIGSAM Bulletintin*, 38(1):23–24, 2004.
- [27] Dov Bulka and David Mayhew. *Efficient C++: Performance Programming Techniques*. Addison-Wesley, 1999.
- [28] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 53–65, New York, NY, USA, 1990. ACM.
- [29] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Not.*, 39(4):328–342, 2004.
- [30] B. D. Chaudhary and H. V. Sahasrabudde. Suggestions about a specification technique. *SIGPLAN Notices*, 13(12):25–28, 1978.
- [31] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectC to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98, New York, NY, USA, 2001. ACM.

- [32] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. *SIGPLAN Notices*, 30(6):279–290, 1995.
- [33] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Brakhage, editor, *Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer-Verlag, Berlin, 1975. Reprinted (with corrections by the author) in: B. F. Caviness and J. R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Springer-Verlag, 1998, pages 85–121.
- [34] George E. Collins and Alkiviadis G. Akritas. Polynomial real root isolation using Descartes’ rule of signs. In R. D. Jenks, editor, *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 272–275. ACM Press, 1976.
- [35] George E. Collins et al. SACLIB User’s Guide. Technical Report 93-19, Research Institute for Symbolic Computation, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria, 1993.
- [36] George E. Collins et al. SACLIB User’s Guide. Technical Report 93-19, Research Institute for Symbolic Computation, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria, 1993.
- [37] George E. Collins and Hoon Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, 1991. Reprinted in: B. F. Caviness, J. R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Springer-Verlag, 1998, pages 174–200.
- [38] Greg Colvin, Beman Dawes, Peter Dimov, and Darin Adler. Boost.SmartPointer. [http://www.boost.org/libs/smart\\_ptr/smart\\_ptr.htm](http://www.boost.org/libs/smart_ptr/smart_ptr.htm).
- [39] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *PLDI ’03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244, New York, NY, USA, 2003. ACM Press.
- [40] Intel Corporation. *A Detailed Look Inside the Intel NetBurst Micro-Architecture of the Intel Pentium 4 Processor*, November 2000.
- [41] Intel Corporation. *The IA-32 Intel Architecture Optimization: Reference Manual*, 2004.
- [42] Intel Corporation. *Intel Pentium D Processor 800 Sequence: Datasheet*, 2006.
- [43] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [44] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative Programming and Active Libraries. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming: International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 2000.
- [45] James C. Dehnert and Alexander Stepanov. Fundamentals of Generic Programming. In M. Jazayeri, R.G.K. Loos, and D.R. Musser, editors, *Generic Programming: International Seminar on Generic Programming, Dagstuhl Castle, Germany, April/May 1998. Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 2000.
- [46] The LinBox Developers. LinBox. <http://www.linalg.org/>.
- [47] Gennadiy Donchyts and Mark Zheleznyak. Object-Oriented Framework for Modelling of Pollutant Transport in River Network. In *Computational Science – ICCS 2003*, volume 2657 of *Lecture Notes in Computer Science*, pages 35–44. Springer-Verlag, 2003.

- [48] Alan Donovan, Adam Kieżun, Matthew S. Tschantz, and Michael D. Ernst. Converting java programs to use generic libraries. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 15–34, New York, NY, USA, 2004. ACM Press.
- [49] J.G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B.D. Saunders, W.J. Turner, and G. Villard. LinBox: A Generic Library for Exact Linear Algebra. In Arjeh M Cohen, Xiao-Shan Gao, and Nobuki Takayama, editors, *Proceedings of the First International Congress of Mathematical Software*. World Scientific, 2002.
- [50] Arno Eigenwillig, Vikram Sharma, and Chee K. Yap. Almost tight recursion tree bounds for the Descartes method. In J.-G. Dumas, editor, *International Symposium on Symbolic and Algebraic Computation*. ACM Press, 2006. To appear.
- [51] Electric Fence Developers. Electric Fence. <http://www.pf-lug.de/projekte/haya/efence.php>.
- [52] I. Z. Emiris, B. Mourrain, and E. Tsigaridas. Real algebraic numbers: Complexity analysis and experimentations. Research Report 5897, INRIA, 2006.
- [53] Úlfar Erlingsson, Erich Kaltofen, and David Musser. Generic Gram-Schmidt orthogonalization by exact division. In *ISSAC '96: Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, pages 275–282, New York, NY, USA, 1996. ACM Press.
