DISSERTATION

A METHODOLOGY FOR AUTOMATED LOOKUP TABLE OPTIMIZATION OF SCIENTIFIC APPLICATIONS

Submitted by

Chris Wilcox

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2012

ABSTRACT

A METHODOLOGY FOR AUTOMATED LOOKUP TABLE OPTIMIZATION OF SCIENTIFIC APPLICATIONS

Tuning the performance of scientific codes is challenging because of their math-intensive nature. Applications such as climate modeling and molecular biology simulate the behavior of natural systems based on scientific equations. Translating these equations into code can often yield expressions that are expensive to evaluate. Trigonometric, logarithmic, and exponential elementary functions are especially problematic because of their high cost relative to ordinary arithmetic. Lookup table (LUT) transformation can expedite function evaluation by precomputing and storing function results, thereby allowing inexpensive memory lookups to replace costly function calls. Practical concerns limit the size and accuracy of LUT data, thus the technique represents a tradeoff between error and performance. Current practice has the programmer apply each LUT transform manually, thereby impacting productivity, obfuscating code, and limiting programmer control over accuracy and performance.

The goal of our research is to help scientific programmers use LUT techniques in a more effective and productive manner. Our approach substantially automates the process of applying LUT transformation via a methodology and its implementation in the Mesa tool. Mesa incorporates algorithms that make adding a LUT transform easier for the programmer, including expression enumeration, domain profiling, error analysis, performance modeling, and code generation. In addition we introduce a novel algorithm for LUT optimization that analyzes application code and helps the programmer identify the most beneficial set of expressions to transform. We demonstrate the effectiveness of our algorithms by using Mesa to perform case studies of scientific applications. Mesa achieves performance speedups of 1.4× to 6.9× without significantly compromising application accuracy. Finally we present evidence that our methodology allows the scientific programmer to be more productive and effective.

ACKNOWLEDGEMENTS

I would like to acknowledge support for our research from Award Number 1R01GM096192 from the National Institute Of General Medical Sciences. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institute Of General Medical Sciences or the National Institutes of Health. Our research is also supported by grant number DESC0003956 from the Department of Energy, with additional seed funding from the Vice President of Research and the Office of the Dean of the College of Natural Sciences at CSU and from a Department of Energy Early Career grant. This research utilized the CSU ISTeC Cray HPC System supported by NSF Grant CNS-0923386.

We gratefully acknowledge the partners who have shared code, including Stefan Sillau, Ryan Croke, and Mark van der Woerd for the SAXS code, Paulette Clancy for the Stillinger-Weber program, Chuck Anderson for the neural network application, and Olaf David and George Leavesley for the PRMS code. I also received ideas from Michael Heroux and Paul Hovland and help with optimization code from Sven Leyffer at Argonne National Labs.

DEDICATION

I would like to give special thanks to Michelle Strout and James Bieman for their limitless patience and help with this dissertation, my research, and every other aspect of graduate school. I would also like to acknowledge Wim Böhm for help on various algorithms, John Labadie for the methodology to proof the local optimization formula, and Sudipto Ghosh for his inspired teaching. Most of all I would like to thank my wife Joann and my daughters Kate and Emma for their understanding of my quixotic desire to pursue a doctorate.

TABLE OF CONTENTS

Chapter 1

Introduction

Scientists depend on increasingly complex software applications to support the advance of human knowledge. Such programs have insatiable performance demands [29], which are hard to fulfill because of the math-intensive nature of scientific codes. Scientific programs frequently call elementary functions that consume many more processor cycles than ordinary arithmetic [41, 46], as shown in Table 1.1. For example, a cosine call is ∼45× to ∼53× costlier than a multiply on current architectures. The topic of our research is a method that improves the performance of scientific codes. Our approach is to automate and extend an existing technique called lookup table (LUT) transformation that exploits the fuzzy redundancy often found in scientific computations. A LUT transform improves performance by precomputing results for a function, then sharing those results between multiple computations with similar input values to eliminate computation.

Table 1.1: Performance of elementary functions versus arithmetic.

(Intel Core 2 Duo, E8300, family 6, model 23, 2.83GHz, single core)

Elementary Function	Single Precision	Double Precision
sin	40 ns	51 ns
cos	45 ns	53 ns
tan	56 ns	71 ns
acos	42 ns	48 ns
asin	43 ns	47 ns
atan	43 ns	49 ns
exp	32 ns	35 ns
log	56 ns	61 ns
sqrt	7.1 ns	5.2 ns
*	1.1 ns	1.9 ns
/	2.0 ns	3.1 ns
+	1.0 ns	1.7 ns
-	1.2 ns	2.0 ns

Programmers often use LUT transformation to improve the sequential performance of math-intensive codes [56]. In current practice, the programmer develops LUT code by hand, often without the benefit of error analysis or a performance model. This makes it difficult for the programmer to gauge the impact of a LUT transform on accuracy and performance. Developing LUT code by hand also represents a substantial effort that can obfuscate application code. The research presented here addresses these problems with a new methodology and algorithms to automate the LUT process, which we implement in a tool called Mesa. Mesa automates many of the steps required for LUT transformation, and it implements a novel algorithm that we call LUT optimization to help the programmer identify optimal sets of expressions to which transformations can be applied. Our evaluation shows that Mesa can achieve application speedups of 1.4× to 6.9×, and we find that automation makes programmers more effective and productive.

1.1 Motivation

Our interest in LUT methods began with an application that we wrote for the Small Angle X-ray Scattering (SAXS) project [64] at Colorado State University (CSU). The program simulates the X-ray scattering of proteins. We initially received a partial implementation of the discrete algorithm based on Debye's formula [26], written in R code. Porting to C++ improved the performance significantly, but still fell short of our performance goals. We reduced execution time by manually incorporating a LUT transform for the dominant calculation, which we found through performance profiling. The process was cumbersome and required lengthy experimentation to determine an effective solution, but the result was a 6.9× speedup. To simplify tuning we developed the Mesa tool to automate the time-consuming and error-prone aspects of LUT transformation. We have since applied Mesa to another version of the SAXS code that implements a continuous algorithm, enabling a 3.0× speedup on the C++ code that we ported from Matlab. For both algorithms the performance improvements were achieved while meeting accuracy requirements.

The current version of Mesa is much more capable than the original, as described in Section 6. We have extended the tool to automatically find and evaluate candidate expressions in source code, and we have applied the tool to additional application areas. Table 1.2 shows the performance improvement and error introduced using Mesa for these applications. The performance speedup is calculated as the ratio of the original time divided by the optimized time, and the maximum error is calculated as the relative difference between the output of the original and optimized versions. The table shows results for the discrete and continuous variants of the SAXS application [64], the Stillinger-Weber molecular dynamics program [31], a CSU neural network application [53], and the PRMS precipitation runoff model [58]. These results were achieved while constraining the LUT data to reside in cache memory on the test system as shown in the last column. The evaluation of these and other applications is described in detail in Section 7.

Table 1.2: Results of application optimization with Mesa.

(Intel Core 2 Duo, E8300, family 6, model 23, 2.83GHz, single core)

Application Name	Original Time	Optimized Time	Performance Speedup	Maximum Error	Memory Usage
Saxs Scattering (discrete)	**196.2s**	**29.0s**	**6.8X**	**4.06×10^{-3}%**	**4MB**
Saxs Scattering (continuous)	**10.1s**	**2.5s**	**4.0X**	**1.48×10^{-4}%**	**4MB**
Stillinger-Weber (simulation)	**14.6s**	**10.4s**	**1.4X**	**2.91×10^{-2}%**	**1MB**
Neural Network (logistics)	**8.0s**	**3.6s**	**2.2X**	**8.70×10^{-2}%**	**4MB**
Neural Network (hypertan)	**10.9s**	**3.9s**	**2.8x**	**6.30×10^{-1}%**	**4MB**
PRMS (slope aspect)	**242ns**	**56ns**	**4.3X**	**8.21×10^{-6}%**	**4MB**
PRMS (solar radiation)	**13.7s**	**6.1s**	**2.2X**	**2.97×10^{-4}%**	**4MB**

1.2 LUT Techniques for Function Evaluation

The context of our research is the use of LUT transformation to improve the performance scientific codes with expressions that contain elementary function calls. Assignment statements and their component expressions can be viewed as mathematical functions, so in this document we use the terms expression and function interchangeably. LUT transformation improves the performance of function evaluation by exploiting the imprecise or *fuzzy reuse* of previously computed results, thereby sharing a computations over a range input values. Computing and storing expression results allows costly function evaluations to be replaced by LUT accesses. Performance improves when LUT accesses are less expensive than the original computation, and when enough reuse occurs to amortize the cost of LUT initialization. The citation that follows from Pharr and Fernando [56] describes the concept succinctly:

> *"Lookup tables (LUTs) are an excellent technique for optimizing the evaluation of functions that are expensive to compute and inexpensive to cache. By precomputing the evaluation of a function over a domain of common inputs, expensive runtime operations can be replaced with inexpensive table lookups. If the table lookups can be performed faster than computing the results from scratch (or if the function is repeatedly queried at the same input), then the use of a lookup table will yield significant performance gains. For data requests that fall between the table's samples, an interpolation algorithm can generate reasonable approximations by averaging nearby samples."*

LUT transformation is only one of several methods that exploit reuse to gain performance. Memoization is a related technique that caches computational results during program execution for future reuse. Memoization supports both precise and fuzzy reuse [2]. When fuzzy reuse is employed, the technique is similar to LUT transformation, but there are two significant differences. First, a LUT transform computes results for the entire domain in advance, whereas memoization waits until computation occurs to cache results. Second, a LUT transform provides a simple and inexpensive indexing operation to retrieve LUT data for a given input. Memoization, in contrast, must provide a mapping function that determines whether a result has been cached, which can introduce significant overhead.

CPU designers have long been aware that scientific codes are dependent on elementary function performance, thus there has been a longstanding debate over the need for dedicated instructions [55]. Early microprocessors contained instructions such as FSIN, FCOS, and FSQRT, and these opcodes are still supported in current architectures [37]. Many current Field-Programmable Gate Arrays (FPGAs) incorporate elementary functions to support image processing and other applications [19]. These hardware implementations of elementary functions often take advantage of hardware LUTs [25]. The fundamentals of applying a LUT transform are the same for hardware and software, but a handful of important differences exist. Section 3.1 lists these differences and presents related work.

Software developers also have a long history of using LUTs, starting with the discovery by early assembly language programmers that table lookups could reduce instruction counts for simple algorithms such as character conversion [36]. Scientific programmers have used LUT transformation for function evaluation for many years. For example, the Fastest Fourier Transforms in the West (FFTW) libraries store LUT data that is computed using cosines and sines, resulting in *"significant reductions in computation time result from table lookup"* [39]. Rapid Radiative Transfer Model (RRTM) software uses LUT transforms for the exponential and tau functions, yielding a 1.75× improvement on code that represents almost 25% of the execution time of a global climate model [59]. Additional evidence of current usage is shown by the LUT programming support in scientific languages such as Matlab [47] and SciPy [66].

Engineering practices related to LUT transformation vary significantly between software and hardware. We are not aware of any tools that support software LUT methods, with the exception of a compiler presented by Zhang et al. [84] that generates hardware and software LUT transforms. The Zhang et al. compiler does not operate directly on application code, but the central idea of the tool is similar to Mesa, as we discuss in more detail in Section 3.1.2. As a result, scientific programmers must perform LUT transformation in an *ad hoc* manner. In contrast, hardware designers have an extensive literature on error analysis [74, 75, 65]. This disparity leads to poor representation of topics that are specific to software methods.

For example, research on LUT error analysis performance appears to be virtually nonexistent, possibly because the topic is not a concern for hardware designers.

Our research objective is to bring significant automation to software LUT methods. We have adopted this goal to support our own development needs by avoid manual tuning, which is known to be unproductive [70]. Without automation, a programmer must explore the space of LUT parameters through unguided experimentation, making it difficult to control the critical tradeoff between error and performance. Manual analysis of this tradeoff may be feasible for a single expression, but the large solution space that is created when considering multiple expressions makes it impractical to analyze by hand.

1.3 Simple Example of LUT Transformation

The following example demonstrates LUT transformation. Consider an application that repeatedly calls the sine function for inputs between 0.0 and 2π. We implement a LUT transform for the sine function by computing a sine table for the specified range of input values, and replacing sine calls with LUT accesses. The resulting LUT transformation decreases application accuracy and improves performance. Error is introduced because the sine table represents a discrete sampling of a continuous function. Table 1.3 shows the memory usage and error statistics for a range of different LUT sizes. For example, a sine table with 65,536 entries uses 256KB to replicate the sine function with a maximum absolute error of $4.88\times10^{-5}\%$. The method used to compute this error is described in Section 5.2. The LUT error is inversely proportional to LUT size, regardless of the function being approximated.

To continue our example, we define LUT benefit as the execution time reduction achieved through LUT transformation. The upper bound of LUT benefit is the evaluation time of the original function multiplied by its call frequency. Table 1.1 shows evaluation time of 40ns or 51ns for a sine call on our test system, depending on the precision. No lower bound exists for LUT benefit because it can actually be negative in cases where the LUT access is slower than evaluation of the original function, for example arithmetic operators on current architectures. We benchmark the LUT transformation described above and find that a LUT

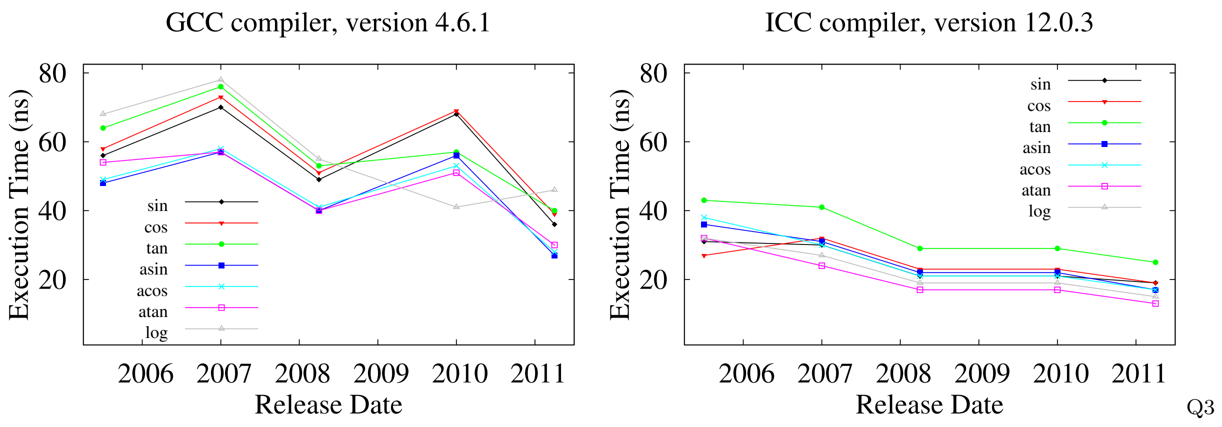
Table 1.3: Lookup table for sine function over domain [0.0,2π].

Table Entries	Memory Usage	Maximum Error	Average Error
256	**1 KB**	**1.25×10^{-2}**	**4.03×10^{-3}**
1024	**4 KB**	**3.12×10^{-3}**	**1.00×10^{-3}**
4096	**16 KB**	**7.79×10^{-4}**	**2.50×10^{-4}**
16384	**64 KB**	**1.95×10^{-4}**	**6.26×10^{-5}**
65536	**256 KB**	**4.88×10^{-5}**	**1.57×10^{-5}**
262144	**1024 KB**	**1.23×10^{-5}**	**3.92×10^{-6}**

access takes 7.4ns. We conclude that the portion of application execution time dedicated to sine evaluation can be improved by approximately 5.4$\times$ to 6.9$\times$. We perform the experiment on our test system and measure a speedup of 6.4$\times$. The LUT sizes in Table 1.3 fit into cache on our test system, thus performance varies only slightly between the table sizes shown. Our example shows that LUT transformation can provide a significant performance increase without incurring an unreasonable amount of error.

1.4 Continued Relevance of LUT Approach

LUT transformation has been around since the 1950's [3] and is still in use today. We claim that research on the automation of LUT methods is still relevant for current computing platforms, because elementary function calls remain a bottleneck for many applications. A recent trend of great concern to scientific programmers is the "power wall", which has ended many years of performance gains achieved solely by increasing clock speed. According to Hennessy and Patterson, processor performance increased annually by 52% from 1986-2002, but only by 6% per year from 2002-2005 [34]. In 1986, the fastest microprocessor was the Intel 80386 at 16Mhz. By 2005, the Intel Pentium was running 200$\times$ faster at 3200Mhz. Since then Intel processors have briefly achieved 4Ghz before scaling back to the current 3Ghz range. Through the entire period Moore's Law has remained in effect, with transistor counts doubling every year. The consequence has been an industry-wide shift to multi-core



2005: AMD Opteron 254 (Troy), family 15, model 37, 2.8Ghz,

Q1 2007: Intel Xeon E5450 (Harpertown), family 6, model 23, 3.0Ghz,

Q2 2008: Intel Core 2 E8300 (Wolfdale), family 6, model 23, 2.83Ghz,

Q1 2010: AMD Opteron 6168 (Magny-Cours), family 15, model 37, 2.8Ghz,

Q2 2011: Intel Xeon E31230 (SandyBridge), family 6, model 42, 3.2Ghz

Figure 1.1: Elementary function performance over time.

architecture and parallelism. This shift has had an effect on elementary function performance and cache availability, both of which impact the viability of LUT transformation.

The power wall has had a large impact on scientific computing, most notably because parallel programming is now required to meet performance demands [6]. However, even within a parallel program, many elementary functions are executed on each processor core. The performance of such functions will no longer benefit from increases in clock frequency. Figure 1.1 shows that elementary function performance has leveled off recent years, based on a sampling of processors available at CSU. The average performance increase of the functions shown is 1.85× over six years, a much slower rate of improvement than previously provided by clock scaling. As a result, many scientific applications may remain limited by elementary function performance, even when executed on multiple cores.

Given the dominance of multi-core architectures, we can justify sequential optimizations only if they are effective in the context of parallel execution. A critical factor for successful LUT optimization on multi-core systems is cache availability, because LUT data must reside in mid-level cache. An increasing number of cores raises the concern that manufacturers

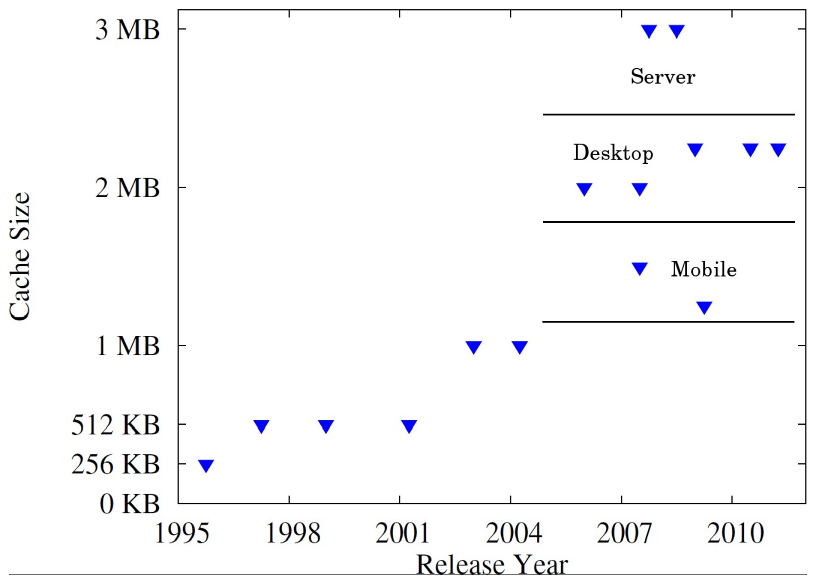


Figure 1.2: Combined L2 and L3 cache per core.

may be unable to maintain the current levels of per core cache. However, recent cache size trends do not support this hypothesis. Figure 1.2 shows the combined size of L2 and L3 cache per core, on historical and recent Intel processors with up to 8 cores. The L2 caches of uniprocessor systems grew quickly from 256KB in 1995 to 1MB or more in 2004. Since then there is no evidence of a reduction in combined L2 and L3 cache size, despite the growth in the number of cores. In our case studies we have confirmed that sufficient cache exists to support LUT transformation on existing multi-core systems.

While the size of cache has remained stable, cache hierarchies are clearly changing. Early dual-core and quad-core processors simply replicated L2 cache for each core. Current multi-cores have reduced L2 cache to 256KB, but designers have added up to 3MB shared L3 cache per core. For example, current server products from Intel have up to 2MB of L2 and 24MB of L3 for eight cores. The recent introduction of L4 cache supports the premise that hardware designers continue to place importance on mid-level cache. Caches have traditionally been built from static memories that provide fast access but require significant die area. A recent trend of using embedded dynamic memory [8] increases the likelihood that mid-level cache size will remain be maintained or increase on future systems.

1.5 Brief Overview of LUT Methodology

To address problems with *ad hoc* practices, we have developed a methodology to help performance programmers with LUT optimization. We briefly summarize the steps here. First, performance profiling tools are employed to identify methods that contain bottlenecks. Mesa processes C functions and C++ methods in the source code, but in this document we use the term function. The programmer then inserts pragmas above the declarations of these functions to identify the scope of LUT optimization, and runs the Mesa to instrument the program. Mesa analyzes the source code within the specified scope and extracts expressions that contain elementary function calls. Each of these expressions represents a candidate for LUT transformation. Mesa finishes by instrumenting the application for domain profiling. The resulting program must be executed by the programmer with one or more representative data sets to gather domain boundaries and call frequency. The data sets must be varied enough to establish robust boundaries for the input domains. This information is stored in a file that the programmer can inspect and edit. After domain information has been gathered, the programmer runs Mesa again to optimize the program.

Mesa starts optimization by evaluating the cost and benefit of each expression through error analysis and performance modeling, then it formulates and solves a numerical optimization problem whose purpose is to identify a set of optimal solutions. The dual objectives of the LUT optimization problem are to maximize performance and minimize error, subject to constraints on memory usage. Solutions are sets of expressions that can be simultaneously transformed while sharing cache resources. Mesa evaluates solutions by combining the benefits and errors of their member transformations. We define an optimal solution as one for which no other solution has more benefit with the same or less error. The LUT optimization problem has competing objectives, thus it produces multiple optimal solutions. Mesa discards suboptimal solutions, leaving the programmer to select from the remaining optimal solutions, which are ranked by accuracy and performance. Mesa realizes the selected solution by generating LUT code for each of its expressions, and by applying a transformation to integrate the code into the application. The programmer completes the process by comparing the performance and accuracy of the original and optimized versions of the program.

1.6 Summary of Contributions

In a workshop paper [76], we described an early version of Mesa that required separate specification of candidate expressions. A programmer using this version had to manually identify candidate expressions and domains, specify the expressions and constituent variables in a file, run Mesa to generate code, and then manually integrate the resulting code back into the application. In a journal paper [77], we presented an approach and newer version of Mesa that used pragmas to apply LUT transformation to expressions in C and C++ source code, thereby operating directly on application source code. This thesis includes the work from these papers, and presents an automated technique for LUT optimization that further minimizes programmer effort. The contributions of this thesis are as follows:

- A **comprehensive methodology** that applies software LUT techniques to scientific codes with expressions that contain elementary functions (Chapter 4).
- A comparison of the accuracy and performance of **error analysis** for LUT data, including analytic and numerical techniques (Chapter 5).
- A novel LUT **optimization technique** that identifies a set of expressions for LUT transformation to maximize performance and minimize error (Chapter 5).
- A set of **case studies** that demonstrates the effectiveness of our LUT methodology and tool in the context of single-core and multi-core execution (Chapter 7).
- A **software tool** that provides automation of domain profiling, error analysis, performance modeling, and code generation for LUT transformation (Chapter 6).

In addition to the above list we provide a background of LUT transformation in Chapter 2 and a review of LUT related work in Chapter 3. We discuss the limitations of our methodology and threats to validity in our empirical research in Chapter 8. We conclude and present future research directions in Chapter 9.

Chapter 2

LUT Background

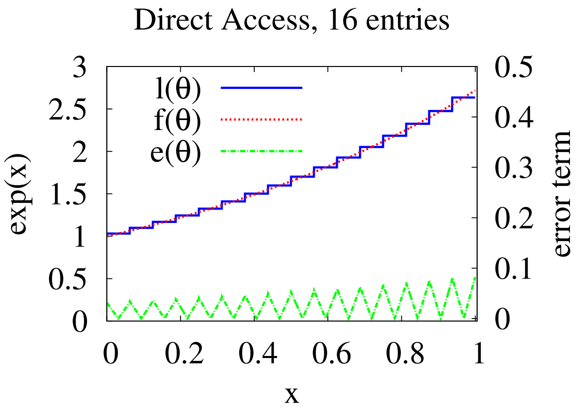
In this chapter we review the fundamentals of LUT transformation by showing an example. We use the example to introduce LUT terminology and explain how LUT transformation affects accuracy and performance. Next we discuss LUT sampling methods, including direct access and linear interpolation. We conclude with a discussion of several important issues related to the implementation of LUT transformation.

2.1 LUT Example and Terminology

Figure 2.1 shows LUT data for the expression $exp(x)$. The input value is restricted to $0 \leq x \leq 1$, thus the LUT *domain* is $[0, 1]$. The left graph shows a table with 16 entries and the right graph shows a table with 32 entries. The original function is plotted as $f(x)$ and its approximation via the LUT data as $l(x)$. The tables are created by partitioning the domain into uniform intervals, and assigning a LUT *entry* for each interval. LUT entries contain results computed using the original function, stored as an array of LUT *data*. The number of intervals is the LUT *size*, and the interval width is the LUT *granularity*. Equation 2.1 shows the relationship between domain, granularity, and size, from which we compute a granularity of 0.0625 for the 16-entry table and 0.03125 for the 32-entry table:

$$Granularity = Domain/Size \tag{2.1}$$

We initialize LUT data by evaluating the expression over each interval and storing the result in the corresponding LUT entry. Each LUT entry stores a single result that must be shared by all inputs in the interval, thus the result is often imprecise. The decision about which output value to select in each interval is an important one, as it will affect accuracy. The literature suggests selecting the output value that corresponds to the center



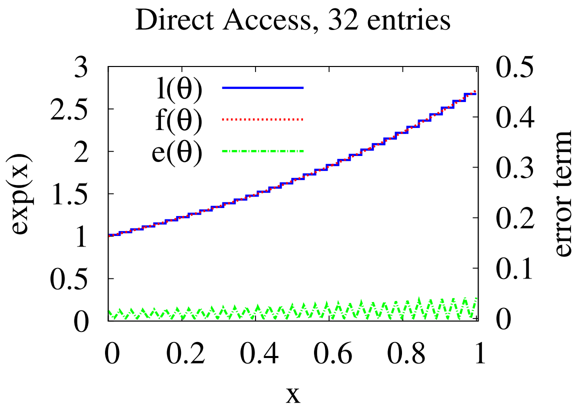


Figure 2.1: LUT data for exponential function, with 16 and 32 entries.

of the interval. Using the mean value of the function over the interval is theoretically more accurate, but expensive to compute. As LUT granularity decreases, the output function in each interval approach linearity. When the output function is linear, the average and center outputs correspond, so computing the mean is unnecessary.

The difference between a function and its approximation represents LUT *error*. Figure 2.1 shows error as the distance between $f(x)$ and $l(x)$, plotted separately as $e(x)$. We can combine the individual error terms to compute the maximum and average error over a LUT entry or the entire table, as described in Section 5.2. Each step in $l(x)$ represents a single LUT entry. For example, the left-most step on the graph assigns the output value 1.0317 to input values in the interval $0.0000 \leq x < 0.0635$. The maximum absolute error is 0.0836 for 16 entries and 0.0421 for 32 entries, found at the right-most step because this is where the exponential function has the highest slope. The graphs illustrate that an increase in LUT size decreases error. As a result of this relationship, we can control LUT error by setting the LUT size, subject to constraints on memory usage.

The amount of error introduced by a LUT transform depends on several factors, including how the LUT data is sampled. The most common sampling methods are *direct access* and *linear interpolation*. Direct access simply finds the interval that contains the input value and returns its LUT entry. Linear interpolation selects the two closest LUT entries and combines them according to their respective distances from the exact input. Linear interpolation improves accuracy, but performance is degraded by the extra LUT access and computation.

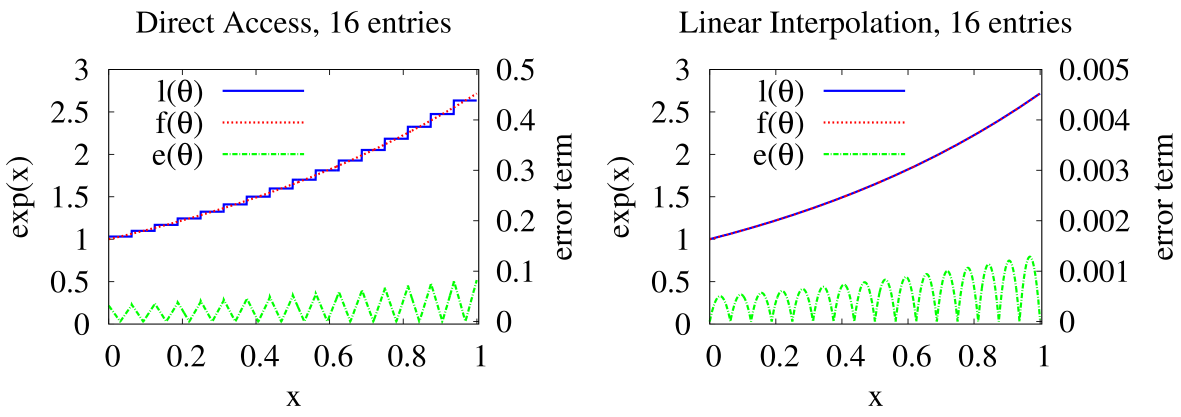


Figure 2.2: LUT data for exponential function, direct access and linear interpolation.

When computing LUT data, the sampling method determines the selection of output values. For direct access we use center selection, as previously described. For linear interpolation, we follow the standard practice of computing output values at interval boundaries.

Figure 2.2 compares the direct access and linear interpolation sampling methods. For a table with 16 entries, the maximum absolute error is 0.0836 for direct access and 0.0013 for linear interpolation, a difference of almost two orders of magnitude. We have adjusted the error scale on the right graph to make the linear interpolation error more visible.

