EXTRACTING DATA-LEVEL PARALLELISM FROM SEQUENTIAL PROGRAMS FOR SIMD EXECUTION

A Dissertation
Presented to
The Academic Faculty

By

Lewis Benton Baumstark, Jr.

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in
Electrical and Computer Engineering

Georgia Institute of Technology

December, 2004

Copyright © Lewis B. Baumstark, Jr. 2004
Extracting Data-Level Parallelism from Sequential Programs for SIMD Execution

Approved by:

Dr. Linda M. Wills, Advisor

Dr. Hsien-Hsin Lee

Dr. Sudhakar Yalamanchili

September 21, 2004
To my parents Susan and Lewis Sr. and my sister Amanda
ACKNOWLEDGEMENT

My success obtaining a PhD, and in general, life, has been approximately 5% luck, 5% work, and 90% the support (and prodding) of family, friends and colleagues. First and foremost, with all the love I have, thank you to Lewis, Sr. and Susan Baumstark, my beloved parents, and to Amanda, my dear sister. Everything I am and everything I have accomplished is because of you.

To Dr. Linda Wills, my advisor, my deepest thanks and gratitude. I would have been stumbling around in the dark (instead of stumbling around in the daylight) without your mentorship and guidance. Thank you also to my doctoral committee for your aid and advice.

Thank you to my friends and colleagues Murat Guler, Chris Lee, Cameron Craddock, Nidhi Shah, Peter Sassone, Dr. Sam Sander, Dr. Antonio Gentile, and Dr. Tarek Taha.

Thank you to my spiritual family at Northside Drive Baptist Church. Understand you have been the stable ground for me here, a place of refuge when all the stress of school threatened to take me out. I am forever grateful.

A special thank-you to Dr. Connie Hood, of Tennessee Technological University, my undergraduate mentor and friend, and one of many who inspired me to teach. (It’s largely your fault I did this…)

And last but not least, thank you Creator God, the spirit that binds us and calls us to unity with your Creation.
# Table of Contents

Acknowledgement iv  
List of Tables viii  
List of Figures ix  
List of Abbreviations xii  
Summary xiv  
Chapter 1 Introduction  
Exploiting Data-Level Parallelism in Multimedia Applications 1  
Reverse Engineering Approach to Data-Parallel Program Development 4  
Extending the Range of Data-Parallel Compilation for Multiple SIMD Targets 5  
Identification of Data-Parallel Code to Focus Retargeting Tasks 6  
Outline of Dissertation 7  
Chapter 2 Extracting an Explicitly Data-Parallel Representation of Image-Processing Programs 8  
Introduction 9  
Explicitly Parallel Target Representation 10  
Outline of Chapter 15  
Related Work 16  
Representation and Recognition 18  
Candidate Loops and Pre-processing 20  
Initial Representation of Source Program 21  
Explicitly Parallel Target Representation 23  
Patterns 24  
Recognition 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Generation</td>
<td>36</td>
</tr>
<tr>
<td>Experimental Validation and Results</td>
<td>40</td>
</tr>
<tr>
<td>Recognition Performance</td>
<td>43</td>
</tr>
<tr>
<td>Conclusions and Future Work</td>
<td>44</td>
</tr>
<tr>
<td>Chapter 3 Retargeting Sequential Code to Multimedia Instruction Set Architectures</td>
<td>45</td>
</tr>
<tr>
<td>Introduction</td>
<td>45</td>
</tr>
<tr>
<td>Misalignment of Mixed-Precision Packed Data Types</td>
<td>47</td>
</tr>
<tr>
<td>Outline of Chapter</td>
<td>49</td>
</tr>
<tr>
<td>Related Work</td>
<td>49</td>
</tr>
<tr>
<td>Vectorization for Multimedia ISAs</td>
<td>49</td>
</tr>
<tr>
<td>Approach</td>
<td>51</td>
</tr>
<tr>
<td>Extracting an Explicitly-Parallel Program Representation</td>
<td>52</td>
</tr>
<tr>
<td>Code Synthesis</td>
<td>61</td>
</tr>
<tr>
<td>Validation and Results</td>
<td>69</td>
</tr>
<tr>
<td>Retargeting of Image Processing Programs to SSE2</td>
<td>70</td>
</tr>
<tr>
<td>Correctness Validation and Performance Evaluation</td>
<td>71</td>
</tr>
<tr>
<td>Code Size Evaluation</td>
<td>74</td>
</tr>
<tr>
<td>Recognition Performance</td>
<td>75</td>
</tr>
<tr>
<td>Conclusions and Discussion</td>
<td>76</td>
</tr>
<tr>
<td>Chapter 4 Lightweight Estimation of Data-Parallel Potential</td>
<td>78</td>
</tr>
<tr>
<td>Introduction</td>
<td>78</td>
</tr>
<tr>
<td>Related Work</td>
<td>79</td>
</tr>
<tr>
<td>Program Performance Estimation</td>
<td>79</td>
</tr>
<tr>
<td>Parallelism Estimators</td>
<td>81</td>
</tr>
<tr>
<td>Methodology</td>
<td>83</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Experimental Results and Validation</td>
<td>92</td>
</tr>
<tr>
<td>Methodology Validation</td>
<td>93</td>
</tr>
<tr>
<td>Execution Performance</td>
<td>99</td>
</tr>
<tr>
<td>Scalability</td>
<td>102</td>
</tr>
<tr>
<td>Conclusions and Discussion</td>
<td>105</td>
</tr>
<tr>
<td>Chapter 5 Conclusion</td>
<td>106</td>
</tr>
<tr>
<td>Extracting an Explicitly Data-Parallel Representation of Image-Processing Programs</td>
<td>106</td>
</tr>
<tr>
<td>Retargeting Data-Parallel Code To Multimedia Instruction Set Architectures</td>
<td>107</td>
</tr>
<tr>
<td>Lightweight Estimation of Data-Parallel Potential</td>
<td>108</td>
</tr>
<tr>
<td>Knowledge Transfers</td>
<td>109</td>
</tr>
<tr>
<td>Program Modeling and Visualization</td>
<td>109</td>
</tr>
<tr>
<td>Visual Programming Languages</td>
<td>110</td>
</tr>
<tr>
<td>Partial Program Recognition</td>
<td>110</td>
</tr>
<tr>
<td>Appendix A SIMD Architectures</td>
<td>111</td>
</tr>
<tr>
<td>Appendix B Reference Graph-Grammar</td>
<td>117</td>
</tr>
<tr>
<td>Appendix C Sobel Source Parallelized by PARRET for SSE2</td>
<td>129</td>
</tr>
<tr>
<td>References</td>
<td>133</td>
</tr>
<tr>
<td>Vita</td>
<td>139</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

Table 1. Node propagation rules 31
Table 2. Benchmark suite 41
Table 3. Recognition times for SIMPil retargeting. 44
Table 4. Available SSE2 instructions. 62
Table 5. Template SSE2 code-generation script. 67
Table 6. IMGLIB test programs. 70
Table 7. Suite coverage test. 71
Table 8. Correctness and Performance Results, PARRET vs. Sequential Execution. 72
Table 9. Performance comparison of PARRET with vectorized benchmarks. 74
Table 10. Code Increase for PARRET-retargeted programs. 75
Table 11. Recognition times for SSE2 retargeting. 76
Table 12. DLPEST results vs. expected results for IMGLIB 94
Table 13. DLPEST Coverage Results. 98
LIST OF FIGURES

Figure 1. 3×3 Convolution source code. 11
Figure 2. 3×3 Convolution as an MDDF model. 13
Figure 3. Recognition process. 19
Figure 4. Sample initial program representation. 22
Figure 5. MDSDF example. 23
Figure 6. (a) Basic count pattern, (b) count plus an expression, (c) count times an expression. 26
Figure 7. Example pointer-based array-read patterns. 27
Figure 8. Array-based load and store patterns: (a) singly-subscripted (b) doubly-subscripted. 28
Figure 9. Quantization source code fragment. 29
Figure 10. Quantization example: (a) original DFG representation (b) recognition of count patterns. 32
Figure 11. Recognition of parallel pointer-based load and store patterns. 33
Figure 12. Merging of transformed inner loop DFG and the outer loop DFG. 34
Figure 13. (a) Recognition of count patterns, (b) recognition of parallel pointer-based load patterns. 35
Figure 14. MDDF quantization algorithm model. 36
Figure 15. Data-parallel ZPL code for the quantization algorithm. 39
Figure 16. Program results comparison, SIMPil vs. sequential 42
Figure 17. Execution speedup of retargeted SIMPil vs. sequential 43
Figure 18. Retargeting process. 52
Figure 19. Sample image-processing algorithm with a MDSDF-like representation. 54
Figure 20. Sobel example source code. 55
Figure 21. Intermediate DFG representation. 56
Figure 22. Pattern examples: (a) Count pattern, (b) array-read pattern based on a count pattern and LOAD operation.

Figure 23. Recognition of a count pattern.

Figure 24. Recognition of array-read and array-write patterns for finalized MDDF representation.

Figure 25. Demotion of an unsigned 32-bit vector type to the lower-half of an unsigned 16-bit vector type.

Figure 26. Estimation overview.

Figure 27. Example of dynamically measuring dependence distance.

Figure 28. Example loop nesting analysis. (a) Source code, (b) inter-procedural loop nesting relationships.

Figure 29. Estimate correlation (TI IMGLIB) between DLPEST and the Simulation-based method

Figure 30. Estimation times comparison.

Figure 31. DLPEST vs. simulator speedup comparison.

Figure 32. DLPEST timing decomposition.

Figure 33. Scaling of DLPEST performance with program size.

Figure 34. Data-parallel addition.

Figure 35. SIMPil NEWS communication network.

Figure 36. Example PSHUFW permutation instruction (SSE).

Figure 37. Graph-grammar graph elements key.

Figure 38. Production rules for constant expression patterns.

Figure 39. Production rules for count patterns, part 1.

Figure 40. Production rules for count patterns, part 2.

Figure 41. Production rule for the pointer-count pattern.

Figure 42. Production rules for array-read patterns, part 1.

Figure 43. Production rules for array-read patterns, part 2.
Figure 44. Production rules for array-write patterns, part 1. 126
Figure 45. Production rules for array-write patterns, part 2. 127
Figure 46. Production rule for an array-sum pattern. 128
LIST OF ABBREVIATIONS

ALU: Arithmetic Logic Unit
AST: Abstract Syntax Tree
CDFG: Control Data Flow Graph
CFG: Control Flow Graph
DLP: Data-Level Parallelism
DLPEST: Data-Level Parallelism Estimator
DFG: Data Flow Graph
DSP: Digital Signal Processor
ICL: Intel C/C++ Compiler
IMGLIB: Image Library (program suite provided by Texas Instruments)
IR: Intermediate Representation
ISA: Instruction Set Architecture
MDDF: Multi-Dimensional Data-Flow
MDSDF: Multi-Dimensional Synchronous Data-Flow
MIMD: Multiple-Input, Multiple-Data
MMX: Multimedia extensions
MOC: Model Of Computation
PARRET: Parallel Retargeter
PBM: Plan-Building Method
PE: Processing Element
PPE: Pixels per Processing Element
RAW: Read-After-Write
SDF: Synchronous Data-Flow

SIMD: Single-Instruction, Multiple-Data

SLOC: Source Lines Of Code

SSE, SSE2: Streaming SIMD Extensions (version 1 and version 2, respectively)

SUIF: Stanford University Intermediate Format

TI: Texas Instruments

VIS: Visual Instruction Set

WAR: Write-After-Read

WAW: Write-After-Write
SUMMARY

A method for retargeting multimedia programs written in sequential languages to architectures with data-parallel execution models is presented. We developed a high-level multi-dimensional dataflow (MDDF) representation that explicitly models data-level parallelism in programs and a recognition method, called PARRET, and pattern library for extracting this representation from sequential programs. Unlike earlier work using program recognition for parallelization, PARRET uses a partial recognition approach to identify only those aspects of the program that hinder parallelization, instead of attempting full recognition of the program as a single entity. PARRET is flexible with respect to multiple data-parallel targets (multi-processor SIMD architectures and multimedia ISA extensions with SIMD functionality) and program variability (e.g., multiple loop nests, conditional statements and mixed data types within a loop).

PARRET was used to retarget a set of test programs from the Texas Instruments IMGLIB suite to SIMPil, a representative multi-processor SIMD architecture, and Intel’s SSE2, a representative multimedia instruction set architecture. The correctness of the retargeted programs is validated by comparing with those of a baseline sequential execution. The performance gains from data-parallel execution are measured. Results of the above indicate PARRET is able to retarget the programs with minimal loss of computational precision in the transformed program and that the transformed programs perform significantly better – 60 to 30,000 faster than sequential for SIMPil targets and 2 to 27 times faster for the SSE2 target.
A method of estimating data-level parallelism, called DLPEST, in sequential programs is also presented, to support the overall retargeting process by focusing recognition on those parts of a program with a large potential for speedup from data-parallel execution. A key requirement is that DLPEST be lightweight in terms of time cost. Our approach is to measure data-level parallelism within an inner loop by measuring its loop-dependence distance using run-time profiling, combined with a static analysis of loop nesting to present the parallelism in a larger program context. A comparison of DLPEST’s results with expected results from the Texas Instruments IMGLIB suite is presented, validating DLPEST accurately measures loop-dependence distance and loop nesting. A comparison of DLPEST with a representative simulation-based method is presented. DLPEST’s results are shown to have a statistical correlation of 0.4 with the representative simulation-based method and to produce those results on average 334 times faster. It is also shown to scale sub-linearly, in terms of its measurement time, with input program size.
CHAPTER 1

INTRODUCTION

Exploiting Data-Level Parallelism in Multimedia Applications

Mainstream multimedia applications, such as video and audio compression, image object recognition, and 3D video games, are demanding in terms of processor performance. For example, an MPEG video playback of 30 frames-per-second at 320×240 pixels per frame requires that over 2.3 million pixels be decoded from the compressed video stream every second. At the same time, these applications often contain a high potential for data-level parallelism (DLP), which can be exploited to address the computational demand. In data-parallel execution, the same operation is applied, independently, to all the elements in a large data set, where the data set might be pixel data in an image or pulse-code modulation sample data in an audio stream.

To address the increased demand for data-parallel processing, processor architectures employing single-instruction, multiple-data (SIMD) functionality have been developed. Recently, a number of processor architectures have emerged which have DLP processing capabilities. These include the multimedia instruction set extensions, such as Intel’s MMX [1], SSE [2], and SSE2 [3][4] AMD’s 3DNow! [5], Motorola’s Altivec [6], and Sun Corporation’s Visual Instruction Set (VIS) [7], and SIMD processor arrays such as Xetal [8], IMAP [9][10], P3I [11], and SIMPil [12][13][14]. The SIMD processing model they employ is well-suited to exploiting DLP, for example speedups of 38,000 times have been estimated for SIMPil [14], while speedups of 36 have been demonstrated for Altivec [15].
The availability of hardware capable of exploiting DLP has created a need for software support for developing applications on, and migrating them to, these architectures. The main challenge in software support for data-parallel processing is that real-world programs are written without explicit specification of the DLP inherent in the algorithms. Mainstream programming languages, such as C, C++, Fortran, and Java, follow an explicitly sequential processing model which has no parallel semantics in the core language. Data-parallel algorithms are written as loops over linear arrays of data, instead of using data-parallel primitives that would facilitate compilation to a SIMD architecture. The particulars of these loop-based implementations obscure the data-parallel nature of the algorithm the program implements, making it difficult for a compiler or other retargeting tool to parallelize the program.

Current techniques for retargeting programs to data-parallel architectures present a trade-off in terms of their cost (e.g., development time) and the range of loops for which they are applicable. Vectorization [16][18][19][20] is the traditional method for parallelizing loops written in mainstream languages, but it can be limited in the types of loops it can handle. The alternate solution is to write programs in an explicitly parallel language, such as ZPL [21][22], or to manually re-write sequential loops using assembly instructions from a multimedia instruction set, such as SSE. This approach can cover a larger range of loops, but with increased training and/or coding time, resulting in higher costs, as well as (in the case of assembly-level programming) non-portability of the application to other architectures. A third solution is to use a program recognition technique, such as PAP [23][24][25] or PARAMAT [26][25], where algorithms are detected in source code and replaced by a parallel representation.
In the case of vectorization, parallelization is being attempted on a program that has no explicit parallel semantics; vectorization itself only examines the data dependencies of the loops as they are written, without attempting any deeper understanding of the program. In the case of programming with multimedia assembly instructions or data-parallel languages, the parallelism is stated explicitly, but with the cost and portability liabilities mentioned before. For previous recognition techniques, their ability to parallelize is limited by the number of algorithms contained in their pattern libraries.

This dissertation presents an approach that automatically extracts a high-level data-parallel representation, based on multi-dimensional dataflow, from programs written in mainstream sequential programming languages. This approach provides the benefits of automated retargeting provided by vectorization, but with a program representation of an explicitly-parallel high-level language. It is a program recognition approach, similar to PAP and PARAMAT. However, it relies on partial recognition – i.e., the program does not need to be recognized as a single algorithm – to provide greater program coverage. Using this representation we achieve both flexibility and generality:

- Hardware-independence allows for code generation on multiple hardware targets (e.g., MMX/SSE and SIMPil),
- Multi-dimensional dataflow program model exposes parallel array regions across which computational operations can be concurrently applied, enabling PARRET to retarget a wider range of loops than vectorization, and
• Partial-recognition approach of only recognizing data-parallel memory access patterns and operations allows PARRET to retarget a wider range of loops than earlier recognition techniques.

Reverse Engineering Approach to Data-Parallel Program Development

The approach taken is to extract a high-level, explicitly-parallel representation of the source program. This representation is based on an extension of the multi-dimensional synchronous dataflow program modeling language [27]. The representation is extracted by dataflow graph pattern-matching, i.e., the program source code is first transformed into a flowgraph representation and then data-parallel memory access patterns are recognized using a pattern library. The resulting program representation is then used to drive code-generation for SIMD processors.