- [54] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of MPI collective communication routines. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402, New York, NY, USA, 2005. ACM.
- [55] Ahmad Faraj, Xin Yuan, and David Lowenthal. STAR-MPI: self tuned adaptive routines for MPI collective operations. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 199–208, New York, NY, USA, 2006. ACM.
- [56] Efi Fogel, Ron Wein, and Dan Halperin. Code Flexibility and Program Efficiency by Genericity: Improving Cgal Arrangements. In Susanne Albers and Tomasz Radzik, editors, *Algorithms - ESA 2004: 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004. Proceedings*, volume 3221 of *Lecture Notes in Computer Science*, pages 664–676. Springer-Verlag, 2004.
- [57] A. Fothi, J. Nyeky-Gaizler, and Z. Porkolab. The structured complexity of object-oriented programs. *Mathematical and Computer Modelling*, 38(7–9):815–827, 2003.
- [58] The Eclipse Foundation. AspectJ. <http://www.eclipse.org/aspectj/>.
- [59] Matteo Frigo. A fast Fourier transform compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 169–180, New York, NY, USA, 1999. ACM.
- [60] Matteo Frigo. A fast Fourier transform compiler. *SIGPLAN Notices*, 39(4):642–655, 2004.
- [61] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.
- [62] Andreas Gal, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. On Aspect-Oriented Distributed Real-time Dependable Systems. In *Proceedings of the Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)*, pages 261–267, 2002.

- [63] Harald Gall, Mehdi Jazayeri, and René Klösch. Research directions in software reuse: where to go from here? In *SSR '95: Proceedings of the 1995 Symposium on Software Reusability*, pages 225–228, New York, NY, USA, 1995. ACM Press.
- [64] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [65] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 115–134, New York, NY, USA, 2003. ACM Press.
- [66] Ronald Garcia and Andrew Lumsdaine. MultiArray: a C++ library for generic programming with arrays. *Software: Practice and Experience*, 35(2):159–188, 2005.
- [67] Jens Gerlach and Joachim Kneis. Generic Programming for Scientific Computing in C++, Java, and C#. In *Advanced Parallel Processing Technologies*, volume 2834 of *Lecture Notes in Computer Science*, pages 301–310. Springer-Verlag, 2003.
- [68] GNU. The GNU Compiler Collection. <http://gcc.gnu.org/>.
- [69] Michael Gong, Charles Zhang, , Vinod Muthusamy, Flannan Lo, and Hans-Arno-Jacobsen. AspectC. <http://www.aspectc.net/>.
- [70] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The*. Addison-Wesley, 3rd edition, 2005.
- [71] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):1–25, 2008.
- [72] Paul Graham. *ANSI Common LISP*. Prentice Hall, 1995.
- [73] Torbjörn Granlund. *GNU MP: The GNU Multiple Precision Arithmetic Library*. Swox AB, September 2004. Edition 4.1.4.
- [74] Torbjörn Granlund. *GNU MP: The GNU Multiple Precision Arithmetic Library*. Swox AB, March 2006. Edition 4.2.
- [75] Douglas Gregor. ConceptGCC: Concept extensions for C++. <http://www.generic-programming.org/software/ConceptGCC>, 2005.
- [76] Douglas Gregor. Frequently Asked Questions: Concepts in C++. <http://www.generic-programming.org/faq/?category=conceptcxx>, 2006.
- [77] Douglas Gregor, Jaakko Järvi, Mayuresh Kulkarni, Andrew Lumsdaine, David Musser, and Sibylle Schupp. Generic Programming and High-Performance Libraries. *International Journal of Parallel Programming*, 33(2–3):145–164, 2005.
- [78] Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 423–437, New York, NY, USA, 2005. ACM Press.
- [79] Douglas Gregor and Jeremy Siek. N1848: Implementing Concepts. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1848.pdf>.
- [80] John V. Guttag, Ellis Horowitz, and David R. Musser. The design of data type specifications. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 414–420, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

- [81] John V. Guttag, Ellis Horowitz, and David R. Musser. Some extensions to algebraic specifications. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 63–67, 1977.
- [82] John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, 1978.
- [83] Guillaume Hanrot, Fabrice Rouillier, Paul Zimmermann, and Sylvain Petitjean. Uspensky’s algorithm. <http://www.loria.fr/equipes/vegas/qi/usp/usp.c>, 2004.