The purpose of LUT transformation is a reduction in execution time that we refer to as LUT *benefit*. We compute the benefit as the difference between the cost of expression evaluation and the cost of LUT access. Expression evaluation is relatively consistent and therefore easy to model. Access cost is harder to characterize because it depends on the location of LUT data in the memory hierarchy. LUT data for an elementary function rarely fits in top-level (L1) cache, which is typically 32-64KB per core on modern systems. Our work shows that LUT transformation performance degrades quickly when the LUT overflows mid-level (L2 or L3) cache, which is typically 1-3MB per core [76]. As a result, we constrain LUT data to reside in mid-level cache. We describe the performance model in Section 5.3.

2.2 LUT Implementation Issues

We now address several issues that arise when implementing a LUT transform. Among these are the LUT data precision, which affects both accuracy and memory usage. Single-precision numbers are accurate to FLT_EPSILON, which is $\pm 1.19\times 10^{-07}$. Double-precision numbers are accurate to DBL_EPSILON, which is $\pm 2.22\times 10^{-16}$. For typical input domains, the LUT granularity is much larger than FLT_EPSILON, so approximation error dominates. It is therefore more effective to store single-precision LUT data and reduce error by increasing the LUT size. Double-precision may be needed when the LUT granularity is close to or smaller than FLT_EPSILON, but this would be practical only for highly restricted domains. All examples and evaluations in this thesis use single-precision LUT data.

Another issue arises with respect to coverage of the LUT domain. To allow the original expression to be completely replaced with a LUT access, a LUT transform must store data for the entire domain, There is another option, which is to store results for a partial domain and add conditional code to choose between the LUT access and the original expression. The advantage of a partial domain is that memory usage can be reduced by storing only the results that are evaluated most frequently, thereby gaining most of the performance advantage. The downside is that the original expression must remain in the code, and conditionals are very expensive. We evaluate the use of partial domains in Section 5.1.5.

A final issue is the uniformity of the intervals used to partition the LUT domain. The use of uniform intervals simplifies the LUT index computation, but non-uniform intervals can selectively provide more accuracy in domain regions where the error rate is higher, thereby decreasing memory usage. However, we have not incorporated non-uniform intervals into our methodology because they reduce the performance benefit by increasing the cost of the LUT index computation. Equation 2.2 shows the index computation for uniform intervals:

$$Index = Input * (Size/Domain) \tag{2.2}$$

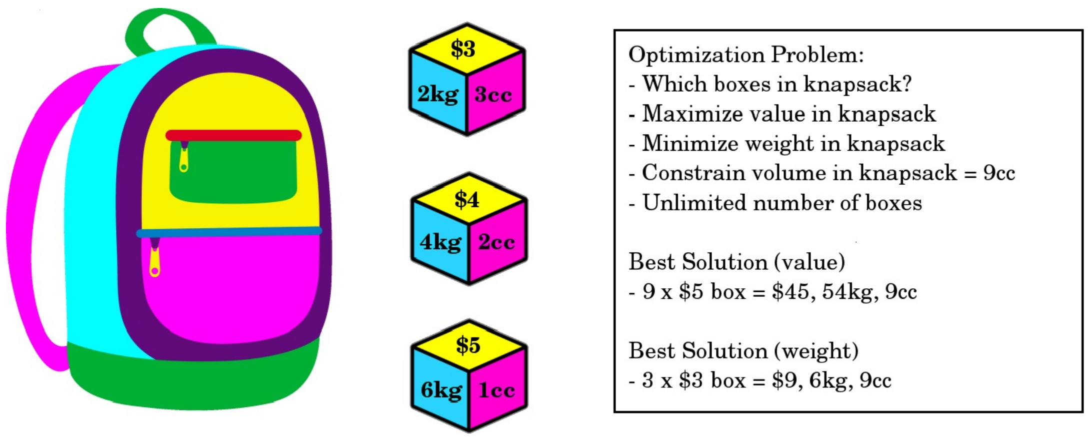
The table index is computed by multiplying the input value by the table *size* divided by the *domain.* The latter is the inverse of the granularity, which can be computed in advance to avoid the division. Finally the table index is used to access the table data to fetch the stored result, which is not shown. An extra subtraction (not shown) is necessary to make the index zero-based when the lower boundary of the domain is non-zero.

2.3 LUT Optimization Problem

In this section we describe how our LUT optimization problem is similar to the knapsack problem, a classic example of combinatorial optimization [45]. The knapsack analogy allows us to frame our problem in terms of traditional numerical optimization languages and terminology. However, there is a significant difference between LUT optimization problem and the knapsack problem, and this has led us to develop a custom solution instead of using existing optimization solvers.

We use optimization methods to find the most effective expressions to which to apply LUT transformation. To do so we must select a set of LUT transforms, then compute the optimal allocation of cache memory for them. The knapsack problem is analogous in that it searches for optimal sets of objects that fit inside a knapsack. In its simplest form, each object has volume and value. The single objective of this problem is to find the set of objects that fit in the backpack to maximize total value, defined as the sum of object values. A constraint on knapsack volume limits the number of objects that can fit in the knapsack. The problem has a unique solution in terms of total value.

We can extend the knapsack problem to include multiple objectives, thus making it closer to the cache memory allocation problem. We do this by adding a weight attribute to objects and a second objective to minimize the sum of object weights. The two objectives compete, since reducing total weight tends to decrease total value and vice versa. As a result there are multiple solutions, some of which do better at the first objective and others that do better at the second. Figure 2.3 shows an example knapsack problem and the two optimal solutions



Optimization Problem:
- Which boxes in knapsack?
- Maximize value in knapsack
- Minimize weight in knapsack
- Constrain volume in knapsack = 9cc
- Unlimited number of boxes

Best Solution (value)
- 9 x $5 box = $45, 54kg, 9cc

Best Solution (weight)
- 3 x $3 box = $9, 6kg, 9cc

Figure 2.3: Example of knapsack optimization problem.

that provide the highest value and lowest weight. Other optimal solutions exist whose total value and weight are in between the ones shown.

The knapsack problem allocates knapsack space and the LUT optimization problem allocates cache memory. For LUT optimization our objects are LUT transforms, with benefit replacing value, error replacing weight, and memory usage replacing volume. The objectives are to maximize benefit and minimize error, subject to a constraint on memory usage. The crux of the problem is that cache resources limit the number and size of LUT transformations, thus we are trying to allocate cache memory in the most effective way possible. Our problem differs from the knapsack problem in that the error associated with a LUT transform is described by a nonlinear equation, as opposed to a constant weight.

As with the knapsack problem, we end up with multiple solutions that represent optimal tradeoffs, in this case between performance and accuracy. Selection of a particular optimal solution allows the programmer to balance these competing demands. By specifying the constraint on cache size, the programmer also gains control over resource usage. Section 5.4 provides an in-depth description of the algorithms that we use to construct and solve the LUT optimization problem.

Chapter 3

Related Work

We start our chapter on related work by citing examples where imprecision is tolerated in exchange for a performance advantage, and we address numerical stability and analysis issues. Next we survey related work on hardware and software LUT transformation, and discuss the difference in requirements between the two disciplines. We briefly review the important issue of the impact of cache memory on LUT performance. We follow this with a discussion of alternate techniques such as memoization and value reuse, and we conclude with a survey of the literature on productivity and the scientific programmer.

3.1 Trading Imprecision for Performance

There is considerable precedence for methods such as LUT transformation that trade accuracy for performance. Computer precision is inherently limited, yet the existing floating-point representations have proven to be satisfactory for many scientific applications. Linderman et al. [43] argue that reducing the precision of computation has benefits, but caution that a careful analysis is necessary to maintain accuracy. Research suggests that single-precision arithmetic can sometimes be used in place of double-precision [10], and several authors propose the intentional reduction of precision to save power [7, 62]. Some libraries give the programmer explicit control over accuracy and performance. For example, the Intel Math Kernel Library (MKL) supports an enhanced performance mode that throws away up to half of the significand bits for single or double-precision [38], and references in the Intel documentation suggest that performance gains can be achieved by using LUT techniques to approximate elementary functions, as quoted below [37]:

The existence of methods that trade accuracy for performance does not necessarily imply that all applications can do so safely. The numerical stability of an application is determined

"Usually, math libraries take advantage of the transcendental instructions (for example, fsin) when evaluating elementary functions. If there is no critical need to evaluate the transcendental functions using the extended precision of 80 bits, applications should consider alternate, software-based approach, such as look-up-table-based algorithm using interpolation techniques."

by many factors [27], so some applications are more sensitive to the introduction of error than others. Techniques such as interval analysis can be used to bound error on sequences of operations [51], but characterizing the error propagation for an entire application is a complex numerical analysis problem that is beyond the scope of this thesis [12]. Our methodology estimates the error introduced by a set of LUT approximations, and we provide support for empirically measuring application error. In Section 5.2 we describe how to quantify the error associated with LUT transformation.

3.1.1 LUT Transformation in Hardware

Hardware LUTs date back to an early IBM computer into which Amdahl [3] incorporated a table for character conversion. More recent hardware LUT examples are found in graphics [32, 56], network routing [1], and logic synthesis [24]. The optimization of elementary functions in hardware also has a considerable history [55, 20], not necessarily limited to LUT methods. The first application of LUT hardware to elementary function evaluation that we are aware of was proposed by Gal [25], who suggested using polynomial reconstruction to increase accuracy. Tang [74, 75] applied LUT methods to elementary functions for number in IEEE format. Many other papers have expanded the use of LUT methods to additional functions [80, 65, 83]. Dedicated memory is expensive, so LUT implementations usually depend on some form of interpolation between table entries to limit LUT size. At a minimum, hardware LUT solutions implement linear interpolation [56], and hardware designers commonly use high-order polynomial reconstruction to reduce error even further [57]. LUT methods for function evaluation in FPGAs has recently become popular [69, 19].

3.1.2 LUT Transformation in Software

Software LUTs have few academic references, but there are a variety of books [60, 56, 36] that discuss the topic. We are aware of only one tool that supports software LUT transformation, as described by Zhang et al. [84]. The Zhang et al. paper presents a standalone compiler that analyzes mathematical expressions written in a language that is similar to MATLAB, and transforms these expressions either into an FPGA design or C/C++ code. Although their compiler does not operate directly on source code, their work provides a unique discussion of software LUT issues. Unlike previous work, the Zhang et al. compiler can transform arbitrary numerical expressions that combine function calls with other operators. The premise of their paper, which we confirm in our work, is that C/C++ code can be generated that will significantly outperform the math library [84]. They also suggest that direct access sampling is impractical, and that cache sizes in current systems do not limit LUT performance. In this work we show that direct access is sufficient for some applications, and we find that cache availability is a critical factor for LUT performance.

Hardware and software environments have fundamental differences with respect to LUT implementation. First, the cost of dedicated memory in a hardware is very high, so LUT sizes are usually limited to 64KB or less. This requires designers to compensate with high accuracy sampling methods such as polynomial reconstruction [19]. Such techniques can be implemented in hardware with multiple computational units that operate in parallel to provide good performance. In addition, hardware is more difficult to modify for specific applications, so designers must pay more attention to accuracy up front. Software, on the other hand, has access to abundant memory resources. LUTs of 6MB or more are feasible on current systems, even when the LUT data is constrained to reside in mid-level cache. Software memory resources are extremely flexible, and can be reconfigured to match the requirements of each application, thus they can address a much broader range of expressions. However, software cannot support the same level of complexity when reconstructing samples as hardware, without risking poor performance.

3.1.3 Cache Impact on LUT Performance

As previously mentioned, there is little published research on software LUT methods. In particular, we are not aware of work on domain profiling, error analysis, or performance modeling in the context of software LUTs, nor have we seen case studies that characterize the performance versus accuracy tradeoff, or that explore the cost versus benefit of the various LUT sampling methods. The issue of whether LUT data must reside in cache to make LUT transformation beneficial is especially important, but the only related work that we are aware of that discusses the impact of cache usage is Defour [18]. Defour combines polynomial reconstruction with small tables, and suggests that LUT data must fit into L1 cache to be effective. Our research has shown a substantial performance gain even when LUT data resides in L2 or L3 cache [76].

3.2 Other Techniques that Exploit Reuse

The reuse of computation is fundamental to a number of optimization techniques. For example, value locality is often exploited by compilers via result caching and other forms of value reuse. In this section we discuss techniques that take advantage of reuse, and we define the terms precise and fuzzy as they apply to reuse. We conclude with a description of memoization.

3.2.1 Value Locality and Value Reuse

Many programs exhibit *value locality*, in which computations are performed repeatedly with a small number of inputs [63]. Value locality creates redundancy that can be exploited to improve performance. Optimizations based on *value reuse* eliminate redundant computation by storing and reusing results from previous computations [40]. Some compilers implement value reuse by selectively caching results to avoid future computation [21]. Hardware can also exploit value reuse by caching results or sequences of instructions, or through speculative value prediction [14]. Most implementations of value reuse apply the method to a small set of precise inputs over localized areas in a program, in contrast to LUT transformation, which exploits value reuse more fully by allowing imprecision.

3.2.2 Explanation of Reuse Types

Value reuse can be exploited even without value locality. Consider a program that repeatedly evaluates one or more expressions with identical or similar inputs. Caching previously computed results can provide a benefit even when the computation is distributed throughout the program, if the reuse is applied globally. Performance gains depend on the reuse inherent to the program, which provides the opportunity to eliminate redundant computation. We use the term *precise reuse* when input values must exactly match [30]. Precise reuse provides the same accuracy as the original expression, if the cache has sufficient precision. Alvarez et al. introduced *fuzzy reuse*, in which input values can match imprecisely [2]. Fuzzy reuse improves performance by sharing cached results between multiple inputs, at the expense of a loss of accuracy. LUT transformation for function evaluation is based on fuzzy reuse.

3.2.3 Memoization versus LUT transformation

Memoization reuses computation by caching results in a data structure called the *reuse table*, which maps input values to results in a similar fashion to a lookup table [63]. At program launch the reuse table is empty, and results are cached only as needed during program execution [30]. When and how tables are built is the primary difference between memoization and LUT transformation, since the latter computes results for the entire domain in advance. Memoization is invoked each time a computation is performed. Before doing the computation, the input value is mapped to its corresponding table entry. If the entry exists its value is returned, otherwise the computation is completed and a new table entry is created to store the result. The mapping function is kept as lightweight as possible, commonly by using associative data structures such as hash tables [21, 30].

Memoization can be combined with precise or fuzzy reuse [2]. Using fuzzy reuse makes the technique similar to LUT transformation, except for the overhead of identifying whether or not a result has been previously cached. In contrast to LUT transformation, memoization only stores values that are used by the computation. As a result, memoization may be able to handle very large domains that are sparsely sampled more easily. On the other hand, LUT transformation provides better performance for small domains that are sampled heavily, because the overhead is smaller. We are not aware of a direct comparison of the two techniques anywhere in the literature.

3.3 Productivity in Scientific Computing

Implementing LUT transforms by hand requires significant time and effort. At the same time the productivity of scientific programmers is a current topic of research [70]. The growing importance of high-performance computing (HPC) has led to an interest in the software development processes and environments used by scientific programmers [11]. Manual tuning methods are thought to be responsible for an increasing amount of the development time and effort [23], and there is concern that such tuning may obscure the original algorithms [44]. One point of agreement is that automation reduces effort by freeing programmers from low-level details [13]. In the parallel programming community, high-level abstractions are being pursued as the path to higher productivity [6]. For example, programming models such as OpenMP decrease the programming effort by providing pragmas to convert sequential programs into parallel ones [16], thereby avoiding costly development of parallel code.

One problem is the difficulty of agreeing on objective measures for productivity [82, 17, 68], particularly for activities such as performance tuning that have no clear endpoint. Empirical studies are needed to understand the high costs often associated with HPC development [35], but these studies face a variety of challenges [67]. Our research centers on the automation of performance tuning, supported by the insertion of pragmas as a substitute for developing code.

Chapter 4

LUT Optimization Methodology

The objectives of our methodology are to significantly increase the performance of compute-bound scientific applications, improve programmer productivity while doing so, and give the programmer control over the tradeoff between accuracy and performance inherent to LUT methods. Our methodology leverages and extends the techniques discussed in Section 2, and we introduce automation to the LUT optimization process to help the programmer identify the most effective set of LUT transformations.

Our approach to LUT optimization is fundamentally different from existing LUT techniques. In current practice, the programmer identifies expressions in the source code that can benefit from LUT transformation, then they manually develop LUT code that is specific for each application. The programmer must often experiment with the LUT implementation to evaluate its effect on accuracy and performance, tweaking the code and data as necessary. Because of the *ad hoc* nature of the process, achieving a solution that is optimal is difficult. In contrast, our approach systematically evaluates expressions in the source code to find the most effective set of potential LUT transforms. We do this by constructing and solving the LUT optimization problem.

The input to the LUT optimization problem is a set of expressions extracted from application source code. These include (1) expressions involving elementary function calls, (2) individual elementary functions that appear more than once, and (3) parameter expressions to elementary function calls. The LUT optimization problem consists of selecting subsets of these expressions for LUT transformation that provide Pareto optimal points in the performance and accuracy tradeoff space. The selection is based on the accuracy and performance impact of the LUT transform for each expression, which we estimate through error analysis and performance modeling. The output of the LUT optimization problem is a list of Pareto optimal solutions that represent sets of potential LUT transforms.

The crux of the LUT optimization problem is to compare solutions to find the best tradeoff between performance benefit and accuracy. We do this by constructing and solving a numerical optimization problem, similar to the knapsack problem described in Section 1. The complexity of this problem makes it impractical to solve without automation, even when the number of expressions is very limited. To compare solutions, we combine the error and benefit estimates of each of their component LUT transforms. We then apply an algorithm that finds the Pareto optimal solutions. These are presented to the programmer along with the estimated accuracy and performance for each solution.

Our methodology is implemented in the Mesa tool, which automates much of the process through a series of stages that perform code analysis and transformation. Figure 4.1 shows the six stages in our methodology and tool and indicates three stages where significant user interaction occurs and one stage with minor user interaction. Throughout this chapter we present the workflow from the viewpoint of the programmer, describing how the Mesa tool is used and what results are achieved. We defer the details of expression enumeration, error analysis, performance modeling, and numerical optimization until Section 5.

4.1 Performance Profiling and Scope Identification

We demonstrate our methodology with an example program that is complex enough to show off our methodology and tool, but simple enough to present in entirety. The program provides a suitable example because it is limited by the computation of elementary functions, and the domain of input values for the program enables us to satisfy accuracy requirements while fitting in cache memory. Figure 4.2 shows the source code for the dominant method in the example program, with the statements numbered from S13 to S29. By inspection, we see that the function has many elementary function calls.

In the first stage of our methodology we use existing *performance profiling* tools to gather information about application performance. Our goal is to identify the code that consumes

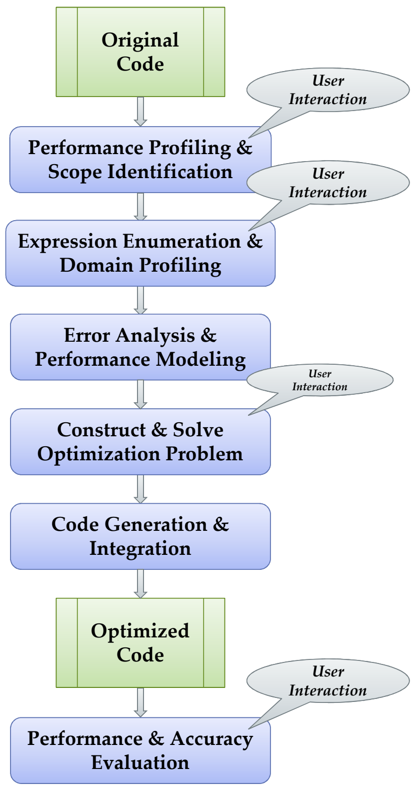

Original Code
User Interaction
Performance Profiling & Scope Identification
User Interaction
Expression Enumeration & Domain Profiling
Error Analysis & Performance Modeling
User Interaction
Construct & Solve Optimization Problem
Code Generation & Integration
Optimized Code
User Interaction
Performance & Accuracy Evaluation

Figure 4.1: Methodology for automated optimization.

```
double Calculate()
{
S13     double x, y, z;
S14     double fResult = 0.0;
S15
S16     for (x = -1.0; x < 1.01; x += 0.05)
S17     {
S18         z = x + 0.33;
S19         fResult += cos(z) / 2.0;
S20         z = x + 0.67;
S21         fResult += exp(z) * 3.0;
S22         for (y = 0.1; y < M_PI; y += 0.05)
S23         {
S24             fResult += sin(x) + cos(x);
S25             fResult += exp(x) + sin(y);
S26             fResult += exp(x) + sqrt(y);
S27         }
S28     }
S29     return fResult;
}
```

Figure 4.2: Source listing for methodology example program.

the majority of the program execution time. Such performance bottlenecks are easily located using commonly available tools such as *gprof* [28]. Profiling tools identify the most expensive functions in an application, which in turn allows the programmer to limit the scope of the optimization. Figure 4.3 shows the gprof output for the example program, which reports that 99.99% of the execution time is spent in 100,000 calls to the Calculate method.

```
Each sample counts as 0.01 seconds.
  %       total
 time     calls
99.99    100000    CExample::Calculate()
00.01         1    CExample::Execute()
```

Figure 4.3: Profiling output for methodology example program.

After performance profiling, the programmer specifies the functions that they want Mesa to analyze. This is referred to as *scope identification*. In our methodology, a programmer identifies scope by inserting pragma statements above function declarations. Mesa considers LUT transformation only within these functions. The pragma identifies the scope as the body of the specified function. The selection of entire functions has the advantage of matching the granularity of profiling data from gprof. Multiple functions can be optimized simultaneously, so the scope could be theoretically be extended to include the entire program. However, the cost of analysis increases quickly as functions are added. Figure 4.4 shows the function declaration and pragma. The pragma causes Mesa to analyze statements S13 through S29 in the example code in Figure 4.2.

```
  #pragma LUTOPTIMIZE
  double CExample::Calculate()
  {
    ...
  }
```

Figure 4.4: Pragma insertion into methodology example program.

4.2 Expression Enumeration and Domain Profiling

The LUT optimization problem depends on the identification of expressions for which LUT transformation may be beneficial. Mesa does this through *expression enumeration*, which uses static analysis of the abstract syntax tree (AST) for the source code to extract expressions. The programmer starts program analysis by running Mesa in instrumentation mode. Mesa identifies and parses statements that contain elementary function calls. Figure 4.5

```
>>> ./Mesa Original.cpp Profile.cpp -profile
Mesa 2.0: Optimization started.
Mesa 2.0: Optimizing Calculate method.
S19: Original.cpp (line 19) fResult +=(cos(z) / 2.0)
S21: Original.cpp (line 21) fResult +=(exp(z) * 3.0)
S24: Original.cpp (line 24) fResult +=(sin(x) + cos(x))
S25: Original.cpp (line 25) fResult +=(exp(x) + sin(y))
S26: Original.cpp (line 26) fResult +=(exp(x) + sqrt(y))
Number of Statements = 5
S19: X0 = cos(z)
S19: X1 = (cos(z)/2.00000000e+00)
S21: X2 = exp(z)
S21: X3 = (exp(z)*3.00000000e+00)
S24: X4 = sin(x)
S24: X5 = cos(x)
S24: X6 = (sin(x)+cos(x))
S25: X7 = exp(x)
S25: X8 = sin(y)
S26: X9 = exp(x)
S26: X10 = sqrt(y)
Number of Expressions = 11
[X0,X1]
[X2,X3]
[X4,X6]
[X5,X6]
Number of Constraints = 4
Mesa 2.0: Generating profiling code
Mesa 2.0: Optimization completed.
```

Figure 4.5: Instrumentation run for methodology example program.

shows the instrumentation run for the example program, including the Mesa invocation. Mesa displays statements, expressions, and constraints extracted from the example program.

Expression enumeration is key to our methodology, because each enumerated expression represents a potential LUT transformation. Mesa parses the candidate statements and extracts expressions according to the criteria described in more detail in Section 5.1. The most important criteria are that each expression must include only one variable and at least one elementary function call. For the example program, Mesa extracts 11 expressions from the 5 statements as shown in Figure 4.5. These include 8 individual elementary functions and 3 more complex expressions. The latter come from the inclusion of arithmetic operators in S19 and S21, and the combined expression $sin(x) + cos(x)$ in S24. Mesa extracts only elementary functions from S25 and S26, because the combined expressions $exp(x) + sin(y)$ and $exp(x) + sqrt(y)$ in these statements have multiple variables.

Expression enumeration decomposes statements into expressions that in some cases can overlap. For example, the expressions X0 and X1 intersect because both of them contain the cosine call from statement S19, and X4 and X6 intersect because they contain the sine call from statement S24. Mesa disallows the simultaneous transformation of overlapping expressions by detecting such intersections and storing them as constraints. Figure 4.5 shows the four intersection constraints {X0 $\cap$ X1, X2 $\cap$ X3, X4 $\cap$ X6, X5 $\cap$ X5} for the example program. The number of expressions and constraints are important because they determine the complexity of the optimization problem.

After expression enumeration, we can construct the LUT optimization problem by generating a set of solutions that consist of the power set of expressions. This set has a theoretical complexity of O(2^N) for N expressions, and the number of expressions can increase because of expression coalescing, or decrease because of intersection constraints, both of which are explained in Section 5.1. However, the complexity of the LUT optimization problem still grows exponentially with the number of expressions. As a result, solving the optimization problem for large numbers of expressions is unfeasible. Our current algorithm can process up to ∼24 expressions in a reasonable amount of time. To handle more expressions we have developed a culling algorithm that reduces the search space, as described in Section 5.4.4.

The next step in our methodology is *domain profiling*, which gathers data concerning program execution for the modeling stage. A LUT domain is defined by the range of input values for which results must be stored. The input values are generally unavailable at compile-time, so Mesa helps the programmer to collect domain information at runtime by providing an instrumented version of the application. The instrumented version replicates the original code and adds profiling calls for each input variable. Additional instrumentation is inserted to gather execution frequency for each candidate statement, which is needed for the performance model. Mesa generates the instrumented program automatically. Figure 4.6 shows a partial listing of the instrumented code for statements S24 through S26. S25 and S26 each profile both the x and y variables. Additional code (not shown) is inserted to write profiling variables and counters to a file when the program exits

To complete the profiling stage, the programmer compiles and runs the instrumented code with one or more representative data sets. The instrumented code writes text files with input domain and call frequency information, and the programmer can edit these files. One

```
S24++;
if (x < S24_x_low) S24_x_low = x;
if (x > S24_x_high) S24_x_high = x;
fResult +=(sin(x) + cos(x));
S25++;
if (x < S25_x_low) S25_x_low = x;
if (x > S25_x_high) S25_x_high = x;
if (y < S25_y_low) S25_y_low = y;
if (y > S25_y_high) S25_y_high = y;
fResult +=(exp(x) + sin(y));
S26++;
if (x < S26_x_low) S26_x_low = x;
if (x > S26_x_high) S26_x_high = x;
if (y < S26_y_low) S26_y_low = y;
if (y > S26_y_high) S26_y_high = y;
fResult +=(exp(x) + sqrt(y));
```

Figure 4.6: Profiling instrumentation for methodology example program.

reason to edit the files is to expand the domains to safely handle data sets that have not been run. Another reason to edit the files is that a programmer may have domain knowledge that allows the setting of domain boundaries without running the instrumented code at all.

Call frequency is stored for each statement, and domain information is stored for each variable and statement. This is necessary because variables can change from one candidate statement to the next. Figure 4.7 lists the domain and call frequency files from a profiling run of the example code. Note the change in the input domain of the variable z in statements S19 to S21 caused by an intervening arithmetic operation. In addition, the three statements in the inner loop have a much higher call frequency.

```
>>> more mesa.domains
S19 z -0.670000 1.330000
S21 z -0.330000 1.670000
S24 x -1.000000 1.000000
S25 x -1.000000 1.000000
S25 y 0.100000 3.100000
S26 x -1.000000 1.000000
S26 y 0.100000 3.100000
>>> more mesa.frequencies
S19 4100000
S21 4100000
S24 250100000
S25 250100000
S26 250100000
```

Figure 4.7: Profiling data for methodology example program.

After domain profiling, the programmer starts the LUT optimization process by running Mesa in optimization mode. Mesa enumerates exactly the same set of expressions as during the instrumentation run, and fetches their domain boundaries and call frequencies from the files shown in Figure 4.7. Mesa then uses an algorithm that we call *expression coalescing* to decrease memory usage and improve accuracy.

Expression coalescing shares LUT data between expressions to conserve scarce cache memory resources. The expressions being coalesced can appear in different statements or methods. The coalescing algorithm searches for sets of expressions that have the same operators, constants, and variables, in the same order. Comparison of variables is based on the input domain, not variable names. For example, $sin(x)$ and $sin(y)$ can share a single LUT transformation for the sine function when the domains of x and y are equal. Even when the domains differ, we can still coalesce expressions by expanding the LUT data to encompass the domains of both input variables, but this may decrease accuracy. Section 5.1.2 describes the heuristic we use to compare domains.

Figure 4.8 shows expression coalescing in the listing of the Mesa optimization run for the example program. Mesa coalesces the sine calls in X0 and X4 into a new expression X11, the cosine calls in X0 and X5 into X12, and so on. The new expressions intersect with the expressions they coalesce, so Mesa must add new intersection constraints. For example, the new expression X11 introduces the constraints $\{$X4 $\cap$ X11, X8 $\cap$ X11, X6 $\cap$ X11$\}$. The latter is inherited from the previously existing X4 $\cap$ X6 constraint. Expression coalescing is often applicable to elementary functions because of their restricted domains. The complexity of the LUT optimization problem increases because of the expressions introduced by coalescing, but the technique can improve accuracy and performance.

Table 4.1 shows the candidate expressions for the example program, after expression coalescing. The domain boundaries and call frequencies added during the profiling stage are shown in bold. The coalesced expressions apply to more than one statement, so multiple statement identifiers are listed. The domain boundaries and call frequencies of the new expressions combine those of their constituent expressions. For example, the expression X12 lists statements S19 and S24, and has a combined domain of [-1.0,1.33] and a combined

```
Mesa 2.0: Coalescing expressions (conservative)
Coalescing X1 from X4 X8
Adding [X4,X11]
Adding [X6,X11]
Adding [X8,X11]
Coalescing X12 from X0 X5
Adding [X0,X12]
Adding [X1,X12]
Adding [X5,X12]
Adding [X6,X12]
Adding [X11,X12]
Coalescing X13 from X2 X7 X9
Adding [X2,X13]
Adding [X3,X13]
Adding [X7,X13]
Adding [X9,X13]
```

Figure 4.8: Expression coalescing for the methodology example program.