Contribution 1: Extracting an Explicitly Data-Parallel Representation of Image-Processing Programs

This contribution presents an explicitly data-parallel program representation, called multi-dimensional dataflow (MDDF) and a recognition algorithm and data-parallel pattern library for extracting MDDF specifications from sequential source code. A suite of programs developed for the Texas Instruments ‘C6200 line of digital signal processors (DSPs) is retargeted to the SIMPil architecture and simulated using the SIMPil simulator. The system is validated by comparing the results of simulated retargeted programs with those produced by a baseline sequential execution. Finally, performance of the retargeted programs is compared to that of the baseline sequential execution, showing speedup from retargeting of 60 to 30,000 times on SIMPil.
Extending the Range of Data-Parallel Compilation for Multiple SIMD Targets

PARRET’s MDDF representation is architecture-independent, giving it the flexibility to target multiple architecture types. Contribution 1 demonstrates its applicability to multi-processor SIMD targets; we further demonstrate PARRET’s multi-target ability by retargeting to the sub-word parallel instructions found in multimedia instruction set extensions.

Compiling for these instruction set extensions is constrained by the unit stride array accesses required for packed vector data. For example, vector data types of mixed precision within a loop body create non-unit stride with respect to each other, complicating the parallelization process as corresponding array elements are unaligned in the packed vectors. PARRET’s high-level multi-dimensional representation facilitates properly aligning these mixed vector data types by exposing the appropriate place in the algorithm to generate data type promotion or demotion code.

Contribution 2: Retargeting Data-Parallel Code to Multimedia Instruction Set Architectures

This research contribution applies the PARRET data-parallel retargeting approach in the context of multimedia ISAs. Image-processing programs from the Texas Instruments IMGLIB suite for its ‘C6200 line of DSPs are retargeted to Intel’s Pentium 4 using Streaming SIMD Extensions 2 (SSE2) primitives. Speedups of two to eight times over baseline sequential execution are shown for the retargeted loop kernels. A comparison with a commercial vectorizing compiler, the Intel C/C++ compiler, is made, demonstrating PARRET’s ability to parallelize a wider range of loop-based applications than traditional vectorization.
Identification of Data-Parallel Code to Focus Retargeting Tasks

Because retargeting a program for parallel execution can be time-consuming and expensive, it is essential to be able to estimate the potential benefit of parallelization beforehand. An automated recognition system can improve the efficiency of the reverse engineering process, but the process can be further streamlined by selective application of recognition to programs or code blocks with high potential for data parallelism. Low-cost techniques for eliminating non-data-parallel code prior to a retargeting activity will facilitate the overall retargeting process, by focusing a parallelization tool (whether based on program recognition or vectorization) only on those code fragments with a high expectation of being data-parallel. These techniques will not only aid the reverse engineering process, but could also be used to focus more traditional vectorization-related analyses, such as Fourier-Motzkin elimination or the Omega test [28].

An approach for estimating data-level parallelism is presented for selection of likely loop candidates for retargeting. Particular to this technique is that it is computationally lightweight. The technique takes a hybrid approach: 1) dependence data is collected by automatically annotating a program with instrumentation code and executing the annotated program and 2) static loop nesting information is collected and combined with the dependence data to produce a parallelism estimate.

In addition to its applicability to the retargeting process, a lightweight technique for data-parallel estimation has benefits in other areas, such as hardware/software co-design. For example, an estimate of program parallelism can be used as part of the architecture selection process in system design, similar to that proposed in [29].
Contribution 3: Lightweight Estimation Of Data-Parallel Potential

This research contribution presents a technique for estimating the amount of data-level parallelism available in loop-based algorithms. The technique is tested on a suite of programs from the TI IMGLIB suite and the MediaBench suite. Correctness is demonstrated by comparing with known expected results. Timing measurements are performed, with the estimation of most benchmarks completing in under ten minutes, and all completing within an hour. Finally, it is shown to have sub-linear scalability in terms of program size.

Outline of Dissertation

Chapter 2 presents the explicitly-parallel program representation, called MDDF, and a recognition technique, implemented in a system called PARRET, for extracting MDDF from source code. Test programs are retargeted to SIMPil [14], a representative multi-processor SIMD architecture. Chapter 3 further demonstrates PARRET’s flexibility in terms of architectural targets by retargeting a test suite of programs to Intel’s SSE2 [3][4], a representative multimedia instruction set extension with SIMD functionality. The particular challenges of parallelization using these instructions and how PARRET and MDDF address these challenges is discussed. Chapter 4 presents a lightweight system for estimating the data-level parallelism in source code. This estimate is used to drive selection of loops as candidates for retargeting. Conclusions and future work are discussed in Chapter 5.
CHAPTER 2

EXTRACTING AN EXPLICITLY DATA-PARALLEL REPRESENTATION OF IMAGE-PROCESSING PROGRAMS

Abstract

The goal of this research is to retarget multimedia programs written in sequential languages (e.g., C) to architectures with data-parallel execution capabilities. Image processing algorithms often have a high potential for data-level parallelism, but the artifacts imposed by the sequential programming language (e.g., loops, pointer variables, linear address spaces) can obscure the parallelism and prohibit generation of efficient parallel code. This research presents a program representation and recognition approach for generating a data-parallel program specification from sequential source code and retargeting it to data-parallel execution mechanisms. The representation is based on an extension of the multi-dimensional synchronous dataflow model of computation. A partial recognition approach identifies and transforms only those program elements that hinder parallelization while leaving other computational elements intact. This permits flexibility in the types of programs that can be retargeted, while avoiding the complexity of complete program recognition.

Central to extracting an explicitly parallel representation from code is uncovering the mapping between iterations and array variables in the source code and the operations over array regions (e.g., rows, columns, tiled blocks) that they implement. Examples are presented to illustrate this mapping. A set of patterns and the process for recognizing these regions is presented. Data-parallel code is generated and executed to
validate the correctness of the retargeted specifications. Speedups due to data-parallel execution of over 60 to over 30,000 times are shown.

Introduction

Image and video processing applications are pervasive, particularly in compact imaging devices, such as smart cameras, biometric security systems, wearable computers, and autonomous vehicle vision systems. These employ a wide range of algorithms, such as convolution, discrete cosine transform, wavelet decomposition, and motion estimation [30]. These algorithms have a high potential for data-level parallelism since they often map operations across all data elements (e.g., individual pixels or regions of pixels) in large two-dimensional image arrays. With the advent of data parallel multimedia extensions to instruction sets (e.g., MMX [1], SSE [2], AMD’s 3DNow! [5], Altivec [6], Sun’s VIS [7],) and new massively parallel, single-instruction, multiple-data (SIMD) architectures (such as SIMPil [14], the IMAP family [9][10], P3T [11], and Xetal [8]), the potential to efficiently exploit this parallelism is emerging.

However, most of these applications are written for traditional sequential or superscalar processors and for digital signal processors (DSPs). Usually they are written in a language like C, which has no mechanism for specifying explicit parallelism. A developer wishing to retarget legacy codes to data parallel execution mechanisms must either hand-optimize the existing code (often at the assembly code level) or else rewrite the data parallel portions in a high-level data parallel language, such as ZPL [21].

This chapter discusses an alternate solution: using automated program recognition techniques to extract a high-level, explicitly parallel representation [31] of the original
source and then to use that representation to drive code generation for data parallel execution. This chapter describes a system, called PARRET (for PARallel RETargeter), to retarget image and video processing applications written in sequential languages, such as C, for data-parallel execution. Sequential code often obscures the data-level parallelism with low-level syntactic and implementation details, for example, an operation on an array element within a loop body may not be readily recognizable as that operation mapped across all the elements of the array. PARRET transforms such low-level constructs to the high-level, explicitly data-parallel operations they implement. The results of this transformation then drive code generation for new and extended hardware platforms that support data parallel execution.

**Explicitly Parallel Target Representation**

Given sequential C source code, our goal is to automatically extract a high-level, explicitly parallel specification and re-implement the algorithm in a data-parallel language for execution on a SIMD processor or a sequential processor with SIMD extensions (e.g., SSE). Consider the source code of Figure 1, from the Eclipse image processing suite [33], which implements a 3×3 convolution routine. Each pixel in the output image is a weighted sum of each of its eight nearest neighbors and itself. This is a common filtering operation that can be used to perform, for example, edge detection or noise removal, depending on the filter weights used.
/* 3x3 convolution from the ESO Eclipse image processing suite. Has minor changes to replace structure references in the loop bodies to scalar variables */

01: image_t * image_filter3x3(image_t *image_in, double *filter) {
02:     image_t *image_out;
03:     int i, j, curr_pos, x, y;
04:     double sum_pix, filter_norm;
05:     pixelvalue *in_data, *out_data;
06:     filter_norm = 0.0;
07:     for (i=0; i<9; i++)
08:         filter_norm += filter[i];
09:     x = image_in->lx; y = image_in->ly;
10:     in_data = image_in->data; out_data = image_out->data;
11:     for (j=1; j<y-1; j++) {
12:         for (i=1; i<x-1; i++) {
13:             curr_pos = i + j*x; sum_pix = 0.0;
14:             sum_pix += filter[0] *
15:                  (double)in_data[curr_pos-1-x] ;
16:             sum_pix += filter[1] *
17:                  (double)in_data[curr_pos-x] ;
18:             sum_pix += filter[2] *
19:                  (double)in_data[curr_pos+1-x] ;
20:             sum_pix += filter[3] *
21:                  (double)in_data[curr_pos-1] ;
23:                  (double)in_data[curr_pos] ;
24:             sum_pix += filter[5] *
25:                  (double)in_data[curr_pos+1] ;
26:             sum_pix += filter[6] *
27:                  (double)in_data[curr_pos-1+x] ;
28:             sum_pix += filter[7] *
29:                  (double)in_data[curr_pos+x] ;
30:             sum_pix += filter[8] *
31:                  (double)in_data[curr_pos+1+x] ;
32:             sum_pix += filter_norm ;
33:             out_data[curr_pos] = (pixelvalue)sum_pix ;
34:         }
35:     return image_out ;
}
Figure 2 shows an explicitly data-parallel specification for this algorithm, which we extract from the code using a recognition-based reverse engineering technique. This specification is captured in a representation based on the multi-dimensional synchronous dataflow (MDSDF) model of computation (MOC) [27]. MDSDF is an extension of the synchronous dataflow (SDF) MOC. Both are implemented in the Ptolemy project [34], which incorporates multiple MOCs (SDF, discrete-event, continuous-time, etc.) within a single heterogeneous modeling and simulation system; its applications include DSP, control systems, and real-time systems modeling. In SDF models, each node is a process that produces (consumes) tokens on its output (input) arcs at integer rates. These integer rates of production and consumption allow the execution rates of each process to be statically determined, leading to a static scheduling of all processes. SDF models are valuable in modeling signal processing applications and generating efficient code for programmable embedded DSP processors [35].
Figure 2. 3x3 Convolution as an MDDF model.
MDSDDF augments SDF with multi-element, multi-dimensional tokens. These tokens provide a useful representation for image processing algorithms, which often operate on regular subregions of an image, such as rows, columns, and tiled blocks. Our work extends MDSDDF in two ways (described further in the section “Explicitly Parallel Target Representation”): by explicitly specifying the bounds (as ranges of values) of the multi-dimensional tokens and by allowing symbolic expressions in dimensions. We refer to this extended MDSDDF as multi-dimensional dataflow (MDDF), since the inclusion of symbolic values lifts the static schedulability of MDSDDF. This fits our needs as our goal is not to schedule processes, but to model data-parallel algorithms. Our MDDF model provides several benefits for representing image processing algorithms:

1. The dimension attributes on the edges indicate the array regions operated upon and, when edge source and edge sink dimensions differ, the interactions between regions (e.g., partitioning into smaller regions, accumulation of scalar values into a vector, or transposing an N×M region into an M×N region).

2. Data-parallel operations can be easily identified. In Figure 2, consider the addition node highlighted by the gray rectangle. Its input and output edge dimensions are all equal, thus it represents a parallel, element-by-element addition over the input arrays. (Similarly, the other addition nodes have the same dimensions.)

3. A complete recognition of the program (here, as a convolution algorithm) is not necessary. A partial recognition approach that identifies the data access patterns at the inputs and outputs of the graph (and in some cases,
certain computational patterns on the interior, such as a sum over array elements) is sufficient. In particular, a distinction is made between those portions of the program that access and store data (in Figure 2, everything outside the dashed box, e.g., the in_data array coupled with the matrix-shift operators) and those that perform computations on the data (everything inside the dashed box).

4. Dataflow-based representations are familiar to systems engineers as these models are a natural way of expressing DSP and image-processing algorithms (e.g., they are used extensively in Cadence’s Signal Processing Worksystem [36]). Hence, in addition to facilitating the retargeting of sequential code to SIMD processors, this representation will aid human understanding of sequential DSP and image-processing programs. This will allow these programs to be imported into common DSP tools for less error-prone evaluation.

Outline of Chapter

This chapter presents an intermediate representation for sequential C programs derived from data and control flow analysis, an MDDF representation for representing explicit data-parallelism, and a recognition system for abstracting the former into the latter. Our partial recognition approach focuses on identifying the array access patterns in loops, while leaving the core algorithmic calculations as-is; in the future, orthogonal recognition and analysis strategies may be applied to the core. A recognition technique using a combination of graph pattern matching and symbolic manipulation to extract an
MDDF model from sequential code is presented. Code generation for translating the
MDDF model to data-parallel code that can execute on a SIMD processor is described.

The remainder of this section discusses related compiler and reverse engineering
methods for extracting data-level parallelism from source code. “Representation and
Recognition” describes our recognition and retargeting process, the internal program
representation and several key patterns used in recognizing array data access patterns.
“Experimental Validation and Results” presents experimental results that validate the
retargeting process. Additionally, it provides a runtime comparison between sequential
execution of the original programs and parallel execution of the retargeted programs to
illustrate the performance gains of the retargeting process. The chapter concludes with a
summary and discussion of the results.

Related Work

Traditional data-level parallelization work (e.g., [37][38][39]) has focused on
vectorization of loops and arrays in scientific applications. The primary goal has been to
achieve performance increases by identifying what data can be piped through composed
operations in long one-dimensional vectors. These techniques have also been adapted for
exploiting one-dimensional data-level parallelism in SIMD instruction set extensions
[18][20][16]. Vectorization attempts only to schedule program statements for parallel
execution; instead of simply finding parallel statements, our technique encodes the
program as a computational structure with explicit specification of parallel operations and
data accesses. Furthermore, most vectorization techniques only recognize parallelism in
one dimension, whereas our techniques recognize both one- and two-dimensional
operations, attaining more efficient exploitation of parallelism for region-based
operations, such as convolution.

Waters [40] abstracts each loop as a “temporal sequence of values,” i.e., a stream
of values produced by the computations in the loop. He defines several plan-building
methods (PBM)s which decompose the loop control and computations into abstractions
over these streams of data. An example is the augmentation PBM which consumes,
element-by-element, a sequence of values and produces an equal-length sequence of
values as some function of the original (e.g., multiplying the elements of an array – the
sequence – by a scalar value). Our work is similar in that we also abstract the temporal
loop structures inherent in sequential programming languages (e.g., iteration, loop body
calculations). However, instead of linear sequences we create temporal abstractions over
two-dimensional array regions.

More recently, PAP [23][24][25] and PARAMAT [26][25] have employed
program recognition to extract parallelism. The PAP system, targeted at distributed
memory architectures, creates an annotated program dependence graph representing the
program, and then performs pattern-matching to identify groups of structures as concepts.
Sets of concepts can then be aggregated into higher-level concepts and, eventually,
transformed into parallelized code. PARAMAT also performs program recognition, but
uses an abstract-syntactic tree as its basic program representation. Both PAP and
PARAMAT attempt to identify sequential algorithms in the code and replace them with
well-known parallel implementations of those algorithms. Like these projects, our work
employs pattern matching and transformations over a graph-based program
representation. The main difference is that their goal is a complete recognition of the
program as a single known algorithm that can be replaced with a parallel implementation, whereas our strategy operates at lower levels of abstraction, without requiring a complete recognition of the program. This efficiently accommodates variability in the overall algorithm while benefiting from the recognition of common building blocks and data reference patterns used in these algorithms.

Nyland et al. [32] present a methodology for top-down design of data-parallel algorithms. It emphasizes early specification of an algorithm in a high-level data-parallel language, avoiding as much as possible architectural details that would force the algorithm into a hardware-specific implementation. As the design process continues, the algorithm is refined for general classes of architectures (e.g., multiprocessor vs. SIMD) through to specific architectures (e.g., MMX). Central to this process is the concept of nested data-parallelism, i.e., data-parallel operations may be performed on the elements of a linear sequence and each of those elements may itself be a data-parallel linear sequence. Our reverse engineering process mirrors their synthesis methodology in reverse by extracting a data-parallel model from the implementation. The concept of array region we employ is compatible with nested-data parallelism, and our explicitly data-parallel representation provides a suitably abstract specification for the algorithm.

**Representation and Recognition**

Figure 3 illustrates the recognition process. Control and data-flow analysis is performed on input C code to produce a hierarchical dataflow graph intermediate representation. Recognition proceeds from the innermost loop towards the outermost. Pattern matching recognizes memory access operations; these sequential artifacts are then replaced with their parallel counterparts. Attribute information produced by pattern
matching is analyzed and propagated to build regions and the transformed graph is annotated with these regions. The transformed MDDF graph (no longer an inner loop) is then merged with the original hierarchical DFG and the selection and recognition processes repeat until the set of nested loops is exhausted. This section elaborates on the steps of the recognition process and the representations used.

Figure 3. Recognition process.
**Candidate Loops and Pre-processing**

Recognition is only performed on loops with a reasonable potential for data-level parallelism. We focus recognition on loops with the following characteristics, common to programs that are inherently data-parallel:

1. No non-local control flow originates inside the loop or enters from outside the loop.

2. Each loop has only one loop exit condition, located in the loop header (e.g., no `continue` statements in C).

3. All addresses generated by the loop must be monotonically increasing or monotonically decreasing functions of the loop counter, i.e., all array accesses, whether through indexed arrays or pointer-accessed arrays, occur with constant stride, though the stride may be greater than one or less than zero.

Constraints (1) and (2) above enforce the property that all iterations must execute under the same control conditions, which is required for data-parallel execution. The third constraint ensures that program input and output data arrays can be mapped to the regular arrangements of processing elements common to SIMD processors.

Additionally, some pre-processing occurs prior to recognition. In particular, any function called from within the loop is in-lined, and any inner loops with a small statically-determinable iteration count (here, fewer than eight) are unrolled.
**Initial Representation of Source Program**

To facilitate recognition, the C source is parsed and translated into an intermediate representation, which is a flowgraph with the following node types:

- **Basic computational operators:** addition, multiplication, logical operations, comparison operations, etc. These are illustrated by circles or ovals in this dissertation.