- [84] Matthew Harren and George C. Necula. Lightweight Wrappers for Interfacing with Binary Code in CCured. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems: Second Next-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003. Revised Papers*, volume 3233 of *Lecture Notes in Computer Science*, pages 209–225. Springer-Verlag, 2004.
- [85] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *OOPSLA ’05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 313–326, New York, NY, USA, 2005. ACM.
- [86] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 143–153, New York, NY, USA, 2005. ACM.
- [87] Hoon Hong, Andreas Neubacher, and Wolfgang Schreiner. The Design of the SACLIB/PACLIB Kernels. *Journal of Symbolic Computation*, 19(1–3):111–132, 1995.
- [88] Tim Horel and Gary Lauterbach. UltraSPARC-III: Designing third-generation 64-bit performance. *IEEE MICRO*, 19(3):73–85, 1999.
- [89] IBM Corporation. Rational Purify. <http://www-306.ibm.com/software/awdtools/purify/>.
- [90] Innovative Computing Laboratory. PAPI. <http://icl.cs.utk.edu/PAPI>.
- [91] British Standards Institute. *The C Standard: Incorporating Technical Corrigendum 1*. John Wiley and Sons, September 2002.
- [92] Intel Corporation. The Intel C++ Compiler. <http://www.intel.com/software/products/compilers/clin/>.
- [93] International Standards Organization, <http://www.iso.org>. *ISO/IEC 9899:1999: Programming languages—C*, 1999.
- [94] International Standards Organization, <http://www.iso.org>. *ISO/IEC 14882:2003: Programming languages—C++*, 2003.
- [95] Jaakko Järvi, Gary Powell, and Andrew Lumsdaine. The Lambda Library: unnamed functions in C++. *Software: Practice and Experience*, 33(3):259–291, 2003.
- [96] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-Controlled Polymorphism. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering: Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, volume 2830 of *Lecture Notes in Computer Science*, pages 228–244. Springer-Verlag, 2003.

- [97] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, New York, NY, USA, 2005. ACM Press.
- [98] Mehdi Jazayeri and Georg Trausmuth. Design concepts as basis for organizing software catalogs. In *SAC '96: Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 558–564, New York, NY, USA, 1996. ACM Press.
- [99] M. Jiménez, J. M. Llabería, A. Fernández, and E. Morancho. A general algorithm for tiling the register level. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 133–140, New York, NY, USA, 1998. ACM.
- [100] Marta Jiménez, José M. Llabería, and Agustín Fernández. Register Tiling in Nonrectangular Iteration Spaces. *ACM Trans. Program. Lang. Syst.*, 24(4):409–453, 2002.
- [101] Elizabeth Johnson and Dennis Gannon. HPC++: experiments with the parallel standard template library. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, pages 124–131, New York, NY, USA, 1997. ACM Press.
- [102] Jeremy R. Johnson, Werner Krandick, Kevin Lynch, David G. Richardson, and Anatole D. Ruslanov. High-performance implementations of the Descartes method. In J.-G. Dumas, editor, *International Symposium on Symbolic and Algebraic Computation*, pages 154–161. ACM Press, 2006.
- [103] Jeremy R. Johnson, Werner Krandick, and Anatole D. Ruslanov. Architecture-aware classical Taylor shift by 1. In M. Kauers, editor, *International Symposium on Symbolic and Algebraic Computation*, pages 200–207. ACM Press, 2005.
- [104] Nicolai M. Josuttis. *The C++ Standard Library*. Addison-Wesley, 1999.
- [105] Erich Kaltofen, Dmitriy Morozov, and George Yuhasz. Generic matrix multiplication and memory management in LinBox. In *ISSAC '05: Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*, pages 216–223, New York, NY, USA, 2005. ACM Press.
- [106] D. Kapur, D. R. Musser, and A. A. Stepanov. Operators and algebraic structures. In *FPCA '81: Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 59–64, New York, NY, USA, 1981. ACM Press.
- [107] Steve Karmesin, Scott Haney, Bill Humphrey, Julian Cummings, Tim Williams, Jim Crotinger, Stephen Smith, and Eugene Gavrilov. Parallel Object-Oriented Methods and Applications. <http://acts.nersc.gov/pooma/>.
- [108] Dimple Kaul and Aniruddha Gokhale. Middleware specialization using aspect oriented programming. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 319–324, New York, NY, USA, 2006. ACM.