Table 4.1: Domains and frequencies for methodology example program.

Expression Identifier	Expression Description	Statement Identifiers	Input Domain	Call Frequency
X0	cos(z)	S19	**-0.67,1.33**	**4,100,000**
X1	cos(z) / 2.0	S19	**-0.67,1.33**	**4,100,000**
X2	exp(z)	S21	**-0.33,1.67**	**4,100,000**
X3	exp(z) * 3.0	S21	**-0.33,1.67**	**4,100,000**
X4	sin(x)	S24	**-1.0,1.0**	**250,100,000**
X5	cos(x)	S24	**-1.0,1.0**	**250,100,000**
X6	sin(x) + cos(x)	S24	**-1.0,1.0**	**250,100,000**
X7	exp(x)	S25	**-1.0,1.0**	**250,100,000**
X8	sin(y)	S25	**0.1,3.1**	**250,100,000**
X9	exp(x)	S26	**-1.0,1.0**	**250,100,000**
X10	sqrt(y)	S26	**0.1,3.1**	**250,100,000**
X11	sin(x or y)	S24, S25	**-1.0,3.1**	**500,200,000**
X12	cos(x or z)	S19, S24	**-1.0,1.33**	**254,200,000**
X13	exp(x or z)	S21, S25, S26	**-1.0,1.67**	**504,300,000**

frequency of 254.1 million. The original expressions remain in the list with the combined expressions, because we want the optimization problem to consider them separately.

4.3 Error and Performance Modeling

The modeling stage estimates the accuracy and performance of potential LUT transformations for each expression. These estimates are used by the subsequent LUT optimization stage to select sets of expressions for which LUT transformation is most effective. The modeling stage consists of *error analysis* and *performance modeling*, both of which are automated.

Error analysis depends on an error equation that characterizes the maximum error for each LUT transformation. We use maximum error as a conservative measure for comparing the accuracy of LUT data. For direct access, the error equation references the LUT domain and size and function slope. For linear interpolation, the change in the slope is used instead of the slope. We explain the error equation and describe analytic and numerical methods for finding the slope and change in slope in Section 5.2. The error equation allows the optimization stage to model the LUT approximation error for a range of LUT sizes.

Performance modeling estimates the benefit of each LUT transformation as a function of the cost of elementary functions, arithmetic operators, and table access. In optimization mode, Mesa runs an embedded benchmark to get timing information for each of these. The expression cost is computed by summing the costs of the expression operators, then subtracting the LUT access time. Benefit models the potential savings in execution time, and the benefit is multiplied by call frequency since each execution saves computation. Our performance model also incorporates the relative performance of linear interpolation versus direct access. Section 5.3 describes the performance model in more detail.

Table 4.2 again lists the candidate expressions for the example code, now with the maximum slope and performance benefit from the modeling stage. The domain column shows the boundaries and extent, computed as the upper bound minus the lower bound. To reduce complexity, we have limited the example to direct access sampling.

To further clarify Table 4.2, we show how the values are calculated. The domain extent is computed as the absolute difference between domain boundaries. For example, the boundaries for X3 are [-0.33,1.67], so the domain magnitude is |1.33-(-0.67)| = 2.00. To find the maximum slope for X2, we look for the maximum of the derivative over the domain. The derivative of exp(x) equals exp(x) for the exponential function. The slope of the exponential increases monotonically, so we know the maximum is at the right boundary. We therefore compute the maximum slope as exp(1.67) = 5.31. Mesa does this with numerically and finds

Table 4.2: Slope and benefits for methodology example program.

Expression Identifier	Expression Description	Statement Identifiers	Input Domain	Maximum Slope	Performance Benefit	Call Frequency
X0	cos(z)	S19	[-0.67,1.33] = 2.00	**0.97**	**0.154s**	4,100,000
X1	cos(z) / 2.0	S19	[-0.67,1.33] = 2.00	**0.49**	**0.162s**	4,100,000
X2	exp(z)	S21	[-0.33,1.67] = 2.00	**5.31**	**0.101s**	4,100,000
X3	exp(z) * 3.0	S21	[-0.33,1.67] = 2.00	**15.94**	**0.105s**	4,100,000
X4	sin(x)	S24	[-1.0,1.0] = 2.00	**1.00**	**8.153s**	250,100,000
X5	cos(x)	S24	[-1.0,1.0] = 2.00	**0.84**	**9.404s**	250,100,000
X6	sin(x) + cos(x)	S24	[-1.0,1.0] = 2.00	**1.41**	**19.660s**	250,100,000
X7	exp(x)	S25	[-1.0,1.0] = 2.00	**2.72**	**6.152s**	250,100,000
X8	sin(y)	S25	[0.1,3.1] = 3.00	**1.00**	**8.153s**	250,100,000
X9	exp(x)	S26	[-1.0,1.0] = 2.00	**2.72**	**6.152s**	250,100,000
X10	sqrt(y)	S26	[0.1,3.1] = 3.00	**1.58**	**-0.075s**	250,100,000
X11	sin(x or y)	S24, S25	[-1.0,3.1] = 4.10	**1.00**	**16.310s**	500,200,000
X12	cos(x or z)	S19, S24	[-1.0,1.33] = 2.33	**0.97**	**9.558s**	254,200,000
X13	exp(x or z)	S21, S25, S26	[-1.0,1.67] = 2.67	**5.31**	**12.410s**	504,300,000

the same value. The performance benefit for X6 is computed by adding the costs of a sine call (40ns), a cosine call (45ns), and an addition (1.0ns), and subtracting the cache access time (7.4ns), which yields a benefit of 78.6ns for a single execution. Multiplying 78.6ns by the call frequency of 2.5×10^8 gives an expression benefit of 19.7s. The performance benefit for X10 is negative because the square root is faster then the LUT access on the test system. For all calculations we have used the costs from Table 1.1.

4.4 Solving the LUT Optimization Problem

After completion of the modeling stage, we now have enough information to build and solve the LUT optimization problem. Figure 4.9 shows a partial listing of the Mesa optimization run. The listing starts with the error and performance modeling stages, and ends with the presentation of Pareto optimal solutions to the programmer. The listing of expressions shown in Table 4.2 is omitted to avoid duplication. Mesa displays the number of possible solutions as 2^{14} or 16,384 for the 14 expressions in the example, and the number of actual solutions as

```
Mesa 2.0: Running error analysis (boundary)
Mesa 2.0: Applying performance model
Mesa 2.0: Solving optimization problem
Optimizing for cache size 4194304
...
16384 solutions (possible)
1040 solutions (actual)
9 solutions (pareto optimal)
Solution  Error      Benefit    Optimizations
C0        0.000e+00  0.000e+00
C1        1.899e+00  1.624e+08  X1
C2        3.373e+02  1.966e+10  X6
C3        1.389e+03  2.781e+10  X6,X8
C4        3.936e+03  3.396e+10  X6,X8,X9
C5        7.779e+03  4.012e+10  X6,X8,X7,X9
C6        8.746e+03  4.028e+10  X6,X8,X7,X9,X1
C7        1.073e+04  4.038e+10  X6,X8,X13,X1
C8        1.436e+04  4.038e+10  X6,X8,X7,X9,X1,X3
Select solution:
```

Figure 4.9: Optimization solution for the methodology example program.

1,040 after intersection culling. The 16× decrease in solutions is not unusual when expression coalescing is enabled, because coalescing introduces new intersection constraints. From the 1,040 potential solutions, our algorithm has discovered 9 Pareto optimal solutions, which are listed in order of benefit. Mesa lists error as an absolute number, benefit in nanoseconds, and shows the list of expressions that comprise each solution.

To calculate the estimated errors shown in Figure 4.9, Mesa first determines the optimal cache allocation for each expression. The programmer specifies the cache size when running Mesa, and the error equations enable computation of error values for any allocation within that size. We have developed a formula derived from the error equation that computes the optimal allocation for a set of expressions, as described in Section 5.4.2. Mesa uses the formula to calculate the allocation, then applies the error equation to compute the expression error based on the allocated size. By summing the expression errors, Mesa gets the estimated solution error. Solution benefit is computed in a similar fashion as the sum of expression benefits. For example, the benefit for C3 is the sum of the benefits for X6 and X8. Adding 1.966 $\times 10^{10}$ns to 8.153 $\times 10^{9}$ns yields a benefit of 2.781 $\times 10^{10}$ns. Thus the performance model estimates that realizing C3 will cause the example program to run 27.8s faster.

Figure 4.10 shows the solution to the optimization problem for the data in Figure 4.9, which we call a Pareto chart. Solution error is on the x-axis, solution benefit is on y-axis,

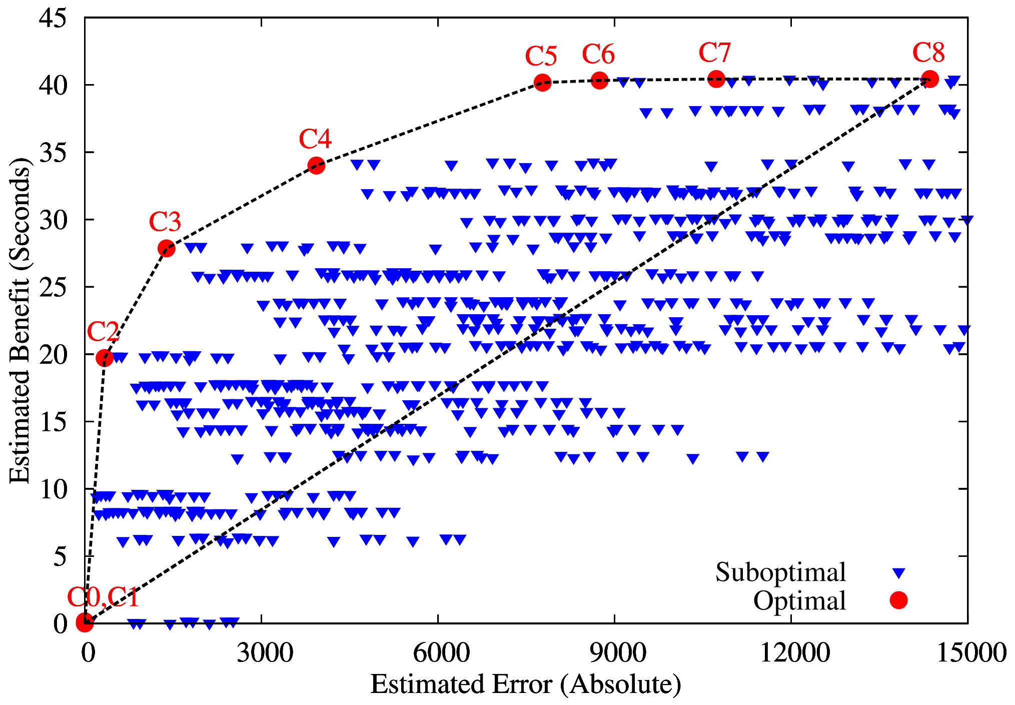


Figure 4.10: Pareto chart for methodology example program.

suboptimal solutions are triangles, and Pareto optimal solutions are circles. Optimal solutions lie above and the left of suboptimal solutions, because they provide more performance with the same or less error. To find the nine Pareto optimal points shown, Mesa uses a convex hull algorithm. The C0 solution is the empty set of expressions, which is equivalent to the original code. The C1 solution contains only the X1 expression, which has so little error and benefit that is almost coincides with C0. The C2 solution adds X6, which has the most benefit of any expression, C3 adds X8 which has the next most benefit, and so on. C5 appears to be the best solution, because it provides 99% of the benefit but only 54% of the error, as compared to C8. This is reflected in the Pareto optimal line in Figure 4.10, which flattens out after C5, thus providing diminishing returns in terms of performance. Figure 4.9 also shows that expression coalescing has not contributed to any solutions except C7.

Mesa completes the optimization stage by presenting the Pareto optimal solutions to the programmer. The solutions are accompanied by the estimated error and performance. These values are intended to represent proxies for application error and performance. Sections 5.2 and 5.3 evaluate the effectiveness of our proxies and find that estimated benefit is generally a good predictor of application performance, but that estimated error often fails to predict application accuracy. The reason for this is the difficulty of estimating how the error introduced by a LUT transformation will propagate through the application. Even so, the error model does correctly indicate the maximum error that can be introduced, and therefore provides a basis for comparing LUT transforms.

Figure 4.11 shows how a solution is selected by the programmer. Mesa lists all of the Pareto optimal choices, and waits for the programmer to select a solution. After the programmer selects solution C5, Mesa shows the optimal LUT sizes for a realization of the solution. For the example run, Mesa was given a cache size of 4MB. Mesa outputs the domain extent (Di), maximum slope (Mi), maximum error (Ei), performance benefit (Bi), and table size (Si). The cache allocations for X7 and X9 are ∼1.2MB each, because both expressions have the same domain and maximum slope. The allocations for X6 and X8 are smaller because the associated LUT transforms are more accurate. The sums of the expression errors and benefits columns are equal to the solution error and benefit for C5, the selected solution.

```
Solution   Error       Benefit     Optimizations
C0         0.000e+00   0.000e+00
C1         1.899e+00   1.624e+08   X1
C2         3.373e+02   1.966e+10   X6
C3         1.389e+03   2.781e+10   X6,X8
C4         3.936e+03   3.396e+10   X6,X8,X9
C5         7.779e+03   4.012e+10   X6,X8,X7,X9
C6         8.746e+03   4.028e+10   X6,X8,X7,X9,X1
C7         1.073e+04   4.038e+10   X6,X8,X13,X1
C8         1.436e+04   4.038e+10   X6,X8,X7,X9,X1,X3
Select solution: 5
X6 Di = 2.00 Mi = 1.41 Ei = 1.620e+03 Bi = 1.966e+10 Si =  873405 (853KB)
X8 Di = 3.00 Mi = 1.00 Ei = 1.668e+03 Bi = 8.153e+09 Si =  899116 (878KB)
X7 Di = 2.00 Mi = 2.72 Ei = 2.246e+03 Bi = 6.152e+09 Si = 1210891 (1183KB)
X9 Di = 2.00 Mi = 2.72 Ei = 2.246e+03 Bi = 6.152e+09 Si = 1210891 (1183KB)
Mesa 2.0: Generating optimized code
Realizing X6 in statement S24
Realizing X8 in statement S25
Realizing X7 in statement S25
Realizing X9 in statement S26
Mesa 2.0: Optimization completed.
```

Figure 4.11: Optimization selection for methodology example program.

4.5 Code Generation and Integration

The next stage includes *code generation* and *code integration*. After the programmer selects a solution, Mesa realizes the necessary code for the specified set of expressions, including the LUT data, constructor, destructor, initialization code, table access, and original function. Mesa then integrates the generated code into the application and replaces original expressions with LUT access calls. An optimized version of the application code is written to the file specified on the command line. The programmer rebuilds the application using the normal build process, and the resulting executable should behave in an identical manner to the original program, except for differences in performance and accuracy. Figure 4.12 shows a partial listing of the code generated by Mesa for the example code. To save space, we show only the code for the X6 expression, and the modifications to the original expressions.

4.6 Performance and Accuracy Evaluation

In the last stage in our methodology, the programmer evaluates the benefit and accuracy of the optimized version of the program against the original version. We compute performance speedup as the original execution time divided by the optimized execution time. A beneficial optimization is defined as a performance speedup greater than 1.0. Figure 4.13 shows our evaluation of the example code. The original and optimized versions of the program are timed, showing a 8.1$\times$ speedup.

Evaluation of accuracy requires a comparison of program results. The original output is assumed to be the oracle, and the error is measured as the deviation of the optimized output. The error values are computed as shown in numerical analysis textbooks [12], with absolute error equal to the absolute difference between the results, and relative error equal to the absolute error divided by the original result. The example code accumulates the values returned from the Calculate function, and displays the result on completion as shown in Figure 4.13. We compare these values to find an absolute error of 9.2×10^{-4} and a relative error of $7.1x10^{-6}\%$, despite having approximated $\sim$2.5 billion elementary function calls.

```
// Start of optimization code generated by Mesa, version 2.0
// LUT constants
const double X6_lower = -1.0000000000e+00;
const double X6_upper = 1.0000000000e+00;
const double X6_granularity = 9.1595542680e-06;
class CLut {
  public:
    // LUT Constructor
    CLut()
    {
        for (double dInput = X6_lower; dInput <= X6_upper; dInput += X6_granularity)
        {
            X6_data.push_back(X6_orig(dInput + (X6_granularity / 2.0)));
        }
    }
    // LUT Destructor
    ~CLut()
    {
        X6_data.clear();
    }
    // LUT functions
    float X6_lut(float X6_param)
    {
        X6_param -= X6_lower;
        int uIndex = (int) (X6_param * (1.0 / X6_granularity));
        return(X6_data[uIndex]);
    }
    // Original functions
    double X6_orig(double x)
    {
        return (sin(x)+cos(x));
    }
  private:
    // LUT data structures
    std::vector<float> X6_data;
};
// Object instantiation
CLut clut;
// End of optimization code generated by Mesa, version 2.0

// Expressions replaced by Mesa
S24 fResult += clut.X6_lut(x);
S25 fResult += clut.X7_lut(x) + clut.X8_lut(y);
S26 fResult += clut.X9_lut(x) + sqrt(y);
```

Figure 4.12: Optimized code generated by Mesa.

```
>>> time ./TestOrig
Accumulated value = 12957.614419
real    0m23.656s
user    0m23.651s
sys     0m0.000s

>>> time ./TestOptd
Accumulated value = 12957.613499
real    0m2.932s
user    0m2.924s
sys     0m0.006s
```

Figure 4.13: Evaluation of the optimized code generated by Mesa.

The evaluation stage is complete when the programmer has the accuracy and performance information necessary to evaluate whether the optimization is worthwhile. The programmer can choose to accept the optimization and use the Mesa generated code, run Mesa again and pick a different solution, or revert to the original code. Mesa has a parameter that specifies the selection of a solution so that the iterative process just described can be scripted.

Chapter 5

LUT Optimization Algorithms

Chapter 4 introduces several algorithms that are complex enough to merit further explanation including expression enumeration, error analysis, performance modeling, and the optimization problem. In this chapter we present detailed information about the algorithms, which we illustrate with the example code shown in Figure 5.1. The example code is similar to the SAXS continuous code evaluated in Section 7.3, but we have modified it to demonstrate several features of our methodology. The main change is the removal of two variables and two constants from the dSum0 and dSum1 computations.

```
#pragma LUTOPTIMIZE
double ScatterSample(Sample sample, vector<Cartesian> &vGeometry)
{
S35     double dProduct;
S36     double dSum0 = 1.0;
S37     double dSum1 = 1.0;
S38
S39     // Iterate geometry
S40     for (int j = 0; j < vGeometry.size(); j++)
S41     {
S42         dProduct = (sample.x * vGeometry[j].x) + (sample.y * vGeometry[j].y);
S43         dSum0 += exp(dProduct) + sin(dProduct);
S44         dSum1 += exp(dProduct) + cos(dProduct);
S45     }
S46
S47     // Return answer
S48     return dSum0 * dSum0 + dSum1 * dSum1;
}
```

Figure 5.1: Source listing for algorithms example program.

5.1 Expression Enumeration

Our first algorithm is expression enumeration, which identifies expressions in the candidate statements that may be suitable for LUT transformation. The enumeration traverses the AST for each statement and extracts subtrees that meet our criteria. The expressions represented by the subtrees are then evaluated as LUT transformation candidates. The algorithm applies four criteria when deciding whether or not to enumerate an expression:

1. The expression must contain **one or more elementary functions** since elementary functions are the focus of our LUT optimization methodology.
2. The expression must contain **exactly one variable** to avoid multi-dimensional LUT data, which we currently do not handle.
3. The expression must be a **contiguous subtree** of the original syntax tree so that we can maintain the original evaluation order.
4. The expression must be a **complete subtree**, as defined by every operator having the arguments it needs for evaluation.

Our algorithm discards expressions that cannot benefit from LUT optimization, and the remaining expressions are considered as candidates for LUT transformation. Per our criteria, a candidate expression must contain one or more elementary functions, and can also contain any number of arithmetic operators. Optimizing arithmetic gains little performance, but can be beneficial in the following cases. First, if the inclusion of an arithmetic operator increases the number of elementary functions, it may be worthwhile. For example, the expression $sin(y) + cos(y)$ has two elementary functions linked by an addition operator. Second, if the inclusion of an arithmetic operator reduces the amount of error for the associated LUT transform, it may be worthwhile. For example, $x = sin(y)/2.0$ has half the maximum slope of $sin(y)$ and the same domain, so it introduces half the error for the same memory usage.

Our strategy is to enumerate all of the expressions that meet the criteria and let the optimization process decide which are the most effective. Our enumeration algorithm works by enumerating all of the contiguous subtrees contained in the statement AST, then culling those that do not match the criteria. The resulting expressions can include everything from a single elementary function call to the entire statement.

Figure 5.2 shows the ASTs for statements S43 and S44 with circles representing operators and rectangles representing variables. The leaves of the subtree are the input variables (or constants) needed to evaluate the expression. Temporary variables that hold a subtree result are introduced to enable decomposition. This allows the subtree to be replaced by a LUT transform that sets the value of the temporary variable, thereby providing an argument for

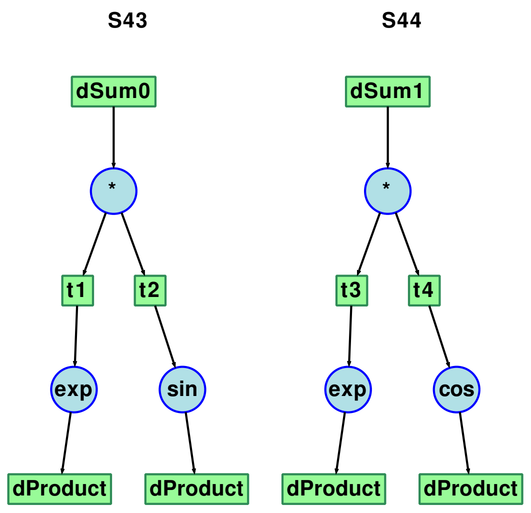


Figure 5.2: Syntax trees for candidate statements S43 and S44.

the operator that depends on evaluation of the subtree. For example, the $t1$ variable in Figure 5.2 enables the creation of a LUT transform for the exponential call, whose return value becomes the first argument to the multiply operator.

Each statement in the example code contains two variables, two elementary function calls, and one arithmetic operator, not counting temporary variables. The $dSum0$ and $dSum1$ variables are assignment variables for the statement, so they are not included in the optimization. For the remaining 5 nodes in each statement, the maximum number of possible subtrees is $2^5 - 1 = 31$. However, because of our criteria, only 3 subtrees are enumerated for each statement. Figure 5.3 shows several examples of expressions from statement S43 that do not meet our criteria. Expression XA ($t1 * t2$) has no elementary function, expression XB ($exp(dProduct)...sin(dProduct)$) is not contiguous, expression XC ($exp(...)$) is not complete because it is missing an argument to the exponential function, and expression XD ($exp(dProduct * t2)$) contains two input variables: $t2$ and $dProduct$.

Figure 5.4 shows the candidate expressions from statement S43 that meet all of the criteria. X0 and X1 are individual elementary functions, and X2 represents the entire statement. Each expressions is a subtree that can be replaced by a LUT transform that returns an approximation of the result previously computed by the subtree.

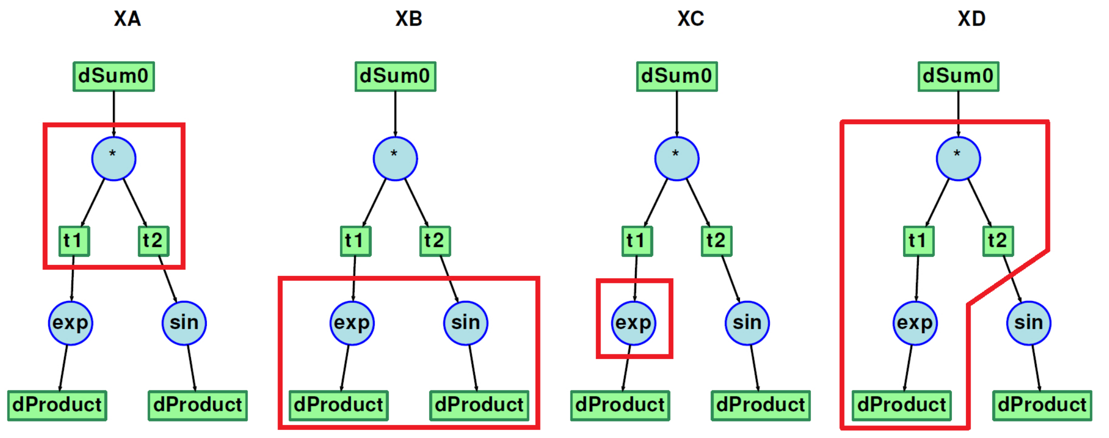


Figure 5.3: Invalid subtrees for candidate statement S43.

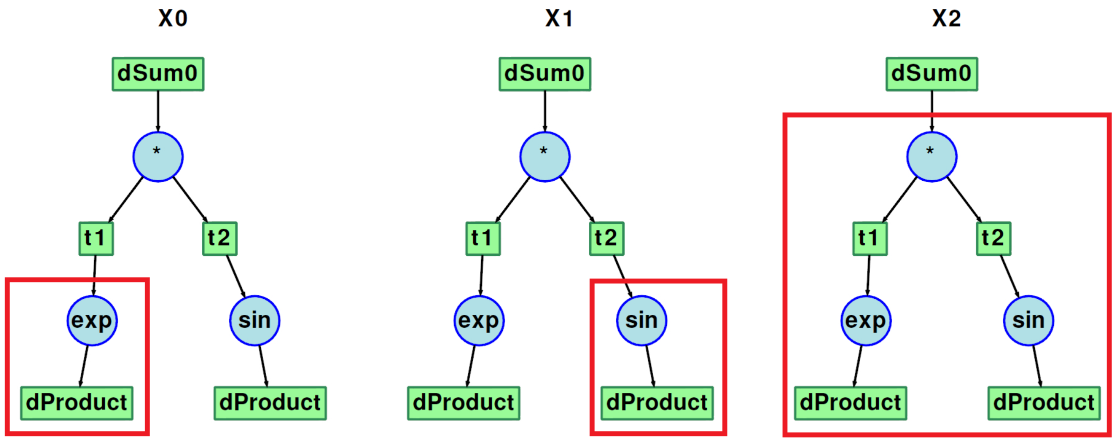


Figure 5.4: Valid subtrees for candidate statement S43.

Mesa 2.0 is the first version of our tool that implements expression enumeration, thereby introducing some new concerns. First, expression enumeration creates list of expressions that can overlap, and this must be taken into account during optimization. Second, expression enumeration improves our ability to improve efficiency and performance through resource sharing. Third, expression enumeration automatically handles certain types of expressions that were problematic for Mesa 1.1. In the subsequent sections we discuss each of these.

5.1.1 Intersection Constraints

Table 5.1 lists the enumerated expressions for both statements numbered from X0 to X5. For each expression we show the identifier, syntax, statements, and input variables. Expressions

Table 5.1: Enumerated expressions for algorithms example program.

Expression Identifier	Expression Description	Statement Identifiers	Input Variables
X0	exp(vProduct)	S43	dProduct
X1	sin(dProduct)	S43	dProduct
X2	exp(vProduct) * sin(dProduct)	S43	dProduct
X3	exp(vProduct)	S44	dProduct
X4	cos(dProduct)	S44	dProduct
X5	exp(vProduct) * cos(dProduct)	S44	dProduct

X0 through X2 come from statement S43 and expressions X3 to X5 come from statement S44. Table 5.1 shows that our enumeration algorithm can extract expressions that overlap with other expressions in the same statement. For example, X0 and X1 are both contained by X2, thus their subtrees intersect as can be seen in Figure 5.4. Overlapping expressions cannot be optimized simultaneously without causing redundant and incorrect computation. To enforce this, Mesa maintains *intersection constraints* to prevent these expressions from being combined in a solution. We define intersection constraints as pairs of expressions joined by the intersection operator. For the example code the intersection constraints are $\{X0 \cap X2, X1 \cap X2, X3 \cap X5, X4 \cap X5\}$. During setup of the optimization problem, solutions that contain expressions associated by an intersection constraint are culled.

5.1.2 Expression Coalescing

Expression enumeration also improves our ability to manipulate the candidate expressions to improve efficiency and performance. For example, we can combine similar expressions through expression coalescing. The purpose of coalescing is to save resources by sharing a LUT transform between two or more expressions. Coalescing is applied globally, so expressions that appear anywhere in the program can share resources. Mesa implements expression coalescing by finding sets of expressions that represent identical computations, i.e. they evaluate the same operators in the same order. The coalescing algorithm does not compare variable names since the domains of variables can change from one statement to the next.

Table 5.2: Coalesced expressions for algorithms example program.

Expression Identifier	Expression Description	Statement Identifiers	Input Domain	Call Frequency
X0	exp(dProduct)	S43	**[-1.246134, 1.1962899]**	**250,000,000**
X1	sin(dProduct)	S43	**[-1.246134, 1.1962899]**	**250,000,000**
X2	exp(dProduct) + sin(dProduct)	S43	**[-1.246134, 1.1962899]**	**250,000,000**
X3	exp(dProduct)	S44	**[-1.246134, 1.1962899]**	**250,000,000**
X4	cos(dProduct)	S44	**[-1.246134, 1.1962899]**	**250,000,000**
X5	exp(dProduct) + cos(dProduct)	S44	**[-1.246134, 1.1962899]**	**250,000,000**
X6	exp(dProduct)	S43, S44	**[-1.246134, 1.1962899]**	**500,000,000**

Instead, the domain extents of variables are compared when deciding whether or not to coalesce, so domain profiling precedes coalescing.

When we run the example code with expression coalescing enabled, the algorithm creates a new expression X6 that combines the identical $exp(dProduct)$ calls in X0 and X3. Table 5.2 shows the expression list after adding the coalesced expression. The X6 expression is associated with S43 and S44 and will be realized into both statements if it is part of the solution selected by the programmer. The call frequency of X6 is the sum of the frequencies of X0 and X3, which will increase its benefit. The domain of X6 combines the domains of X0 and X3, which could cause X6 to contribute more error. However, since these domains are identical in the example the X6 error term will be the same as the individual expressions. X0 and X3 remain in the list to allow the optimization problem to consider them separately. A new set of intersection constraints is introduced for X6, which overlaps with X0 and X3 and inherits their constraints. Figure 5.5 shows the expanded set of intersection constraints.