- **Memory operators:** load and store, with appropriate variants depending on whether the access is through a pointer, a singly-subscripted array, or a doubly-subscripted array. These are indicated by rectangles in our figures.

- **Interfaces:** input and output nodes, representing data flows into or out of the flowgraph. These are used to match input and output flows when merging a recognized inner loop MDDF graph with an outer loop. Interfaces (and constant numeric inputs) appear as rounded rectangles.

- **Control blocks:** abstractions over embedded loops and conditionals. These appear as rectangles with named ports. An example is shown in Figure 4.

Dataflow edges connect node outputs to node inputs. For loops that contain back edges, the back edges (here, represented as dashed arcs) are annotated separately so they can be avoided during graph traversals (a simple definition-use analysis during source parsing is sufficient to identify back edges).

The memory operators are required to represent accesses through pointer and array types. The simplest load, “LOAD_ptr,” takes a pointer address as an input and produces the value stored at the address. The analogous store takes a pointer address
edge and the data edge to be stored as its inputs, and produces no output. More complex load and store operators include the singly- and doubly-subscripted versions. For instance, the one-dimensional array-access load “LOAD_1” takes an array (or pointer) and an indexing expression as its inputs, producing the array element as its output; the corresponding one-dimensional store has a data input in addition to the array and index inputs and produces an array as its output.

*Control blocks* encapsulate loops and conditionals, creating a hierarchical DFG representation. Each contains its own DFG (i.e., the loop body), which could itself contain other control blocks. The control blocks are annotated with information from the source code, including the block type (loop or conditional), loop exit or branching condition, and any loop initializations (e.g., the “\(i=0;\)” in “*for*(*i*=0;*i*<N;*i*++)”). Figure 4 illustrates the nesting of DFGs within a loop control block.

![Diagram of control block](image)

*Figure 4. Sample initial program representation.*
Explicitly Parallel Target Representation

Our goal is to abstract the intermediate representation into the higher-level, explicitly data-parallel MDSDF representation. In MDSDF, upon which MDSDF is based, edges may carry multi-element tokens as rectangular multi-dimensional arrays. Further, MDSDF allows for edge sources and their connected edge sinks to produce and consume, respectively, tokens of differing dimensions. These unmatched token dimensions implicitly specify the relative rates at which the nodes in the graph must execute. Figure 5 gives an example. Since the multiply node consumes two 1×8 arrays (performing eight parallel multiplies each time it executes), it must execute eight times for every 8×8 token produced by the input A.

![Figure 5. MDSDF example.](image-url)
For our goal of retargeting sequential code for data-parallel execution, the multi-dimensional edge annotations encapsulate the loop control flow of the original source without requiring the back edges of the original graph. Back edge removal provides the additional benefit of allowing us to flatten the hierarchical DFG representation such that the final model is a single flowgraph with multi-dimensional annotations. Additional processing is then applied to the MDDF representation to generate SIMD code. In the example of Figure 5, a code generator produces 64 concurrent multiply operations (assuming 64 available processors) by distributing the results of the function at node C and the corresponding elements of input A to 64 different processors.

Our MDDF representation includes two extensions to the MDSDF representation. The first is that edge annotations include the bounds information, e.g., \( (1:8:1, 0:8:2) \) specifies a two-dimensional token with eight elements in the first dimension, beginning at position one, and having a stride of one. Similarly, the second dimension has eight elements, begins at position zero, and has a stride of two between each element. This preserves information from the original source concerning loop iteration ranges and array boundary conditions. The second is that our representation allows for symbolic expressions – necessary in an analysis of source code – in dimension annotations, while MDSDF does not. (The MDSDF design goal of static schedulability of graph nodes as communicating processes requires exact knowledge of array dimensions, thus symbolic expressions are not part of the MDSDF representation.)

**Patterns**

This section presents several common patterns useful for transforming the initial program representation to the MDDF specification. These patterns are concerned
primarily with understanding data accesses to arrays both indirectly through pointers and directly through explicit array references. For the remainder of this dissertation, graph elements corresponding to recognized patterns will be represented as a rectangle with double vertical sidebars (see the left-hand side of Figure 6). The full pattern library, as a reference graph grammar, is given in Appendix B.

The patterns detected provide array region information. Here, a region is notated as \( \text{base}):(\text{start}:\text{range}:\text{step}) \) for one-dimensional regions and \( \text{base}):(\text{start}_y:\text{range}_y:\text{step}_y, \text{start}_x:\text{range}_x:\text{step}_x) \) for two-dimensional regions. The \text{base} is the base address variable (either a pointer or array variable in the original C code) of the region. The \text{start}, \text{range}, and \text{step} fields are symbolic expressions representing, respectively, the first element position of the region, the number of elements in the region (including the start element), and the stride. For example “\text{img}_in):(2:256:4)” would indicate a region of 256 data elements beginning at address \text{img}_in+2 and with a stride of 4, i.e. \{\text{img}_in+2, \text{img}_in+6, \text{img}_in+10, \ldots, \text{img}_in+1022\}. This region notation is similar to array region data structures used in the ZPL parallel programming language [21] and the parallelizing compiler work of [41].

The latter notate each region dimension as \( (\text{lb}:\text{ub}:\text{step}) \) where \text{lb} and \text{ub} are the lower and upper bounds, respectively and \text{step} is the region stride; ZPL uses a similar bounds-based format. We chose to use the range field instead of the upper bound as this facilitates reasoning about the regions (while still maintaining equivalent semantics). For example, all regions defined in a given loop nest will have the same range value, even if their steps and bounds differ, thus once the range is found for one induction variable it is available for all.
Count Pattern

The count pattern aggregates the values enumerated by a loop induction variable; it may be a component of other memory access patterns. For a count pattern, a cyclic path exists such that \( \text{var}_{i+1} = \text{var}_i + \text{expr} \) holds true for all iterations \( i \) where \( \text{var} \) is a loop counter variable (encoded by a input-output node pair connected by a back edge) and \( \text{expr} \) is an additive expression (i.e., only addition, subtraction, or negation operations allowed). Figure 6 (a) shows a case where \( \text{expr} \) is one; Figure 6 (b) and (c) show the effects of adding a variable to, or multiplying a variable by, respectively, a count pattern.

Figure 6. (a) Basic count pattern, (b) count plus an expression, (c) count times an expression.

Array-Read and -Write Patterns from Pointer-Based Loads and Stores

A parallel array-read or array-write pattern can match from one of two cases of pointer-based load and store structures. The first is similar to the count pattern: a pointer
variable participates in a cyclic path such that $ptr_{i+1} = ptr_i + expr$, leading to an
enumeration of addresses for the array pointed to by $ptr$. Figure 7 (a) shows an example
for an array-read. The second case occurs when the address input of a pointer-based load
or store node is a linear function (i.e., only additive operations and multiplication) of the
elements enumerated by a count pattern, as in the example of Figure 7 (b).

![Diagram](image)

**Figure 7.** Example pointer-based array-read patterns.

**Array-Read and –Write Patterns from Subscripted Loads and Stores**

The array-read and array-write patterns matching singly- and doubly-subscripted
load and store structures are similar to pointer-based loads and stores: they occur when
the index inputs are linear functions of the elements enumerated by count patterns.
Figure 8 (a) and (b) illustrate the singly- and doubly-subscripted load cases, respectively.
Figure 8. Array-based load and store patterns: (a) singly-subscripted (b) doubly-subscripted.

Recognition

PARRET employs graph pattern matching to detect patterns implementing array data-access regions. It identifies the loop condition and any induction variables present in the loop and, along with address calculation operations, determines their effects on array and pointer variables.

Block Quantization Example

The example code of Figure 9, a block quantization algorithm distributed by Texas Instruments (TI) for their C6200 line of DSPs [42], will be used to illustrate the implementation details. Block quantization is used in applications such as JPEG encoding to round each pixel to the nearest quantization value. This rounding results in a smaller range of pixel values for the image, meaning the pixels can be represented in fewer bits, facilitating compression. A particularly interesting feature of the example code is that the data array in the inner loop is being accessed in strides of blk_size instead
of one as an architecture-specific optimization to avoid redundant load operations from the `recip_tbl` array. Application of our recognition process will expose the core algorithm here, specifically that individual sub-blocks of the `data` array are being multiplied, element-by-element, with the values in the `recip_tbl` array. This recognition effectively removes the optimization, allowing the algorithm to be retargeted to a different processor where such an optimization might not be desired. Note that the variables `recip_tbl` and `data` are both declared by the function to be pointers, not arrays, so the example will treat them as pointers even though the code sample uses array subscript notation.

```
01:   for (i = 0; i < blk_size; i++) {
02:       recip = recip_tbl[i];
03:       k = i;
04:       for (j = 0; j < num_blks; j++) {
05:           quot = data[k] * recip + round;
06:           data[k] = quot >> q_pt;
07:           k += blk_size;
08:       }
09:   }
```

**Figure 9.** Quantization source code fragment.

Recognition Process

Recognition proceeds as illustrated in Figure 3. The algorithm is as follows:

1. **Select innermost loop DFG.** Recognition is first performed on the innermost loop DFG, and each subsequent iteration of Figure 3 focuses on the next-highest loop DFG. Figure 10 (a) shows the DFG for the innermost loop of the quantization example.
2. **Pattern matching.** The patterns are matched based on their structural characteristics. Count patterns and the cyclic pointer load and store patterns produce partial region information; each defines a region with the start field equal to the initial value of the variable and a step field equal to the net increment of the variable update expression. Figure 10 (b) shows the recognition of count patterns for the $j$ and $k$ variables. Parsing the loop conditional results in an upper bound (or lower bound, if the step expression is negative) for one of the count patterns; this information provides region range information. In the example, the conditional “$j < num_blks$” results in an upper bound of $num_blks - j$ for the count pattern corresponding to $j$. Dividing the upper bound by the step for $j$ (one) and substituting the initial value of $j=0$ gives the range field, i.e., the number of loop iterations, for all count patterns, leading to the completed region information shown in Figure 10 (b).

3. **Attribute synthesis.** This step aggregates start, range, and step information into region annotations, simplifying symbolic expressions as needed.

4. **Removal of sequential artifacts.** Replace the recognized count, load, and store patterns with the corresponding graph elements from the initial loop DFG. Edges are annotated with region information. The transformed graph for the quantization example is shown in Figure 11.
5. *Embed explicitly parallel form.* The transformed inner loop DFG is merged with the parent loop DFG by matching corresponding flow edges.

Figure 12 shows this merge for the quantization example.

The recognition process now repeats for the composite DFG, as shown in Figure 13. During the subsequent attribute synthesis step, the inner loop region $(k: num\_blks: blk\_size)$ is resolved with the outer loop region $(0: blk\_size: 1)$ over $i$ as $k$ is assigned $i$ in the outer loop. Evaluation reveals that the composite region is $(0: num\_blks * blk\_size: 1)$, i.e., for every element in the $i$-region, there is an instance of the entire $k$-region.

After all loop DFGs have undergone recognition and a flat MDDF model created, a final post-processing step ("*Build Parallel Operations*" in Figure 3) is required to annotate operations in the core of the algorithm with region information. Regions from array-read nodes are propagated along graph edges to add, multiply, etc. operator nodes according to rules given in Table 1. The results of this annotation operation are shown in the final MDDF model of our example in Figure 14.

<table>
<thead>
<tr>
<th>Table 1. Node propagation rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unary operator (negate, logical NOT)</strong></td>
</tr>
<tr>
<td><strong>Binary operator (add, mult, etc.)</strong></td>
</tr>
<tr>
<td>where MIN\textsubscript{$i=1$}(R1,R2) is the smallest non-unity (i.e., range &gt; 1) region between R1 and R2</td>
</tr>
</tbody>
</table>
Figure 10. Quantization example: (a) original DFG representation (b) recognition of count patterns.
Figure 11. Recognition of parallel pointer-based load and store patterns.
Figure 12. Merging of transformed inner loop DFG and the outer loop DFG.
Figure 13. (a) Recognition of count patterns, (b) recognition of parallel pointer-based load patterns.
Figure 14. MDDF quantization algorithm model.

**Code Generation**

To validate our recognition approach, we retarget the extracted MDDF model to code for a data-parallel processing platform. In particular, we target the SIMPil processor [14], which is representative of emerging SIMD architectures. SIMPil is a two-dimensional grid array of processing elements (PEs), with each PE consisting of a local register file, an integer arithmetic logic unit (ALU), local memory, and communication links with its four nearest neighbor PEs (above, below, left, and right). All PEs receive instructions from a control unit which performs instruction fetch, decode, and dispatch, and all PEs operate in lock-step. For image processing applications, the PE array receives data from an image source such that each PE is assigned a block of image
pixels for processing; the number of pixels each PE receives is called the pixel-per-
processing element (PPE) ratio.

The image processing algorithms under investigation divide the image into blocks
(blocks may be a single pixel), performing a set of sequential operations within each
block. Parallelism is achieved by operating on multiple blocks simultaneously. Our code
generator matches block operations specified by the retargeted program model to a
specific PPE ratio in SIMPil.

The input to the code generator is an MDFF program model, such as the one
shown in Figure 14. Because the image size for a given implementation of SIMPil is
fixed, the actual (not symbolic) image bounds must be specified as well as values for
`num_blks` and `blk_size`, if this information is not available in the original C source code.
(In our example, a 256x256 pixel image size is chosen, with `blk_size`=16 and
`num_blks`=256x256/16=4096.) The code generator first examines all nodes in the graph,
looking for regions that appear to be a subset of the image size; in the example, it finds
the region `(0:blk_size:1)=(0:16:1)` to be the region upon which the core calculations are
operating and chooses it as the base block size for mapping operations to SIMPil. SIMPil
requires that square image regions be mapped to a PE, so the code generator expands the
`(0:16:1) base region to `(0:16:1,0:16:1)`, i.e., a 256-PPE ratio. In other words, sixteen
instances of each `(0:16:1) operation will be mapped to each PE. Thus our ideal speedup
due to parallel execution is `(256x256)/(16x16)=256`.

The code generator identifies the input region
`data::(0:num_blks*blk_size:1)=data::(0:65536:1)` as a linear region equivalent to the
256x256 image region and converts it to its two-dimensional representation
data::(0:256:1,0:256:1). This repeats for the corresponding output region. It can now convert the data array input and output operations to those that read and write, respectively, the entire image.

Having adjusted all regions to be compatible with the SIMPil processor, the code generator outputs all operations in breadth-first order (ensuring no operation is executed before the operations that supply its inputs). If the head of an edge has a region different from that of its tail, region-matching code must be generated. In our example, the edge connecting input recip_tbl and the multiplication node has a head region of (0:16:1) and a tail region of (0:16:1,0:16:1), meaning a for-loop must execute on each PE to replicate the recip_tbl array:

```c
for( ii=0; ii<16; ii++ ) {
    for( jj=0; jj<16; jj++ ) {
        /* tl4 is the input variable to the multiply operator */
        tl4[ii][jj] = recip_tbl[jj];
    }
}
```

This edge-matching is not required between an image region and a PE region, as is the case from the data input to the multiply node, as the distribution of such block regions to PEs implicitly performs the edge-matching.

The code generator produces code in the extended ZPL [21] language used as the input to the Retargetable SIMD Compiler (RSC) [22] which generates SIMPil assembly code. The generated code is shown in Figure 15 (variable declarations are excluded for brevity). RSC allows operations to be specified at the level of a single pixel (i.e., the operation is performed for every pixel in the image, indicated by the [PIXEL] notation
in Figure 15) or at the granularity of a PE (i.e., the operation is performed for every PE in the system, indicated by [PE] in Figure 15).

```plaintext
01: [PE] t3 := 1; /* PE-level operation */
02: [PE] t2 := q_pt;
03: [PE] t4 := t2 - t3;
04: [PE] t1 := 1;
05: for x_tmp3 := 0 to 0 do
06:     for x_tmp4 := 0 to 15 do
07:         [PE] t14 [x_tmp3*16+x_tmp4] :=
08:             recip_tbl[x_tmp3*16+x_tmp4];
09:     end;
10: end;
11: read(data);
12: [PIXEL] t24 := data; /* pixel-level operation */
13: [PE] t5 := t1 <= t4;
14: for x_tmp1 := 0 to 15 do
15:     for x_tmp2 := 0 to 15 do
16:         [PE] t14_26 [x_tmp1*16+x_tmp2] :=
17:             t14 [x_tmp2];
18:     end;
19: end;
20: [PIXEL] t26 := t24 * t14_26;
21: [PE] t2 := q_pt;
22: [PIXEL] t28 := t26 + t5;
23: [PIXEL] t30 := t28 >> t2;
24: [PIXEL] data := t30;
25: write(data);
```

**Figure 15. Data-parallel ZPL code for the quantization algorithm.**

Though not present in this example, some applications, such as image convolution and edge detection, exhibit spatial dependencies within a neighborhood of pixels, i.e., the value of an output pixel may depend on the values of several input pixels spatially local to it. These patterns are revealed in the detected regions as offsets from zero in the start fields. For example, a load in an image convolution might load in the region `img_in:=(1:255:1,-1:256:1)`, indicating an output pixel dependent on this value would
require input pixel data from the pixel one row below (downward is taken as the positive y-direction) and one position to the left. This would result in a statement of “[PIXEL]
tmpvar := img_in@[1, -1]” in the extended-ZPL syntax used by RSC.

Experimental Validation and Results

To validate the recognition system, we analyzed several image processing algorithms from the TI IMGLIB library with PARRET (PARallel RETargeter), our prototype recognition and retargeting tool. The resulting specifications were used to generate SIMD code and execute it on the SIMPil architectural simulator. The quality of the retargeted results and their run-time performance were compared with those of a baseline sequential execution.