- [109] Christopher E. Kees and Cass T. Miller. C++ implementations of numerical methods for solving differential-algebraic equations: design and optimization considerations. *ACM Transactions on Mathematical Software*, 25(4):377–403, 1999.
- [110] William E. Kempf. Boost.Threads. <http://www.boost.org/doc/html/threads.html>.
- [111] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, 2001.

- [112] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.
- [113] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videria Lopes, Jean-Marc Loingtier, and John Irwin. An Overview of AspectJ. In M. Aksit and S. Matsuoka, editors, *proc. 13th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 2000.
- [114] Robert Klarer, John Maddock, Beman Dawes, and Howard Hinnant. N1720: Proposal to Add Static Assertions to the Core Language (Revision 3). <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html>, 2004.
- [115] Werner Krandick. Isolierung reeller Nullstellen von Polynomen. In J. Herzberger, editor, *Wissenschaftliches Rechnen*, pages 105–154. Akademie Verlag, Berlin, 1995.
- [116] Werner Krandick and Kurt Mehlhorn. New bounds for the Descartes method. *Journal of Symbolic Computation*, 41(1):49–66, 2006.
- [117] Jeffrey M. Lane and R. F. Riesenfeld. Bounds on a polynomial. *BIT*, 21(1):112–117, 1981.
- [118] Weiguo Liu and Bertil Schmidt. A Generic Parallel Pattern-Based System for Bioinformatics. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference, Pisa, Italy, August 31- September 3, 2004. Proceedings*, volume 3149 of *Lecture Notes in Computer Science*, pages 989–996. Springer-Verlag, 2004.
- [119] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic Advice: On the Combination of AOP with Generative Programming in AspectC++. In *Proceedings of GPCE'04*, 2004.
- [120] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. *SIGOPPS Operating Systems Review*, 40(4):191–204, 2006.
- [121] Daniel Lohmann and Olaf Spinczyk. On Typesafe Aspect Implementations in C++. In *Proceedings of Software Composition (SC 2005)*, 2005.
- [122] Daniel Lohmann and Olaf Spinczyk. Developing embedded software product lines with AspectC++. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 740–742, New York, NY, USA, 2006. ACM.
- [123] Daniel Lohmann, Jochen Streicher, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Interrupt synchronization in the CiAO operating system: experiences from implementing low-level system policies by AOP. In *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, page 6, New York, NY, USA, 2007. ACM.
- [124] Rüdiger Loos. Algebraic algorithm descriptions as programs. *SIGSAM Bulletin*, (23):16–24, 1972.
- [125] A. Lumsdaine, J. Siek, and L.-Q. Lee. The Matrix Template Library. <http://www.osl.iu.edu/research/mtl/>.
- [126] John Maddock. Boost.StaticAssert. [http://boost.org/doc/html/boost\\_staticassert.html](http://boost.org/doc/html/boost_staticassert.html).

- [127] Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In L. Bacellar, P. Puschner, and S. Hong, editors, *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 249–256. IEEE Computer Society Press, 2002.
- [128] Vincent Le Maout. Cursors. In S. Yu and A. Pun, editors, *Implementation and Application of Automata : 5th International Conference, CIAA 2000, London, Ontario, Canada, July 24-25, 2000, Revised Papers*, volume 2088 of *Lecture Notes in Computer Science*, pages 195–207. Springer-Verlag, 2000.
- [129] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *ICS '00: Proceedings of the 14th International Conference on Supercomputing*, pages 88–99, New York, NY, USA, 2000. ACM Press.
- [130] Yukihiro Matsumoto. *Ruby In A Nutshell*. O'Reilly, 2001.
- [131] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.
- [132] Brian McNamara and Yannis Smaragdakis. Functional programming in C++. In *ICFP '00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 118–129, New York, NY, USA, 2000. ACM Press.
- [133] Brian McNamara and Yannis Smaragdakis. Functional programming in C++ using the FC++ library. *SIGPLAN Notices*, 36(4):25–30, 2001.
- [134] Scott Meyers. *Effective STL*. Addison-Wesley, 2001.
- [135] Michael B. Monagan, Keith O. Geddes, K. Michael Heal, George Labahn, Stefan M. Vorkoetter, James McCarron, and Paul DeMarco. *Maple 10 Programming Guide*. Maplesoft, Waterloo ON, Canada, 2005.
- [136] Gabriel A. Moreno. Creating custom containers with generative techniques. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 29–38, New York, NY, USA, 2006. ACM.