Different heuristics can be used for domain comparison, depending on how aggressively we wish to coalesce. Mesa currently supports three levels of coalescing. If the domains have no intersection, then coalescing is at best unproductive because no resources are shared.

```
Original Constraints:  {X0 ∩ X2, X1 ∩ X2, X3 ∩ X5, X4 ∩ X5}
Coalesce Constraints:  {X0 ∩ X6, X3 ∩ X6}
Inherited Constraints: {X2 ∩ X6, X5 ∩ X6}.
```

Figure 5.5: Intersection constraints for algorithms example program.

The *conservative* heuristic coalesces only when the domains are identical, since the resulting expression is guaranteed to have at least equal error and more benefit in this case than the component expressions. The *aggressive* heuristic indiscriminately coalesces all expressions with overlapping domains, but this can end up reducing accuracy instead of increasing it.

For example, consider the coalescing of two expressions with domain $[0.0, 0.1]$ and $[0.0, 10.0]$, using the same amount of memory for the combined expression as was allocated for the individual expressions. The result is to slightly increase the accuracy of the latter expression and greatly decrease the accuracy of the former. For this reason we have implemented a *moderate* heuristic in Mesa that coalesces only when the domain overlap is at least half of both original domains.

5.1.3 Parameter Merging

Another benefit of expression enumeration is *parameter merging*, which avoids the realization of unnecessary multi-dimensional LUT transformations by merging input variables. To illustrate this we consider optimization of the statement $x = exp(y/z)$. Mesa extracts expressions for the exponential by itself and the entire expression with the division. Optimizing just the exponential is achieved with a single-dimensional LUT indexed by (y/z), whereas the entire expression requires a multi-dimensional LUT indexed by y and z. The latter requires significantly more memory to achieve the same level of accuracy. Ideally, we would enumerate both expressions and let the optimization problem decide which was more advantageous. However, Mesa does not support multi-dimensional data, so it culls the expression instead, leaving only the preferred single-dimensional expression.

The next step in our methodology is error analysis, which follows expression enumeration and domain profiling. We now discuss two methods that can reduce the error in a LUT transform: *domain conditioning* and *partial domains*. Both techniques make use of domain information, so they are applied after domain profiling but before error analysis.

5.1.4 Domain Conditioning

Domain conditioning takes advantage of the fact that some elementary functions are cyclical, thus providing the opportunity to reduce the input domain of the associated LUT transform. Reducing the error domain decreases the error associated with a LUT transform. For example, the sine and cosine functions are completely represented by the interval from 0 to 2π radians. Input values outside of this domain can be mapped back into the interval by a modulo operation or repeated addition and subtraction. We call this technique domain conditioning, also referred to in the literature as *parameter folding* or *range reduction* [69]. We have experimented with a command line option to Mesa that generates domain conditioning code for cyclical functions.

Figure 5.6 shows LUT access code with domain conditioning code based on the iterative method. On our test system, the performance of a sine lookup with a 4MB table is 6.97ns with domain conditioning and 6.29ns without, constituting an 11% overhead. However, the LUT transformation becomes $\sim 3\times$ more accurate with domain conditioning, assuming that the original domain is $[-\pi, 3\pi]$ and the conditioned domain is $[0.0, 2\pi]$. The speedup for the domain conditioning is $\sim 5.4\times$ over the unoptimized code, as compared to $\sim 6.0\times$ without domain conditioning, so the tradeoff of performance for more accuracy is reasonable. Some care needs to be taken with domain conditioning, for at least two reasons. First, executing a conditional repeatedly is very expensive on current architectures. Second, performance and accuracy are degraded by iterative subtractions and additions.

```
// LUT Function
float X1_lut(float X1_param)
{
    // Domain conditioning
    while (X1_param < 0.0) X1_param += (2.0 * M_PI);
    while (X1_param > (2.0 * M_PI)) X1_param -= (2.0 * M_PI);

    // Table access
    int uIndex = (int) (X1_param * (1.0 / X1_granularity));
    return (X1_data[uIndex]);
}
```

Figure 5.6: Source listing for domain conditioning code.

5.1.5 Partial Domains

Partial domains are another method for reducing error. Domain profiling finds the boundaries of input variables, but does not capture distribution data for the input values over the domain. The distribution of input values is important because it determines the sampling of LUT data, and can therefore affect accuracy and performance. For example, some applications may sample the domain at regular intervals. For such an application the LUT initialization code could potentially decrease memory usage and increase accuracy by constructing LUT entries that exactly correspond to these intervals.

We have prototyped profiling code that captures the distribution of input data. Figure 5.7 shows the result for the SAXS discrete scattering code. The distribution is sampled heavily at the left side of the domain, near 0.0, and lightly sampled on the right side of the domain, near 20.0. This suggests that the corresponding LUT transform could reduce memory usage by storing only a partial domain. This requires the generation of conditional code to decide between a LUT access and the original function. An experimental version of partial domain code is shown in Figure 5.8.

We tested the code by creating an application that samples the sine function heavily in the range $[0, 1\pi]$ and lightly in the range $[1\pi, 2\pi]$. We then implement a LUT transformation for sine with the partial domain $[0, 1\pi]$ and the code shown in Figure 5.8. On our test system, the performance of a sine lookup with a 4MB table is 5.71ns with the partial domain code,

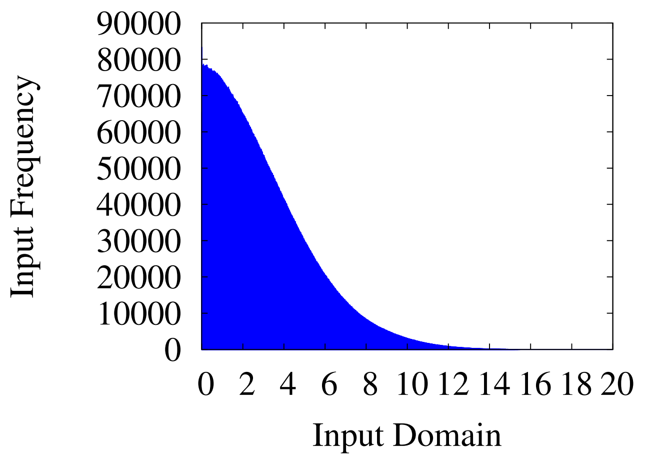


Figure 5.7: Domain distribution for SAXS discrete scattering code.

```
// LUT Function
float X1_lut(float X1_param)
{
    // Partial domain
    if (X1_param > X1_upper)
        return sin(X1_param);

    // Table access
    int uIndex = (int) (X1_param * (1.0 / X1_granularity));
    return (X1_data[uIndex]);
}
```

Figure 5.8: Source listing for partial domain code.

and 3.34ns without, constituting an 71% overhead. The accuracy varies only negligibly between the two versions, but the partial table uses half the memory. The speedup for the partial domain code is ∼5.0× over the unoptimized code, as compared to ∼ 8.5× without, so the optimization remains viable. Our code only checks the upper boundary because our partial domain includes the lower boundary, but some partial domains would require both boundaries to be checked in the conditional.

5.2 Error Analysis

Error analysis is the next algorithm in terms of the LUT optimization workflow. Error analysis follows domain profiling, which has provided the domain information needed to characterize the error for a LUT transform. LUT transforms represent continuous functions with discrete values, thereby introducing a fundamental source of error. With unlimited memory for LUT data, we could theoretically match processor precision. The IEEE 754 format is accurate to $\pm 1.19 \times 10^{-07}$ for single-precision and $\pm 2.22 \times 10^{-16}$ for double-precision. To provide the same precision as the processor for an optimization with domain $[0.0, 1.0]$ would therefore require $\sim 8.4 \times 10^{6}$ LUT entries (32 MB) for single-precision, and $\sim 4.5 \times 10^{15}$ LUT entries (16PB) for double-precision. The former exceeds cache memory on most systems, and the latter exceeds the memory capability of modern computers. We conclude that for most domains, a LUT transformation will provide significantly less precision than the processor. This motivates the need for a reliable method to predict and control LUT error.

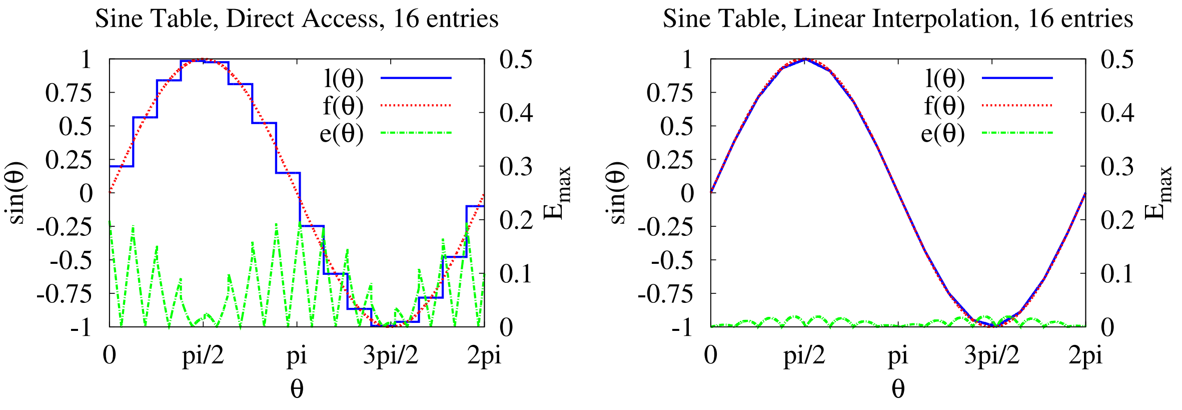
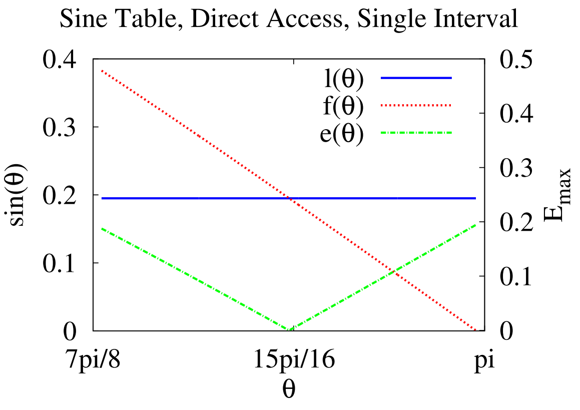


Figure 5.9: Lookup tables for sine function.

Fast and accurate error analysis is needed to support automation, and to help the programmer decide critical tradeoff between accuracy and performance for LUT transforms. This section presents the algorithms we have explored for LUT error analysis including analytic, exhaustive, stochastic, and boundary methods. As an example, Figure 5.9 illustrates the error introduced by LUT transformation for the sine function with a domain of $[0.0, 2\pi]$. The left graph shows direct access sampling, and the right graph shows linear interpolation. On both plots $f(\theta)$ is the original function, $l(\theta)$ is the approximation, and $e(\theta)$ is the error. The error is the absolute difference of the function and its approximation, computed as $e(\theta) = |f(\theta) - l(\theta)|$. The error can be interpreted graphically as the area between $f(\theta)$ and $l(\theta)$. As with the exponential function shown in Figure 2.1, linear interpolation considerably reduces the error magnitude.

For error analysis, we consider the following error statistics: E_{max}, the worst case error, and E_{avg}, the average error. These statistics can be computed over a single LUT interval or the entire domain. E_{max} is the largest possible value $\mathrm{e}(\theta)$, which bounds the error that can be introduced by any single table access. E_{avg} is the arithmetic mean of $\mathrm{e}(\theta)$ when the table is sampled uniformly. The maximum error provides a conservative measure for comparing LUT accuracy. The average error is an alternative measure for comparing accuracy, but it costs more to compute and is less reliable, since it depends on the distribution of input values. As a result, Mesa no longer uses the average error for LUT optimization.



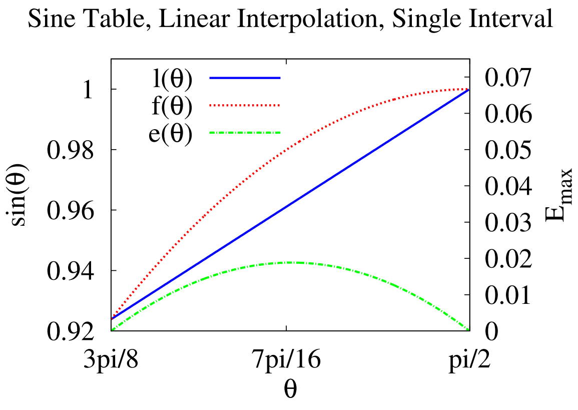


Figure 5.10: Error illustration in table interval.

Figure 5.9 shows the maximum error in the left graph at $0, \pi, 2\pi$, and in the right graph close to $\pi/2, 3\pi/2$, because the location of the maximum error differs for direct access and linear interpolation. Figure 5.10 expands the LUT interval that includes the maximum error to illustrate why this is the case. The left graph shows the interval $[7\pi/8, \pi]$ for direct access. In this case the error is related to the slope of the function, there is no error at the interval center, and E_{max} is at the interval boundaries. The right graph shows the interval $[3\pi/8, \pi/2]$ for linear interpolation. In this case the error is related to the curvature of the function, so there is no error at the interval boundaries, and E_{max} is at the interval center.

We investigated analytic and numerical approaches for computing error statistics. Analytic techniques apply differential or integral calculus to the functions being optimized. Numerical approaches apply sampling techniques to the functions being optimized to find the same information. Analytic techniques are generally more efficient and accurate, however they require the ability to do calculus on arbitrary functions, which may not always be possible. Numerical approaches, on the other hand, are relatively straightforward to implement and can handle arbitrary functions. The disadvantage of numerical sampling is the amount of time it requires. The following sections describe both approaches.

5.2.1 Analytic Error Method

To understand the source of error in a LUT transform, we provide a geometric interpretation. The left graph of Figure 5.10 illustrates the error for direct access sampling. Consider the function $f(x)$, which is approximated by $l(x)$ in an interval bounded by $[x, x+\epsilon]$,where ϵ represents the granularity of the LUT data. The left graph shows the specific case where $f(x)$=$sin(x)$ and ϵ is $pi/8$. If the function is monotonically increasing or decreasing in the interval, the worst case error for the LUT interval will depend on the slope and granularity. LUT data is uniformly distributed over the domain, so the granularity is the domain divided by the size. The approximation value $l(x)$ is determined by sampling $f(x)$ at the center of the interval. Assuming that $f(x)$ is linear throughout the interval, the maximum error will actually be half of the worst case, and it will lie at the interval boundaries. We show the equation for the maximum error calculation for direct access in Equation (5.1). MaxSlope is the maximum value of the function slope over the domain.

$$MaxError = (Granularity * MaxSlope)/2 = (Domain/Size) * (MaxSlope/2) \tag{5.1}$$

From the equation we see that the error is a function of the domain, size, and maximum slope. Note that the error for direct access is inversely proportional to size. This means that a 2× increase in size will decrease the error by 2×. The optimization problem requires an equation that solves for maximum error based on size. The domain is known for each expression, so error analysis only has to compute the maximum slope. The analytic method for computing the slope is to find the first derivative of the function and solve for its maximum absolute value over the domain. For the elementary functions, we know the derivatives in advance, and can therefore compute the maximum slope for arbitrary intervals.

The right graph of Figure 5.10 shows the error for linear interpolation. Again we have the function $f(x)$, which is approximated by $l(x)$ in an interval bounded by $[x, x+\epsilon]$,where ϵ represents the granularity of the LUT data. If the function $f(x)$ were linear through the interval, there would be no error, thus the error is caused by the curvature of the function. The curvature is determined by the delta in slope over the interval, which is the second derivative of the function. This leads to the definition of error shown in Equation (5.2) and cited many places in the literature [22]. MaxDelta is the maximum change in the slope of the function over the domain. Because the error term contains the square of the granularity, a 2× increase in size will decrease the error by 4×.

$$MaxError = (Granularity^2 * MaxDelta)/8.0 = (Domain/Size)^2 * (MaxDelta/8) \quad (5.2)$$

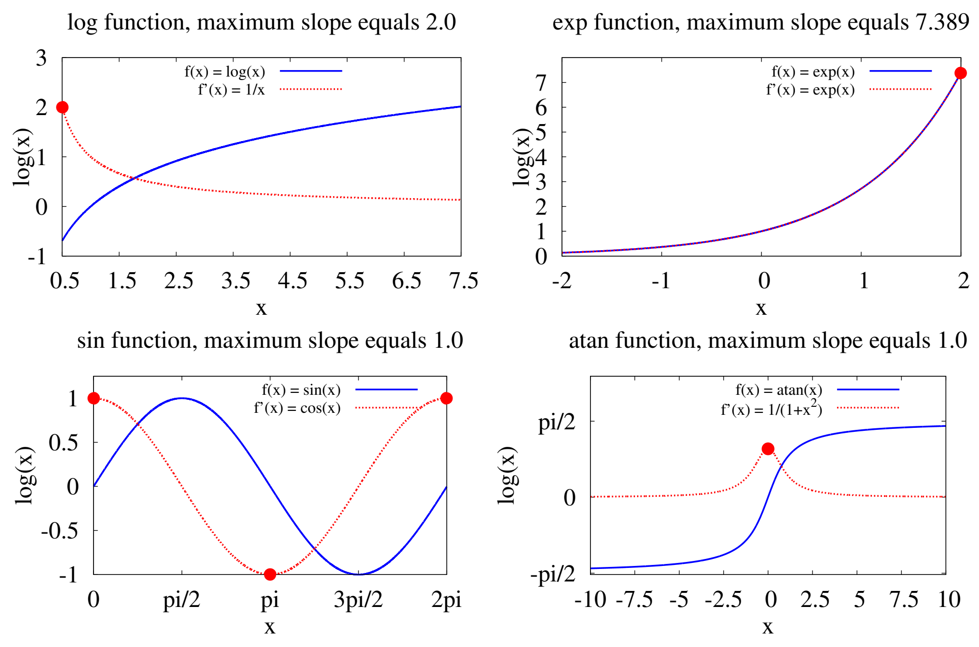
Earlier versions of Mesa computed E_{max} and E_{avg} by evaluating the original function and approximation value at each LUT entry. This required Mesa to construct a LUT of a specified size. Equations 5.1 and 5.2 compute the maximum error for any LUT size, based on the maximum slope or delta slope of the function. As a result, error analysis now consists of evaluating the original function to find these values.

Figure 5.11 shows the maximum slope calculation for several common elementary functions. The functions and their derivatives are plotted over the domain, and the maximum slope is highlighted with a small circle. Note that all functions are continuous in the selected domains, thus the slopes are never undefined. One advantage of analytic techniques is that they can flag discontinuous functions. The same method as shown in Figure 5.11 can be used to compute the maximum delta, except that the second derivative is used. Table 5.3 shows the results of the error computation for several elementary functions. For each function we evaluate a LUT transform of the domain and size shown, and we calculate the maximum slope and error using both sampling methods.

Analytical techniques derived from calculus can also compute the average error over an interval. Unfortunately these methods require the integral and in some cases the derivative of the function being analyzed, which is not trivial to compute for arbitrary expressions. Even for the maximum error, analytical techniques are useful only for the functions that we specifically handle in the code, i.e. individual elementary functions. For this reason we generally use numerical methods, which are slower but very robust.

5.2.2 Exhaustive Method

The most straightforward numerical method is an emphexhaustive traversal of the input domain at a granularity close to the precision of the LUT data. For single-precision an



Plots elementary functions and derivatives, maximum slope is shown by a circle.

Figure 5.11: Maximum slope for elementary functions.

Table 5.3: Maximum error for elementary functions.

Lists maximum slope and error for LUT transforms with specified domain and size.

			Direct Access		***Linear Interpolation***	
Function	***Domain***	***Size***	***Maximum Slope***	***Maximum Error***	***Maximum Delta***	***Maximum Error***
sqrt(x)	[1.0, 4.0]	64KB	**0.500**	**4.58E-05**	**0.250**	**1.05E-09**
sqrt(x)	[2.0, 5.0]	128KB	**0.354**	**1.62E-05**	**0.088**	**9.28E-11**
exp(x)	[0.0, 1.0]	4KB	**2.718**	**1.33E-03**	**2.718**	**3.24E-07**
exp(x)	[0.0, 2.0]	512KB	**7.389**	**5.65E-05**	**7.389**	**2.15E-10**
log(x)	[0.5, 1.0]	64KB	**2.000**	**3.05E-05**	**4.000**	**4.66E-10**
sin(x)	[0, 6.28]	32KB	**1.000**	**3.84E-04**	**1.000**	**7.35E-08**
tan(x)	[1.75, 3.14]	256KB	**31.474**	**3.34E-04**	**347.517**	**1.95E-08**

interval of 2^{-23} or 1.19×10^{-07} produces very accurate results. To compute the slope, the algorithm evaluates the function at the left and right boundary of the interval. The maximum delta is found in an almost identical manner except that three points are required, so the algorithm samples an additional point in the middle of the interval. The advantage of the technique is high accuracy, and the drawback is the cost of sampling at such fine granularity.

5.2.3 Stochastic Method

To address the performance limitations of exhaustive traversal, we experimented with *stochastic* sampling. This method samples randomly across the domain, accumulating samples in the same manner as numerical traversal. The sampling terminates when the maximum slope converges to within a specified criteria. The stochastic technique has the advantage of completing more quickly and with fewer samples than exhaustive traversal. The drawback of stochastic sampling is that it can miss the maximum slope by a wide margin, and no method exists to decide how many samples are required to get a reliable answer.

5.2.4 Boundary Method

Exhaustive and stochastic methods both require extensive domain sampling. The boundary method samples error only at the *boundary* of LUT intervals. For direct access, the method evaluates the original function at the left and right boundary of each LUT interval, which gives us the two values needed compute the slope. For linear interpolation, the method adds an evaluation at the interval center, which gives us the three values needed to compute the delta slope. Boundary methods have proven to be very accurate for tables of all sizes. The advantage of computing error at the boundaries is that the number of samples is proportional to the LUT size instead of the domain size. The boundary method does not require the actual LUT data, instead the technique works by evaluating the function at intervals equal to the LUT granularity. The boundary method does not support the computation of average error.

5.2.5 Comparison of Error Methods

Table 5.4 compares error analysis performance and accuracy for the LUT transforms in Table 5.3. The stochastic method is an order of magnitude faster than the exhaustive method, and the boundary method is generally even better. The performance of the exhaustive and stochastic methods is proportional to the domain size, and the performance of the boundary method is proportional to the table size. The analytic method is extremely fast, but handles only individual elementary functions. Table 5.5 compares accuracy against the analytic

Table 5.4: Comparison of the Performance of Error Methods.

(Intel Core 2 Duo, E8300, family 6, model 23, 2.83GHz, single core)

			Analytic Method	*Exhaustive Method*	*Stochastic Method*	*Boundary Method*
Expression	*Function*	*Domain*	*Analysis Time*			
X0	**sqrt(x)**	[1.0, 4.0]	~0.001s	19.334s	1.960s	0.814s
X1	**sqrt(x)**	[2.0, 5.0]	~0.001s	19.468s	1.953s	0.827s
X2	**exp(x)**	[0.0, 1.0]	~0.001s	7.331s	0.739s	0.937s
X3	**exp(x)**	[0.0, 2.0]	~0.001s	14.376s	1.427s	0.899s
X4	**log(x)**	[0.5, 1.0]	~0.001s	3.571s	0.381s	0.911s
X5	**sin(x)**	[0, 6.28]	~0.001s	41.733s	6.626s	0.886s
X6	**tan(x)**	[1.75, 3.14]	~0.001s	10.264s	1.057s	0.934s

Table 5.5: Comparison of the Accuracy of Error Methods.

			Analytic Method	*Exhaustive Method*	*Stochastic Method*	*Boundary Method*
Expression	*Function*	*Domain*	*Maximum Slope*			
X0	**sqrt(x)**	[1.0, 4.0]	0.500000	0.500000	0.500000	0.500000
X1	**sqrt(x)**	[2.0, 5.0]	0.363553	0.363553	0.363553	0.363553
X2	**exp(x)**	[0.0, 1.0]	2.718281	2.71828**2**	2.718**278**	2.718281
X3	**exp(x)**	[0.0, 2.0]	7.389056	7.389056	7.3890**46**	7.3890**49**
X4	**log(x)**	[0.5, 1.0]	2.000000	2.000000	**1.999995**	**1.999999**
X5	**sin(x)**	[0, 6.28]	1.000000	1.000000	1.000000	1.000000
X6	**tan(x)**	[1.75, 3.14]	31.474594	31.4745**74**	31.474**520**	31.474**564**

result, which we believe is the most accurate. Error digits that do not match the analytic result are shown in bold. The data shows reasonable accuracy for all methods.

5.3 Performance Model

The performance model estimates the benefit of applying LUT transformation to expressions. The model takes into account the performance of arithmetic operators, elementary function calls, and table access. This information comes from an benchmark implemented in Mesa that measures execution times for each of these items. The table access benchmark includes both the index calculation and cache access execution times. In addition, the benchmark measures the relative performance of direct access and linear interpolation. Mesa automatically runs the benchmark and writes the execution times to a file. The performance file is read before applying the performance model.

To apply the performance model, Mesa counts the arithmetic operators and elementary function calls in each expression. These counts are multiplied by the cost of each operation stored in the performance file, and the table access time is subtracted. For linear interpolation, the result is then divided by the relative performance compared to direct access. This produces an estimate of the potential benefit of replacing the expression with a LUT access. For example, optimizing an expression with a sine call on our test system should save 40ns for the sine call, minus 7.4ns for the table access, for a total of 32.6ns per execution. We multiply this number by the call frequency of the expression to get overall expression benefit. If the call frequency is 10^8 for the same expression, the benefit would be 32.6ns$\times 10^8$, or 3.26s. When the expression is optimized with linear interpolation, the benefit would be reduced by the relative factor, which is $\sim$1.8$\times$ on our test system, so 3.26s / 1.8 = 1.81s. Equation 5.3 shows the computation just described.

$$Benefit = ((Cost(Operator) * Count(Operator)) - Cost(Access)) * Frequency \tag{5.3}$$

The purpose of the performance model is not to predict the exact performance, but to establish the relative performance to allow comparison of different solutions. Even so, we are often within 5-10% when estimating performance speedup for individual elementary functions. For example, we compile and run a program on our test system that calls a single sine function 2.52×10^8 times, and we measure performance as 9.5s. Mesa optimizes the program, and estimates a savings of 32.6ns x 2.52 $\times10^8$ = 8.2s. We run the optimized code and measure performance as 0.9s. We then compare the estimated savings of 8.2s to the actual savings of 9.5s - 0.9s = 8.6s a $\sim$5% difference. The same experiment on the exponential function yields a $\sim$9% difference between the estimated benefit of 6.2s and the actual benefit of 6.8s.

When optimizing code with multiple elementary functions, the model tends to overestimate the benefit of LUT optimization, mainly for the following reasons. First, elementary function performance can vary based on input values. This introduces inaccuracy because our benchmark uses a fixed domain of input values for measurement. To measure sine and cosine, for example, we evaluate random inputs in the domain $[0.0, \pi]$. The resulting performance on our test system is around 40ns per call. By changing the domain to $[0.0, 0.02]$ the execution time becomes 17ns. Thus the performance of sine is dependent on the domain and distribution of input values. We have seen this same behavior with other elementary functions, independent of the compiler used. We theorize that math libraries take advantage of specific input conditions to accelerate computations. For example, we have noted that the *pow* function is considerably faster when the exponent is an integer. We have also seen a considerable slowdown in the math library implementation of *sin* and *cos* when the parameters are far outside the range 0 to 2π.

Second, the compiler and hardware optimize combinations of elementary functions aggressively. For example, the code shown in this section makes one sine and cosine and two exponential calls, thus we would expect an execution time of around 40ns + 45ns + 32ns + 32ns = 149ns. The actual time of the loop as measured on our test system is 108ns. As a result, our performance model predicts an 8.6s benefit, but the actual savings is 5.9s. We speculate that the increase in performance of combinations of elementary functions is a result of Instruction-level Parallelism (ILP). We have also seen other compiler optimizations that speed up combinations of elementary functions, such as replacing a sine and cosine call with a single sincos function. However, our case studies have shown that our estimates ere usually within $\sim$1-2$\times$ of the performance benefit, even for complex expressions.

5.4 Optimization Problem

We construct and solve the LUT optimization problem after the profiling and modeling stages are complete. These prior stages have provided the information needed to find the most effective set of LUT transforms. The essence of the problem is to allocate cache memory for LUT data in a way that maximizes performance and minimizes error. Cache memory is a constrained resource that limits the number and size of LUT transformations active at

Table 5.6: Input data for optimization problem.

X_i	D_i	M_i	B_i	*Expression Description*
X0	2.44	3.31	6.15 s	exp(dProduct)
X1	2.44	1.00	8.15 s	sin(dProduct)
X2	2.44	3.67	16.40 s	exp(dProduct) + sin(dProduct)
X3	2.44	3.31	6.15 s	exp(dProduct)
X4	2.44	0.95	9.40 s	cos(dProduct)
X5	2.44	2.38	17.65 s	exp(dProduct) + cos(dProduct)
X6	2.44	3.31	12.30 s	exp(dProduct)

any given time. The optimization problem is responsible for the global problem of selecting the set of expressions that will be optimized, and the local problem of finding the optimal allocation of cache memory for those expressions.

The dual objectives of the optimization problem are to maximize performance (benefit) and minimize error (cost). As with the knapsack problem we presented in Section 2.3, these objectives conflict with each other, so multiple solutions exist with different tradeoffs between performance and error. The goal is to find the set of solutions that have the most effective tradeoff and allow the programmer to choose between them. The programmer is also given error and performance estimates for each solution, thereby supporting an informed decision about which solution is the best. The optimization problem additionally constrains memory usage, with the goal of keeping LUT data in cache memory.

We discuss the optimization algorithm by continuing the example introduced at the beginning of this chapter, which has 7 expressions and 8 intersection constraints. Table 5.6 shows the input data to the optimization problem for the example, after the previous stages have completed. We use the following terminology: X_i is the expression, D_i is the domain size (from domain profiling), M_i is the maximum slope (from error analysis), and B_i is the benefit (from performance modeling). The independent variables are S_i, the LUT size, and E_i, the LUT errors. Solutions are subsets of expressions from the table.

The complexity of the optimization problem is exponential because it evaluates all subsets of expressions. The solution space for the problem is the power set of expressions, which has size $O(2^N)$ for N expressions. The actual number of subsets is smaller, because we disallow solutions with intersecting expressions. For our example we have 7 expressions that yield a power set of size $2^7 = 128$, but intersection constraints reduce that number.