Recognition was applied to the seven benchmarks listed in Table 2. Each application was tested using a 256 row by 256 column, 8-bit grayscale image as input. Figure 16 compares the results of each retargeted program with those produced by a baseline sequential execution (i.e., compiled C executable). The results are given as the average distance of the output images, defined as:

$$\text{avg dist} = \frac{1}{MN} \sum_{n=1}^{N} \sum_{m=1}^{M} |a_{nm} - b_{nm}|$$  \hspace{1cm} (1)

where $a$ and $b$ are the $M\times N$ images produced by the original sequential execution and retargeted data-parallel code, respectively. The average distance measure here ranges over grayscale pixel values 0 to 255 (i.e., 8-bit pixels); the closer the average distance is to zero, the more closely the results match. For all tested applications, our data-parallel results are quantitatively close to those produced by sequential execution of the C code. Most of the limited errors observed are due to border conditions at the edge of the image.
In some cases (e.g., corr_3x3), the C code was written in such a way that a pixel at the end of a row was dependent on the pixel at the beginning of the next row (wrap-around), which is an artifact of treating a two-dimensional image as a one-dimensional vector of values. The generated SIMPil code would assume no wrap-around and treat the dependence as an edge border condition, supplying a zero for the needed value. A similar case (e.g., perimeter) exists when the C version may ignore the last one to two pixels in a row, in this case to avoid the wrap-around mentioned earlier. The SIMPil code generated would assume operations on every pixel in the row, resulting in pixel errors in the last one to two columns of the image. For the image-processing applications described, these errors are acceptable as their qualitative effect will be small.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Application(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv_3x3</td>
<td>Convolution of image with a 3x3 filter mask</td>
<td>Noise removal, image smoothing</td>
</tr>
<tr>
<td>corr_3x3</td>
<td>Correlation of a 3x3 reference image with the input image</td>
<td>Motion estimation</td>
</tr>
<tr>
<td>perimeter</td>
<td>Build image object perimeter</td>
<td>Object detection/recognition</td>
</tr>
<tr>
<td>pix_sat</td>
<td>Clip (saturate) pixels to ceiling value</td>
<td>Compression (i.e., clip values to a certain bit precision)</td>
</tr>
<tr>
<td>quantize</td>
<td>Quantize pixel values to a smaller range of discrete values</td>
<td>Compression (e.g., used in JPEG)</td>
</tr>
<tr>
<td>sobel</td>
<td>Object edge-detection</td>
<td>Object detection/recognition</td>
</tr>
<tr>
<td>threshold</td>
<td>All pixels above threshold value set to high value, all others set to low value</td>
<td>Pre-process for image dilation/erosion, perimeter detection</td>
</tr>
</tbody>
</table>
Figure 16. Program results comparison, SIMPil vs. sequential

Figure 17 compares the execution times of the retargeted SIMPil programs with those of the original sequential code, provided by execution on the SimpleScalar [43] simulator (sim-outorder). As Figure 17 illustrates, speedups from over 60 times to over 30,000 times are possible. In all cases, the retargeted code provides a clear performance benefit over sequential execution.
Figure 17. Execution speedup of retargeted SIMPil vs. sequential

Recognition Performance

Table 3 lists the size of each test program (in terms of the number of nodes in the input graph) and the amount of time recognition and code generation required. The timing was measured using the UNIX time command on a 4-CPU Sun E-450 server with 4 GB of RAM running SunOS 5.8. For these programs and this pattern library, the retargeting process is efficient.
Table 3. Recognition times for SIMPil retargeting.

<table>
<thead>
<tr>
<th>Program</th>
<th># Graph Nodes</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv_3x3</td>
<td>270</td>
<td>3.50</td>
</tr>
<tr>
<td>corr_3x3</td>
<td>131</td>
<td>2.93</td>
</tr>
<tr>
<td>perimeter</td>
<td>151</td>
<td>2.02</td>
</tr>
<tr>
<td>pix_sat</td>
<td>75</td>
<td>0.59</td>
</tr>
<tr>
<td>quantize</td>
<td>107</td>
<td>1.27</td>
</tr>
<tr>
<td>sobel</td>
<td>246</td>
<td>6.70</td>
</tr>
<tr>
<td>threshold</td>
<td>43</td>
<td>0.42</td>
</tr>
</tbody>
</table>

Conclusions and Future Work

In this chapter, we have presented a dataflow representation for explicitly representing data-parallel algorithms and have described the recognition process for extracting those representations. As Figures 16 and 17 show, our representation and recognition process can extract, from sequential source code, accurate code for data-parallel execution on SIMD processors, resulting in a large speedup in execution times. Automatically exposing the inherent data parallelism in image processing applications can fuel a new generation of high performance, high efficiency imaging systems.
CHAPTER 3

RETARGETING SEQUENTIAL CODE TO MULTIMEDIA INSTRUCTION SET ARCHITECTURES

Abstract

This research addresses challenges for retargeting loop-based code for multimedia instruction set extensions. An important issue is that vector data types of mixed precision within a loop body complicate the parallelization process since corresponding array elements are misaligned in the packed vectors. This work presents an approach based on program recognition to create a multi-dimensional dataflow graph-based representation with explicit parallel semantics. The multi-dimensional annotations facilitate generating vector data type conversion code during code synthesis. This representation is independent of sequential artifacts, allowing code synthesis to proceed based on an abstract data parallel model of the program and the constraints imposed by the architecture (vector length, available data types, etc.). Our results show this representation facilitates parallelization of a wider range of loops than traditional vectorization. The results of this parallelization indicate loop speedups of 2 to 26 times over a sequential execution are possible.

Introduction

Multimedia software applications, such as video playback, image processing, and 3D video games often have a high potential for data-level parallelism (DLP), i.e., they perform the same operation(s) over all the elements of a large data set. To address the performance needs of these applications, many processor vendors have extended their instruction sets with multimedia instructions incorporating single-input, multiple-data
(SIMD) functionality. Examples are Intel’s MMX [1], SSE[2], and SSE2 [3][4], AMD’s 3DNow! [5], Motorola’s AltiVec [6], and Sun Corporation’s Visual Instruction Set (VIS) [7]. These SIMD instructions simultaneously operate on multiple data elements packed into a single register, resulting in execution speedup. For example, the PADDW (“Packed Add Word”) instruction included with Intel’s SSE2 [3][4] extensions adds the corresponding elements of two vectors of eight 16-bit data elements, providing an ideal speedup of eight over a sequential execution.

Challenges still exist in developing and porting applications for these instruction sets. Typically, loop kernels employing multimedia instructions must be hand-optimized using in-lined assembly code or intrinsic functions. Traditional loop vectorization has been employed to generate SIMD code automatically, but is in general a transliteration technique without any notion of a high-level program representation that can be applied to multiple data-parallel targets.

We present an explicitly-parallel, multi-dimensional dataflow (MDDF) graph-based representation of the program that abstracts over the sequential details used to implement the algorithm. This representation facilitates multi-target code generation (e.g., SIMD processors arrays and multimedia instruction set extensions). PARRET is a system we developed for extracting MDDF from sequential source code and, from that representation, synthesizing data-parallel code. We show that this method is capable of parallelizing a range of loop kernels beyond that of traditional vectorization. We identified a set of test programs from the Texas Instruments (TI) IMGLIB suite that exhibit parallelization challenges particularly in misaligned mixed-precision data types (as described below); all nine were successfully parallelized by PARRET while only two
were parallelized by a vectorizing compiler. We obtain an average speedup of two over sequential across our test suite, with a maximum of 26.

**Misalignment of Mixed-Precision Packed Data Types**

The main architectural constraint that must be considered when parallelizing loops for multimedia instruction sets is that all accesses to the elements of an array must be consecutive, i.e., having a stride of one. This follows from the parallel load and store operations common to these instruction sets where an entire vector of consecutive data is read from or written to memory in a single instruction (e.g., 16 consecutive bytes in SSE2[3][4]).

A loop kernel that is not carefully written can easily violate this unit stride constraint. The simplest case is a loop where the array indices are monotonically increasing by a stride greater than one, e.g.:

```c
for( i=0; i<N; i+=8 ) {
    A[i] = B[i] + C[i];
}
```

In certain cases, such loops can be restructured, using MDDF representation, to have a unit stride. Chapter 2 presented an example where PARRET was able to derive a unit stride access pattern from a doubly-nested loop arrangement where the inner loop had a non-unit stride.

A more common case where unit stride becomes an issue is the presence in a loop body of vector data types of different precisions. For example, a loop having both 16- and 32-bit types in the body will not have corresponding elements aligned properly; in this case, the 16-bit vectors have unit stride but the 32-bit vectors have a stride of two
with respect to the smaller type. The compiler must deal with multiple parallel factors (e.g., 8-way and 4-way parallelism for the above types, assuming a 128-bit SIMD register size) in the same loop. PARRET’s MDDF representation explicitly encodes array region annotations that facilitate resolving differing vector data types along graph edges.

This research addresses the above issues, increasing the range of loops that can be optimized using multimedia instruction sets. Specifically, we take a reverse-engineering approach where the program is abstracted to a dataflow-based representation, data-parallel access patterns are recognized, and the appropriate code for multimedia ISAs is generated from the resulting specification. This front-end analysis abstracts away architectural details specific to particular hardware and thus attempts to find best-case parallelism at a platform-independent level, for example, using knowledge of loop nesting to normalize loops with non-unit stride into unit stride, if possible. The back-end code generator then translates the abstract representation into machine-readable form, using knowledge of array regions formed by each vector data type to generate type-conversion code to properly align mixed-precision vector types. We compare our results with those of a widely-used commercial vectorizing compiler and show that this approach is capable of parallelizing a wider range of loops.

By retargeting to a multimedia ISA, we demonstrate, along with the results of Chapter 2, PARRET’s ability to target multiple SIMD architectural models. The key to this flexibility is that the multi-dimensional dataflow (MDDF) representation PARRET extracts from source code is architecture-independent, representing the program as an abstract data-parallel execution model.
Outline of Chapter

The next section presents related work on vectorization to parallelize program loops for multimedia ISA extensions. The “Approach” section gives a brief overview of MDDF, PARRET’s recognition process, and synthesis of code for multimedia ISAs from MDDF. The “Validation and Results” validates the correctness of programs retargeted to Intel’s SSE2 [3][4] by PARRET, compares their speedup over sequential, and compares the ability of PARRET to parallelize loops with that of ICL [16], Intel’s commercial vectorizing compiler. The last section presents conclusions and discussion.

Related Work

The traditional method for parallelization of loops is vectorization, i.e., partitioning the iteration space and data set of a loop such that each partition can be executed on parallel hardware. The main constraint for vectorization is that loops not have any loop-carried dependencies [44][38][28][45]. Once a vectorizing compiler is satisfied that dependence constraints have been met, it can schedule concurrent iterations for parallel execution based on the hardware resources available.

Vectorization for Multimedia ISAs

Several projects have implemented vectorization for generating code using multimedia ISAs, including the Intel C/C++ compiler [16][17][46], the MOM project [47], and work by Sreraman et al. [18], Gustin et al. [19], and Cheong et al. [20]. In general, these projects attempt to vectorize the inner loop of a loop nest. First, loop dependency analysis is performed. Then any applicable loop transforms are applied, such as loop distribution to isolate dependent statements into separate loops and scalar expansion (distributing the value of a scalar to an array that replaces the scalar) to break
unnecessary loop-carried dependencies. Finally, if the loop is determined to be vectorizable, it is strip-mined to vector length, i.e., the loop step is changed from one to the vector length, such that each iteration operates on a single vector [44][48]. Code generation usually produces inlined assembly instructions from the multimedia ISA extension (or function calls that execute those instructions). Some, like the Intel compiler, perform low-level pattern recognition to identify code fragments, such as reduction operations (e.g., summation, minimum/maximum, etc.), that contain true dependencies, but that can still be parallelized when understood as the higher-level operation.

The Matrix-Oriented Multimedia (MOM) project [47] takes a unique approach to both multimedia ISAs and their compiler support. The MOM ISA provides two-dimensional, i.e., matrix, packed data types. These matrix types are modeled as a vector (the matrix rows) of packed words (the elements in each row). The MOM compiler builds these matrices by extending vectorization into two-dimensions. First, the inner loop (of a doubly-nested loop) is vectorized in the usual manner. Then the outer loop is itself vectorized, using the results of the first step, to produce the matrices as vectors of vectors.

The main limitation of these types of compilers is their dependence on the syntactic details of the program. Compiler intermediate representations (IRs, e.g., abstract syntax trees, data-dependence graphs, triples/quads, register-transfer language, control-flow graphs, static single-assignment, etc.) are tightly coupled to the sequential details of the program (use of variables, loops, assignment statements, etc.) and as such still follow an inherently sequential computational model. They have knowledge of array
accesses, but not data-parallel array regions. They see a temporary variable written to in one statement, and read from in another, but not an explicit dataflow link connecting two operations. These sequential IRs force vectorizing compilers to parallelize loop body statements as-written, instead of treating the whole loop body as a single function over composite (e.g., array or matrix) data types.

**Approach**

Figure 18 illustrates the retargeting process. Our recent work [49][31][50] has investigated using reverse engineering techniques, in particular program recognition, to identify data-parallel memory access and computation patterns in sequential source code. Once identified, these patterns are used to create a high-level program representation with explicitly data-parallel semantics, from which data-parallel source code for SIMD processors can be synthesized. The remainder of this section will give a brief overview.
Extracting an Explicitly-Parallel Program Representation

Target Representation

The goal in recognizing data-parallel patterns in source code is to create a high-level, explicitly parallel program representation. We chose a representation based on the multi-dimensional synchronous dataflow (MDSDF) [27] model of computation (MOC) developed as part of the Ptolemy project [34]. MDSDF specifies a task-concurrency model where tasks, represented as nodes in a flow graph, communicate by sending tokens along the graph edges. These tokens are multi-dimensional entities where each dimension has a known integer size, e.g., a particular token may have three rows and two columns. Additionally, the producer for an edge may produce a token with different sizes and dimensions than the consumer of a node, resulting in differing rates of execution for
each task. For example, if task \( T_1 \) produces a \( 4 \times 2 \) token onto edge \( E_{1,2} \) and task \( T_2 \) consumes a \( 2 \times 1 \) token from \( E_{1,2} \), then \( T_2 \) must execute four times for every execution of \( T_1 \). These execution rates can be used by a task scheduler to create a static execution schedule for all tasks in the system.

Even though our domain is data-parallel program representation and not task-concurrency, the MDSDF model comes very close to providing a data-parallel program representation. If the multi-dimensional tokens are viewed as regions of array data, they become a natural way to express the data structures common to multimedia applications, such as images and audio streams, as well as the partitions of these data structures that are often operated upon in isolation, e.g., columns and rows of images, sub-blocks of images and data-streams, etc. For example, consider the abstract algorithm representation in Figure 19. This algorithm reads in a \( 512 \times 512 \)-pixel image and a \( 1 \times 128 \)-element coefficient array, partitions the image into \( 1 \times 128 \)-pixel sub-blocks, multiplies the sub-blocks element-wise with the elements of the coefficient array, and then re-assembles the resulting blocks into the \( 512 \times 512 \)-pixel output array. With the program in this representation, data-parallel code can be generated such that the multiplication operation is performed in parallel on SIMD hardware.
We refer to our representation as multi-dimensional dataflow (MDDF) since we do not use it for synchronous task scheduling. We refer to the edge annotations describing the tokens as regions, similar to the notion of array regions described in [21] and [41], and notate our regions as 

\[ (\text{start}_i:\text{range}_i:\text{step}_i, \text{start}_j:\text{range}_j:\text{step}_j) \]

where \text{start}_i, \text{range}_i, and \text{step}_i refer to the first index of the region, the number of elements in the region, and the stride between elements, respectively, in each dimension \( i \). For one-dimensional regions, the \( j \)-fields are removed for convenience, e.g., \( (\text{start}_i:\text{range}_i:\text{step}_i) \).

**Pattern Recognition**

The recognition begins by performing control- and data-flow analyses on the source code, producing a dataflow graph-based intermediate form. For example, the loop of Figure 20, a Sobel edge-detection algorithm from the TI IMGLIB suite [42], is converted into the intermediate dataflow graph (DFG) of Figure 21. For brevity only
representative portions of the code are shown. This example points out the mixed vector
data type problem discussed in the introduction (see annotations in Figure 21). Chapter 2
presents an example where the loop stride is not one, but where PARRETT is able to use
deeper reasoning about the loop nesting structure to discover the program is operating on
contiguous data with a stride of one.

```
01: void sobel() {
02:     unsigned char * in;
03:     unsigned char * out;
04:     short cols, rows;
05:     int H, O, V, i, j;
06:     int i00=0, i01=0, i02=0;
07:     int i10=0, i12=0;
08:     int i20=0, i21=0, i22=0;
09:     int w = cols;
10:     for (i = 0; i < cols*(rows-2) - 2; i++) {
11:         i00=in[i    ];
12:         i01=in[i +1 ];
13:         i02=in[i +2 ];
14:         i10=in[i+   w ];
15:         i12=in[i+ w+2];
16:         i20=in[i+2*w ];
17:         i21=in[i+2*w+1];
18:         i22=in[i+2*w+2];
19:         H = -100 - 2*i01 - i02 + i20 + 2*i21 + i22;
20:         V = -100 + i02 - 2*i10 + 2*i12;
21:         A = abs(H) + abs(V);
22:         if (A > 255) A = 255;
23:         out[i + 1] = A;
24:     }
```

Figure 20. Sobel example source code.
Figure 21. Intermediate DFG representation.
Once the intermediate representation has been built, pattern matching can begin. Using a graph-matching technique (see Chapter 2 and Appendix B), the recognizer searches for instances of data-parallel patterns. Example patterns are shown in Figure 22; a full description of the pattern library is given in Chapter 2. Figure 23 shows the recognition of a \texttt{count} pattern (a function that generates a linear sequence of values) from the cyclic structure created by the induction variable \( i \). This corresponds to the shaded region in Figure 21. Figure 24 then shows the recognition of parallel input and output primitives (\texttt{array-read} and \texttt{array-write}, respectively) from the memory access sub-graphs (the \texttt{LOAD} and \texttt{STORE} blocks with their address calculation inputs) and the \texttt{count} pattern (see shaded regions in Figure 23), producing the final MDDF specification for this loop.

![Diagram](image)

**Figure 22.** Pattern examples: (a) Count pattern, (b) \texttt{array-read} pattern based on a count pattern and \texttt{LOAD} operation.
Figure 23. Recognition of a count pattern.
Figure 24. Recognition of array-read and array-write patterns for finalized MDDF representation.
In this example, the SELECT node (and every other computational node, i.e., those not involved in memory address calculations) persists from the original DFG intermediate representation. Its semantics are that if the first input is a “true” (non-zero in C) value the second input is quoted to the output, otherwise the third input is quoted to the output. This operator extends to a data-parallel domain by making arrays of all inputs and its output, applying the basic SELECT operation to corresponding elements of its inputs. This operation is equivalent to the bit-masking transformations used in mainstream vectorizing compilers [17][18][20].