- [137] B. Mourrain, J. P. Pavone, P. Trébuchet, and E. Tsigaridas. SYNAPS: A library for symbolic-numeric computation. Software presentation. MEGA 2005, Sardinia, Italy, May 2005. <http://www-sop.inria.fr/galaad/logiciels/synaps/>.
- [138] Bernard Mourrain, Fabrice Rouillier, and Marie-Françoise Roy. The Bernstein basis and real root isolation. In J. E. Goodman, J. Pach, and E. Welzl, editors, *Combinatorial and Computational Geometry*, volume 52 of *Mathematical Sciences Research Institute Publications*, pages 459–478. Cambridge University Press, 2005.
- [139] David Musser, Sibylle Schupp, and Rüdiger Loos. Requirement Oriented Programming Concepts, Implications, and Algorithms. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming: International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 12–24. Springer-Verlag, 2000.
- [140] David R. Musser. Introspective sorting and selection algorithms. *Software: practice and experience*, 27(8):983–993, 1997.
- [141] David R. Musser and Atul Saini. *STL Tutorial & Reference Guide: C++ Programming With the Standard Template Library*. Addison-Wesley, 1996.

- [142] David R. Musser and Zhiqing Shao. Concept Use or Concept Refinement: An Important Distinction in Building Generic Specifications. In C. George and H. Miao, editors, *Formal Methods and Software Engineering: 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002. Proceedings*, volume 2495 of *Lecture Notes in Computer Science*, pages 132 – 143. Springer-Verlag, 2002.
- [143] David R. Musser and Alexander A. Stepanov. A library of generic algorithms in Ada. In *SIGAda '87: Proceedings of the 1987 Annual ACM SIGAda International Conference on Ada*, pages 216–225, New York, NY, USA, 1987. ACM Press.
- [144] GC Necula, J Condit, M Harren, S McPeak, and W Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.
- [145] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, New York, NY, USA, 2002. ACM Press.
- [146] Gor V. Nishanov and Sibylle Schupp. Garbage collection in generic libraries. In *ISMM '98: Proceedings of the 1st International Symposium on Memory Management*, pages 86–96, New York, NY, USA, 1998. ACM Press.
- [147] OpenEye Scientific Software. OEChem. <http://www.eyesopen.com/products/toolkits/ochem.html>.
- [148] Zhelong Pan and Rudolf Eigenmann. PEAK—a fast and effective performance tuning system via compiler optimization orchestration. *ACM Transactions on Programming Languages and Systems*, 30(3):1–43, 2008.
- [149] William E. Perry. *Effective Methods for Software Testing*. John Wiley & Sons, third edition, 2006.
- [150] Open Directory Project. The Open Directory Project. [http://dmoz.org/Computers/Programming/Languages/Garbage\\_Collected/](http://dmoz.org/Computers/Programming/Languages/Garbage_Collected/).
- [151] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”*, 93(2):232–275, 2005.
- [152] Richard Rasala. A model C++ tree iterator class for binary search trees. In *SIGCSE '97: Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, pages 72–76, New York, NY, USA, 1997. ACM Press.
- [153] Reginaldo Ré, Otávio Augusto Lazzarini Lemos, and Paulo Cesar Masiero. Minimizing stub creation during integration test of aspect-oriented programs. In D. Xu and R. T. Alexander, editors, *Proceedings of the Third Workshop on Testing Aspect-Oriented Programs*, pages 1–6. ACM Press, 2007.
- [154] G. Dos Reis, B. Mourrain, F. Rouillier, and Ph. Trébuchet. SYNAPS. <http://www-sop.inria.fr/galaad/software/synaps/>.
- [155] André Restivo and Ademar Aguiar. Towards detecting and solving aspect conflicts and interferences using unit tests. In *Proceedings of the Fifth Workshop on Software Engineering Properties of Languages and Aspect Technologies*. Article no. 7, 5 pp. ACM Press, 2007.

- [156] David G. Richardson. DU-CS-05-14: Compiler-Enforced Memory Semantics in the SACLIB Computer Algebra Library. [http://www.cs.drexel.edu/index.php?option=com\\_page&Itemid=91](http://www.cs.drexel.edu/index.php?option=com_page&Itemid=91).
- [157] David G. Richardson. Compiler-Enforced Memory Semantics in the SACLIB Computer Algebra Library. Master's thesis, Drexel University, 2005. Published as Department of Computer Science Technical Report DU-CS-05-14.