Figure 5.12 shows the members of the power that remain after enforcing intersection constraints. The resulting set has 29 actual solutions out of 128 possible, a 77% reduction in complexity for this example. We number the solutions from C0 to C28. The example solutions include the empty set and all 7 single transformations, since individual transformations are never culled because of intersection constraints. In addition there are 13 out of 21 possible subsets of two transforms, 7 out of 35 subsets of three transforms, 1 out of 35 subsets of four transforms, and no solutions with 5 or more transforms. Intersection constraints are therefore responsible for significantly decreasing the complexity of the optimization problem. For example, a single intersection constraint reduces the solution space for a set of 9 expressions from 512 to 384.

5.4.1 Mathematical Definition of the Problem

Figure 5.13 presents the LUT optimization problem in mathematical terminology. The independent variables for each solution include the table sizes S_i and the approximation errors E_i for each transform. Selection variables are named X_i, same as the expressions they represent. Error is computed by $E_i = (D_i/S_i) * (M_i/2.0)$, as defined by Equation 5.5, where D_i is the domain size, M_i is the maximum slope, and S_i is the table size. The $TotalError$ and $TotalBenefit$ variables are summations of the error and benefit terms for all of the selected expressions. The objectives to maximize performance and minimize error have equal weight. The primary constraint is the LUT data for all expressions must fit into cache. We write this constraint as $\sum S_i = CS$ to ensure that the entire cache is used. Secondary constraints are added to enforce the intersection constraints. For example, $X0 + X2 \leq 1$ keeps the associated expressions from being optimized simultaneously.

To evaluate solvers on the NEOS framework [52] we translated the optimization problem into AMPL [4], an input language used by many optimization frameworks. Figure 5.14 shows

```
Power Set (culled):
{
      C0  = { }, // 0 transformations
      C1  = { X0 }, // 1 transformations
      C2  = { X1 },
      C3  = { X2 },
      C4  = { X3 },
      C5  = { X4 },
      C6  = { X5 },
      C7  = { X6 },
      C8  = { X0, X1 }, // 2 transformations
      C9  = { X0, X3 },
      C10 = { X0, X4 },
      C11 = { X0, X5 },
      C12 = { X1, X3 },
      C13 = { X1, X4 },
      C14 = { X1, X5 },
      C15 = { X1, X6 },
      C16 = { X2, X3 },
      C17 = { X2, X4 },
      C18 = { X2, X5 },
      C19 = { X3, X4 },
      C20 = { X4, X6 },
      C21 = { X0, X1, X3 }, // 3 transformations
      C22 = { X0, X1, X4 },
      C23 = { X0, X1, X5 },
      C24 = { X0, X3, X4 },
      C25 = { X1, X3, X4 },
      C26 = { X1, X4, X6 },
      C27 = { X2, X3, X4 },
      C28 = { X0, X1, X3, X4 } // 4 transformations
};
```

Figure 5.12: Culled power set for optimization problem.

the AMPL code that we developed to use existing multi-objective optimization software. Our problem is mixed-integer, because we combine integer and real parameters, and non-linear, because of the error equations and performance model. Optimization frameworks exist that handle both of these attributes. However, our problem is unusual because the solver must simultaneously determine which expressions are included (selection) and the allocation for each optimization (size). As a result, we have not yet been able to solve the LUT optimization problem with existing solvers including SYMPHONY [73], Couenne [15], Bonmin [9], and MINLP [49]. The remainder of this section discusses our methodology for solving the optimization problem without using an existing solver.

5.4.2 The LUT Optimization Algorithm

Our optimization algorithm divides into three steps. First, we enumerate the set of potential solutions. Second, for each enumerated solution we compute the cache allocation that min-

INPUTS

D_i: real - input domain for expression
M_i: real - maximum slope for expression
B_i: real - potential benefit for expression
CS: integer - cache size

UNKNOWNS

X_i: boolean - expression selector, 0 when S_i is 0, and 1 otherwise
S_i: integer - table size, computed by the solver as $0 < S_i < CS$
E_i: real - computed as $(D_i/S_i) * (M_i/2.0)$

OBJECTIVES

maximize TotalBenefit $= \sum(X_i * B_i)$ - search for solutions that maximize benefit
minimize TotalError $= \sum(X_i * E_i)$ - search for solutions that minimize error

CONSTRAINTS
$\sum S_i = CS$ - must fit into cache and use entire cache

Intersection constraints
$X0 + X2 \leq 1$, $X1 + X2 \leq 1$, $X3 + X5 \leq 1$, $X4 + X5 \leq 1$,
$X0 + X6 \leq 1$, $X2 + X6 \leq 1$, $X3 + X6 \leq 1$, $X5 + X6 \leq 1$

Figure 5.13: Mathematical definition of optimization problem.

imizes error. Third, we use the benefit and error associated with each solution to identify Pareto optimal solutions. During enumeration we cull solutions that violate intersection constraints, thereby avoiding enumeration of the entire power set.

Enumeration of Solution Space To efficiently enumerate the solution space, we iterate an integer with N bits over the interval [0,2^N-1] and use its value as a bitmask. Each value of the integer is a solution whose transforms are selected based on the nonzero bits. For example, the mask for C27 from Figure 5.12 has a binary value of 00011100b to select {X2,X3,X4}. We use an unsigned 64-bit integer to support up to 64 expressions. The intersection constraints are enforced by a mask operation during enumeration. For example, the constraint $X0 + X1 \leq 1$ causes us to discard solutions in which both Bit 0 and 1 are set.

```
# AMPL data
param:  EXPRESSIONS:   Benefit   Domain     Slope     :=
        "X0"                23    2.000     2.718
        "X1"                39    2.000     1.000
        "X2"                62    2.000     3.259
        "X3"                23    1.500     7.389
        "X4"                41    1.500     1.000
        "X5"                64    1.500     8.298
        "X6"                46    3.000     7.389
param CS := 6291456; # 6 MB

# AMPL code
set EXPRESSIONS ordered;
param Benefit  {EXPRESSIONS} >= 0;
param Domain   {EXPRESSIONS} >= 0;
param Slope    {EXPRESSIONS} >= 0;

# Unknown variables
var Size {i in EXPRESSIONS} integer >= 0, <= CS, := 1;
var Select {i in EXPRESSIONS} integer >= 0, <= 1;

# Objective functions
maximize TotalError:   sum {i in EXPRESSIONS} (Slope[i] * Domain[i]) / (Size[i] * 2.0);
maximize TotalBenefit: sum {i in EXPRESSIONS} (Select[i] * Benefit[i]);

# Constraints
subject to ComputeSelect {i in OPTIMIZATIONS}: Select[i] <= Size[i];
subject to CacheSize: CS <= sum {i in OPTIMIZATIONS} Size[i] <= CS;

# Intersection constraints
X0 + X2 <= 1;
X1 + X2 <= 1;
X3 + X5 <= 1;
X4 + X5 <= 1;
X0 + X6 <= 1;
X2 + X6 <= 1;
X3 + X6 <= 1;
X5 + X6 <= 1;

# AMPL commands
solve;
display Size;
display TotalBenefit;
display TotalError;
display CacheSize.body;
```

Figure 5.14: AMPL code for optimization problem.

Determining Cache Allocation We can compute the solution benefit by summing the benefits of the expressions it contains. To find solution error, we must compute the cache allocation. We derived a closed-form analytic formula to compute the LUT sizes and errors for each subset of expressions. The formula shown in Equation (5.4) computes the precise size for each transformation that minimizes solution error, and Equation (5.5) computes the precise error. Our algorithm only has to compute one equation or the other, since size and error are related through Equation (5.1), which is less computationally expensive.

$$S_i = CS/\left(\sum_{j=1..n} \sqrt{(M_j D_j / M_i D_i)}\right) \tag{5.4}$$

$$E_i = M_i D_i \left(\sum_{j=1..n} \sqrt{(M_j D_j / M_i D_i)}\right) /CS \tag{5.5}$$

Figure 5.15 shows the derivation of Equation (5.4) for an arbitrary number of expressions. The function we minimize is the sum of the errors based on the error formula in Equation (5.1), and subject to the constraint of sharing the cache (1). We create the Lagrangian function that combines the minimization with a lambda expression that represents the constraint (2). Next we take the derivative of the Lagrangian function w.r.t. to the size and set it to zero (3) to get the Stationarity constraint. This lets us solve for the size squared (4) and the size (5) in terms of lambda.

Next we return to the Lagrangian function and substitute all instances of the size with the lambda equivalent (6). We then simplify by distributing lambda and dividing $(M_i D_i/2)^1$ by $(M_i D_i/2)^{1/2}$ to get $(M_i D_i/2)^{1/2}$ and λ^1 by $\lambda^{1/2}$ to get $\lambda^{1/2}$ (7), and combining similar terms (8). We again take the derivative of the Lagrangian function, this time w.r.t. lambda and set it to zero (9). We can now solve for lambda by adding CS to both sides (10), dividing by the summation term (11) and squaring the result to compute the inverse of lambda (12). Note that this value is defined in terms of the known values CS, M_j, and D_j. Returning to the equation in (4), we substitute our new value for lambda (13). Next we take the square root (14) and divide by $sqrt(M_i D_i/2)$ (15) to match the formula in Equation (5.4). Equation (5.5) is then derived from Equation (5.4).

$$\text{Minimize} \sum_{i=1..n} E_i = \sum_{i=1..n} \frac{M_i D_i}{2S_i}, \text{ subject to} \sum_{i=1..n} S_i = CS \tag{1}$$

$$L = \sum_{i=1..n} \frac{M_i D_i}{2S_i} + \lambda\Big[\sum_{i=1..n} S_i - CS\Big] \quad \text{// Lagrangian Function} \tag{2}$$

$$L' = \frac{\delta L}{\delta S_i} = -\frac{M_i D_i}{2S_i^2} + \lambda = 0 \quad \text{// Stationarity Condition} \tag{3}$$

$$\frac{1}{S_i^2} = \frac{2\lambda}{M_i D_i} \text{ or } S_i^2 = \frac{M_i D_i}{2\lambda} \tag{4}$$

$$\frac{1}{S_i} = \left(\frac{2\lambda}{M_i D_i}\right)^{1/2} \text{ or } S_i = \left(\frac{M_i D_i}{2\lambda}\right)^{1/2} \tag{5}$$

$$L = \sum_{j=1..n} \frac{M_j D_j}{2}\left(\frac{2\lambda}{M_j D_j}\right)^{1/2} + \lambda\Big[\sum_{j=1..n}\left(\frac{M_j D_j}{2\lambda}\right)^{1/2} - CS\Big] \tag{6}$$

$$L = \sum_{j=1..n}\left(\frac{M_j D_j}{2}\right)^{1/2}\lambda^{1/2} + \sum_{j=1..n}\left(\frac{M_j D_j}{2}\right)^{1/2}\lambda^{1/2} - \lambda CS \tag{7}$$

$$L = \sum_{j=1..n} 2\left(\frac{M_j D_j}{2}\right)^{1/2}\lambda^{1/2} - \lambda CS \tag{8}$$

$$L' = \frac{\delta L}{\delta \lambda} = \sum_{j=1..n}\left(\frac{M_j D_j}{2}\right)^{1/2}\lambda^{-1/2} - CS = 0 \tag{9}$$

$$CS = \sum_{j=1..n}\left(\frac{M_j D_j}{2}\right)^{1/2}\lambda^{-1/2} \tag{10}$$

$$\lambda^{-1/2} = CS/\sum_{j=1..n}\left(\frac{M_j D_j}{2}\right)^{1/2} \tag{11}$$

$$\lambda^{-1} = CS^2/\Big[\sum_{j=1..n}\left(\frac{M_j D_j}{2}\right)^{1/2}\Big]^2 \tag{12}$$

$$S_i^2 = \frac{M_i D_i}{2\lambda} = CS^2\frac{M_i D_i}{2}/\Big[\sum_{j=1..n}\left(\frac{M_j D_j}{2}\right)^{1/2}\Big]^2 \tag{13}$$

$$S_i = CS\left(\frac{M_i D_i}{2}\right)^{1/2}/\Big[\sum_{j=1..n}\left(\frac{M_j D_j}{2}\right)^{1/2}\Big] \tag{14}$$

$$S_i = CS/\Big[\sum_{j=1..n}\left(\frac{M_j D_j}{M_i D_i}\right)^{1/2}\Big] = CS/\Big(\sum_{j=1..n}\sqrt{\frac{M_j D_j}{M_i D_i}}\Big) \tag{15}$$

Figure 5.15: Derivation of the analytic size equation

We can use Equations (5.4) and (5.5) to find the optimal size and error for each LUT transform in a solution. The results for the example code are shown in Table 5.7. Sizes and errors are listed for each LUT transform in the solution, along with the total benefit B_i, in nanoseconds, and total error E_i for each solution. The total benefit is a summation of the benefits as shown in Equation (5.6). The total error is a summation of the errors as shown in Equation (5.7). The solutions in Table 5.7 are sorted by benefit.

$$TotalBenefit = \sum_{i=1..n} X_i B_i \tag{5.6}$$

$$TotalError = \sum_{i=1..n} X_i E_i \tag{5.7}$$

Table 5.7: Errors and Sizes for algorithms example program.

C_i	X_i	B_i	E_i	S_0	S_1	S_2	S_3	E_0	E_1	E_2	E_3
C0	**Empty Set**	**0.00E+00**	**0.00E+00**								
	X0	6.15E+09	9.63E+02	4194304				9.63E+02			
	X3	6.15E+09	9.63E+02	4194304				9.63E+02			
	X1	8.15E+09	2.91E+02	4194304				2.91E+02			
C1	**X4**	**9.40E+09**	**2.76E+02**	**4194304**				**2.76E+02**			
	X6	1.23E+10	1.93E+03	4194304				1.93E+03			
	X0,X3	1.23E+10	3.85E+03	2097152	2097152			1.93E+03	1.93E+03		
	X0,X1	1.43E+10	2.31E+03	1488006	2706297			8.21E+02	1.49E+03		
	X1,X3	1.43E+10	2.31E+03	1488006	2706297			8.21E+02	1.49E+03		
	X0,X4	1.56E+10	2.27E+03	1462350	2731953			7.92E+02	1.48E+03		
	X3,X4	1.56E+10	2.27E+03	1462350	2731953			7.92E+02	1.48E+03		
	X2	1.64E+10	1.07E+03	4194304				1.07E+03			
	X1,X4	1.76E+10	1.13E+03	2069022	2125281			5.59E+02	5.75E+02		
C2	**X5**	**1.77E+10**	**6.92E+02**	**4194304**				**6.92E+02**			
	X1,X6	2.05E+10	3.81E+03	1488006	2706297			8.21E+02	2.99E+03		
	X0,X1,X3	2.05E+10	6.26E+03	904436	1644933	1644933		1.35E+03	2.46E+03	2.46E+03	
	X4,X6	2.17E+10	3.75E+03	1462350	2731953			7.92E+02	2.96E+03		
	X0,X3,X4	2.17E+10	6.19E+03	885549	1654377	1654377		1.31E+03	2.44E+03	2.44E+03	
	X2,X3	2.26E+10	4.06E+03	2152133	2042170			2.09E+03	1.98E+03		
	X0,X1,X4	2.37E+10	4.19E+03	1076736	1106014	2011552		1.08E+03	1.10E+03	2.01E+03	
	X1,X3,X4	2.37E+10	4.19E+03	1076736	1106014	2011552		1.08E+03	1.10E+03	2.01E+03	
	X0,X5	2.38E+10	3.29E+03	1924322	2269981			1.51E+03	1.78E+03		
C3	**X1,X5**	**2.58E+10**	**1.88E+03**	**2544168**	**1650135**			**1.14E+03**	**7.40E+02**		
	X2,X4	2.58E+10	2.43E+03	1412800	2781503			8.19E+02	1.61E+03		
	X1,X4,X6	2.99E+10	6.20E+03	1076736	1106014	2011552		1.08E+03	1.10E+03	4.02E+03	
	X0,X1,X3,X4	2.99E+10	9.17E+03	727725	747513	1359532	1359532	1.59E+03	1.63E+03	2.97E+03	2.97E+03
	X0,X1,X5	3.20E+10	5.54E+03	1483018	961878	1749407		1.96E+03	1.27E+03	2.31E+03	
	X2,X3,X4	3.20E+10	6.46E+03	867132	1707200	1619971		1.34E+03	2.63E+03	2.49E+03	
C4	**X2,X5**	**3.41E+10**	**3.48E+03**	**1869832**	**2324471**			**1.55E+03**	**1.93E+03**		

Determination of Pareto Optimal The final step in the optimization problem is to find the Pareto optimal solutions by using the Graham Scan [33] algorithm for convex hull detection. The input data has estimated error on the x-axis and estimated benefit on the y-axis. The algorithm sorts the solutions by the cosine of angle between the x-axis and the line between the origin and the solution. The solutions are traversed in order, and a cross-product is computed for each contiguous set of three points. The cross-product differentiates between a left and right turn. A right turn maintains the points being evaluated, and a left turn culls the center point. The result is to cull all of the non-optimal solutions that lie within the convex hull, leaving only solutions that lie on the Pareto optimal line. The algorithm requires the removal of solutions that lie below the diagonal line between the origin and the solution with the most benefit, to avoid adding solutions outside the convex hull.

Figure 5.16 shows the Mesa output for the optimization run on the example code. Mesa has found 5 Pareto optimal solutions and presented them to the programmer, with the corresponding error and benefit estimates. The C4 solution that has the maximum benefit is a combination of the X2 ($exp(dProduct) * sin(dProduct)$) and X5 ($exp(dProduct) * cos(dProduct)$) expressions, each of which combines two elementary functions. Expression X6 ($exp(dProduct)$), which is the coalesced exponential call, does not appear in the solution so expression coalescing has not helped the example code. The programmer selects the maximum benefit solution C4, and Mesa realizes the corresponding expressions. Note

```
128 solutions (possible)
29 solutions (actual)
5 solutions (pareto optimal)
Solution  Error      Benefit    Optimizations
C0        0.000e+00  0.000e+00
C1        2.759e+02  9.400e+09  X4
C2        6.921e+02  1.765e+10  X5
C3        1.881e+03  2.580e+10  X5,X1
C4        3.483e+03  3.405e+10  X5,X2
Select solution: 4
X5 Di = 2.44 Mi = 2.38 Ei = 1.553e+03 Bi = 1.765e+10 Si = 1869833 (1826KB)
X2 Di = 2.44 Mi = 3.67 Ei = 1.930e+03 Bi = 1.640e+10 Si = 2324471 (2270KB)
Mesa 2.0: Generating optimized code
Realizing X5 in statement S44
Realizing X2 in statement S43
Mesa 2.0: Optimization completed.
```

Figure 5.16: Optimization run for algorithms example program.

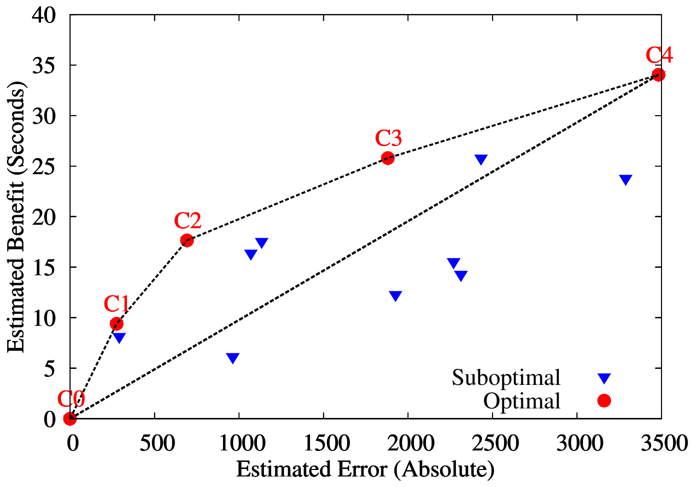


Figure 5.17: Pareto chart for algorithms example program.

that X2 receives a larger cache allocation because of its higher slope. The result is that the LUT optimization has identified the most effective LUT transforms based on the error and performance model, and allocated the ideal amount of cache for them.

Figure 5.17 plots all 29 solutions in the example, with error on the x-axis and benefit on the y-axis. The Pareto optimal solutions are shown as circles, and suboptimal solutions are shown as triangles. Only 14 points are visible because many of solutions have identical error and benefit. The 5 solutions on the curved line are Pareto optimal. We observe that the Pareto optimal solutions include the most *effective* transformations, i.e. those estimated to have the most performance for the least error. Table 5.7 also shows the Pareto optimal solutions in bold type. Only Pareto optimal solutions have identifiers, for consistency with the Mesa output.

The Pareto optimal solutions include C0, which has zero benefit and error. This corresponds to to the original application code. X5 is the highest benefit LUT transform, so it appears repeatedly in the Pareto optimal solutions, both by itself and in combination with other solutions. C4 is the solution with the maximum benefit, which corresponds to accepting expressions X2 and X5. We now compare the actual performance and accuracy of the example code, in Figure 5.18. The optimized code is 8.1× faster, and the relative error is 1.13×10^{-7}, based on the accumulation of return values from the ScatterSample function.

```
>>> time ./Original
Accumulated value = 1329836312500.0
real     0m27.299s
user     0m27.292s
sys      0m0.001s

>>> time ./Optimized
Accumulated value = 1329836314000.0
real     0m3.364s
user     0m3.357s
sys      0m0.006s
```

Figure 5.18: Optimization results for algorithms example program.

Local optimization and the Pareto algorithm take ∼3.2us each per LUT transform on our test system, and expression enumeration is ∼0.4us. For 24 candidate expressions without intersection constraints the solution space has 2^{24} = 16 million solutions. Mesa can run problems of this size in ∼100s. The Pareto algorithm selects the most effective solutions in terms of the estimated performance and error, so its accuracy is entirely dependent on the validity of these models.

5.4.3 More Examples of Optimization Results

Results from the Pareto algorithm vary based on the homogeneity of the input data. To further explore optimization results we present two synthetic data sets. The first data set introduces a set of LUT transforms that vary in effectiveness more widely than in our example code. Table 5.8 shows the inputs for the disparate data set, and Table 5.9 lists the Pareto optimal solutions. There are no intersection constraints, so the number of solutions is exactly 2^7=128. Just two LUT transforms provide the majority of the benefit for this data set with minimal error. Figure 5.19 graphs the Pareto optimal solutions. C1 has only the most effective transformation X4, C2 adds the next most effective X3, and so on. The Pareto curve is steep on the left side, and flat on the right side, so the programmer may want to select solutions C2, C3, or C4 to capture most of the performance with less error.

The second data set introduces a set of LUT transforms that have an almost identical benefit and error. Table 5.10 shows the inputs for the similar data set, and Table 5.11 lists the Pareto optimal solutions. As with the disparate data set, the problem has no intersection

Table 5.8: Input data for disparate data set.

X_i	B_i	D_i	M_i
X0	23 s	2.0	71.718
X1	39 s	2.0	87.000
X2	62 s	2.0	19.259
X3	175 s	1.5	2.389
X4	141 s	1.5	1.000
X5	4 s	1.5	23.928
X6	47 s	3.0	15.389

Table 5.9: Optimization solution for disparate data set.

C_i	X_i	B_i	E_i
C0	∅	0 s	0.000E+00
C1	X4	141 s	1.192E-07
C2	X4,X3	316 s	7.725E-07
C3	X4,X3,X2	378 s	6.909E-06
C4	X4,X3,X2,X6	425 s	2.065E-05
C5	X4,X3,X2,X6,X1	464 s	6.827E-05
C6	X4,X3,X2,X6,X1,X0	487 s	1.355E-04

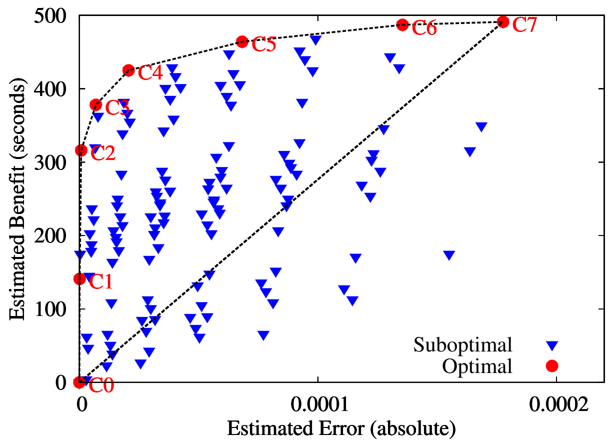


Figure 5.19: Pareto chart for disparate data set.

constraints. Figure 5.20 graphs the Pareto optimal solutions computed for the similar data set. Note that all LUT transforms are essentially equal in benefit and error, so each optimal point is very close to another set of points. This means that any set of LUT transforms can be selected at each optimal point, with only minor changes to the overall performance and accuracy. The similarity creates a Pareto optimal curve that is much flatter. The decision for the programmer in this case is a straightforward tradeoff between performance and error.

Table 5.10: Input data for similar data set.

X_i	B_i	D_i	M_i
X0	20 s	2.8	3.100
X1	21 s	2.7	3.300
X2	22 s	2.6	3.200
X3	23 s	2.5	3.400
X4	24 s	2.4	3.500
X5	25 s	2.3	3.600
X6	26 s	2.2	3.000

Table 5.11: Optimization solution for similar data set.

C_i	X_i	B_i	E_i
C0	∅	0 s	0.000E+00
C1	X6	26 s	5.245E-07
C2	X6,X5	51 s	2.358E-06
C3	X6,X5,X4	75 s	5.534E-06
C4	X6,X5,X4,X3	98 s	1.008E-05
C5	X6,X5,X4,X3,X2	120 s	1.590E-05
C6	X6,X5,X4,X3,X2,X1	141 s	2.332E-05

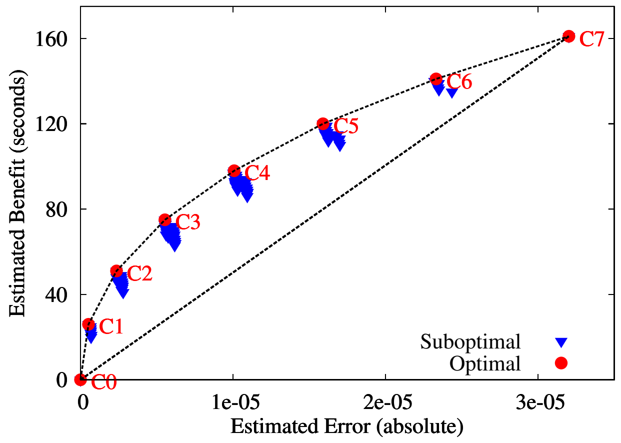


Figure 5.20: Pareto chart for similar data set.

5.4.4 Rank Culling Algorithm

Our algorithm optionally prunes solutions to avoid exponential growth in complexity for large numbers of transformations. We call the algorithm we have developed for this *rank culling*. Rank culling is integrated with power set generation to avoid storing the entire exponential space of solutions. To efficiently support rank culling, we use a different algorithm for solution enumeration. The goal of rank culling is to reduce the complexity of the optimization problem without changing the outcome. On current systems we start rank culling when the number of expressions is greater than a tuning parameter, currently set to 24 expressions.

To show the need for culling we measure the performance of the Mesa optimization algorithm. Table 5.12 shows the overall performance, and the timing of each step in the optimization problem: solution enumeration, solution optimization, and selection of Pareto optimal points. The table lists the number of expressions, possible solutions, and optimal solutions. We have measured performance for 8 to 24 expressions. The performance follows the expected 2^N growth, where N is the number of optimizations. We can derive the performance from the table. Generating subsets takes ∼0.4 us per transformation, solving for the optimal size and error takes ∼3.2us per transformation, and selection of the optimal solutions takes ∼3.2us. By extrapolation we can predict that solving the optimization problem for 32 expressions would take around 8 hours. This motivates the need to cull the solution space for this size of problem.

Table 5.12: Optimization problem performance without culling.

Number of Expressions	*Number of Solutions*	*Optimal Solutions*	*Performance*			
			Solution Enumeration	*Cache Allocation*	*Pareto Selection*	*Overall Performance*
8	256	11				0.002 s
10	1 K	13				0.003 s
12	4 K	17				0.007 s
14	16 K	19				0.031 s
16	64 K	21				0.142 s
18	256 K	25				0.843 s
20	1 M	28	0.2 s	2.4 s	1.9 s	4.5 s
22	4 M	33	0.6 s	11.4 s	9.8 s	21.8 s
24	16 M	39	2.6 s	53.8 s	53.0 s	109.4 s

The goal of the culling algorithm is to avoid consideration of solutions that are not Pareto optimal. We do this by avoiding enumeration of solutions based on transforms that we predict to be less effective. Prediction requires a metric to rank LUT transform effectiveness. Such a metric must increase when benefit increases, and decrease when accuracy decreases. We have evaluated several metrics, and found that the most accurate one is the benefit divided by the square root of the error. The reasoning behind this metric is that performance benefit is absolute, but error can be reduced by an increased cache allocation.

We compute the metric before solving the optimization problem, when the precise error is still unknown. For this reason the metric uses M_iD_i as a proxy for error as shown in Equation 5.4.4. This is equivalent to normalizing the error by setting the size for all transformations to 1.0. We represent the metric by the symbol V_i.

$$V_i = B_i/\sqrt{D_i * M_i} \tag{5.8}$$

We call the algorithm that is based on the above metric *rank culling*, because it depends an a metric-based ranking. The rank culling algorithm is designed to prune solutions that lie far from the Pareto optimal line. The assumption we make is that optimal solutions primarily consist of sets of the most effective LUT transforms. This assumption works only if the metric accurately predicts the effectiveness of a LUT transform. The justification for the metric as currently defined is empirical. To use the metric we sort LUT transforms and prune the least effective ones. The current algorithm uses several parameters to decide on how many solutions to prune, based on the number of expressions. We do this to favor solutions that lie along the Pareto optimal line.