This example also illustrates PARRET’s knowledge of mixed vector data types and how they can be resolved. Consider the edge labeled Edge I on Figure 21 and Figure 24. We chose the data type of the loop body calculations to be 16-bits (twice that of the input data, to handle the multiplies) and signed (because of the subtraction/negations). This means that the head of Edge I is 8-bit unsigned (the input type) and its tail is 16-bit signed. In a multimedia ISA (assuming 128-bit vector registers), the head type would have 16 elements and the tail type would have 8 elements. The semantics of the representation dictate that two instances of the node supplying data to Edge I (the array-read node) would have to execute for each instance of the node that receives data from Edge I (the negation node). A back-end code generator uses these relative execution ratios to generate the appropriate number of instances of data-parallel instructions for each operation.
**Code Synthesis**

**Target ISA: Intel’s Streaming SIMD Extensions 2**

Intel’s Streaming SIMD Extensions 2 (SSE2) \[3][4\] was chosen for analysis as a representative ISA with SIMD extensions. SSE2 is implemented in the Pentium 4 generation of x86 processors, facilitating analysis on commodity desktop workstations. It is supported at the assembly level by Microsoft’s Visual Studio, and at the source level, via vectorization, by Intel’s C/C++ Compiler \[16][17][46\] and Codeplay’s VectorC compiler \[51\].

SSE2 registers are 128-bits wide, allowing its instructions to operate on one of seven packed data types: 16 8-bit integers, 8 16-bit integers, 4 32-bit integers, 2 64-bit integers, 4 32-bit floating point values, 2 64-bit floating-point values, or a 128-bit bit vector. This work focuses on 8-, 16-, and 32-bit integer operations, though the methodology can be applied to the other data types as well. Table 4 summarizes the available integer arithmetic, logical, comparison, and shift operations defined for SSE2, based on the data types they operate upon. In addition, efficient memory operations are provided to move entire vectors to and from memory if the data is aligned to 16-byte boundaries; less efficient unaligned loads are stores are also available. Data shuffling and conversion operations are also available, respectively, to re-order data elements within vectors and to promote or demote data types.
Table 4. Available SSE2 instructions.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Datatype (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unsigned</td>
</tr>
<tr>
<td></td>
<td>8  16  32</td>
</tr>
<tr>
<td>addition</td>
<td>Yes  Yes  Yes</td>
</tr>
<tr>
<td>subtraction</td>
<td>Yes  Yes  Yes</td>
</tr>
<tr>
<td>multiplication</td>
<td>Yes</td>
</tr>
<tr>
<td>logical AND</td>
<td>(Available via their bitwise counterparts)</td>
</tr>
<tr>
<td>logical OR</td>
<td></td>
</tr>
<tr>
<td>logical NOT</td>
<td></td>
</tr>
<tr>
<td>bitwise AND</td>
<td>Yes  Yes  Yes</td>
</tr>
<tr>
<td>bitwise OR</td>
<td>Yes  Yes  Yes</td>
</tr>
<tr>
<td>bitwise XOR</td>
<td>Yes  Yes  Yes</td>
</tr>
<tr>
<td>bitwise NOT</td>
<td></td>
</tr>
<tr>
<td>right shift</td>
<td>Yes  Yes  Yes</td>
</tr>
<tr>
<td>left shift</td>
<td>Yes  Yes  Yes</td>
</tr>
<tr>
<td>greater-than</td>
<td></td>
</tr>
<tr>
<td>greater-than-or-equal</td>
<td></td>
</tr>
<tr>
<td>less-than</td>
<td></td>
</tr>
<tr>
<td>less-than-or-equal</td>
<td></td>
</tr>
<tr>
<td>equal</td>
<td>Yes  Yes  Yes</td>
</tr>
<tr>
<td>not-equal</td>
<td></td>
</tr>
</tbody>
</table>

Mapping MDDF Operations to SSE2 Instructions

Once the recognition front-end has recognized the parallel operations in code, the back-end must generate machine code using only the instructions available. When the front-end representation requires an instruction and data type combination not available in the actual instruction set, a pseudo-instruction must be used in its place. For example, SSE2 does not include a bitwise-NOT at the instruction level, however it can easily be created by an exclusive-OR (SSE2 instruction PXOR) of the input with a bit vector of all ones. The set of pseudo-instructions is either in-lined by the compiler during generation of assembly code or else is linked to the executable as part of a static or shared library.
Promotion and demotion of vector data types may be required when retargeting a loop that has mixed data types. For example, if a loop has operations between 32-bit and 16-bit unsigned integers (which result in, respectively, four element and eight element vectors, assuming SSE2), the eight 16-bit values (contained in a single register) must be promoted to two vectors of four 32-bit values so that corresponding elements can be properly matched. Sreraman et al. [18] discuss the mechanics of promoting one vector data type to another. We extend demotion beyond their use of vector packing instructions in the following section.

Data Type Demotion to Lower Precision

The symmetric operation to data type promotion is demotion, i.e., converting vectors of a higher precision data to a vector of lower-precision data. These conversions often arise when the inputs and outputs of a series of parallel operations are of the lower precision, but intermediate values may be higher precision (e.g., results of multiply operations). Demotion, of course, results in a loss of precision; we address this in typical packed-vector SIMD fashion by saturating the results if necessary.

For some cases, such as demotion of two signed 32-bit-by-4-element vectors to one signed 16-bit-by-8-element vector, SSE2 has the needed instruction, in this case PACKSSDW. SSE2 does not support operations to pack unsigned values, however. Figure 25 shows the implementation of an unsigned demotion of 32-bit vectors to a 16-bit vector. First, the values to be demoted are compared to see if they are greater than the saturation value for 16-bit unsigned integers, 0xffff (Figure 25 (a)). Each element of the result mask is all-ones if the result should be saturated, all-zeros if not. This mask is then added with saturation, as a 16-bit vector type, to the original input (Figure 25 (b)). Thus
values lower than the saturation point are passed through – in the appropriate location, with zeros padding to 32-bits. The “TRUE” values of the mask are effectively already saturated and are passed through as-is. Finally, a signed saturating pack is used to pack the 32-bit data into, here, the lower half of a 16-bit vector types (Figure 25 (c)). The signed saturation property ensures that values of 0xffffffff are correctly saturated to 0xffff.

<table>
<thead>
<tr>
<th>a0</th>
<th>a1</th>
<th>a2</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) \(\Rightarrow\) (unsigned, 32-bit)

| 0x0 | 0xffff | 0x0 | 0xffff | 0x0 | 0xffff | 0x0 | 0xffff |

\[
\]

| 0xffffffff | 0x0 | 0x0 | 0xffffffff |

(b) \(\Rightarrow\) (unsigned, 16-bit, saturating)

<table>
<thead>
<tr>
<th>a0</th>
<th>a1</th>
<th>a2</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 0xffff | 0xffff | 0x0 | a1 | 0x0 | a2 | 0xffff | 0xffff |

(c) (signed, saturating)

| 0xffff | a1 | a2 | 0xffff |

Figure 25. Demotion of an unsigned 32-bit vector type to the lower-half of an unsigned 16-bit vector type.
Code Generation for SSE2

Our approach to code generation for SSE2, particularly in the presence of mixed data types, is to generate code based on the highest-precision type. This could mean, for instance, that we load in 8-bit unsigned data, promote to 16-bit unsigned to perform higher-precision calculations such as multiplication, and then demote back to 8-bit unsigned for write-back, as must happen for the example of Figure 20 through Figure 24. The main constraint imposed by the architecture is that the parallel loads and stores must have the data packed consecutively, e.g., if all internal calculations are performed at 16-bit precision and the original program specifies output to an array of 32-bit integers, promotion must occur before write-back.

The first step is an analysis to determine the precision (in bits) and vector length (in number of elements, here, 128/precision) of all inputs, outputs, and the internal calculations. For our example of Figure 20, the input and output are both of unsigned char type in C, which has precision of 8-bits (unsigned) and a corresponding vector length of 16 under SSE2. We want to maximize for parallelism without sacrificing precision. Since the body of the loop contains multiplication operations, we choose the internal precision of the loop body to be twice that of the inputs, 16-bits, in order to keep the best possible precision from the multiply results. This requires that the inputs be promoted to a 16-bit/8-element type and that the outputs be demoted from that back to the 8-bit/16-element type. In this situation, the loop is effectively unrolled twice – a single 8-bit/16-element load is matched with two instances of every loop body calculation, which are 16-bit/8-element operations.
The second step is to generate a template script for the algorithm. The template specifies a valid partial ordering of all operations in the algorithm, with each operation annotated with its region information. A breadth-first traversal of the example MDDF graph is sufficient, generating the script of Table 5.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Type</th>
<th>Vector Length</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>array-read</td>
<td>8-bits</td>
<td>16</td>
<td>(0:cols*(rows-2)-2:1)</td>
</tr>
<tr>
<td>array-read</td>
<td>8-bits</td>
<td>16</td>
<td>(1:cols*(rows-2)-2:1)</td>
</tr>
<tr>
<td>array-read</td>
<td>8-bits</td>
<td>16</td>
<td>(2:cols*(rows-2)-2:1)</td>
</tr>
<tr>
<td>array-read</td>
<td>8-bits</td>
<td>16</td>
<td>(w:cols*(rows-2)-2:1)</td>
</tr>
<tr>
<td>array-read</td>
<td>8-bits</td>
<td>16</td>
<td>(w+2:cols*(rows-2)-2:1)</td>
</tr>
<tr>
<td>array-read</td>
<td>8-bits</td>
<td>16</td>
<td>(2<em>w:cols</em>(rows-2)-2:1)</td>
</tr>
<tr>
<td>array-read</td>
<td>8-bits</td>
<td>16</td>
<td>(2<em>w+1:cols</em>(rows-2)-2:1)</td>
</tr>
<tr>
<td>array-read</td>
<td>8-bits</td>
<td>16</td>
<td>(2<em>w+2:cols</em>(rows-2)-2:1)</td>
</tr>
<tr>
<td>integer-constant(2)</td>
<td>8-bits</td>
<td>16</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>integer-constant(2)</td>
<td>8-bits</td>
<td>16</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>integer-constant(2)</td>
<td>8-bits</td>
<td>16</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>integer-constant(2)</td>
<td>8-bits</td>
<td>16</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>integer-constant(0)</td>
<td>8-bits</td>
<td>16</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>integer-constant(0)</td>
<td>8-bits</td>
<td>16</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>integer-constant(255)</td>
<td>8-bits</td>
<td>16</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>integer-constant(255)</td>
<td>8-bits</td>
<td>16</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>multiplication</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>multiplication</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>multiplication</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>multiplication</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>negate</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>subtract</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>subtract</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>subtract</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>add</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>subtract</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>add</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>add</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>less-than</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>add</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>negate</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>select</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>add</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>greater-than</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>select</td>
<td>16-bits</td>
<td>8</td>
<td>(0:0:1)</td>
</tr>
<tr>
<td>array-write</td>
<td>8-bits</td>
<td>16</td>
<td>(1:cols*(rows-2)-2:1)</td>
</tr>
</tbody>
</table>
The last step is to produce the output SSE2 code as a C loop using the intrinsic functions corresponding to SSE2 instructions. Iterating over the lines of the template script, the first item is an 8-bit/16-element array-read. It specifies a region, (0:cols×(rows-2):2:1), of more than one element, so it is necessary to instantiate a for-loop of the form “for (i=0;i<cols×(rows-2)-2;i+=16),” i.e., iteration over the entire region, but with step equal to the vector length (strip-mining). The load is instantiated (SSE2 instruction MOVQ or intrinsic function _mm_loadu_si128()) inside the loop and its result assigned to a temporary variable. Since this row has a vector length twice that for the internal loop calculations, we next output two promotion operations to convert the lower and upper halves of the loaded value into 16-bit/8-element values and assign the outputs to temporary variables of the appropriate vector type (for all SSE2 intrinsic functions compiled with ICL, the 128-bit vector type __m128i is used). This process is repeated for all the array-read operations.

Each of the next 29 operations has region (0:0:1), i.e., a single element, meaning they are parallelizable over any non-singular region and thus inherit the region of the loop they are in. We output two lines of source code for each (one instance of the operation for each 8-element vector), filling in the appropriate temporary variables as inputs, assigning the output to a new temporary variable, and then moving to the next row. For constant inputs, the constant must be distributed into a vector form (scalar distribution). Here, the intrinsic function _mm_set1_epi16() is appropriate.

Finally, the array-write row is encountered. Since it has a precision and vector length different from that of the internal calculations, some adjustment must be performed. First, its input (16-bits) must be demoted to the write-back type (8-bits).
Since the loop body is operating on pairs of vectors, the demotion takes the resultant two 8-element vectors and creates a single 16-element vector, which is stored using the _mm_storeu_si128() intrinsic. The complete generated code is given in Appendix C.

**Validation and Results**

This section presents our experiments retargeting loops with inherent DLP to SSE2, a representative multimedia instruction set with SIMD functionality. We show PARRET is capable of synthesizing optimized code for SSE2 from the architecture-independent MDDF program representation.

As a baseline for comparison, we chose Intel’s C/C++ Compiler (ICL), version 8.0 [16][17][46]. It employs traditional loop vectorization to generate SIMD code as well as providing a set of intrinsic functions for inlining SIMD operations in C/C++ source code. Our analysis will make use of ICL’s vectorizing capabilities in characterizing whether or not applications can be parallelized using traditional methods. The intrinsic functions provide a convenient set of primitives for PARRET to use when generating its own data-parallel code.

Table 6 lists the loop-based applications in our test suite. These are taken from the TI IMGLIB [42] suite of image processing applications. All have been examined by hand and determined to be candidates for retargeting to a data-parallel architecture.
Table 6. IMGLIB test programs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Application(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv_3x3</td>
<td>Convolution of image with a 3×3 filter mask</td>
<td>Noise removal, image smoothing</td>
</tr>
<tr>
<td>corr_3x3</td>
<td>Correlation of a 3×3 reference image with the input image</td>
<td>Motion estimation</td>
</tr>
<tr>
<td>perimeter</td>
<td>Build image object perimeter</td>
<td>Object detection/recognition</td>
</tr>
<tr>
<td>pix_sat</td>
<td>Clip (saturate) pixels to ceiling value</td>
<td>Compression (i.e., clip values to a certain bit precision)</td>
</tr>
<tr>
<td>quantize</td>
<td>Quantize pixel values to a smaller range of discrete values</td>
<td>Compression (e.g., used in JPEG)</td>
</tr>
<tr>
<td>sobel</td>
<td>Object edge-detection</td>
<td>Object detection/recognition</td>
</tr>
<tr>
<td>threshold</td>
<td>All pixels above threshold value set to high value, all others set to low value</td>
<td>Pre-process for image dilation/erosion, perimeter detection</td>
</tr>
<tr>
<td>fdct_8x8</td>
<td>8×8 discrete cosine transform</td>
<td>Image compression (e.g., used in JPEG)</td>
</tr>
<tr>
<td>mad_16x16</td>
<td>16×16 minimum absolute difference</td>
<td>Video compression (e.g., MPEG)</td>
</tr>
</tbody>
</table>

Retargeting of Image Processing Programs to SSE2

Table 7 reports on the ability of PARRET and ICL to parallelize the test programs. For each of the programs, ICL was executed using the “/QxW /O3 /Qvec_report3” flags. These flags specify, respectively, Pentium 4 code generation (including vectorization for SSE2), speed optimizations, and detailed reporting on vectorization attempts.
Table 7. Suite coverage test.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PARRET</th>
<th>ICL</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv_3x3</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>corr_3x3</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>perimeter</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>pix_sat</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>quantize</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>sobel</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>threshold</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>fdct_8x8</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>mad_16x16</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Correctness Validation and Performance Evaluation

In order to validate correctness, each of the programs in the test suite retargeted by PARRET were executed for a 256-by-256 pixel input image and the results compared with those of a baseline execution of the original program as compiled by ICL. Average distance (Equation 2) was used to compare the matrix results of each trial, and these are listed in column 2 of Table 8. Here, lower values are better, i.e., the matrix results closely match.

\[
\text{avg \_ \_dist} = \frac{1}{MN} \sum_{n=1}^{N} \sum_{m=1}^{M} |a_{nm} - b_{nm}|
\]  

(2)

The mad_16x16 benchmark produced a scalar result, so average distance was not an appropriate measure. The scalar results for both the PARRET and ICL versions matched exactly.

The performance gains of using parallelized code generated by PARRET are shown in column 3 of Table 8. These were calculated as the time taken to execute the sequential version (as compiled by ICL) one hundred times divided by the time taken to execute the PARRET version one hundred times. Execution time was measured using

71
the \texttt{ftime()} function available from the Microsoft C/C++ runtime library (required for the Microsoft Windows version of ICL). All execution time tests were performed on a Pentium 4 1.80 GHz with 512 MB of RAM running Windows 2000.

Table 8. Correctness and Performance Results, PARRET vs. Sequential Execution.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average Distance</th>
<th>Vector Width</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv_3x3</td>
<td>0.0000</td>
<td>8</td>
<td>3.4299</td>
</tr>
<tr>
<td>corr_3x3</td>
<td>0.0000</td>
<td>8</td>
<td>1.9278</td>
</tr>
<tr>
<td>perimeter</td>
<td>0.1072</td>
<td>16</td>
<td>4.0920</td>
</tr>
<tr>
<td>pix_sat</td>
<td>0.0000</td>
<td>8</td>
<td>1.0940</td>
</tr>
<tr>
<td>quantize</td>
<td>0.0000</td>
<td>4</td>
<td>8.8130</td>
</tr>
<tr>
<td>sobel</td>
<td>0.0087</td>
<td>8</td>
<td>2.7154</td>
</tr>
<tr>
<td>threshold</td>
<td>0.0000</td>
<td>16</td>
<td>2.4436</td>
</tr>
<tr>
<td>fdct_8x8</td>
<td>0.0000</td>
<td>4</td>
<td>1.6447</td>
</tr>
<tr>
<td>mad_16x16</td>
<td>scalar result, exact match</td>
<td>4</td>
<td>26.7263</td>
</tr>
</tbody>
</table>

In several cases, notably \texttt{sobel}, \texttt{corr_3x3}, \texttt{perimeter}, \texttt{fdct_8x8}, and \texttt{quantize}, useful speedup over sequential execution was obtained with a small or non-existent deviation in computational results. The small deviations seen with \texttt{sobel} and \texttt{perimeter} are likely due to precision errors in the vectorized instructions, for example, using saturation arithmetic in the vectorized version.