- [158] David G. Richardson and Werner Krandick. Compiler-Enforced Memory Semantics in the SACLIB Computer Algebra Library. In V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, editors, *International Workshop on Computer Algebra in Scientific Computing*, volume 3718 of *Lecture Notes in Computer Science*, pages 330–343. Springer-Verlag, 2005.
- [159] Guido Van Rossum and Fred L. Drake, Jr. *The Python Language Reference Manual*. Network Theory, 2003.
- [160] Fabrice Rouillier and Paul Zimmermann. Efficient isolation of a polynomial's real roots. *Journal of Computational and Applied Mathematics*, 162:33–50, 2004.
- [161] Anatole D. Ruslanov. *Architecture-aware Taylor shift by 1*. PhD thesis, Drexel University, 2007.
- [162] W. Schreiner, W. Danielczyk-Landerl, M. Marin, and W. Stöcher. A generic programming environment for high-performance mathematical libraries. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming: International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 256–267. Springer-Verlag, 2000.
- [163] Sibylle Schupp, Marcin Zalewski, and Kyle Ross. Rapid performance prediction for library components. In *WOSP '04: Proceedings of the 4th International Workshop on Software and Performance*, pages 69–73, New York, NY, USA, 2004. ACM Press.
- [164] Christoph Schwarzweiler. *Towards Formal Support for Generic Programming*. Habilitation thesis, Wilhelm-Schickard-Institute for Computer Science, Universität Tübingen, Sand 14 72076 Tübingen, Germany, 2003.
- [165] Thomson Scientific. Web of Science. <http://www.isinet.com/>.
- [166] Ravi Sethi. Complete Register Allocation Problems. *SIAM Journal on Computing*, 4(3):226–248, 1975.
- [167] Mary Shaw. The coming-of-age of software architecture research. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, page 656, Washington, DC, USA, 2001. IEEE Computer Society.
- [168] Victor Shoup. NTL. <http://www.shoup.net/ntl/>.
- [169] Victor Shoup. *NTL: A Library for Doing Number Theory*. <http://www.shoup.net/ntl>.
- [170] Victor Shoup. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation*, 20(4):363–397, 1995.
- [171] Jamal Siadat, Robert J. Walker, and Cameron Kiddle. Optimization aspects in network simulation. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 122–133, New York, NY, USA, 2006. ACM.
- [172] Jeremy Siek. Boost Concept Check Library. [http://www.boost.org/libs/concept\\_check/concept\\_check.htm](http://www.boost.org/libs/concept_check/concept_check.htm).

- [173] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. N1758: Concepts for C++0x. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1758.pdf>.
- [174] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. Boost.GraphLibrary. [http://boost.org/libs/graph/doc/table\\_of\\_contents.html](http://boost.org/libs/graph/doc/table_of_contents.html).
- [175] Jeremy G. Siek. Modular generics. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 54–55, New York, NY, USA, 2004. ACM Press.
- [176] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison Wesley Longman, Inc., 2001.
- [177] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2001.
- [178] Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra. In D. Caromel, R.R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments: Second International Symposium, ISCOPE 98, Santa Fe, NM, USA, December 1998. Proceedings*, volume 1505 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, 1998.
- [179] Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: Generic Components for High-Performance Scientific Computing. *Computing in Science & Engineering*, 1(6):70–78, 1999.
- [180] Jeremy G. Siek, Andrew Lumsdaine, and Lie quan Lee. Generic programming for high performance numerical linear algebra. In *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO<sub>98</sub>)*. SIAM Press, 1998.
- [181] Volker Simonis. Adapters and binders: overcoming problems in the design and implementation of the C++ STL. *SIGPLAN Notices*, 35(2):46–53, 2000.
- [182] Murali Sitaraman, Greg Kulczycki, Joan Krone, William F. Ogden, and A. L. N. Reddy. Performance specification of software components. In *SSR '01: Proceedings of the 2001 Symposium on Software Reusability*, pages 3–10, New York, NY, USA, 2001. ACM Press.
- [183] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 215–228, New York, NY, USA, 1999. ACM.
- [184] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems*, pages 53–60. Australian Computer Society, Inc., 2002.
- [185] Olaf Spinczyk and Daniel Lohmann. Using AOP to develop architectural-neutral operating system components. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 34, New York, NY, USA, 2004. ACM.