Figure 5.21 presents a larger example with 16 expressions and $2^{16} = 65536$ solutions. The culled points are shown as triangles and the unculled points are shown as circles. The optimal points are shown as squares that lie along the Pareto optimal line. Note that some culled solutions lie near the Pareto optimal curve, thus culling presents the risk that some optimal solutions may be culled. We have evaluated sets of random transformations in the size range from 16 to 32 transformations, and found that no Pareto optimal solutions are

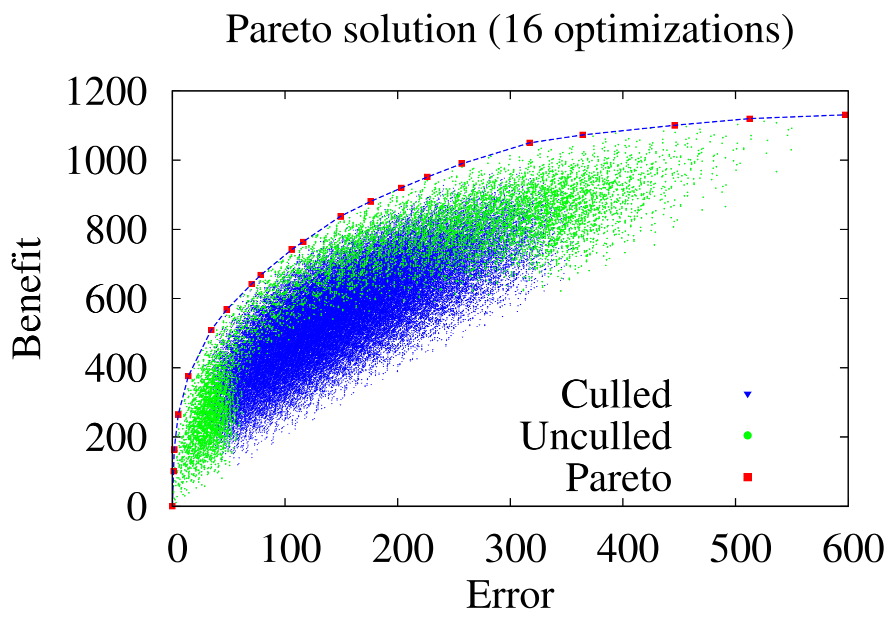


Figure 5.21: Illustration of culling algorithm.

culled. When an optimal solution is culled, our experiments shows that an almost optimal solution replaces the culled solution, so the outcome of the optimization is almost identical.

The benefit of rank culling is a significant increase in performance as shown in Figure 5.22. The table shows the performance with culling, for 16 to 48 transformations. The set of solutions produced during the experiment was identical with and without culling, however as discussed above this may not always be the case. Note that the increase in time and complexity with culling grows approximately in a quadratic fashion, as opposed to the exponential growth of the original problem. Without culling we can handle only 24 expressions in a small amount of time. With culling we can solve optimization problems with up to 40 expressions in seconds instead of hours as shown in Figure 5.22.

5.5 Evaluation of Performance and Error Models

Our methodology relies on error analysis and performance modeling to make decisions about the effectiveness of individual LUT transforms. In this section we evaluate our models by comparing their estimates to the measured performance and error of the optimized application. Results from the optimized application represent the ground truth against which

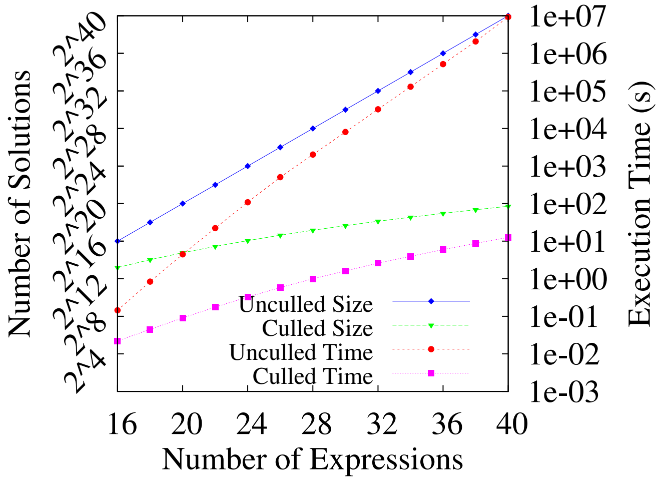


Figure 5.22: Optimization problem performance with culling.

we evaluate our estimates. Performance benefit is measured as the decrease in execution time from the original to the optimized code. Application accuracy is based on the program output that the programmer decides needs to be compared.

5.5.1 Model versus Actual Performance

We evaluate the performance model on the example code by disabling the Pareto optimal selection to gather error and performance estimates for all solutions. We use a script to invoke Mesa to realize all solutions, including optimal and suboptimal, then compile and run each solution to measure application time for comparison with the performance model. Figure 5.23 graphs the estimated benefit against the actual benefit. The solutions are listed in order of increasing performance as estimated by the model. The performance model slightly overestimates the benefit, but the estimated and actual numbers have the same slope, showing that the prediction correlates with real performance gain.

5.5.2 Model versus Actual Error

We evaluate the error model on the example code at the same time as performance. As performance is measured for each solution, we also compute the relative error. Figure 5.24 graphs the error model estimate against the actual application error. The solutions are listed in order of increasing error, as estimated by the model. The model estimate is absolute

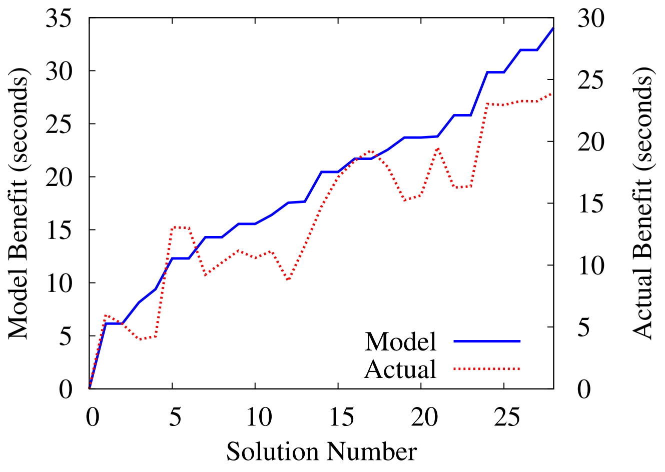


Figure 5.23: Comparison of model benefit with application benefit.

error, and the application measures relative error, so no direct comparison of the magnitude is possible. There appears to be some correlation, but many solutions have much lower error than estimated. We theorize that this is related to the accumulation of values in the example code, which tends to reduce error through cancellation. We observe this behavior in both of the SAXS applications. The reason for the cancellation is that LUT approximations introduce negative and positive error in close to equal amounts. We provide an evaluation of the error model in which error values are not accumulated in Section 7.2.2.

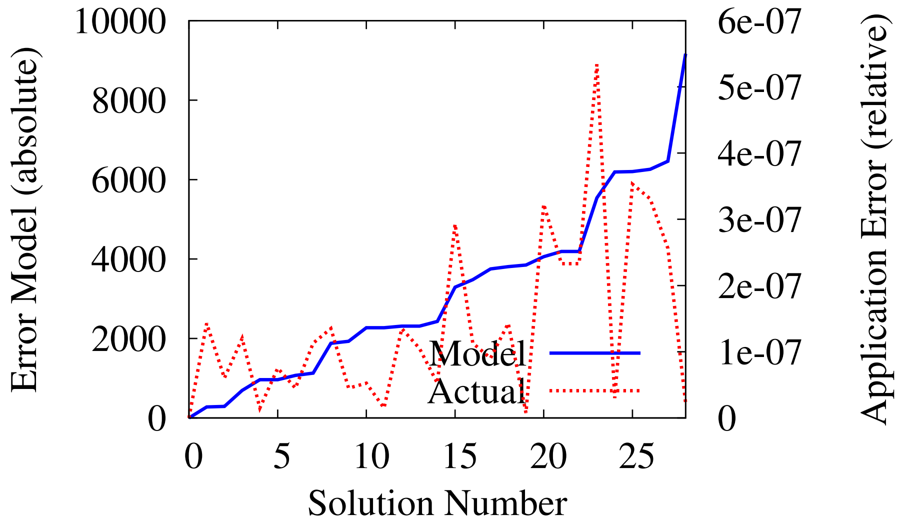


Figure 5.24: Comparison of error model with application accuracy.

Chapter 6
Mesa Tool

As previously stated, the manual development of LUT transformation code can lower productivity, obfuscate code, and limit control over performance and accuracy. We have developed the Mesa tool to address these concerns. Automation helps the programmer cope with the complexity of expression enumeration, error analysis, and performance modeling. These kinds of analysis are not practical with pencil and paper, except for trivial examples. Solving optimization problems by hand is also impractical for more than a handful of expressions, because the number of solutions grows exponentially with the number of expressions. Mesa can process up to 40 expressions in a couple of minutes, thus it can apply LUT transformation at a much larger scale than was previously possible. The current version of Mesa handles expressions with the arithmetic operators: +, -, *, and /, and double-precision elementary functions: sin, asin, sinh, cos, acos, cosh, tan, atan, tanh, exp, log, and sqrt. Mesa can also handle the associated single-precision functions when they exist, for example sinf, cosf, and tanf. Adding new functions requires fewer than 20 lines of new code.

6.1 Mesa Implementation

Mesa is a standalone tool written in C++ that incorporates the ROSE compiler infrastructure [61, 42] to support source-to-source transformations on C and C++ application code. The workflow for the tool shown in Figure 6.1. After the programmer inserts a #pragma LUTOPTIMIZE above the function(s) to be profiled, the steps shown below are followed:

Step 1: Run Mesa with a command-line option requesting program instrumentation.
Step 2: Mesa generates instrumented code that is compiled into a profiling executable.
Step 3: Compile and run the instrumented code, which writes the domain profiles.
Step 4: Run Mesa with a command-line option requesting program optimization.

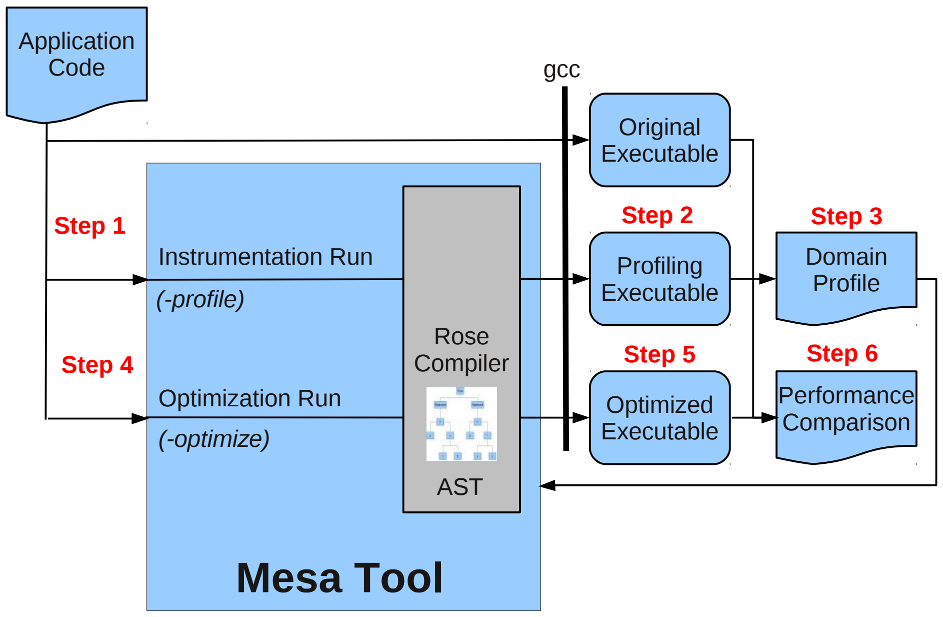


Figure 6.1: Workflow Diagram for Mesa Tool.

Step 5: Mesa generates optimized code that the programmer compiles into an executable.

Step 6: Compare the performance and accuracy of the original and optimized programs.

The instrumentation and optimization runs are implemented by calling the Rose libraries to read the original code and parse it into an abstract syntax tree (AST). Mesa analyzes the portion of the AST identified by the pragma to find the set of expressions that may benefit from LUT transformation, then either instruments or optimizes the code by modifying the AST and calling Rose to unparse it back into C and C++ application code.

Mesa 2.0 currently consists of ∼7000 lines of C++ code in 9 modules. The code is freely distributed and available on our project website [48]. Expression enumeration and code generation modules manipulate the AST through inheritance of the node traversal mechanism in Rose, which is based on the visitor pattern. A performance module implements a benchmark to measure the performance of elementary functions and arithmetic operators. An error module implements analytic and numerical analysis using double-precision math for accuracy. An optimization module implements the algorithms presented in Section 5, without making use of an optimization framework. Processing occurs in a pipelined fashion,

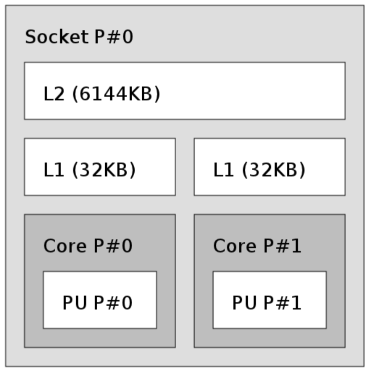


Figure 6.2: Output of *lstopo* command from *hwloc* suite.

with each module contributing to the solution. Expression data is stored in a global object that can be accessed from all modules.

Mesa has a command line option for setting the memory usage constraint. The tool will operate within any specified constraint, but the usual usage is to specify a portion of the mid-level cache for LUT data. Mesa does not determine the size of mid-level cache, but this can be done through external tools such as *lstopo*, which is part of the Hardware Locality Suite (*hwloc*) [72]. Figure 6.2 shows the output of the lstopo command on our test system. From the diagram we see that our test system has two cores with 32KB of dedicated L1 cache, and a shared L2 cache of size 6MB. The programmer is responsible for deciding how much cache to allocate, taking into account the cache requirements of the application. For the examples and case studies in this thesis we set a limit of 4MB on memory usage.

Another tool concern is that the embedded benchmark ties the optimization to the platform on which the tool runs. The Mesa default behavior is to use the performance file instead of running the benchmark each time. Using a performance file has two advantages. First, it provides stability over a series of Mesa executions. Second, performance files can be moved to another system. This allows Mesa to optimize for target systems other than the one on which it executes. The programmer can explicitly request a benchmark run via the command line. Deleting the performance file will also force a benchmark run. Mesa will not perform optimization without the performance file. The benchmark is a straightforward measurement of operators in a loop using randomly generated data in a specified range. The benchmark accumulates results to avoid removal of the loop by code optimization.

6.2 Mesa Evolution

Mesa 2.0 is the only version that supports the LUT optimization algorithm with global scope. The original version of Mesa was 1.0, which required specification of the expression, variables, and constants in a separate file. Programmers using Mesa 1.0 had to extract candidate expressions from the source code, use the tool to generate LUT code and data, then integrate the resulting code back into the application manually. Mesa 1.1, the only other version, operated directly on application code, optimizing individual statements marked by a pragma. The purpose of Mesa 1.1 was to partially automate the application of LUT transforms. Using Mesa 1.1, the programmer had to pinpoint the exact location of expressions. Mesa 2.0 can optimize entire methods that are preceded by a pragma, thereby automatically identifying sets of potential LUT transforms in the source code.

6.3 Effect of Cache Misses

We used Mesa 1.0 to investigate the performance of LUT transforms with respect to the memory hierarchy. In this research we noted that the performance of a LUT transform degrades when LUT data overflows mid-level cache. This behavior has been replicated on newer versions of Mesa and with different compilers including gcc 4.6.1 and icc 12.1.2. It has also been confirmed on a variety of 32-bit and 64-bit Intel and AMD processors, including processors from the Pentium, Xeon and Opteron families. Figure 6.3 compares the performance of the original and optimized SAXS discrete scattering code [76]. On the y-axis, $T_{ORIGINAL}$ is the performance of the original code, and $T_{OPTIMIZED}$ is the performance of the code optimized by Mesa 1.0. The x-axis varies the size of LUT data, from 256KB to 32MB, with a vertical line showing the L2 cache size. The notable result is that the performance of the optimized code is relatively flat until the L2 cache overflows, then it quickly degrades.

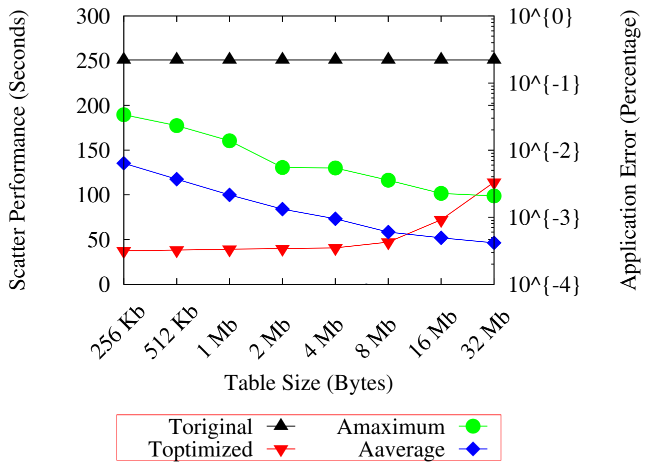


Figure 6.3: SAXS discrete scattering simulation results.
(Intel Xeon E5450, 3.00Ghz, 6MB L2 cache, single core)

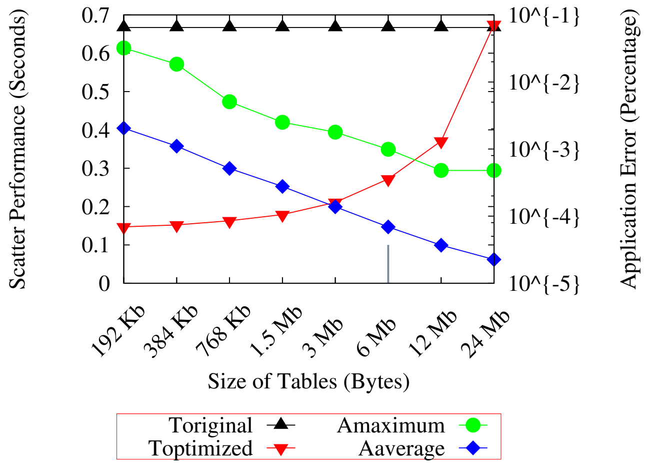


Figure 6.4: SAXS continuous scattering simulation results.
(Intel Xeon E5450, 3.00Ghz, 6MB L2 cache, single core)

The application maximum error $A_{MAXIMUM}$ and average error $A_{AVERAGE}$ is also shown, with the expected improvement as LUT size grows.

Figure 6.4 plots the same information for the SAXS continuous scattering code. The results are similar, except that optimized performance degrades even more quickly. We attribute the degradation in performance to L2 cache misses, and we have documented these using the PAPI library [54].

6.4 Evaluation of Multi-Dimensional Tables

Mesa 1.0 had limited support for multi-dimensional LUT data that we used to experiment with multi-dimensional LUT transforms for the SAXS discrete scattering code [76]. This code originally computed the Debye's formula based on two variables: the distance between atoms (r), and the scattering angle (θ). We compared a 1-dimensional LUT transform that combined the variables to a 2-dimensional transform as shown in Figures 6.5 and 6.6. The latter required significantly more memory to meet the same accuracy, which caused the

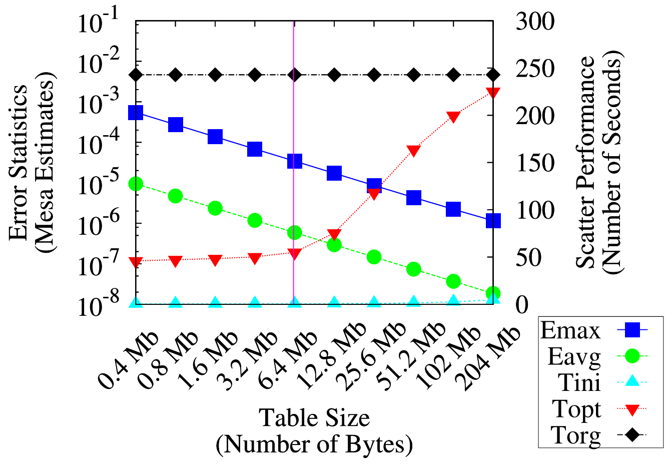


Figure 6.5: Performance of 1-dimensional LUT data.
(Intel Xeon E5450, 3.00Ghz, 6MB L2 cache, single core)

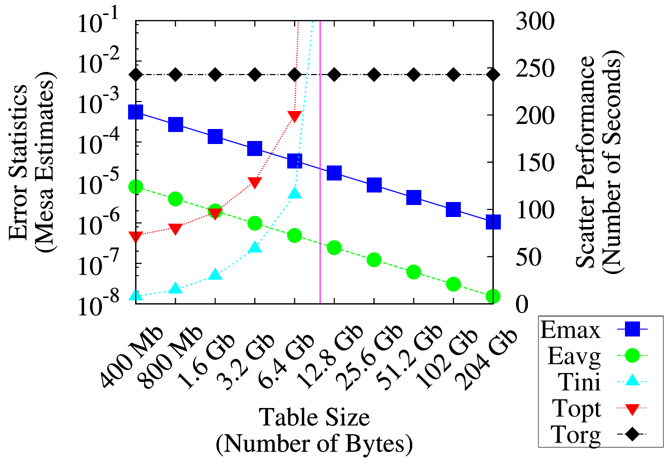
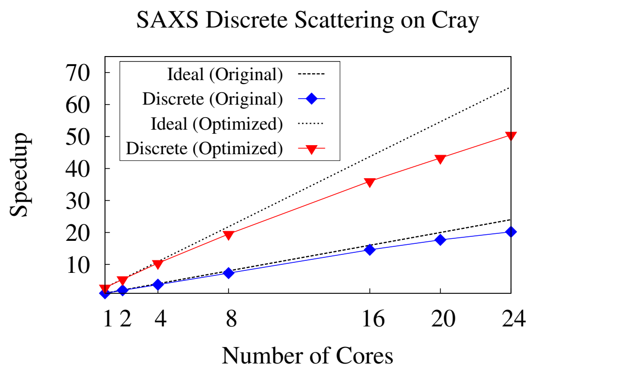


Figure 6.6: Performance of 2-dimensional LUT data.
(Intel Xeon E5450, 3.00Ghz, 6MB L2 cache, single core)

performance to degrade quickly. LUT initialization time, which is generally insignificant on our 1-dimensional LUT transforms, quickly became the dominant factor in the 2-dimensional case. We do not claim that these results generalize to other applications. However, due to these problems, Mesa 1.1 and Mesa 2.0 do not optimize expressions with multiple variables. There are a number of papers that describe the use of multi-dimensional tables [50].

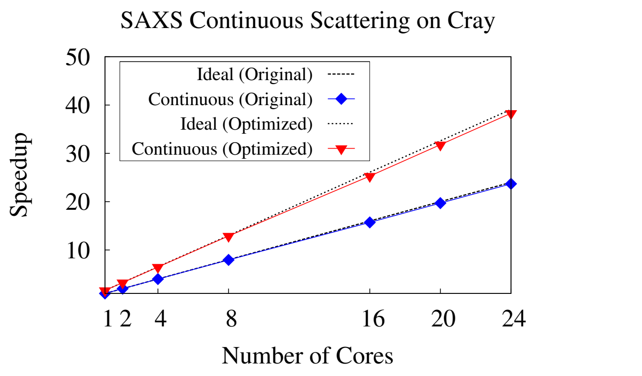
6.5 Parallel Efficiency of Generated Code

We used Mesa 1.1 to evaluate the parallel performance of the automatically generated LUT code To do so, we parallelized the SAXS discrete and continuous scattering loops with OpenMP directives. Figure 6.7 and 6.8 show that Mesa generates code with parallel efficiencies of 84% to 98% using 24 cores on a Cray XT6m computer [77]. We have replicated this on other multi-core systems including 4 and 8-core Xeons. We conclude that our single-core optimizations are independent from and complementary to parallelization.



(Cray XT6m, AMD Opteron 6100, 2.5Ghz, 512KB L2, 6MB L3, 24 cores)

Figure 6.7: Parallel efficiency of SAXS discrete scattering application.



(Cray XT6m, AMD Opteron 6100, 2.5Ghz, 512KB L2, 6MB L3, 24 cores)

Figure 6.8: Parallel efficiency of SAXS continuous scattering application.

Chapter 7

Case Studies

We evaluate our methodology in terms of ease of use, accuracy, and performance by using Mesa to optimize six scientific applications. The first two case studies evaluate *slope aspect* and *solar radiation* computations from the Precipitation-Runoff Modeling System (PRMS), developed by the United States Geologic Survey [58] for hydrologic modeling. The third and fourth case studies come from two applications written for the SAXS project [64], a multi-disciplinary project at CSU between the Molecular Biology, Mathematics, Statistics, and Computer Science departments. The fifth application is Stillinger-Weber, a molecular dynamics program developed and used for research at Cornell University [31]. The sixth application is neural network code [53] developed by a faculty member in our Computer Science department and used in [5]. Some of the applications were originally written in a different language, and we have ported them to C++. We have also made slight modifications to the C applications to support C++ compilation and Mesa optimization.

7.1 Evaluation Methodology

To evaluate applications we use Mesa 2.0 to optimize the performance dominant code in the same manner as a programmer would. We performance profile to find bottleneck functions, which we identify with a pragma. We ask Mesa to instrument the application, and we compile and run the resulting code to profile input domains and call frequencies. Next we ask Mesa to optimize the application. Mesa lists the statements, expressions, and constraints for the problem, and it constructs and solves the optimization problem. Part of our evaluation is to examine the number of possible and actual solutions for each application, and we measure tool processing time. We also produce a Pareto chart that provides insight into the optimization problem by reporting the number and location of Pareto solutions in the error and performance space. Next we select a solution for Mesa to realize. We finish by compiling and running the original and optimized code and comparing the benefit and error predicted by Mesa to the measured application performance and accuracy.

To further evaluate our tool, we use case studies to enable different features of Mesa. For example, we show SAXS discrete scattering with direct access and interpolation. The reason for selecting this application is that it has the highest performance speedup of our case studies, thus there is headroom to support the increased computational cost of linear interpolation. We use the PRMS slope aspect program to compare LUT optimization with and without expression coalescing. The reason for selecting this application is that it repetitively calls sine and cosine with similar domains, so coalescing is especially effective.

Table 7.1 summarizes the performance and results of our case studies. The first five columns show the program and tool statistics: lines of code analyzed, number of expressions, number of possible and actual solutions, and the number of Pareto optimal solutions, followed by the processing time. The next two columns show the sampling method (direct access or linear interpolation), and whether expression coalescing is enabled or disabled. The final two columns report the performance speedup and error relative to the original output. For example, Mesa extracts 9 expressions from the PRMS slope aspect code, from which it analyzes 384 solutions, and finds 9 to be Pareto optimal. Mesa then generates optimized code that achieves a 4.5$\times$ speedup at the expense of 2.67$\times 10^{-1}$% error.

Table 7.1: Application results from Mesa optimization.
(Intel Core 2 Duo, E8300, family 6, model 23, 2.83GHz, single core)

Application Name	Lines of Code Analyzed	Number of Expressions	Possible Solutions	Actual Solutions	Pareto Solutions	Processing Time	Sampling Method	Expression Coalescing	Performance Speedup	Relative Error
PRMS Slope Aspect	35	9	512	384	9	13.7s	Direct	Disabled	**4.4x**	**2.67E-01%**
PRMS Slope Aspect	35	11	2048	425	9	15.5s	Direct	Aggressive	**4.3x**	**8.21E-06%**
PRMS Solar Radiation	7	6	64	64	8	14.1s	Direct	Moderate	**2.2x**	**2.97E-04%**
SAXS Discrete	60	3	8	4	3	11.2s	Direct	Disabled	**6.8x**	**4.06E-03%**
SAXS Discrete	60	3	8	4	3	16.5s	Linear	Disabled	**3.0x**	**5.55E-04%**
SAXS Continuous	30	5	32	20	4	10.8s	Direct	Conservative	**4.0x**	**1.48E-04%**
Stillinger-Weber	44	6	64	36	3	9.3s	Direct	Disabled	**1.4x**	**2.91E-02%**
Neural Network (logistics)	5	2	4	3	2	4.9s	Direct	Disabled	**2.2x**	**8.70e-02%**
Neural Network (hypertan)	5	1	2	2	2	2.8s	Direct	Disabled	**2.8x**	**6.30e-01%**

7.2 Hydrology Modeling

We present two case studies of LUT optimization using computations from the PRMS application. The first function we optimize computes the slope aspect for a point on a terrain grid based a variety of parameters including the latitude and declination. The second function we optimize computes the solar radiation based on the latitude and the slope aspect. The source code is 1500 lines of C++ that we ported from Java code. The functions are called from a test program that randomly varies the input data, provides timing, and maintains the results to allow a comparison of accuracy.

7.2.1 Slope Aspect Computation

Our first case study is a slope aspect computation from the PRMS application. Figure 7.1 shows a partial listing of the slope aspect function. We initially run Mesa without expression coalescing, specifying direct access sampling, and a 4MB memory usage constraint. Mesa identifies 9 expressions from 8 statements as shown in Table 7.2, and an intersection constraint X0 $\cap$ X1. The list of expressions shows the domain extent, maximum slope, and estimated benefit. The maximum slope is computed by error analysis, and is not related to the slope aspect computation itself. The call frequency for all expressions is 3.65 $\times 10^7$. The estimated benefit is similar for all of the expressions, but there is variance in the domains and slopes, so some of the associated LUT transformations will be more accurate than others.

```
#pragma LUTOPTIMIZE
float CGeospatial::Calculation(int julDay, double aspect, double slope, double latitude)
{
S103   // Declination calculation
S104   double decline  = 0.4095*sin(0.01720*(julDay-79.35));
       ...
S110   double sin_decline  = sin(decline);
S111   double cos_decline  = cos(decline);
S112   double sin_latitude = sin(latitude);
S113   double cos_latitude = cos(latitude);
S114   double sin_slope    = sin(slope);
S115   double cos_slope    = cos(slope);
S116   double cos_aspect   = cos(aspect);
S117   double slope = (sin_decline*sin_latitude*cos_slope) -
                      (sin_decline*cos_latitude*sin_slope*cos_aspect) +
                      (cos_decline*cos_latitude*cos_slope) +
                      (cos_decline*sin_latitude*sin_slope*cos_aspect);
}
```

Figure 7.1: Source code for slope aspect computation.

Table 7.2: Expressions from slope aspect code.