Also of interest is the large speedup seen with the \texttt{quantize} and \texttt{mad_16x16} benchmarks. This \texttt{quantize} benchmark was originally written as an inner loop over sub-blocks of an array, and an outer loop over the elements of each sub-block, resulting in an inner loop stride greater than one. The 2-byte data array coupled with an inner loop stride of 16 (the number of quantization elements used) creates an effective memory stride of 32 bytes each iteration of the inner loop. L1 cache lines on the Pentium 4 are 64
bytes [52], meaning quantize required bringing a new cache line into L1 every two iterations, reducing the performance of the sequential version. PARRET, however, was able to recognize the loop nest as a sequence of operations on stride-1 data and generate the appropriate optimized code. The high performance gains seen in the mad_16x16 benchmark were likely due to the loop structure that resulted when PARRET generated its parallel code: it generated four copies each of 4-way parallel versions of the instructions needed to complete the calculations, effectively unrolling the loop entirely (the loop was written as a 16-iteration loop). This eliminated the loop overhead and several expressions based on the loop index (now a constant) would have been calculated at compile-time instead of run-time. The trade-off is a larger code size.

Table 9 lists the two benchmarks that ICL was able to vectorize, and their performance as compared to PARRET. An interesting result is that PARRET produced the same performance gain for threshold as the ICL vectorized version. PARRET performed worse than ICL for mad_16x16. From an examination of the generated assembly code, ICL was able to recognize a sum of absolute differences operation in the original C loop and replace that code with the special-purpose instruction PSADBW (“packed sum of absolute differences”), saving several instructions that PARRET had to generate explicitly. In the future, such recognition could be incorporated into PARRET by expanding its recognition pattern library. In addition, the intrinsic functions PARRET uses for optimized C code generation can result in as many as seven assembly instructions for a single operation instead of one or two that might result if code generation were performed at the assembly level. We used these intrinsic functions for
convenience only; future versions of PARRET could generate assembly code directly, producing a faster executable.

Table 9. Performance comparison of PARRET with vectorized benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PARRET Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>threshold</td>
<td>1.0000</td>
</tr>
<tr>
<td>mad_16x16</td>
<td>6.5419</td>
</tr>
</tbody>
</table>

**Code Size Evaluation**

The code-generation method presented here was expected to increase code size due to overhead of using pseudo-instructions, vector data type promotion and demotion, multiple instances of the same operation (to operate on separate parts of a vector), and the use of intrinsic functions. Table 10 reports the increase in code size of each retargeted loop (not the overall application) caused by PARRET. The average increase of 4.3 times could possibly be reduced by generating code at the assembly level and not by the intrinsic functions.
Table 10. Code Increase for PARRET-retargeted programs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PARRET size (bytes)</th>
<th>ICL size (bytes)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv 3x3</td>
<td>6291</td>
<td>1136</td>
<td>5.5379</td>
</tr>
<tr>
<td>corr 3x3</td>
<td>6194</td>
<td>1379</td>
<td>4.4917</td>
</tr>
<tr>
<td>perimeter</td>
<td>4352</td>
<td>995</td>
<td>4.3739</td>
</tr>
<tr>
<td>pix_sat</td>
<td>4695</td>
<td>1230</td>
<td>3.8171</td>
</tr>
<tr>
<td>quantize</td>
<td>4731</td>
<td>1202</td>
<td>3.9359</td>
</tr>
<tr>
<td>sobel</td>
<td>5595</td>
<td>1376</td>
<td>4.0661</td>
</tr>
<tr>
<td>threshold</td>
<td>4518</td>
<td>1701</td>
<td>2.6561</td>
</tr>
<tr>
<td>fdet 8x8</td>
<td>6915</td>
<td>1414</td>
<td>4.8904</td>
</tr>
<tr>
<td>mad_16x16</td>
<td>5668</td>
<td>1155</td>
<td>4.9074</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td>4.2974</td>
</tr>
</tbody>
</table>

ICL options: -c -Drestrict= -QxW -O3

Recognition Performance

Table 11 lists the size of each test program (in terms of the number of nodes in the input graph) and the amount of time recognition and code generation required. The timing was measured using the UNIX time command on a 4-CPU Sun E-450 server with 4 GB of RAM running SunOS 5.8. The results are comparable with those for retargeting to SIMPil (Chapter 2, Table 3), indicating that not only is the process efficient for the SSE2 target, but that the change in architectural target does not incur significant overhead. This is expected as the recognition is performed in an architecture-independent manner.
### Table 11. Recognition times for SSE2 retargeting.

<table>
<thead>
<tr>
<th>Program</th>
<th># Graph Nodes</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv_3x3</td>
<td>270</td>
<td>3.22</td>
</tr>
<tr>
<td>corr_3x3</td>
<td>131</td>
<td>2.22</td>
</tr>
<tr>
<td>perimeter</td>
<td>151</td>
<td>1.39</td>
</tr>
<tr>
<td>pix_sat</td>
<td>75</td>
<td>1.03</td>
</tr>
<tr>
<td>quantize</td>
<td>107</td>
<td>1.14</td>
</tr>
<tr>
<td>sobel</td>
<td>246</td>
<td>5.15</td>
</tr>
<tr>
<td>threshold</td>
<td>43</td>
<td>0.42</td>
</tr>
<tr>
<td>fdct_8x8</td>
<td>170</td>
<td>1.00</td>
</tr>
<tr>
<td>mad_16x16</td>
<td>34</td>
<td>0.19</td>
</tr>
</tbody>
</table>

### Conclusions and Discussion

PARRET is able to optimize loop-based code for multimedia ISAs beyond the capabilities of traditional vectorization, particularly in those cases where mixed-precision vector data types must be aligned for vector operations in the loop. Those properties which make PARRET effective are:

- Abstract dataflow representation of recognized algorithms,
- Ability to recognize more complex iteration spaces (e.g., the `quantize` benchmark) and normalize them into vectorizable form,
- Explicit representation of how mixed data types compose, and
- Support for data-driven conditional statements.

In the future, it would be interesting to integrate PARRET’s MDDF representation and recognition with traditional vectorization approaches; the MDDF representation and recognition is highly amenable to data-dependence analysis on which the traditional techniques are based. Similarly, the observation that using intrinsic functions possibly increases the number of assembly instructions required suggests that if
assembly was generated directly by PARRET (eliminating extraneous instructions), even more speedup could be obtained. Other work will focus on extending the recognition system for an even wider range of loops, in particular, analysis of deeper loop nests and fusing adjacent loops into a unified abstract algorithm.
CHAPTER 4

LIGHTWEIGHT ESTIMATION OF DATA-PARALLEL POTENTIAL

Abstract

Retargeting to data parallel mechanisms is difficult, but can provide significant increases in efficiency. It is desirable to estimate potential parallelism before undertaking the expensive process of reverse engineering and retargeting. The goal is to narrow down the search space to a select set of loops which have a high likelihood of being data-parallel. This work presents a hybrid static/dynamic approach, called DLPEST, for estimating the data-level parallelism in sequential program loops. Loop data-dependence distance is measured using code instrumentation and run-time profiling. Loop nesting information is collected from a static analysis of the program. These values are then used to estimate lower and upper bounds on available data-level parallelism for each loop. We demonstrate the correctness of the DLPEST’s estimates, show that estimates for programs of 25 to 5000 lines of code can be performed in under 10 minutes and that estimation time scales sub-linearly with input program size.

Introduction

Processor designers have spent over 30 years creating clever methods to execute inherently sequential machine instructions in parallel. But few automatic retargeting tools exist for data parallelism. Since companies are reluctant to abandon significant sequential code assets, programs must be reverse engineered and retargeted for explicitly parallel execution mechanisms. This often manual process is time consuming and
expensive and potential performance improvements are difficult to estimate until lengthy, often undocumented programs are reverse engineered.

The automated recognition approach described in Chapters 2 and 3 will assist in this reverse engineering process, but must be applied selectively to programs or code blocks with high potential for data parallelism. Low-cost techniques for eliminating non-data-parallel code prior to a retargeting activity will facilitate the overall retargeting process, by focusing a parallelization tool (whether based on program recognition or vectorization) only on those code fragments with a high expectation of being data-parallel. These techniques will not only aid the reverse engineering process, but could also be used to focus more traditional vectorization-related analyses, such as Fourier-Motzkin elimination or the Omega test [28].

This chapter presents a new method, combining a syntactic analysis of the code with a code-instrumentation and dynamic profiling technique (i.e., the executable is instrumented to collect its own profile information) for detecting regions of program text that are likely to be data-parallel. It is shown to be lightweight in both time and memory requirements compared to a previous technique representative of simulation-based approaches.

Related Work

Program Performance Estimation

Chabini et al. [53] describe a system for estimating program execution time. Their technique begins by converting a source program (written in C) into a control data flow graph (CDFG). A CDFG embeds data flow graphs (DFGs) – created from the basic blocks of the program – within the nodes of a control flow graph (CFG). Each node in a
DFG is assigned a latency based on the cycle time of its analogous machine instruction. Then, the longest and shortest paths through the DFG are found, and the corresponding path latencies (as best- and worst-case execution times) are found for the DFG. Finally, upper and lower bounds on the program execution time are found by multiplying each DFG’s best- and worst-case latencies by some execution weight for the DFG’s enclosing CFG node. For example, if a loop structure in the CDFG graph is known to execute one hundred times, its corresponding weight would be one hundred.

The approach taken by Allara et al. [54] is similar. Instead of creating a CDFG, however, their technique examines the program statement-by-statement. For example, for each assignment statement, the operations involved are assigned some weight. The estimates for program execution times are calculated by finding the shortest and longest paths through the program.

The approach taken by Ghazal et al. [55] expands on the static techniques presented above. Instead of statically determining loop weights, however, some program execution is performed to more accurately model the loop effects. Further, their technique allows compiler optimizations to be applied to the source, more accurately modeling their effects in the estimate.

Stolberg et al. [56] instrument C code with profiling information and compile the annotated code. An execution of the resulting binary builds a “complexity profile” of the frequencies of program tasks and the time each is required to execute. Summing the products of the frequencies and measured execution times produces the estimate of execution time.
Bjuréus and Jantsch [57] dynamically generate a trace of assembly instructions executed by a program. They then apply a static analytical model to determine the total execution time based on the latencies of the individual instructions encountered. By varying the latencies of the instruction types, estimates can be calculated for many different architectures.

As these techniques work well for identifying portions of code ("hotspots") that consume large fractions of the overall execution time, they could be used for loop selection for parallelization purposes. They make no distinction, however, between time-consuming loops that can be parallelized and those that cannot, which would result in a larger number of loops selected for parallelization that would not be parallelizable. Our technique makes this distinction by measuring parallelism (in the form of loop dependence distance) as part of the selection process. We do, however, incorporate the idea of weighting portions of code based on their execution properties.

**Parallelism Estimators**

Larus [58] presents a system targeted at recognizing loop-level parallelism. Using an idealized parallel machine model (no limit on processors, etc.) it seeks to find an upper bound on the available program parallelism. The system scans through an externally generated program trace. When it identifies a loop, it tracks register- and memory-based data dependencies for the loop. The parallelism estimate is computed from the number of loop iterations without loop-carried dependencies. The system also offers the option of ignoring loop-carried data-dependencies, which provide insight into the amount of parallelism available if a compiler can recognize "false" loop-carried dependencies and remove them. We incorporate their tracking of loop-carried dependencies, but at the
source-code level instead of the assembly-level. This allows us to distinguish between array- and pointer-based memory references (which are more likely to cause dependencies that foil parallelization) and memory references to scalar program variables, which can often (as in the case of loop induction variables) cause loop-carried dependencies despite being parallelizable.

Kumar [59] offers a source-code instrumentation scheme for Fortran programs. The added code allows the program to self-compute the earliest time slot for each source code statement that would complete execution on a parallel machine with unlimited resources, based on satisfying data dependence constraints (i.e., the statement’s variable and/or memory reads) and control constraints (e.g., a statement inside an if-then body cannot execute before the conditional is resolved). Under this model, potentially many statements can fall into the same time slot, yielding a parallelism metric. We perform source-code instrumentation as well, but focus only on calculating dependence distance which requires only counting the number of iterations for which a dependency does not exist. This avoids the complexity of tracking every memory access and when its values would be ready for consumption in an ideal machine.

This research expands on existing work by presenting a technique that is lightweight in both execution time and memory required, is explicitly targeted at data-parallel execution, and tailors the ranking of selected loops based on execution weights and the type of architecture selected.
Methodology

Figure 26 gives an overview of the estimation process. Dynamic profiling measures the total number of iterations executed by each loop in the program and, for inner loops, the inter-iteration dependence distance. Static analysis is used to build a model of how loops are nested within the program. Call-graph information is used to expand the static model across function boundaries, e.g., if loop \( L1 \) appears in function \( F0 \) and loop \( L2 \) calls function \( F0 \), then the model understands this as a loop nest with \( L1 \) as the inner loop and \( L2 \) as the outer loop. The combination of dynamic and static information produces an estimate of the amount of data-level parallelism – as the number of loops that can be executed in parallel – that may be available for each loop nest under consideration. This estimate is then used as the loop selection criterion.

![Diagram](image)

**Figure 26.** Estimation overview.
Dynamic Profiling

The key value in data-parallel estimation is *loop dependence distance*, defined as the number of iterations between the source of a dependency and its target [45]. For example, if a write to memory location $m$ occurs in iteration $j$ and a read from $m$ occurs in iteration $j+4$, iteration $j$ must execute before $j+4$; the net loop dependence distance in this case is four. The loop dependence distance represents an upper bound on the number of loop iterations that can be executed in parallel. The exception is a loop with a dependence distance of zero, meaning the only dependence is within the same iteration, i.e., the loop is fully parallelizable.

A dynamic profiling approach incorporating ideas from [59] (code instrumentation for native execution and self-profiling) and [58] (dependence distance measurement) is used to measure loop dependence distance during program execution. C source code is automatically annotated with profiling code that tracks dependence distance for each loop. It associates a data structure $Prof := \{Wr,Rd,Dd\}$ with each loop, where $Wr$ is the set of all memory addresses written to since the first iteration of that loop, $Rd$ is the set of all memory addresses read in the current iteration, and $Dd$ is the loop dependence distance calculated as of the current iteration. The algorithm to calculate dependence distance for a given loop is:
\(Wr = \emptyset;\)
\(Rd = \emptyset;\)
\(Dd = 1;\)

for each iteration of loop \(L:\)
\(Rd = \{\text{all addresses read in } L\};\)
\(Collision = Wr \cap Rd;\)
if \(Collision = \emptyset\) then
\(Wr = Wr \cup \{\text{all addresses written in } L\};\)
\(Dd = Dd + 1;\)
else stop;

This algorithm compares addresses read in the current iteration with those written in all past iterations of the loop. Addresses are recorded for data types reported as arrays or pointers by the original C code. If the intersection of the two sets is the empty set, no dependence has been found for the current iteration, so increase the measured dependence distance and continue, otherwise, end and use the last dependence distance. Note that if multiple loop-carried dependencies exist, this technique will detect the one with the shortest dependence-distance, as this is the dependence constraining parallelization.

Consider the example of Figure 27, where each row of the table represents a snapshot of \(Prof\) during the current iteration. The profiler records all memory addresses written by the loop up through and including the current iteration \(i\), shown in the second column. This list of memory writes is compared with the addresses of all memory read accesses of the current iteration (third column). A true loop-carried dependence is detected when a read address in the current iteration matches an address in the cumulative set of write addresses. The dependence distance for the loop is increased by one (fourth column) for every iteration until a dependence is found, at which time the maximum dependence distance for the loop is known and measurement ceases for the loop. In the example of Figure 27, a dependence is detected in iteration 4 when address 504, written
in iteration 0, is read. The final dependence distance of 4 is the value recorded in the prior iteration.

```c
void main() {
    int i, A[100], B[100];
    for(i=0; i<96; i++) {
        A[i+4] = 5*A[i];
    }
}
```


<table>
<thead>
<tr>
<th>Iteration (i)</th>
<th>Addresses written, iterations 0 through i</th>
<th>Addresses read, this iteration</th>
<th>Dependence distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>({504,700})</td>
<td>({500,706})</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>({504,505,700,701})</td>
<td>({501,707})</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>({504,505,506,700,701,702})</td>
<td>({502,708})</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>({504,505,506,507,700,701,702,703})</td>
<td>({503,709})</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>({504,505,506,507,508,700,701,702,703,704})</td>
<td>({504,710})</td>
<td>4</td>
</tr>
</tbody>
</table>

**Figure 27. Example of dynamically measuring dependence distance.**

Note that loop dependence distance is measured only in terms of read-after-write (RAW, or true) dependencies caused by memory access operations. Write-after-read (WAR, or “anti”) dependencies and write-after-write (WAW, or “output”) dependencies are not tracked since a compiler could eliminate these with techniques such as alpha renaming. The other dependence situation that could arise is a RAW, WAR, or WAW dependence caused by a scalar (non-array) program variable, since the address of the variable remains the same for every iteration. In many cases, these dependences can be
eliminated using a DFG-based retargeting system. For example, a variable accumulating the result of a vector sum operation will cause a loop-carried dependence. Similarly, any induction variables for the loop will necessarily cause a loop-carried dependence because of their update operation. Both cases can be abstracted into a dependency-free form during retargeting, thus the loop-carried dependencies caused by scalar variables are not tracked.

The primary time and space requirements for this technique are the memory required to store the cumulative set of write addresses and the corresponding time required to search the set. For inner loops (those that do not contain another loop in their body, including those transitively containing another loop via a series of procedure calls), a threshold $T_L$ is used to limit the dependence distance; if analysis shows that a loop’s dependence distance could be larger, it is assumed to be fully parallel and the analysis is halted after $T_L$ iterations of the loop. This constrains the maximum memory required to analyze each loop to be $T_L \times N_W$, where $N_W$ is the number of static write operations in the loop body, i.e., each iteration will generate a maximum of $N_W$ new dynamic addresses from the static write operations and these addresses are inserted into the set of write addresses. Without the threshold constraint, a fully-parallel loop would record every address written by every iteration (since the exit condition of a detected dependence would not cause an early halt to measurement). All other tracked variables ($Iter$, $Rd$, and $Dd$) remain constant in memory usage from iteration to iteration.