- [186] Olaf Spinczyk and Daniel Lohmann. Advances in AOP with AspectC++. In *SoMeT 2005*. IOS Press, 2005.
- [187] Olaf Spinczyk, Matthias Urban, Daniel Lohmann, Georg Blaschke, Rainer Sand, and Horst Schirmeier. AspectC++. <http://www.aspectc.org/>.

- [188] R. E. Kurt Stirewalt and Laura K. Dillon. A component-based approach to building formal analysis tools. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 167–176, Washington, DC, USA, 2001. IEEE Computer Society.
- [189] Bjarne Stroustrup. Why C++ is not just an object-oriented programming language. In *OOPSLA '95: Addendum to the Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum)*, pages 1–13, New York, NY, USA, 1995. ACM Press.
- [190] Bjarne Stroustrup. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000.
- [191] Bjarne Stroustrup. *The C++ Standard: Incorporating Technical Corrigendum No. 1*. John Wiley and Sons, 2003.
- [192] Bjarne Stroustrup and Gabriel Dos Reis. N1782: A concept design (Rev. 1). <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1782.pdf>.
- [193] Sun Microsystems. Sun Studio Collection. <http://www.sun.com/software/products/studio/>.
- [194] Sun Microsystems. *UltraSPARC III Cu: User's Manual*, 2004.
- [195] Vitaly Surazhsky and Joseph (Yossi) Gil. Type-safe covariance in C++. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1496–1502, New York, NY, USA, 2004. ACM Press.
- [196] Herb Sutter. Eliminate False Sharing. *Dr Dobbs Journal*, May 2009.
- [197] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.
- [198] Valgrind Developers. Valgrind. <http://valgrind.kde.org>.
- [199] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley, 2003.
- [200] Todd Veldhuizen. Blitz++. <http://www.oonumerics.org/blitz/>.
- [201] Todd L. Veldhuizen. Arrays in Blitz++. In D. Caromel, R.R. Oldehoeft, and M. Tholburn, editors, *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer-Verlag, 1998.
- [202] Dennis M. Volpano and Richard B. Kieburtz. Software templates. In *ICSE '85: Proceedings of the 8th International Conference on Software Engineering*, pages 55–60, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.
- [203] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.
- [204] Karsten Weihe. A software engineering perspective on algorithmics. *ACM Computing Surveys*, 33(1):89–134, 2001.
- [205] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.
- [206] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Softw. Pract. Exper.*, 35(2):101–121, 2005.

- [207] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM.
- [208] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286, Washington, DC, USA, 1996. IEEE Computer Society.
- [209] Stephen Wolfram. *The Mathematica Book*. Wolfram Research Inc., 2000.
- [210] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. SPL: a language and compiler for DSP algorithms. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 298–308, New York, NY, USA, 2001. ACM.
- [211] Dianxiang Xu and Weifeng Xu. State-based incremental testing of aspect-oriented programs. In H. Masuhara and A. Rashid, editors, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 180–189. ACM Press, 2006.
- [212] Guoqing Xu. A regression tests selection technique for aspect-oriented programs. In *Proceedings of the Second Workshop on Testing Aspect-Oriented Programs*, pages 15–20. ACM Press, 2006.
- [213] Guoqing Xu and Atanas Rountev. Regression Test Selection for AspectJ Software. In *Proceedings of the 29th International Conference on Software Engineering*, pages 65–74. IEEE Computer Society Press, 2007.
- [214] Zhen Yao, Qi long Zheng, and Guo liang Chen. GOOMPI: A Generic Object Oriented Message Passing Interface. In Hai Jin, Guang R. Gao, Zhiwei Xu, and Hao Chen, editors, *Network and Parallel Computing: IFIP International Conference, NPC 2004, Wuhan, China, October 18-20, 2004. Proceedings*, volume 3222 of *Lecture Notes in Computer Science*, pages 261–271. Springer-Verlag, 2004.
- [215] W. Zhang, T. Hou, X. Qiao, and X. Xu. Some Basic Data Structures and Algorithms for Chemical Generic Programming. *Journal of Chemical Information and Computer Sciences*, 44(5):1571–1575, 2004.
- [216] Jianjun Zhao, Tao Xie, and Nan Li. Towards regression test selection for AspectJ programs. In *Proceedings of the Second Workshop on Testing Aspect-Oriented Programs*, pages 21–26. ACM Press, 2006.