Expression Identifier	**Statement Identifier**	**Expression Syntax**	**Domain Extent**	**Maximum Slope**	**Estimated Benefit**
X0	S104	sin(dAngle)	6.27	1.00	1.24s
X1	S104	0.40954 * sin(dAngle)	6.27	0.41	1.36s
X2	S110	sin(declRad))	6.28	1.00	1.24s
X3	S111	cos(declRad)	6.28	1.00	1.39s
X4	S112	sin(latRad)	1.05	1.00	1.24s
X5	S113	cos(latRad)	1.05	0.87	1.39s
X6	S114	sin(slopeRad)	1.05	1.00	1.24s
X7	S115	cos(slopeRad)	1.05	0.87	1.39s
X8	S116	cos(aspRad)	6.28	1.00	1.39s

Figure 7.2 shows the Mesa optimization run for the PRMS slope aspect code. The intersection constraint limits the problem to 384 actual out of 512 possible solutions. Mesa finds 9 Pareto optimal solutions, which for this application consist of adding of one transform at a time with increasing benefit and error. We select the maximum benefit solution (C8) and Mesa realizes it, reporting the error, benefit, and size for each expression. Summing these values gives the solution error and benefit shown in Figure 7.2. For example, we sum the error (E_i) and benefit (B_i) columns to compute a total error of $8.148{\times}10^{-05}$ and a total benefit of 10.66s. The size column (S_i) shows the cache allocation for LUT transforms.

```
Optimizing for cache size 4194304
512 solutions (possible)
384 solutions (actual)
9 solutions (pareto optimal)
Solution  Error      Benefit    Optimizations
C0        0.000e+00  0.000e+00
C1        4.324e-07  1.391e+09  X5
C2        1.730e-06  2.781e+09  X5,X7
C3        4.088e-06  4.026e+09  X5,X7,X4
C4        7.445e-06  5.271e+09  X5,X7,X4,X6
C5        1.470e-05  6.632e+09  X5,X7,X4,X6,X1
C6        3.097e-05  8.023e+09  X5,X7,X4,X6,X1,X3
C7        5.323e-05  9.413e+09  X5,X7,X4,X6,X1,X3,X8
C8        8.148e-05  1.066e+10  X5,X7,X4,X6,X1,X3,X8,X2
Select solution: 8
X5 Di = 1.05 Mi = 0.87 Ei = 5.936e-06 Bi = 1.391e+09 Si = 305556 (298KB)
X7 Di = 1.05 Mi = 0.87 Ei = 5.936e-06 Bi = 1.391e+09 Si = 305558 (298KB)
X4 Di = 1.05 Mi = 1.00 Ei = 6.379e-06 Bi = 1.245e+09 Si = 328342 (321KB)
X6 Di = 1.05 Mi = 1.00 Ei = 6.379e-06 Bi = 1.245e+09 Si = 328344 (321KB)
X1 Di = 6.27 Mi = 0.41 Ei = 9.985e-06 Bi = 1.361e+09 Si = 513995 (502KB)
X3 Di = 6.28 Mi = 1.00 Ei = 1.562e-05 Bi = 1.391e+09 Si = 804119 (785KB)
X8 Di = 6.28 Mi = 1.00 Ei = 1.562e-05 Bi = 1.391e+09 Si = 804271 (785KB)
X2 Di = 6.28 Mi = 1.00 Ei = 1.562e-05 Bi = 1.245e+09 Si = 804119 (785KB)
Mesa 2.0: Generating optimized code
```

Figure 7.2: Optimization results for slope aspect code without coalescing.

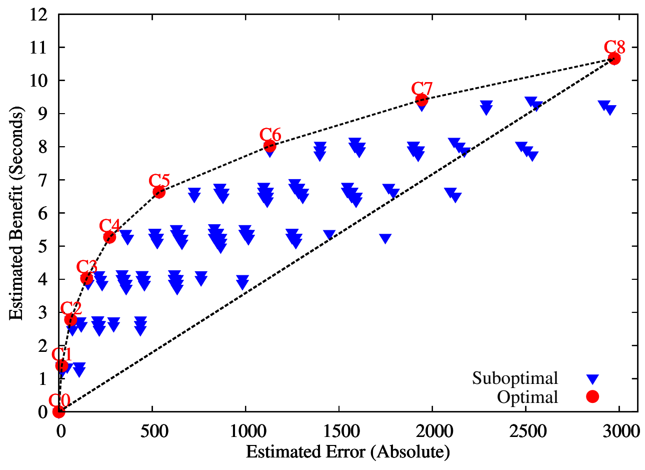


Figure 7.3: Pareto chart for slope aspect code without coalescing.

```
Coalescing X9 from X0 X2 X4 X6
Adding [X0,X9]
Adding [X1,X9]
Adding [X2,X9]
Adding [X4,X9]
Adding [X6,X9]
Coalescing X10 from X3 X5 X7 X8
Adding [X3,X10]
Adding [X5,X10]
Adding [X7,X10]
Adding [X8,X10]
```

Figure 7.4: Expression coalescing for slope aspect code.

Figure 7.3 shows the chart of Pareto optimal solution for the PRMS source code, without coalescing. The Pareto curve climbs steeply, then flattens after solution C5. The reason for this is that the performance benefit is similar for all of the transforms, but the error rate grows quickly after C5. As a result, the programmer must decide whether to accept the additional error to get all of the possible benefit.

We repeat the experiment with the same parameters except that we enable aggressive expression coalescing. Mesa identifies the same candidate statements, expressions, constraints, domains, and slopes. The sine and cosine calls are coalesced into two new expressions, and a series of intersection constraints are introduced as shown in Figure 7.4. The processing time is 13% longer because coalescing introduces new expressions that require error analysis and increase the complexity of the optimization problem.

```
Optimizing for cache size 4194304
2048 solutions (possible)
425 solutions (actual)
9 solutions (pareto optimal)
Solution   Error       Benefit     Optimizations
C0          0.000e+00  0.000e+00
C1          1.578e+01  1.391e+09  X5
C2          6.314e+01  2.781e+09  X5,X7
C3          1.492e+02  4.026e+09  X5,X7,X4
C4          2.717e+02  5.271e+09  X5,X7,X4,X6
C5          5.367e+02  6.632e+09  X5,X7,X4,X6,X1
C6          9.568e+02  8.052e+09  X4,X6,X10
C7          1.749e+03  1.054e+10  X10,X9
C8          2.439e+03  1.066e+10  X4,X6,X10,X1,X2
Select solution: 7
X10 Di = 6.28 Mi = 1.00 Ei = 8.748e+02 Bi = 5.563e+09 Si = 2097350 (2048KB)
X9  Di = 6.28 Mi = 1.00 Ei = 8.746e+02 Bi = 4.979e+09 Si = 2096954 (2048KB)
Mesa 2.0: Generating optimized code
```

Figure 7.5: Optimization results for slope aspect code with coalescing.

Because of expression coalescing, the solution space expands to 425 actual out of 2048 possible solutions. Mesa finds 9 Pareto optimal solutions, several of which use the coalesced transformations as shown in Figure 7.5 because they provide the same benefit with less error. We select solution C7 because it has 99% of the benefit and approximately ∼28% less error than C8, the maximum benefit solution. Solution C8 has omitted one of the coalesced expressions to get slightly more performance, thereby introducing more error. Mesa realizes LUT transformations for the two coalesced expressions with allocations of around 2MB each, which is much larger than the allocations of the uncoalesced expressions. As a result, we can expect higher accuracy from the version optimized with expression coalescing.

Figure 7.6 shows the chart of solutions for the PRMS slope aspect code, with coalescing. The top three Pareto solutions use one or both of the coalesced expressions. The chart illustrates why C7 provides a better tradeoff between performance and accuracy than C8. The maximum benefit solution C8 lies a small distance above, but a larger distance to the right of C7, meaning that is has only slightly more performance but significantly more error.

Now we can compare the performance and accuracy of the optimization with and without expression coalescing. Table 7.3 shows the execution times, performance speedup, and maximum absolute and relative error of each version of the program. The optimized slope

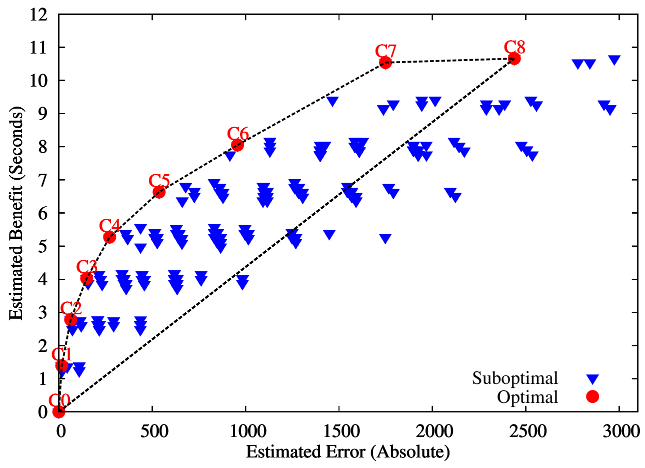


Figure 7.6: Pareto chart for slope aspect with coalescing.

Table 7.3: Comparison of results for slope aspect code.
(Intel Core 2 Duo E8300, 2.83GHz, 6MB L2 cache, single core)

Program Versions	**Execution Time**	**Performance Speedup**	**Absolute Error**	**Relative Error**
Original Code	8.85s	n/a	0.0	n/a
Optimized Code: No Coalescing	2.00s	4.4×	83498.7	$2.67 \times 10^{-1}\%$
Optimized Code: Coalescing	2.07s	4.3×	2.6	$8.21 \times 10^{-6}\%$

aspect code achieves a 4.4× speedup without coalescing, and 4.3× with aggressive coalescing. Thus both versions are very close in performance as would be expected since both optimize all of the elementary functions in the source code and both constrain LUT data to reside in cache. However, the optimized slope aspect code without coalescing has a relative error of $2.67 \times 10^{-1}\%$, as compared to $8.21 \times 10^{-6}\%$ with coalescing. The more than four orders of magnitude improvement in accuracy is due to the superior allocation of cache resources achieved by expression coalescing.

7.2.2 Solar Radiation Computation

Our second case study is a solar radiation computation from the PRMS application. Figure 7.7 shows a listing of the performance dominant function, which combines a series of elementary function calls. Note that unlike our previous examples, the function result is not a simple accumulation of values returned from these calls. Instead, the values approximated

```
#pragma LUTOPTIMIZE
double CSolarRadiation::func3r(double v, double w, double x, double y, double r1, double d)
       {
S212     double dResult = r1*PI_12*(sin(d)*sin(w)*(x-y)+cos(d)*cos(w)*(sin(x+v)-sin(y+v)));
S213     return dResult;
       }
```

Figure 7.7: Source code for solar radiation computation.

by the LUT optimization are multiplied together. To see what effect this may have we revisit the correspondence of our error model to application accuracy in this section.

We run Mesa with expression coalescing, direct access sampling, and a 4MB memory usage. Mesa identifies 6 expressions from statement S212 and no intersection constraints. Table 7.4 shows 4 sine expressions and 2 cosine expressions. The domain extents and maximum slopes are dissimilar enough that moderate coalescing does not combine any expressions. The call frequency for all expressions is 4.58 $\times 10^7$.

Table 7.4: Expressions from solar radiation code.

Expression Identifier	**Statement Identifier**	**Expression Syntax**	**Domain Extent**	**Maximum Slope**	**Estimated Benefit**
X0	S212	sin(d)	0.82	1.00	1.49s
X1	S212	sin(w)	1.03	0.93	1.49s
X2	S212	cos(d)	0.82	0.40	1.72s
X3	S212	cos(w)	1.03	0.99	1.72s
X4	S212	sin(x+v)	4.68	1.00	1.49s
X5	S212	sin(y+v)	4.89	1.00	1.49s

Figure 7.8 shows the Mesa optimization run for the PRMS solar radiation code. The solution space has all 64 possible solutions because there are no intersection constraints. Mesa finds 8 Pareto optimal solutions, with the expressions X2 and X0 added first because of their higher benefit, and X4 and X5 added last because of their higher error. We select the C7 solution, which has the maximum benefit, and Mesa realizes its LUT transforms. Mesa then reports the error, benefit, and size for each expression. Note that the least accurate expressions X4 and X5 get more than 1MB of memory, and the others get 512KB or less.

Table 7.5 shows the execution times, performance speedup and error statistics for the original and optimized code. The error is calculated by a program that compares all of

```
Optimizing for cache size 4194304
64 solutions (possible)
64 solutions (actual)
8 solutions (pareto optimal)
Solution   Error       Benefit     Optimizations
C0         0.000e+00  0.000e+00
C1         7.117e+00  1.723e+09  X2
C2         4.756e+01  3.217e+09  X2,X0
C3         5.460e+01  3.446e+09  X2,X3
C4         1.350e+02  4.940e+09  X2,X3,X0
C5         2.623e+02  6.435e+09  X2,X3,X0,X1
C6         6.921e+02  7.929e+09  X2,X3,X0,X1,X4
C7         1.342e+03  9.423e+09  X2,X3,X0,X1,X4,X5
Select solution: 7
X2 Di = 0.82 Mi = 0.40 Ei = 9.775e+01 Bi = 1.723e+09 Si =  305395 (298KB)
X3 Di = 1.03 Mi = 0.99 Ei = 1.730e+02 Bi = 1.723e+09 Si =  540470 (528KB)
X0 Di = 0.82 Mi = 1.00 Ei = 1.549e+02 Bi = 1.494e+09 Si =  484062 (473KB)
X1 Di = 1.03 Mi = 0.93 Ei = 1.677e+02 Bi = 1.494e+09 Si =  523955 (512KB)
X4 Di = 4.68 Mi = 1.00 Ei = 3.705e+02 Bi = 1.494e+09 Si = 1157625 (1130KB)
X5 Di = 4.89 Mi = 1.00 Ei = 3.786e+02 Bi = 1.494e+09 Si = 1182797 (1155KB)
Mesa 2.0: Generating optimized code
```

Figure 7.8: Optimization results for solar radiation code without coalescing.

Table 7.5: Comparison of results for solar radiation code.
(Intel Core 2 Duo E8300, 2.83GHz, 6MB L2 cache, single core)

Program Versions	**Execution Time**	**Performance Speedup**	**Absolute Error**	**Relative Error**
Original Code	13.67s	1.0×	0.0	0.0%
Optimized Code	6.09s	2.2×	1.56×10^{-1}	2.97×10^{-4}%

the individual output values to find the maximum and average errors. The optimized solar radiation code achieves a 2.2× speedup with a relative error of 2.97×10^{-4}%.

We now use the solar radiation application to evaluate our error and performance model in the same fashion as in Sections 5.5.2 and 5.5.2. Figure 7.9 shows that the performance model predicts the slope of the actual performance, but slightly overestimates the benefit. Some of the solutions are significantly below the estimated performance. We attribute this again to ILP and other compiler optimizations. Figure 7.10 compares the estimated maximum (absolute) error from the error analysis against the actual maximum (relative) error of the optimized application. The error model appears better in this case than for the example code in Section 5, possibly because the approximation results are not accumulated.

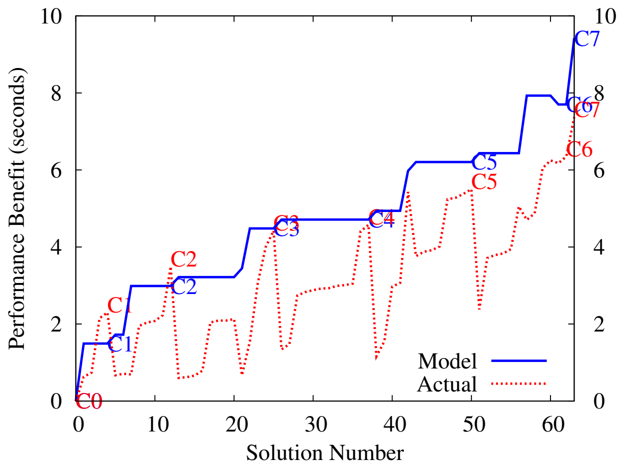


Figure 7.9: Performance model versus application accuracy for solar radiation code.

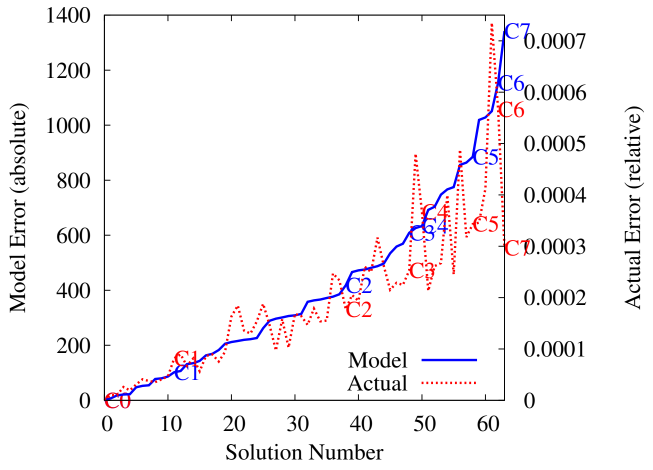


Figure 7.10: Error model versus application accuracy for solar radiation code.

7.3 Molecular Biology

Our next two case studies are from the SAXS project. SAXS is an experimental technique that explores the structure of molecules [26]. SAXS results can be simulated on a computer via discrete or continuous algorithms that operate on molecular models. A partial discrete scattering simulation was written in the *R* language by members of the Statistics department, then ported to C++ and completed by the author. A complete simulation of continuous scattering was written in *MATLAB* by members of the Math department, then ported to C++ by various people in the Computer Science department. The current source base for both programs is now around 6000 lines, excluding documentation, scripts, and test programs and data. Separate applications exist for the discrete and continuous scattering algorithms. In this section we evaluate the LUT optimization of both applications.

7.3.1 Saxs Discrete Scattering

Our third case study is the SAXS application that simulates the discrete scattering of a molecular model using Debye's formula [26] shown in Equation (7.1).

$$I(\theta) = 2\Sigma_{i=1}^{N-1}\Sigma_{j=i+1}^{N}F_i(\theta)F_j(\theta)sin(4\pi r\theta)/(4\pi r\theta) \tag{7.1}$$

A partial listing of the code that implements Debye's formula is shown in Figure 7.11. The $fDistance$ variable represents the distance r between atoms in the above formula. The $fTheta$ variable is the scattering angle θ in the above formula. The $rTheta$ variable combines the variables to avoid an expression with two variables. The only elementary function called in the loop is the sine function. Running gprof on the SAXS scattering code shows that the listed code dominates the computation time, so we insert a pragma above the function.

```
      // Iterate steps (outer loop)
      for (step = 0; step < 1000; ++step) {
        // Iterate atoms (middle loop)
        for (atom1 = 0; atom1 < vecAtoms.size(); ++atom1) {
          // Iterate atoms (inner loop)
          for (atom2 = atom1; atom2 < vecAtoms.size(); ++atom2) {
S193        // Compute distance between atoms
S194        float fDistance = distance(atom1, atom2);
S195
S196        // Compute scattering angle
S197        float fTheta = m_fStep * (float)(step + 1);
S198
S199        // Combine parameters to scatter
S200        float rTheta = fDistance * fTheta;
S201
S202        // Optimize subexpression shown below
S203        fIntermediate = sinf(FOURPI * rTheta) / (FOURPI * rTheta);
          }
        }
      }
```

Figure 7.11: Dominant calculation from SAXS discrete scattering code.

Table 7.6 shows the expressions extracted during the Mesa optimization run. Mesa considers only the statement S203, from which it extracts three expressions, all of which include the sine function and varying amounts of associated math. Since all expressions come from the same statement, they share the call frequency of 4.66 $\times 10^9$. However, the domain extents and maximum slopes vary widely. For example, the expressions X0 and X2 have the same domain as the *rTheta* parameter, which is [0.0,20.0], but expression X1 has a range that is 4π larger as shown by its expression syntax. The maximum slope of X1 is 1.0, since X1 represents the sine function. Expression X0 represents 4π times the sine function, so it has a maximum slope of 4π. Expression S2 has a high maximum slope, because of the division. However, Mesa estimates more performance for X2 because it handles two multiplies and a division, as well as the sine function.

Table 7.6: Expressions from SAXS discrete scattering code.

Expression Identifier	**Statement Identifier**	**Expression Syntax**	**Domain Extent**	**Maximum Slope**	**Estimated Benefit**
X0	S203	sin(4πrTheta)	20.00	12.57	172.3s
X1	S203	sin(rTheta)	251.33	1.00	157.4s
X2	S203	sin(4πrTheta)/(4πrTheta)	20.00	52428.80	207.2s

Mesa finds two Pareto optimal solutions in addition to the empty solution. These trivially consist of selecting either expressions X0 or X2, each with an allocation of the entire 4MB. The expression X1 does not appear as Pareto optimal, because X0 provides more benefit for the same amount of error. The Pareto optimal solutions present an interesting tradeoff for the programmer, since X2 gives 20% more performance, but almost 2× the error. Figure 7.12 shows the solutions and the selection of solution C2 to maximize the performance benefit. We have omitted the Pareto chart because of the small number of solutions.

Figure 7.13 shows the performance and accuracy evaluation of the original and optimized versions of the SAXS discrete scattering code. Both versions of the program are run, then our comparison program evaluates the maximum relative error between the program results. The performance speedup is 6.8×, with a maximum relative error of $4.06X10^{-3}$%. This meets our accuracy requirements, so we do not have to consider optimizing expression X0 instead to decrease the error. The performance benefit comes from reuse in the SAXS code.

```
Mesa 2.0: Solving optimization problem
Optimizing for cache size 4194304
8 solutions (possible)
4 solutions (actual)
3 solutions (pareto optimal)
Solution   Error      Benefit    Optimizations
C0         0.000e+00  0.000e+00
C1         5.580e+05  1.723e+11  X0
C2         2.328e+09  2.072e+11  X2
Select solution: 2
X2 Di = 20.00 Mi = 52428.80 Ei = 2.328e+09 Bi = 2.072e+11 Si = 4194304 (4096KB)
Mesa 2.0: Generating optimized code
```

Figure 7.12: Optimization results for SAXS discrete scattering code.

```
>>> ./DiscreteOriginal ../Data/1xib.pdb 1xib.int.orig
Scatter computation: 196.2 seconds.

>>> ./DiscreteOptimized ../Data/1xib.pdb 1xib.int.optd
Scatter computation: 29.0 seconds.

>>>./Compare 1xib.int.orig 1xib.int.optd 0.0
0.00000e+00\% minimum relative error
4.05608e-03\% maximum relative error
8.13219e-04\% average relative error
```

Figure 7.13: Comparison of results for SAXS discrete code.

The reuse occurs because we evaluate the function 4MB / sizeof(float) = $\sim 1\times 10^6$ times to initialize the LUT data, but the optimized program accesses the LUT data 4.6 billion times. Thus each LUT entry is reused more than 4,600 times on the average. Mesa estimates a performance benefit of 207.2s as compared to an actual benefit of 167.2, a difference of 19%.

To conclude our evaluation of the SAXS discrete code, we optimize the code with linear interpolation instead of direct access sampling. Mesa extracts the identical set of expressions and finds the same Pareto optimal solutions. However, Mesa estimates that the benefits of each expression are reduced by the measured difference between direct access and linear interpolation, which is 1.8× on our test system. The measured performance of the linear interpolation code is 65.5s, which is 2.3× slower than direct access. The accuracy improves by slightly less than an order of magnitude to 5.55×10^{-4}. Thus linear interpolation increases accuracy, but degrades performance significantly.

7.3.2 Saxs Continuous Scattering

Our fourth case study uses SAXS project code that implements another set of equations that model scattering, using a continuous instead of a discrete algorithm. Performance profiling shows that the ScatterSample function dominates the computation time. The C++ code for the inner loop of ScatterSample is shown in Figure 7.14. The equations that define continuous scattering are shown in Equation (7.2).

$$I(q,\psi) = \left(\sum_{j=1}^{N} d_j e^{-\sigma_j^2 q\cdot q/2} cos(q\cdot\mu_j(\psi))\right)^2 + \left(\sum_{j=1}^{N} d_j e^{-\sigma_j^2 q\cdot q/2} sin(q\cdot\mu_j(\psi))\right)^2 \tag{7.2}$$

We invoke Mesa 2.0 with direct access, conservative expression coalescing, and a 4MB memory constraint. No other elementary functions exist in the function, so S145 and S146 are the only candidates statements. Table 7.7 shows the expressions extracted during the Mesa optimization run, after error analysis and performance modeling. The exponential expressions from both statements are identical, and the sine and cosine expressions are similar but with slightly different benefit. Expressions X0 and X2 are coalesced into X4, and the intersection constraints X0 ∩ X4 and X2 ∩ X4 are added. The expression X4 has the same domain and maximum slope as X0 and X2, but twice the benefit.

The intersection constraints limit the number of actual solutions to 20 out of 32 possible solutions. From these Mesa finds three Pareto optimal solutions in addition to the empty solution as shown in Figure 7.15. The X4 expression that coalesces the exponential function is picked up first, then the X3 cosine expression, and finally the X1 sine expression. The X0 and X2 expressions are suboptimal because the coalesced X4 has higher benefit for the same amount of error, so they do not appear in the Pareto optimal list. The X1 and X3 expressions

```
        for (int j = 0; j < m_vecGeometry.size(); j++) {
          ...
          // Scattering equation

S145      dSum0 += m_vecGeometry[j].fDensity * exp(fProduct) * sin(vecProduct);
S146      dSum1 += m_vecGeometry[j].fDensity * exp(fProduct) * cos(vecProduct);
          ...
        }
```

Figure 7.14: SAXS continuous scattering code.

Table 7.7: Expressions from SAXS continuous scattering code.

Expression Identifier	**Statement Identifier**	**Expression Syntax**	**Domain Extent**	**Maximum Slope**	**Estimated Benefit**
X0	S145	exp(fProduct)	0.01	0.99	4.88s
X1	S145	sin(vecProduct)	30.0	1.00	6.47s
X2	S146	exp(fProduct)	0.01	0.99	4.88s
X3	S146	cos(vecProduct)	30.0	1.00	7.46s
X4	S145, S146	exp(fProduct)	0.01	1.00	9.766s

```
Mesa 2.0: Solving optimization problem
Optimizing for cache size 4194304
32 solutions (possible)
20 solutions (actual)
4 solutions (pareto optimal)
Solution   Error        Benefit      Optimizations
C0          0.000e+00   0.000e+00
C1          1.893e+00   9.764e+09   X4
C2          2.996e+03   1.723e+10   X4,X3
C3          1.167e+04   2.370e+10   X4,X3,X1
Select solution: 3
X4 Di =   0.01 Mi = 1.00 Ei = 2.092e+02 Bi = 9.764e+09 Si =   37942 (37KB)
X3 Di = 30.00 Mi = 1.00 Ei = 5.730e+03 Bi = 7.462e+09 Si = 2078181 (2029KB)
X1 Di = 30.00 Mi = 1.00 Ei = 5.730e+03 Bi = 6.470e+09 Si = 2078181 (2029KB)
Mesa 2.0: Generating optimized code
```

Figure 7.15: Optimization results for SAXS continuous scattering code.

approximately the same maximum slope, but with a much more extensive domain, so their cache allocation is more than 50 times larger. We show selection of solution C3, which has the maximum benefit. Figure 7.16 shows the Pareto chart.

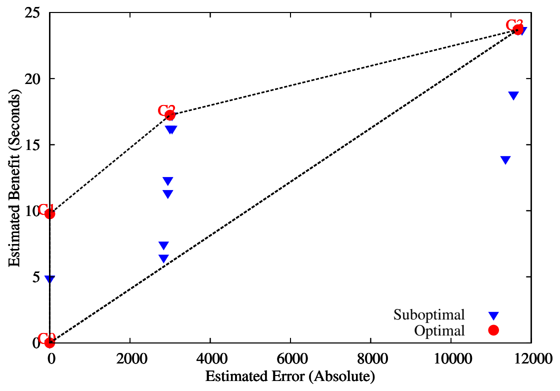


Figure 7.16: Pareto chart for SAXS continuous scattering code.

```
>>> ./ContinuousOriginal ../Data/1xib.xyzd 1xib.orig 25
10.1 seconds.

>>> ./ContinuousOptimized ../Data/1xib.xyzd 1xib.optd 25
2.5 seconds.

>>> ./Compare 1xib.orig 1xib.optd 0.0 -xyi
0.00000e+00\% minimum relative error
1.47644e-04\% maximum relative error
5.14799e-05\% average relative error
```

Figure 7.17: Optimization results for SAXS continuous scattering code.

Figure 7.17 shows the performance and accuracy evaluation of the original and optimized versions of the SAXS continuous scattering code. The performance speedup is 4.0×, with a maximum relative error of $1.48X10^{-4}\%$. The reuse analysis is similar to the SAXS discrete case, because LUT initialization requires ∼1 million function evaluations, but the LUTs are accessed approximately 800 million times. Mesa estimates a performance benefit of 23.7s as compared to the actual benefit of 7.6s. In addition to the factors such as ILP that are mentioned in Section 5.3, the overestimate occurs partially because the compiler uses common subexpression elimination (CSE) to avoid calling the exponential function with the same arguments in subsequent statements. Examination of the generated assembly code shows that the compiler issues a single exponential call in this case.

7.4 Molecular Dynamics

Our fifth case study is Stillinger-Weber, a molecular dynamics program that models the physical movement of atoms and molecules by computing the potential energy and interaction forces of particles. The code consists of around 3000 lines of C developed by Mohit Haran, James Catherwood, and Paulette Clancy [31]. The molecular dynamics simulation is performed over a series of time steps to predict particle trajectories. Many molecular dynamics applications exist, but we have chosen Stillinger-Weber because the code contains a manual LUT transformation done by the original authors. The calculations in Stillinger-Weber are based on the potential energy equations [71] of the same name, which take into

```
S149   sqrt(r2j)
S150   sqrt(r2k)
       ...
S159   expgj = exp(g * (1.0/(rrj-a)));
S160   expgk = exp(g * (1.0/(rrk-a)));
```

Figure 7.18: Stillinger-Weber molecular dynamics program.

account 2-body (ϕ_2) and 3-body (ϕ_3) interactions that call the exponential function as shown in Equations (7.3) and (7.4):

$$E = \sum_{i}\sum_{j>i}\phi_2(r_{ij}) + \sum_{i}\sum_{j\neq i}\sum_{k<i}\phi_3(r_{ij}r_{ik}\theta_{ijk}) \tag{7.3}$$

$$\phi_2(r_{ij}) = A_{ij}\epsilon_{ij}[B_{ij}(\frac{\sigma_{ij}}{r_{ij}})^{p_{ij}}]exp(\frac{\sigma_{ij}}{r_{ij}-a_{ij}\sigma_{ij}}) \tag{7.4}$$

$$\phi_3(r_{ij}, r_{ik}\theta_{ijk}) = \lambda_{ijk}\epsilon_{ijk}[cos\theta_{ijk} - cos\theta_{0ijk}]^2 exp(\frac{\gamma_{ij}\sigma_{ij}}{r_{ij}-a_{ij}\sigma_{ij}})exp(\frac{\gamma_{ik}\sigma_{ik}}{r_{ik}-a_{ik}\sigma_{ik}})$$

The original version of Stillinger-Weber optimized the 2-body and 3-body calculation by precomputing multiple lookup tables for series of expressions. To evaluate Stillinger-Weber we removed LUT transformation code from the original version and inserted straightforward implementations of the Stillinger-Weber equations into the 2-body and 3-body loops. We used the resulting unoptimized version of Stillinger-Weber as a baseline for performance and accuracy. The elementary function calls in the 3-body code are shown in Figure 7.18. Each statement is executed 5.8 $\times 10^7$ times.