The threshold $T_L$ is chosen based on the current (or projected) state of processor technology with respect to DLP-exploiting ability. For multimedia ISAs capable of
speedup factors in the tens, or multi-processor SIMD architectures in the hundreds to thousands, a measuring dependence distance for a few hundred iterations is sufficient.

In addition to measuring dependence distance, the total number of iterations each loop executes is recorded. This measurement is performed for every loop, regardless of whether it is an inner loop or not. This information is used by the loop selection heuristics.

**Static Analysis**

For non-inner loops, dependence distance measurement is not performed as this would require tracking too many addresses. (Not only must the outer loop addresses be tracked, but also every address generated by the inner loop which is executed for every iteration of the outer loop.) Instead, information provided by the static analysis is used to model the effect of these loops on the aggregate nested structure.

The static analysis extracts loop nesting information. Inter-procedural analysis is performed to reveal loop nests that occur transitively via procedure calls. The product is a database of the following relationships:

1. `loop_in_loop(n,m)`: Specifies that loop $m$ is, in the abstract syntax tree (AST) of the program, a parent node of loop $n$, i.e., $n$ is an inner loop to $m$, and both are defined within the same procedure.

2. `loop_in_procedure(n,m)`: Loop $n$ is defined inside procedure $m$. Here, only the outermost loops in $m$ will be recorded using `loop_in_procedure`; inner loops are specified transitively via `loop_in_loop`.
3. $call_{in\_loop}(n,m)$: A call statement $n$ appears inside loop $m$, but not within a loop contained in $m$.

4. $call_{in\_procedure}(n,m)$: A call statement $n$ appears inside procedure $m$, but not within a loop contained in $m$.

By following the relationships among procedures and loops defined above, loops nested within an arbitrarily long chain of procedure calls can be extracted. Figure 28 (a) shows code with loops and procedure calls and Figure 28 (b) shows the call chain those relationships describe. A graph traversal identifies the system as a doubly-nested loop nest, even though the loop statements are in separate procedures.
Figure 28. Example loop nesting analysis. (a) Source code, (b) inter-procedural loop nesting relationships.
**Loop Selection**

A ranking function is used to order loops based on their appropriateness for data-parallel retargeting. Along with the dynamic loop information (loop dependence distance and iteration count) and the static nesting information, the number of parallel functional units available on an architecture is used in the ranking function.

A large measured loop dependence distance implies a significant amount of exploitable parallelism and should be given a high ranking. If, however, loop dependence distance is small, i.e., on the order of the vector width of a multimedia ISA, the loop would only be a candidate for this kind of architecture (and not a large-scale SIMD processor). Additionally, it would only be a candidate if the parallelized version of the loop (i.e., strip-mined to execute multiple iterations in parallel) executes a large number of times. For example, several loops in the MPEG2 encoding program from the MediaBench suite had exactly eight iterations with a loop dependence distance of eight (fully parallel). Examined in isolation, these would not be good candidates for retargeting, as the speedup would be small in relation to the overall program. However, if such a loop were embedded in another loop that executed 100 times, then the overall fraction of execution time affected by parallelizing the inner loop becomes significant, making it a parallelization candidate.

With the above observation in mind, the rank function should be based on the number of loop iterations that can be executed on parallel hardware. This ensures that loops that may technically be data-parallel (exhibiting no data-dependences) are not considered for retargeting if they only execute a few iterations total. Thus the loops are selected and ranked according to the following criteria:
1. If the dependence distance of the loop is less than the number of parallel resources (e.g., number of processor nodes, vector width) on the target machine, it is removed from consideration.

2. Loops that satisfy the first criterion are ordered according to the size of the iteration space they cover. In the case of an inner loop, this is simply the number of iterations measured dynamically. If static analysis reveals nesting, this is the number of inner loop iterations multiplied by the number of outer loop iterations.

**Experimental Results and Validation**

The dynamic profiler is implemented by automatically annotating C source code with profiling code. The annotated source is compiled and executed normally, producing a file of results. Both the annotation and static analysis tools are based on the Stanford University Intermediate Format (SUIF) compiler system [60]. The complete annotation, static analysis, and dynamic profiling system is called DLPEST (Data-Level Parallelism ESTimator).

This section presents experimental results that

1. validate the applicability of our method for estimation of data-parallelism,

2. show that our method is quantitatively faster than a previous simulation-based approach while producing better results, and

3. demonstrate the scalability of our method.
Methodology Validation

A set of benchmark applications from the Texas Instruments (TI) IMGLIB suite [42] was used to validate our data-parallel estimation method.

The first validation test was a comparison of the estimation results of DLPEST profiling with the expected estimation results. The expected results were calculated by manual examination of the source code to determine data-dependence distances for each loop.

Table 12 compares the expected and measured results (here, the “(a)” and “(b)” indicate that certain benchmarks contain two loops, so a separate entry exists for each one). For all benchmarks except histogram (a) the measured and expected results match exactly. In the case of histogram (a), the expected value was impossible to predict due to non-linear array references (an indexed array was used as the index to another array).
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Nesting Depth</th>
<th>Estimated Dependence Distance</th>
<th>Measured</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundary</td>
<td>2</td>
<td>65536</td>
<td>65536</td>
<td></td>
</tr>
<tr>
<td>conv_3x3</td>
<td>2</td>
<td>65024</td>
<td>65024</td>
<td></td>
</tr>
<tr>
<td>corr_3x3</td>
<td>1</td>
<td>65536</td>
<td>65536</td>
<td></td>
</tr>
<tr>
<td>errdif_bin</td>
<td>2</td>
<td>65536</td>
<td>65536</td>
<td></td>
</tr>
<tr>
<td>dct (a)</td>
<td>2</td>
<td>8192</td>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>dct (b)</td>
<td>1</td>
<td>8192</td>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>idct (a)</td>
<td>2</td>
<td>8192</td>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>idct (b)</td>
<td>2</td>
<td>8192</td>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>histogram (a)</td>
<td>1</td>
<td>16384</td>
<td></td>
<td>nonlinear</td>
</tr>
<tr>
<td>histogram (b)</td>
<td>1</td>
<td>256</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>mad 16x16</td>
<td>2</td>
<td>256</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>mad_8x8</td>
<td>2</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>median 3x3</td>
<td>2</td>
<td>65024</td>
<td>65024</td>
<td></td>
</tr>
<tr>
<td>perimeter</td>
<td>2</td>
<td>65536</td>
<td>64516</td>
<td></td>
</tr>
<tr>
<td>pix sat</td>
<td>1</td>
<td>65536</td>
<td>65536</td>
<td></td>
</tr>
<tr>
<td>quantize</td>
<td>2</td>
<td>65536</td>
<td>65536</td>
<td></td>
</tr>
<tr>
<td>sad 16x16</td>
<td>2</td>
<td>256</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>sobel</td>
<td>1</td>
<td>65022</td>
<td>65022</td>
<td></td>
</tr>
<tr>
<td>thr_le2thr</td>
<td>1</td>
<td>65536</td>
<td>65536</td>
<td></td>
</tr>
<tr>
<td>wave_horz (a)</td>
<td>2</td>
<td>1024</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>wave_horz (b)</td>
<td>2</td>
<td>1024</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>wave_vert (a)</td>
<td>2</td>
<td>2048</td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>wave_vert (b)</td>
<td>2</td>
<td>2048</td>
<td>2048</td>
<td></td>
</tr>
</tbody>
</table>
The results for IMGLIB were also compared with an approach [29] representative of simulation-based techniques. This technique models all available types of parallelism—thread-level, instruction-level, and data-level—using a modified version of the SimpleScalar simulator [43]. After executing each machine instruction, the modified simulator places the instruction into a schedule grid (where columns are time slots and rows are instructions scheduled in parallel into the time slots) based on its dynamic data-dependency constraints. This schedule modeled a processor with theoretically unlimited (but practically limited) hardware resources (functional units, registers, memory, etc.). Each instruction placed into the schedule was scheduled as early as possible, i.e., one time slot after the producers of its operand registers were scheduled (all instructions were assumed to have a latency of a single time slot). Value prediction and perfect branch prediction were used to measure the maximum theoretical DLP possible. A parallelism metric was formed by dividing the total number of instructions exhibiting parallelism (i.e., those that occur more than once in a given time slot) by the total number of instructions in the program.

Figure 29 shows the results of each method for the TI IMGLIB library. The statistical correlation between the two result sets was calculated as 0.4164, a moderate positive correlation. Note that this comparison required correlating the results of individual program loops, so only programs with one or two easily-identifiable loops were compared. Larger programs with many loops (such as those in Mediabench) made matching and comparing the results for individual loops difficult. Also note that the comparison is between two different types of metric: the number of parallel loop iterations (DLPEST) and the number of parallel machine instructions (simulation) for
each loop. The machine instructions, however, are grouped by their program counter (PC) value, i.e., only multiple instructions in a time slot with the same PC are measured as “parallel.” Under this model, parallel instructions indicate that a static program instruction has no loop-carried dependence (including transitive dependencies) with itself, measuring dependence distance at a finer granularity, hence the positive correlation with DLPEST’s dependence distance measurements. The simulation method aggregates these measurements for many groups of parallel instructions, some which can be larger than the dependence distance and some of which can be smaller, resulting in a correlation with DLPEST of less than one.

Benchmarks such as boundary and histogram (a) match closely between the two techniques. Those where the simulation technique reports higher parallelism, such as wave_horz (b), likely point out its strengths in finding data-parallel instructions past dependence boundaries. For example, if iteration $i+1$ is dependent on iteration $i$, the simulator can still pick out individual instructions within each iteration that are not dependent and mark them as parallel. On the other hand, the simulator falls short on benchmarks such as perimeter and pix_sat, most likely because it is limited in the number of machine instructions it can practically schedule at any given time (based on the memory requirements needed for the schedule data structure). DLPEST only records memory addresses and dependence distance and thus can record information about many more iterations than the simulator.
Figure 29. Estimate correlation (TI IMGLIB) between DLPEST and the Simulation-based method
To validate the quality of DLPEST’s results in terms of number of program loops covered, we evaluated its coverage of the MPEG2 encoding benchmark from the MediaBench suite. This program was chosen as it has a large number of candidate loops (101) of which a significant number (50) were found by a manual examination to be data-parallel. Table 13 reports DLPEST’s coverage of this program and compares it with the coverage of the simulation-based method. The upper portion of Table 13, labeled “Overall,” examines DLPEST’s overall ability to locate data-parallel loops. Since DLPEST incorporates dynamic profiling, this means some loops may be not be selected since they may not actually be executed during profiling. The lower portion of Table 13, labeled “Normalized,” reports on DLPEST’s loop-selection ability when only those loops that are executed are considered.

<table>
<thead>
<tr>
<th>Metric</th>
<th>DLPEST</th>
<th>Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td># loops selected</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td># false positives</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td># false negatives</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td># data-parallel loops correctly identified</td>
<td>41</td>
<td>31</td>
</tr>
<tr>
<td>coverage</td>
<td>82%</td>
<td>62%</td>
</tr>
</tbody>
</table>

**Normalized**

<table>
<thead>
<tr>
<th>Metric</th>
<th>DLPEST</th>
<th>Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td># false negatives due to loops not executed</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>net false negatives</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td># data-parallel loops correctly identified</td>
<td>41</td>
<td>31</td>
</tr>
<tr>
<td>coverage</td>
<td>98%</td>
<td>74%</td>
</tr>
</tbody>
</table>

Total program loops (inner): 101
Total data-parallel program loops: 50
Total data-parallel program loops executed: 41
**Execution Performance**

A comparison of estimation times, measured using the UNIX `time` command, for DLPEST and the simulator across the IMGLIB and Mediabench suites is given in Figure 30. This timing was performed on a 4-CPU Sun E-450 server with 4 GB of RAM running SunOS 5.8. DLPEST outperforms the simulator – often by one to three orders of magnitude – in all but two cases (`epic-unepic` and `jpeg-djpeg`) and the performance gains of the simulator in those cases is small. Figure 31 shows the performance gains of DLPEST in terms of its speedup over the simulator. Across the suite, the average speedup is 334.14.
Figure 31. DLPEST vs. simulator speedup comparison.
Scalability

Figure 32 decomposes the operation of DLPEST based on its stages of operation:

1. \textit{C2SUIF}: Conversion of C source files to the SUIF [60] intermediate format.
2. \textit{LINK}: Link all SUIF files into one aggregate by resolving their symbol tables; this insures the full call structure of the program can be analyzed statically.
3. \textit{ANNOTATE}: Perform the loop annotation and output the modified C source code.
4. \textit{COMPILE}: Compile the modified C source code.
5. \textit{EXECUTE}: Native execution of the modified program.

From Figure 32, it is evident most of the estimation time is spent in the \textit{LINK} and \textit{ANNOTATE} stages, meaning program size is most likely the primary characteristic governing estimation time. A comparison of program size, in terms of source lines of code (SLOC, which excludes all non-executable source text such as comments and blank lines), to estimation time is given in Figure 33. The slope $m$ of the line in Figure 33 (using a linear regression) is 0.87 and the y-intercept $b$ is -0.66. Since this is a log-log scale graph, solving $\log y = m \log x + b$ yields the curve $y = 0.22x^{0.87}$. Based on the data, DLPEST scales better than linearly with respect to input program size.
Figure 32. DLPEST timing decomposition.
Figure 33. Scaling of DLPEST performance with program size.
Conclusions and Discussion

As part of a development cycle, techniques that can estimate program characteristics such as execution time, energy consumption, and parallelism are invaluable in making early decisions to prune the design space, choosing appropriate compiler transformations to get the best optimization strategy, or, in a reverse-engineering task, limiting the focus of recognition to the appropriate sections of code. For the estimation process to be useful, however, it must provide a savings in terms of development time or computational resources. This chapter has presented a technique for lightweight estimation of DLP in program source code.

We have shown that DLPEST produces results comparable to those of earlier simulation-based data-level parallelism estimation techniques, with performance increases of over 300 times on average. The technique scales sub-linearly with program size, making it suitable for a large range of programs.
CHAPTER 5

CONCLUSION

This dissertation has presented work on retargeting sequential multimedia applications to architectures with data-parallel capabilities. Methods have been presented to retarget to multi-processor SIMD architectures as well as single-processor architectures incorporating multimedia ISA extensions with SIMD capability. To support the overall retargeting process, a method for estimating data-level parallelism in applications has also been presented. The remainder of this chapter summarizes these methods and the experimental results obtained. It describes areas of future work and how the research can be applied to other problems beyond data-parallel retargeting.

Extracting an Explicitly Data-Parallel Representation of Image-Processing Programs

Chapter 2 describes the implementation of a system, called PARRET, for retargeting sequential multimedia programs to data-parallel architectures. The source code of a program is first converted to a dataflow graph representation. Next, graph pattern-matching is applied to recognize data-parallel memory access patterns, producing an explicitly data-parallel MDDF representation of the program. This multidimensional dataflow representation is then used to generate data-parallel code. Here, an extended version [22] of ZPL [21] was used as the target language so that an existing compiler could be used to generate code for SIMPil [12][13][14], a representative target SIMD architecture. A comparison between execution of the retargeted programs and a baseline sequential execution of those programs showed speedups of 30,000 times are possible by retargeting for SIMPil execution with only a small reduction in the quality of the results.
Future work in this area includes extending the abilities of PARRET, i.e., allowing for more complex loop nests, incorporating patterns for non-linear memory accesses (e.g., the increasing strides that occur with pyramid encoding), and being able to recognize exclusive array regions, e.g., regions such as border areas of an image that may undergo a different set of operations than those on the interior of an image. PARRET could also be extended to retarget data-parallel code to multiple-input/multiple-data (MIMD) systems, such as symmetric multi-processor systems or distributed systems.

**Retargeting Data-Parallel Code To Multimedia Instruction Set Architectures**

Chapter 3 extends the retargeting work to cover processor architectures that include multimedia instructions with SIMD functionality, demonstrating PARRET’s architectural flexibility. These processors achieve data-level parallelism by packing multiple data values into long (e.g., 128-bit) registers, allowing a single instruction to operate on several data values simultaneously. Challenges (e.g., mixed data types) to traditional parallelization methods (vectorization) are discussed, and the effectiveness of PARRET’s explicitly-parallel program representation for overcoming these issues is demonstrated.

PARRET was applied to a suite of programs and code generated for the SSE2 extensions to the Intel x86 architecture (Pentium 4). Execution speedups of 2 to 27 times over sequential due to retargeting were shown. The correctness was validated by comparing the outputs of the retargeted programs with those of the sequential baseline; most errors were small or zero. Further, it has been shown that PARRET can retarget a wide range of programs due to its deep understanding of data types and operations (including multidimensional mixed types and masking operations) required for successful
program retargeting, as well as its ability to tightly connect producers and consumers of data via dataflow graph semantics.

Future work in retargeting data-parallel programs to multimedia ISAs includes better code-generation, i.e., at the assembly level instead of using intrinsic functions. In particular, some multimedia ISA extensions contain instructions for optimizing cache accesses, such as prefetch instructions; with its knowledge of memory layout and access patterns, PARRET would be a good candidate for leveraging these instructions in compiled code. Finally, it would be interesting to apply PARRET to two-dimensional multimedia ISA targets such as MOM [47] and CSI [61]. These would be good retargeting candidates for PARRET due to its ability to recognize DLP in multiple dimensions.

**Lightweight Estimation of Data-Parallel Potential**

Chapter 4 describes a method to filter non-data-parallel source code from a program, leaving PARRET with only those portions of code which have a high likelihood of being data-parallel. A lightweight estimation process is presented for estimating the degree of data-level parallelism in program loops. The method automatically annotates loops with instrumentation code at the source level. The modified source is then compiled and executed natively on a desktop processor to produce a profile of dependence-distance information for each loop. A static pass over the program source collects data about loop nesting. Together, the nesting model and dependence-distance profile are used to rank program loops with respect to its applicability for parallel retargeting. The results show the correctness of this method, its increased performance
— on average, 334 times faster over earlier simulation-based techniques – and that it scales sub-linearly with input program size.