We invoke Mesa 2.0 with direct access, expression coalescing disabled, and a 1MB cache constraint. Table 7.8 shows the expressions extracted during the Mesa optimization run, after error analysis and performance modeling. Two versions of each exponential call are listed, with and without associated math. The square root calls have negative benefit, so we do not expect them to show up in an optimal solution.

The intersection constraints limit the number of actual solutions to 36 out of 64 possible solutions. From these Mesa finds only two Pareto optimal solutions in addition to the empty solution as shown in Figure 7.19. We show selection of solution C2, which has the maximum

Table 7.8: Expressions from Stillinger-Weber program.

Expression Identifier	**Statement Identifier**	**Expression Syntax**	**Domain Extent**	**Maximum Slope**	**Estimated Benefit**
X0	S149	sqrt(r2j)	2.43	0.56	-0.17s
X1	S150	sqrt(r2k)	2.43	0.56	-0.17
X2	S159	exp(1.2*(1.0/(rrj - 1.8)))	0.90	0.45	1.69s
X3	S159	exp(rrj)	1260084.28	0.15	1.44s
X4	S160	exp(1.2*(1.0/(rrk - 1.8)))	0.90	0.45	1.69s
X5	S160	exp(rrk)	1260084.28	0.15	1.44s

benefit. The X2 and X4 expressions are the exponential calls, with associated math. They have equal benefit and therefore each receive a 512KB allocation. We have omitted the Pareto chart because it is trivial.

The Mesa version achieves 4.5× better accuracy with 5.6× less memory usage than the original code with the manual LUT optimization, but the performance is 18% slower. Table 7.9 shows the results. We modified the Mesa-generated code by hand to include all of the expressions optimized by the original version. By doing so we were able to closely match the performance of the original version with 3.5× better accuracy and 2.8× less memory usage. The manual optimizations were needed because Mesa was unable to generate code for two of the complex expressions in the Stillinger-Weber program, and do not reflect a limitation of our methodology.

```
Mesa 2.0: Solving optimization problem
Optimizing for cache size 4194304
64 solutions (possible)
36 solutions (actual)
3 solutions (pareto optimal)
Solution  Error      Benefit     Optimizations
C0        0.000e+00  0.000e+00
C1        1.137e+01  1.697e+09   X4
C2        4.550e+01  3.394e+09   X2,X4
Select solution: 2
X2 Di = 0.90 Mi = 0.45 Ei = 2.275e+01 Bi = 1.697e+09 Si = 524289 (512KB)
X4 Di = 0.90 Mi = 0.45 Ei = 2.275e+01 Bi = 1.697e+09 Si = 524289 (512KB)
Mesa 2.0: Generating optimized code
```

Figure 7.19: Optimization results for Stillinger-Weber program.

Table 7.9: Stillinger-Weber accuracy and performance comparison.
(Intel Core 2 Duo E8300, 2.83GHz, 6MB L2 cache, single core)

Program Version	**Execution Time**	**Performance Speedup**	**Application Error**	**Memory Usage**
Original	8.79s	1.66×	0.135%	5.6MB
Mesa (manual)	8.99s	1.63×	0.038%	2.0MB
Mesa (automated)	10.37s	1.41×	0.030%	1.0MB
Unoptimized	14.62s	1.00×	0.000%	0.0MB

We also investigated the difference in accuracy between the original and Mesa versions and found that the *ad hoc* transformation propagates and magnifies error terms by combining LUT values in successive expressions. Mesa avoids this problem by optimizing only the critical expressions. The disparity in memory usage between these versions is because (1) Mesa stores single-precision values and the original tables were double-precision, and (2) fewer expressions were optimized in the Mesa version. Mesa allowed us to experiment with different LUT sizes, and we discovered that we could improve accuracy significantly with a relatively small table. Another benefit from using Mesa is that the optimization required no coding except for the addition of the pragma.

7.5 Neural Network

Our sixth case study evaluates a neural network program [53] developed by Chuck Anderson at CSU. The program consists of around 1000 lines of C. Profiling shows that the evaluation of transfer functions in the neural network is a performance bottleneck that consumes approximately 47% of the execution time. Two commonly used transfer functions are logistics $f = 1.0/(1.0 + e^x)$, and hyperbolic tangent $f = tanh(x)$, both of which call elementary functions as shown in Figure 7.20.

Mesa analysis of the neural network code is trivial for the hyperbolic tangent case, since the only candidate expression is an individual elementary function with a single argument. The logistics function is slightly more interesting because of the associated math as shown in Figure 7.21. Because of the simplicity of the transfer functions, Mesa 2.0 achieves the same

```
#pragma LUTOPTIMIZE
// logistic function
float logistic(float x) {
  float fReturn = (1.0 / (1.0 + exp(-x)));
  return fReturn;
}
#pragma LUTOPTIMIZE
// hyperbolic tangent
float logistic(float x) {
  float fReturn = (tanh(x));
  return fReturn;
}
```

Figure 7.20: CSU neural network transfer functions.

```
Mesa 2.0: Optimization started.
S1016: Train.cpp (line 1016) fReturn =((1.0 /(1.0 + exp(((-x))))))
Number of Statements = 1
S1016: X0 = exp(x)
S1016: X1 = (1.00000000e+00/(1.00000000e+00+exp(x)))
Number of Expressions = 2
[X0,X1]
Number of Constraints = 1
Mesa 2.0: Solving optimization problem
Optimizing for cache size 4194304
2 optimizations
X0 Di =    8.13 Mi =   29.14  Ei = 1.492e+04 Bi = 3.419e+09 Si = 4194304
X1 Di =    8.13 Mi =    0.25  Ei = 1.280e+02 Bi = 4.752e+09 Si = 4194304
Mesa 2.0: rank culling is disabled
4 solutions (possible)
3 solutions (actual)
2 solutions (pareto optimal)
Solution   Error      Benefit    Optimizations
C0         0.000e+00  0.000e+00
C1         1.280e+02  4.752e+09  X1
Select solution: 1
X1 Di = 8.13 Mi = 0.25 Ei = 1.280e+02 Bi = 4.752e+09 Si = 4194304 (4096KB)
Mesa 2.0: Generating optimized code
```

Figure 7.21: CSU neural network, logistics function.

Table 7.10: Mesa results on neural network code
(Intel Core 2 Duo E8300, 2.83GHz, 6MB L2 cache, single core)

Transfer Function	Original Time	Optimized Time	Performance Speedup	Maximum Error
Logistics	8.0s	3.6s	2.2×	$8.7X10^{-2}\%$
Hyperbolic Tangent	10.9s	3.9s	2.8×	$6.3X10^{-1}\%$

results on the hyperbolic function, as reported with Mesa 1.1 [77]. The logistics function gains a slight advantage with Mesa 2.0 because of the additional math that it incorporates. The results for the neural network are shown in Table 7.10.

7.6 Summary and Evaluation

The first criterion we use to evaluate our methodology is the performance speedup and accuracy achieved by the optimized applications. These are measured quantitatively through a comparison with the original program. The second criterion is programming effort, which we define qualitatively as the effort needed to apply one or more LUT transforms with Mesa. We gauge programmer effort in terms of the ease of use of the tool as compared to the manual process. Our case studies demonstrate that LUT optimizations are effective on the applications shown, because they show significant performance improvements while meeting the accuracy requirements of the application. Table 7.1 show performance speedups ranging from $2.3\times$ to $6.8\times$, while degrading accuracy by at most a fraction of a percent.

Mesa reduces the programming effort significantly in several ways. First, our tool automates source source code analysis, thereby avoiding the need for the programmer to search for candidate expressions. Second, the programmer is presented with error and performance estimates without having to instrument the code manually. Third, our tool automatically generates and integrates LUT code, freeing the programmer from coding. Fourth, our tool computes effective cache allocations for LUT data, using a method that would be very cumbersome if not impossible by hand. Finally, the tool simplifies experimentation with different solutions. We commonly call Mesa from a script to repeatedly optimize, compile, and run a program until we are satisfied with the outcome. For example, we scripted the generation, compilation, and execution of all 64 optimal and suboptimal solutions for the solar radiation code, and were able to get results in under an hour.

Mesa has improved our own process for tuning applications whose performance is bound by elementary functions. Our original *ad hoc* LUT implementation for the SAXS discrete code required several weeks of development time and experimentation, even after the base algorithm was implemented and tested. Characterization of error and performance was especially time-consuming, because it required multiple runs of the entire SAXS application. In addition, we simply had no way to estimate the performance or error impact of our LUT transforms. In contrast, the SAXS continuous code was never optimized manually because Mesa has been available during its development period. We can now revisit the optimization of the SAXS discrete and continuous code, or any other application, in a matter of minutes.

Finally, Mesa error analysis allows us to quickly identify efficient set of LUT parameters while constraining the LUT data to reside in mid-level cache. We find that the Mesa code very closely matches the performance of the manually developed SAXS code. New applications are equally easy to optimize with Mesa, requiring at most the insertion of a few pragmas. Currently the most time consuming aspect of using Mesa is domain profiling, but even this is aided by the automatic generation of instrumented code.

Chapter 8

Limitations and Threats to Validity

In this chapter we evaluate the limitations of our methodology and tool and the threats to validity in the empirical research presented in this thesis.

8.1 Limitations

Several of the limitations of our methodology are inherent to LUT transformation, automated or otherwise. The first of these is that LUT transformation only benefits programs that are performance-limited by elementary functions or other expensive computations. Despite frequent calls to elementary functions, some applications are still memory-limited because of the extensive data sets involved. When memory is the bottleneck, computational cost can often be ignored because of overlap with memory operations. For such applications, the additional memory allocated for LUT data can actually reduce performance. However, performance profiling tools usually make it easy to determine whether performance bottlenecks are computational or memory related. Programs that are not clearly limited by either computation or memory access can be evaluated as potential candidates for LUT optimization by applying tools based on hardware performance counters, such as PAPI [54]. These tools report the cost of computation and memory overhead with high precision. The *Roofline* performance model also offers insight into the balance between computation and memory access in an application [78].

A second limitation of LUT transformation is that decreased accuracy may violate the accuracy requirements of some applications. Many scientific programs require at least double-precision, and some codes are even less tolerant of error, requiring quad-precision and beyond [81]. The approximation of functions with LUT transformation can often reduce accuracy to below that of single-precision. In contrast, applications such as image processing and multimedia are known to have reduced requirements for precision, and can therefore achieve a significant benefit from LUT transformation without compromising results. Numerous precedents for reducing precision to gain performance exist, as discussed in Section 3.1.

The third limitation of applying LUT transformation on current systems is that LUT data must share mid-level cache memory with the application to avoid cache penalties. Some applications may depend on complete ownership of cache memory, in which case using cache resources for LUT data could reduce performance. However, the applications that we have studied share the mid-level cache with LUT data without serious performance degradation as shown by our case studies. Mesa allows the programmer to specify the amount of memory used for LUT transformation. Thus the programmer can explicitly share cache memory between the LUT data and the application.

A couple of additional limitations are specific to our methodology. In particular, it can be difficult to model application error. Despite being able to quantify many aspects of the error introduced by a LUT transformation, a general method to compute the effect on application accuracy remains an open problem. The propagation of error through arbitrary sections of application code poses a complex numerical analysis problem that is unique to each application. This implies that some level of experimentation will still be necessary to evaluate the effect of a LUT transformation on accuracy.

One limitation of the current Mesa tool is that it parses only C and C++ code, and the generated code must be compiled by a C++ compiler. Mesa also only handles a single source module. Beyond that, there are many syntactic elements that are not handled, including type casts, structure and pointer access, and *const* variables, and the tool handles only the elementary functions listed in Section 6. In addition, Mesa optimizes assignment expressions, including operators in the set {=, +=, -=, *=, /=}, and initializers, but will not find computation in other constructs. Mesa does not analyze or generate LUT transformations with multiple variables. Cache size detection is not automatic, and the total number of expressions is limited to 64. Finally, error analysis for linear interpolation is limited to the boundary method, and the analytic method works only for elementary functions, not arbitrary expressions. These are limitations of the tool, not the methodology.

8.2 Threats to Validity

This section is organized according to the four types of validity cited by Wohlin et al. that must be examined in empirical research: conclusion validity, internal validity, construct validity, and external validity [79].

Conclusion validity describes the degree to which conclusions made about the relationship between variables is justified. Our study exhibits conclusion validity because we clearly demonstrate the relationship between LUT optimization and program behavior. More specifically, we show (1) that programs with LUT optimization are faster than those without, (2) that programs with LUT optimization are less accurate than those without, and (3) that automated LUT optimization takes less programmer effort and produces more accurate results than manual methods.

Internal validity is achieved when an empirical study shows a cause and effect relationship between the treatment and external variables. One threat to internal validity is to have no explanation for why the treatment would produce the results shown. Another threat is the failure to control independent and dependent variables. In our research, we present a clear rationale for the causal relationship between LUT optimization and the dependent variables: performance, accuracy, and programming effort. In addition, the new and modified code introduced into a program by our methodology is the only change to the program, thus the changes in the dependent variables must be caused by our LUT optimization.

Construct validity is achieved when the treatment accurately represents the concepts that we intend to study, and when the measurements of dependent variable are valid. Our automated tool performs LUT optimization in a manner consistent with the LUT techniques described in the literature. In addition, the measurement of performance as execution time, and accuracy as the absolute or relative error from the original results, are consistent with the extensive body of research on program performance. In a similar manner, using programming time as a proxy for programming effort is consistent with current practices in the area of productivity measurement.

External validity addresses whether the results of the study will generalize beyond the scope of the study data. Empirical research is always limited with respect to the number and scope of the applications that can be evaluated. Our empirical evaluation consists of case studies of six applications in four scientific areas, of which two were partially written by the authors. Further research is required to demonstrate applicability to other domains. However, we expect that our results will generalize to applications that have the same limitations on performance caused by elementary functions, assuming that other environmental factors (compilers, languages, hardware) are consistent.

Chapter 9

Conclusions and Future Work

In this chapter we list our conclusions and describe future work in the area of LUT optimization.

9.1 Conclusions

The contributions of this thesis are (1) a comprehensive methodology for LUT transformation, (2) an investigation of error analysis and performance modeling methods, (3) a novel approach to LUT optimization, and (4) a software tool that automates many aspects of LUT optimization. We describe our methodology and tool that substantially automate the application of LUT optimization to scientific programs. We find that our methodology is effective at speeding up code that is performance limited by elementary functions. Our case studies demonstrate speedups from 1.4× to 6.8× with reasonable accuracy. We further demonstrate that automation improves programmer productivity by reducing the effort required to identify and implement sets of LUT transforms, and by providing information that helps the programmer make the critical tradeoff between error and performance.

Our research extends the literature on software LUT optimization. More specifically, we present a variety of techniques for fast and accurate error estimation, and we introduce a performance model that predicts the benefit of a LUT transform. Our work investigates some of practical aspects of LUT implementation, including domain reduction, partial domains, sampling methods, and the parallel performance of LUT code. Furthermore, we present a novel approach to LUT optimization that performs analysis of expressions over the scope of one or more functions. The Mesa tool provides an alternative to *ad hoc* practices that require significant programmer effort. Mesa automates the time-consuming and error-prone tasks associated with LUT methods. We find that the tool makes it easy to evaluate whether

LUT tuning is beneficial for a given application, and it helps us to identify sets of expressions that can benefit from LUT transformation.

9.2 Future Work

Continued work on making the error model more accurate is the primary area for future work. Successful prediction of application error would improve the ability of Mesa to correctly choose the most effective set of LUT transformations. One approach would be to selectively introduce error for each individual expression, then measure the resulting application error, thereby quantifying the effect of each transformation on application accuracy. The results could be incorporated into the error model to improve its estimation capability.

Applications often have performance bottlenecks that are encountered one after another, but Mesa assumes that the cache memory allocation persists throughout program execution. We could extend our work by taking into account the temporal aspect of cache allocation. For example, we could successively allocate and initialize LUT data for different functions over time, thereby providing more benefit over the lifetime of the program.

Another area for future work is the characterization of performance on multi-core systems with shared caches. As shown by our work, the performance of LUT transformations depends the availability of mid-level cache memory for LUT data. The evaluation of performance on a single-core system is straightforward, and we see a clear degradation of performance when LUT exceeds the cache size. Multi-core architectures are more complex, making the evaluation more difficult. Current systems generally have dedicated L2 and shared L3 caches, and recent systems now incorporate L4 cache. The performance characteristics of such systems is made more complicated by cache sharing between cores.

Additional work could focus on research questions that have not been addressed in the current literature. For example, we are not aware of a study that compares memoization to LUT transformation. While similar in many aspects, these approaches differ in important ways. The goal would be to characterize the conditions under which each technique is more effective. The use of polynomial reconstruction in software LUT transformation is another area that needs research. The hardware literature has made an extensive study of this, but the usefulness of the technique in software is unknown.

REFERENCES

[1] Mohammad J. Akhbarizadeh and Mehrdad Nourani. Hardware-based IP routing using partitioned lookup table. *IEEE/ACM Transactions on Networking*, 13(4):769–781, 2005.

[2] Carlos Alvarez, J. Corbal, and Mateo Valero. Fuzzy Memoization for Floating-Point Multimedia Applications. *IEEE Transactions on Computers*, 54(7):922–927, 2005.

[3] Gene M. Amdahl. Computer Architecture and Amdahl's Law. *Solid-State Circuits Newsletter*, 12(3):4–9, Summer 2007.

[4] AMPL: A Modeling Language for Mathematical Programming, 2011. `http://www.ampl.com/`.

[5] Charles W. Anderson, Saikumar V. Devulapalli, and Erik A. Stolz. Determining mental state from EEG signals using parallel implementations of neural networks. *Scientific Programming*, 4:171–183, Sep. 1995.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, CA, Dec. 2006.

[7] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[8] J. Barth, D. Plass, E. Nelson, C. Hwang, G. Fredeman, M. Sperling, A. Mathews, T. Kirihata, W.R. Reohr, K. Nair, and Nianzheng Caon. A 45 nm SOI Embedded DRAM Macro for the POWER Processor 32 MByte On-Chip L3 Cache. *IEEE Journal of Solid-State Circuits*, 46(1):64 –75, Jan. 2011.

[9] Bonmin Solver Project, 2012. `http://projects.coin-or.org/Bonmin`.

[10] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy. *ACM Transations on Mathematical Software*, 34:1–22, 2008.

[11] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. Software Development Environments for Scientific and Engineering Software: A Series of Case Studies. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 550–559, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Steven C. Chapra and Raymond P. Canale. *Numerical Methods for Engineers: With Programming and Software Applications.* McGraw-Hill, New York, NY, USA, 3rd edition, 1997.

[13] G. Cong, S. Seelam, I. Chung, H. Wen, and D. Klepacki. Towards a Framework for Automated Performance Tuning. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Daniel A. Connors and Wen-mei W. Hwu. Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 158–169, Washington, DC, USA, 1999. IEEE Computer Society.

[15] Couenne Solver Project, 2012. `http://projects.coin-or.org/Couenne`.

[16] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-memory Programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan. 1998.

[17] Catalina Danis, John Thomas, John Richards, Jonathan Brezin, Cal Swart, Christine Halverson, Rachel Bellamy, and Peter Malkin. *Towards Applying Complexity Metrics to Measure Programmer Productivity in High Performance Computing.* SE-CSE '08. ACM, New York, NY, USA, 2008.

[18] D. Defour. Cache-optimised methods for the evaluation of elementary functions. Technical Report 2002-38, Ecole normale superieure de Lyon, Lyon, France, 2002.

[19] L. Deng, C. Chakrabarti, N. Pitsianis, and X. Sun. Automated Optimization of Look-up table Implementation for Function Evaluation on FPGAs. In *Proceedings of SPIE*, volume 7444, 2009.

[20] Jeremie Detrey, Florent de Dinechin, and Xavier Pujol. Return of the Hardware Floating-point Elementary Function. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 161–168, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Yonghua Ding and Zhiyuan Li. A Compiler Scheme for Reusing Intermediate Computation Results. In *Proceedings of the International Symposium on Code Generation and Optimization*, page 279, Washington, DC, USA, 2004. IEEE Computer Society.

[22] James Epperson. *An Introduction to Numerical Methods and Analysis.* John Wiley & Sons, New York, NY, USA, 2007.

[23] Stuart Faulk, Adam Porter, John Gustafson, Walter Tichy, Philip Johnson, and Lawrence Votta. Measuring HPC productivity. *International Journal of High Performance Computing Applications*, 2004:459–473, 2004.

[24] Robert J. Francis. A tutorial on logic synthesis for lookup-table based FPGAs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 40–47, Los Alamitos, CA, USA, 1992. IEEE Computer Society.

[25] Shmuel Gal. Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance. In *Proceedings of the Symposium on Accurate Scientific Computations*, pages 1–16, London, UK, 1986. Springer-Verlag.

[26] O. Glatter and O. Kratky, editors. *Small angle x-ray scattering.* Academic Press, London, UK, 1982.

[27] David Goldberg. What every Computer Scientist should know about Floating-point Arithmetic. *ACM Computing Surveys*, 23:5–48, Mar. 1991.

[28] Brian Gough. *An introduction to GCC for the GNU compilers gcc and g++.* Network Theory Ltd., Bristol, IK, 2004.

[29] S. Graham and M. Snir. The NRC Report on the Future of Supercomputing. *CTWatch Quarterly*, 1, Feb. 2005.

[30] Marty Hall and McNamee Paul. Improving Software Performance with Automated Memoization. *The Johns Hopkins APL Technical Digest*, 18:254–260, 1997.

[31] Mohit Haran, James A. Catherwood, and Paulette Clancy. Diffusion of Group V Dopants in Silicon-Germanium Alloys. *Applied Physics Letters*, 88(17):173502, Apr. 2006.

[32] David Harris. An Exponentiation Unit for an OpenGL Lighting Engine. *IEEE Transactions on Computers*, 53(3):251–258, 2004.

[33] Thomas H.Cormen, Charles E. Leiserson, Ronald L.Rivest, and Clifford Stein. *Introduction to Algorithms (Second Edition).* The MIT Press, Cambridge, MA, USA, 2001.

[34] John L. Hennessy and David A. Patterson. *Computer Architecture.* Morgan Kaufmann, Waltham, MA, USA, 2007.

[35] Lorin Hochstein, Taiga Nakamura, Victor R. Basili, Sima Asgari, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, Forrest Shull, Jeffrey Carver, Martin Voelp, Nico Zazworka, and Philip Johnson. Experiments to Understand HPC Time to Development. *CTWatch Quarterly*, 2:24–32, Nov. 2006.

[36] R. Hyde. *The Art of Assembly Language.* No Starch Press, San Francisco, CA, USA, 2003.

[37] Intel. *IA-32 Intel Architecture Optimization Reference Manual.* Beaverton, OR, USA, 2006.

[38] Intel. *Intel Math Kernel Library Reference Manual.* Beaverton, OR, USA, 2011.

[39] Douglas L. Jones. Efficient FFT Algorithm and Programming Tricks. *Connexions*, 2007. `http://cnx.org/content/m12021/1.6/`.

[40] K. V. Seshu Kumar. Value Reuse Optimization: reuse of Evaluated Math Library Function Calls through Compiler Generated Cache. *SIGPLAN Notices*, 38:60–66, Aug. 2003.

[41] Dong-U Lee, Altaf Abdul Gaffar, Oskar Mencer, and Wayne Luk. Optimizing Hardware Function Evaluation. *IEEE Trans. Comput.*, 54:1520–1531, Dec. 2005.

[42] Chunhua Liao, Daniel J. Quinlan, Thomas Panas, and Bronis R. de Supinski. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In *IWOMP*, pages 15–28, 2010.

[43] Michael D. Linderman, Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan. Towards program optimization through automated analysis of numerical precision. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 230–237, New York, NY, USA, 2010. ACM.

[44] Eugene Loh, Michael L. Van De Vanter, and Lawrence G. Votta. Can Software Engineering Solve the HPCS Problem? In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, pages 27–31, New York, NY, USA, 2005. ACM.

[45] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, New York, NY, USA, 1990.

[46] Mathematical Acceleration Subsystem (MASS), 2012. `http://www-01.ibm.com/software/awdtools/mass`.

[47] Matlab Lookup Tables, 2010. `http://www.mathworks.com/help/toolbox/simulink/ug/bqiqpa3.html`.

[48] MESA Project, 2012. `http://www.cs.colostate.edu/hpc/MESA`.

[49] MINLP Solver Project, 2012. `//www.neos-server.org/neos/solvers/minco:MINLP/AMPL.html`.

[50] Michael Mishchenko, Brian Cairns, Greg Kopp, Carl Schueler, Bryan Fafaul, James Hansen, Ronald Hooker, Tom Itchkawich, Hal Maring, and Larry Travis. Accurate Monitoring of Terrestrial Aerosols and Total Solar Irradiance. 88(5):677–691, May 2007.

[51] Ramon E. Moore and Fritz Bierbaum. *Methods and Applications of Interval Analysis)*. Society for Industrial and Applied Math (SIAM), Philadelphia, PA, USA, 1979.

[52] NEOS Solverss, 2012. `http://www.neos-server.org/neos/solvers/index.html`.

[53] Neural Network Software, 2011.
`http://www.cs.colostate.edu/~anderson/meOther.html`.

[54] PAPI Project, 2012. `http://icl.cs.utk.edu/papi/index.html`.

[55] George Paul and M. Wayne Wilson. Should the Elementary Function Library Be Incorporated Into Computer Instruction Sets? *ACM Trans. Math. Softw.*, 2:132–142, Jun. 1976.

[56] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley Professional, Boston, MA, USA, 2005.

[57] J. A. Piñeiro, J. D. Bruguera, and J. M. Muller. Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree. In *ARITH '01: Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, Washington, DC, USA, 2001. IEEE Computer Society.

[58] PRMS Project, 2010. `http://water.usgs.gov/software/PRMS`.

[59] Rapid Radiative Transfer Model, 2010. `http://rtweb.aer.com/rrtm_frame.html`.

[60] John Riley. *Writing Fast Programs: A Practical Guide for Scientists and Engineers.* Cambridge International Science Publishing, Cambridge, UK, 2006.

[61] ROSE Project, 2011. `http://www.rosecompiler.org/`.

[62] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[63] S. Subramanya Sastry, Rastislav Bodik, and James E. Smith. Characterizing Coarse-Grained Reuse of Computation. In *3rd ACM Workshop on Feedback Directed and Dynamic Optimization*, pages 16–18, 2000.

[64] SAXS Project, 2010. `http://www.cs.colostate.edu/hpc/SAXS`.

[65] M.J. Schulte and Jr Swartzlander, E.E. Hardware designs for exactly rounded elementary functions. *IEEE Transactions on Computers*, 43(8):964–973, 1994.

[66] Scientific Python Interpolation, 2011.
`http://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html`.

[67] F. Shull, J. Carver, L. Hochstein, and V. Basili. Empirical study design in the area of high-performance computing (HPC). In *Proceedings of International Symposium on Empirical Software Engineering*, pages 305–314, Nov. 2005.

[68] Marc Snir, David A. Bader, James C. Browne, Brad Chamberlain, Peter Kogge, John Mccalpin, Rami Melhem, and David Padua. A framework for measuring supercomputer productivity. *The International Journal of High Performance Computing Applications*, 18(4):417–432, 2004.

[69] K. Sobti, L. Deng, C. Chakrabarti, N. Pitsianis, X. Sun, J. Kim, P. Mangalagiri, K. Irick, M. Kandemir, and V. Narayanan. Efficient Function Evaluations with Lookup Tables for Structured Matrix Operations. In *2007 IEEE Workshop on Signal Processing Systems*, Oct. 2007.

[70] S. Squires, M. Van De Vanter, and L. Votta. Software Productivity Research In High Performance Computing. *CTWatch Quarterly*, 2, Nov. 2006.

[71] F.H. Stillinger and T.A. Weber. Computer Simulation of Local Order in Condensed Phases of Silicon. *Physical Review B*, 31(8):5262–5271, 1985.

[72] Portable Hardware Locality Suite, 2011. `http://www.open-mpi.org/projects/hwloc`.

[73] SYMPHONY Solver Project, 2012. `http://projects.coin-or.org/SYMPHONY`.

[74] Ping-Tak Peter Tang. Table-driven Implementation of the Exponential Function in IEEE Floating-point Arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, 1989.

[75] Ping-Tak Peter Tang. Table-lookup Algorithms for Elementary Functions and their Error Analysis. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, 1991.

[76] C. Wilcox, M. Strout, and J. Bieman. Mesa: Automatic Generation of Lookup Table Optimizations. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, New York, NY, USA, 2011. ACM.

[77] C. Wilcox, M. Strout, and J. Bieman. Tool support for software lookup table optimization. *Scientific Programming*, 19(4):213–229, Dec. 2011.

[78] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65–76, Apr. 2009.

[79] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.

[80] W. F. Wong and E. Gogo. Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers. *IEEE Transactions on Computers*, 43(3):278–294, 1994.

[81] X.S. Li Y. Hida and D.H. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. Technical Report Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, Berkeley, CA, Oct. 2000.

[82] Marvin Zelkowitz, Victor Basili, Sima Asgari, Lorin Hochstein, Jeff Hollingsworth, and Taiga Nakamura. Measuring productivity on high performance computers. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, pages 6–, Washington, DC, USA, 2005. IEEE Computer Society.

[83] M Zhang, J.G. Delgado-Frias, and S. Vassiliadis. Table driven Newton scheme for high precision logarithm generation. In *Proceedings of the IEEE Computers and Digital Techniques*, volume 141, 1994.

[84] Yuanrui Zhang, Lanping Deng, P. Yedlapalli, S.P. Muralidhara, Hui Zhao, M. Kandemir, C. Chakrabarti, N. Pitsianis, and Xiaobai Sun. A Special-Purpose Compiler for Look-up Table and Code Generation for Function Evaluation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1130 –1135, Mar. 2010.