In the future, the dynamic estimation portion of this technique should be extended to better estimate loop dependence for arbitrary loop nests. This would have applications for supporting the appropriate choice of parallelizing loop transformations at compile-time, estimating communication patterns in multi-processor systems, as well as improving the ability to retarget to DLP architectures (such as SIMPil [13]) with two-dimensional (or better) parallelism resources.

Knowledge Transfers

This research benefits from work in other areas, for example, the MDSD model of computation was not originally intended for modeling data-parallel programs but has been modified here for that purpose. This section describes areas of research outside of automated data-parallel program retargeting that could benefit from the contributions of this dissertation.

Program Modeling and Visualization

In situations where automated parallelization is not possible, the MDDF representation described here still has applications for modeling and visualizing programs. Such a visual model could facilitate interactive parallelization where a designer could modify and optimize code based on suggestions made by the development environment. The program model can also provide implementation-independent documentation of the algorithm, which can be referred to by a software maintainer as the target platform and programming languages evolve.
Visual Programming Languages

Though not textual, the MDDF representation is still a high-level programming language, given it is descended from other visual specification and implementation languages, e.g., SDF and MDSD. With an appropriate development environment it could be used for the design and implementation of data-parallel programs as part of formal data-parallel design process as proposed by [32].

Partial Program Recognition

A partial recognition approach was key to extracting a parallel program representation with PARRET. In this case, it resulted in both a large coverage of algorithms with a small pattern library. This provides a benefit over past pattern-based program recognition systems which attempted to recognize the entire program as a single named entity and require larger pattern libraries for increased program coverage. The only constraint in this partial recognition approach is that recognition successfully abstract over the program details that hinder the program maintenance task (retargeting, documentation, etc.). This approach can be applied in domains other than data-parallel program retargeting, for example, recognizing independent threads in sequential code by focusing on the detection of synchronization points.
APPENDIX A

SIMD ARCHITECTURES

Single-instruction, multiple-data (SIMD) computer architectures achieve parallelism by applying the same operation across multiple elements of data. An example would be the element-by-element addition of two arrays of length $N$, as shown in Figure 34. To perform the same calculation on a typical sequential (or superscalar) processor, the processor would have to perform each of the $N$ additions separately. A SIMD architecture with adequate hardware resources (functional units, registers, etc.) could potentially perform all $N$ additions simultaneously, yielding a potential net speedup of $N$ over the sequential processor. When a program or algorithm describes such operations that are applied, independently, over a large set of data, this is referred to as data-level parallelism (DLP). SIMD architectures are designed to improve the performance of applications with large inherent DLP, such as image processing and scientific algorithms.

![Diagram showing data-parallel addition](image)

**Figure 34.** Data-parallel addition.
Multi-Processor SIMD

Multi-processor SIMD architectures are characterized by an array of processing elements (PEs) that are each assigned some sub-set of data from the input set. A separate control unit broadcasts instructions to all PEs, which operate in lock-step (i.e., every PE executes the same instruction at the same time). Lock-step operation allows each PE to have simple decoding and control logic, since the bulk of the instruction decoding and branching is taken care of by the control unit; similarly, a single decode results in many instances of the instruction being executed (one for each PE), reducing the number of decodes required to process a large data set. Each PE may have a dedicated register file, or may share with other PEs. Similarly, memory may be local to each PE, or may be global to all PEs. Architectures in this class include Thinking Machines Corporation’s CM-2 Connection Machine [62], MasPar Corporation’s MP-1 [63][64], the Kestrel processor [65], IMAP[9][10], P3I [11], Xetal [8], and the SIMPil focal-plane image processing architecture [12][13][14].

Communication in multi-processor SIMD architectures is provided by a connection network among PEs. The connectivity varies by architecture, but typically the network is laid out such that neighboring PEs (i.e., those assigned adjacent data sets) are connected. For example, the SIMPil processor has a mesh-style NEWS (“north, east, west, south”) network (Figure 35) where each (non-border) PE is connected, respectively, to the PE above it, to its left, to its right, and below it [12][13][14]. The MasPar MP-1 is similar, but with connections to all eight neighboring PEs [63][64]. Such nearest-neighbor networks exploit the two-dimensional data locality present in algorithms such as
a 3×3 image convolution, where each output pixel in the image is a function of the eight pixels surrounding it.

![Diagram of SIMPil NEWS communication network](image)

**Figure 35. SIMPil NEWS communication network.**

A problem faced by designers of multi-processor SIMD systems was dealing with conditionals. As long as a SIMD program consists of straight-line code, every PE can execute the same instruction, but in the presence of if-then-else conditionals, different PEs may be required to take different branches. Multi-processor SIMD architectures use PE masking to control which PEs execute which branches. These masking operations, such as the SLEEP and WAKEUP instructions for SIMPil [13], disable PEs that will not participate in a taken branch. When the branch has completed, the disabled PEs are enabled and processing continues.
Multimedia ISA Extensions

Multimedia ISA extensions make use of wide (e.g., 32-, 64-, or 128-bit) architectural registers, within a sequential superscalar processor, to hold multiple elements of data (e.g., four 32-bit data elements in a 128-bit register). Extensions to the instruction set architecture (ISA) provide useful SIMD operations on the packed data. ISA extensions such as Intel’s multimedia extensions (MMX) [1], streaming SIMD extensions (SSE) [2], and SSE2 [3][4], Motorola’s Altivec [6], Advanced Micro Devices’ 3Dnow! [5], Sun Corporation’s visual instruction set (VIS) [7], and the MAX-2 extensions for PA-RISC [66] all implement SIMD instructions. Such ISA extensions allow a modest ability to exploit DLP in a sequential processor, but often with a very small increase in chip area due to re-use of existing architectural registers and functional units. This type of parallelism is often referred to as subword parallelism [66].

Depending on the vendor, a multimedia ISA extension will contain some or all of the following operation types:

- **Arithmetic:** Operations in this class include element-by-element addition and subtraction, logical operations, multiplication, etc. Overflow and underflow (for integer addition and subtraction) are typically dealt with by saturating the result, as throwing an exception for a single data element in a packed word complicates the instruction logic. As most subword-parallel ISA extensions are targeted at multimedia applications, saturated results are acceptable.

- **Permutation/mix:** A data element produced by a SIMD algorithm may depend on the values of other spatially-local elements. Whereas a multi-processor SIMD architecture provides an inter-PE communication network, a subword-
parallel SIMD architecture provides permutation operations that can rearrange the packed data with a single instruction. Figure 36 gives an example of the SSE general permutation instruction, PSHUFW [2]. Here, R2 specifies the elements of word V1 to insert into the appropriate elements of V2 (e.g., the ‘2’ in the left-most position of R2 instructs PSHUFW to insert element two of V1 [with value 64] into the left-most position of V2). Some permutation implementations, as Figure 36 illustrates, allow duplication of values in the permuted result. A mix is similar to a permutation, except that the result contains values from multiple source registers.

- Pack/unpack: Pack (unpack) operations convert data from higher (lower) precision to lower (higher) precision.

- Prefetch instructions: As these ISA extensions are designed to operate on long streams of data, some (notably SSE [2] and 3DNow! [5]) include prefetch instructions intended to give the processor “hints” about the data required for the application. Using prefetch instructions, future data can be brought into the cache while current data is being operated upon, thus reducing cache miss penalties.

- Multimedia-specific: Some subword-parallel architectures include instructions to support common multimedia algorithms. AMDs 3Dnow!, for example, includes instructions for performing approximate reciprocal and reciprocal square root [5]. VIS includes an instruction for computing the absolute difference between the elements of two packed words, useful in motion estimation for video compression [7].
The commercial subword-parallel ISA extensions described above all operate on one-dimensional vectors of data. In addition, at least two research projects exist that attempt to extend such ISAs into two dimensions: the complex streaming instructions (CSI) [61] and matrix oriented multimedia (MOM) [47] extensions. Both ISAs include instructions that operate on two-dimensional matrices of data in order to better accommodate image and video applications that operate on two-dimensional data sets. MOM is modeled as a vector (the matrix rows) of packed words (the elements in each row). A single instruction can operate on the entire matrix at once, but it is pipelined such that only one row at a time is processed, i.e., a single matrix-addition operation may still take multiple cycles. CSI allows operations on semi-arbitrary blocks of data in memory (row and column strides must be constant for each block, but may be greater than one). By incorporating stride and array-length information into the instructions, CSI can fetch an instruction one time and repeatedly apply it to all the elements of a data stream.
APPENDIX B

REFERENCE GRAPH-GRAMMAR

Overview

This appendix describes the reference graph-grammar used to implement MDDF graph extraction. The grammar is presented as a set of production rules of the form

\[ <\text{non-terminal}> \rightarrow <\text{pattern}> \]

where \(<\text{non-terminal}>\) is a single node abstracting over the graph of \(<\text{pattern}>\) which can contain terminal as well as non-terminal nodes. Additionally, each rule may include text describing additional constraints required to complete the pattern.

Figure 37 provides the key to all graph elements used in the production rules. Note that back-edges are shown as edges for descriptive purposes only. In terms of recognition, they are actually graph annotations indicating a cyclic relationship between two flow edges. Note also that addition and multiplication nodes are commutative for all patterns that include them.

![Graph-grammar graph elements key.](image-url)

Figure 37. Graph-grammar graph elements key.
Edges have a type equal to that of the data type of the data they carry. This type is determined from the original source language and carried through to the intermediate DFG representation. For certain production rules, type is important in distinguishing between edges carrying an address (i.e., pointer or array type) and those that do not.

Some production rules use a notation of the form \((n_1:r_1:s_1)\) to describe recognized array regions. For clarity, these are shown as labels near the head or tail of an edge. Formally, they are associated with the node ports connected to the corresponding edge head/tail. In this notation, the first field ("n") is the start value for the region, the second ("r") is the range value, and the third ("s") is the step value. The two-dimension version, \((n_1:r_1:s_1, n_2:r_2:s_2)\), is also used. See Chapter 2 for a full description of these notations. The array region notation \((R)\) is used in production rules to match array regions, regardless of dimensionality, directly between right-hand-side patterns and left-hand-side non-terminals when no modification is required. Finally, if a "1" appears at the head or tail of an edge, this is short-hand for a scalar (single-element) region.

**Constant Expression Pattern**

The `const-expr` ("constant expression") pattern collects groups of operator nodes over constants (symbols that are constant for the loop as well as numeric values) into a single expression involving addition and/or multiplication operations. The pattern attribute (notated inside the curly braces) is a symbolic expression representing the expression. The production rules for `const-expr` are listed in Figure 38.
constraint: \( c \) is constant for the loop

Figure 38. Production rules for constant expression patterns.
Count Pattern

The count pattern represents an enumeration of values. These patterns, given in Figure 39 and Figure 40, commonly arise from loop induction variables. When recognized, the count pattern inherits attributes from the enclosing loop environment to build an array region representing the enumeration. The initial value ("n" here) is the initial value of the induction variable as defined in the loop header. (If not defined explicitly, it is given as the symbolic variable itself.) The range of the region ("r" here) is calculated based on the initial value, the step ("s", synthesized as shown in Figure 39 and Figure 40) and the loop condition as \( r = (u-n)/s \) where \( u \) is the upper bound of the loop as determined by the loop condition.
Figure 39. Production rules for count patterns, part 1.
constraint: \( s_1 = r_2 \cdot s_2 \)

constraint: \( r_1 = r_2 \)

constraint: \((n_1: r_1: s_1 \text{ region was recognized first})\) (inner loop region)

Figure 40. Production rules for count patterns, part 2.
**Pointer-Count Pattern**

The `ptr-count` ("pointer-count") pattern represents an enumerated range of memory addresses. It inherits region information (here, “n” and “r”) from the loop environment in the same way as the `count` pattern (i.e., from the induction variable initialization and update statements, usually part of the loop header). The production rule is given in Figure 41.

![Figure 41. Production rule for the pointer-count pattern.](image)

**Array-Read Pattern**

The `array-read` pattern represents a parallel load of consecutive values from memory. The production rules are given in Figure 42 and Figure 43. The terminal nodes `LOAD_ptr`, `LOAD_1`, and `LOAD_2` represent, respectively, memory reads via a pointer, a singly-indexed array, and a doubly-indexed array.
Figure 42. Production rules for array-read patterns, part 1.
Figure 43. Production rules for array-read patterns, part 2.
Array-write patterns

The **array-write** pattern is a parallel store of values to consecutive memory addresses. The production rules are given in Figure 44 and Figure 45. The terminal nodes STORE_ptr, STORE_1, and STORE_2 refer, respectively, to memory stores via a pointer, a singly-indexed array, and a doubly-indexed array.

![Diagram of array-write patterns](image)

*Figure 44. Production rules for array-write patterns, part 1.*
Figure 45. Production rules for array-write patterns, part 2.
Array-Sum Pattern

The array-sum represents the summation of an array of values into a scalar.

The production rule is given in Figure 46.

Figure 46. Production rule for an array-sum pattern.
APPENDIX C

SOBEL SOURCE PARALLELIZED BY PARRET FOR SSE2

The source code presented in this section was generated by parallelizing the sobel benchmark from the Texas Instruments IMGLIB suite [42] using PARRET. The generated code is C source making use of optimized intrinsic functions provided by ICL version 8.0 as well as a few custom functions which are composed of ICL intrinsic functions. The former are all functions prefixed with “_mm_” and are documented in the help utility included with ICL [46]. The latter are as follows:

- **promote_uchar2ushort**: Packed promotion of eight unsigned 8-bit integers to unsigned 16-bit integers. Selection of the upper half or lower half of the input sixteen elements is via the _UPPER_ and _LOWER_ macros, respectively.

- **__ia32_pnegatew**: Packed negation of 16-bit integers.

- **__ia32_pcmpltw**: Packed less-than comparison of signed 16-bit integers.

- **pselectw**: Packed trinary selection operation for signed or unsigned 16-bit integers. Inputs are a condition mask (each 16-bit element is all ones for true, all zeros for false), a packed value represented the “yes” side of a branch and a packed value for the “no” side of a branch. The result is a packed value where the “yes” elements for which the mask is true are interleaved with the “no” elements for which the mask is false.
- demote_short2uchar: Demotion with saturation of two packed signed vectors of 16-bit elements to a single packed vector of unsigned 8-bit elements.

```c
for( x_tmp1=0;
     x_tmp1<(0-2+-2*cols+cols*rows);
     x_tmp1+=16 ) {
  x_tmp2[0] =
    _mm_loadu_si128( (_m128i*)&in[x_tmp1+0+0] );
  x_tmp3[0] =
    promote_uchar2ushort( x_tmp2[0], _LOWER_ );
  x_tmp3[1] =
    promote_uchar2ushort( x_tmp2[0], _UPPER_ );
  x_tmp4[0] =
    _mm_loadu_si128((_m128i*)&in[x_tmp1+1+0] );
  x_tmp5[0] =
    promote_uchar2ushort( x_tmp4[0], _LOWER_ );
  x_tmp5[1] =
    promote_uchar2ushort( x_tmp4[0], _UPPER_ );
  x_tmp6[0] =
    _mm_loadu_si128((_m128i*)&in[x_tmp1+2+0] );
  x_tmp7[0] =
    promote_uchar2ushort( x_tmp6[0], _LOWER_ );
  x_tmp7[1] =
    promote_uchar2ushort( x_tmp6[0], _UPPER_ );
  x_tmp8[0] =
    _mm_loadu_si128((_m128i*)&in[x_tmp1+w+0] );
  x_tmp9[0] =
    promote_uchar2ushort( x_tmp8[0], _LOWER_ );
  x_tmp9[1] =
    promote_uchar2ushort( x_tmp8[0], _UPPER_ );
  x_tmp10[0] =
    _mm_loadu_si128((_m128i*)&in[x_tmp1+2+w+0] );
  x_tmp11[0] =
    promote_uchar2ushort( x_tmp10[0], _LOWER_ );
  x_tmp11[1] =
    promote_uchar2ushort( x_tmp10[0], _UPPER_ );
  x_tmp12[0] =
    _mm_loadu_si128((_m128i*)&in[x_tmp1+2*w+0] );
  x_tmp13[0] =
    promote_uchar2ushort( x_tmp12[0], _LOWER_ );
  x_tmp13[1] =
    promote_uchar2ushort( x_tmp12[0], _UPPER_ );
  x_tmp14[0] =
    _mm_loadu_si128((_m128i*)&in[x_tmp1+1+2*w+0] );
  x_tmp15[0] =
    promote_uchar2ushort( x_tmp14[0], _LOWER_ );
  x_tmp15[1] =
    promote_uchar2ushort( x_tmp14[0], _UPPER_ );
  x_tmp16[0] =
```
_mm_loadu_si128( __m128i*) & in[x_tmp1+2+2*w+0]
); 
_x_tmp17[0] = 
_promote_uchar2ushort( x_tmp16[0], _LOWER_ ); 
_x_tmp17[1] = 
_promote_uchar2ushort( x_tmp16[0], _UPPER_ ); 
_x_tmp26[0] = 
__mm_mullo_epi16( x_tmp5[0], x_tmp27[0] ); 
_x_tmp26[1] = 
__mm_mullo_epi16( x_tmp5[1], x_tmp28[1] ); 
_x_tmp29[0] = 
__mm_mullo_epi16( x_tmp15[0], x_tmp30[0] ); 
_x_tmp29[1] = 
__mm_mullo_epi16( x_tmp15[1], x_tmp31[1] ); 
_x_tmp32[0] = 
__mm_mullo_epi16( x_tmp9[0], x_tmp33[0] ); 
_x_tmp32[1] = 
__mm_mullo_epi16( x_tmp9[1], x_tmp34[1] ); 
_x_tmp35[0] = 
__mm_mullo_epi16( x_tmp11[0], x_tmp36[0] ); 
_x_tmp35[1] = 
__mm_mullo_epi16( x_tmp11[1], x_tmp37[1] ); 
_x_tmp38[0] = __ia32_pnegatew( x_tmp3[0] ); 
_x_tmp38[1] = __ia32_pnegatew( x_tmp3[1] ); 
_x_tmp39[0] = __mm_subs_epi16( x_tmp7[0], x_tmp3[0] ); 
_x_tmp39[1] = __mm_subs_epi16( x_tmp7[1], x_tmp3[1] ); 
_x_tmp40[0] = 
__mm_subs_epi16( x_tmp38[0], x_tmp26[0] ); 
_x_tmp40[1] = 
__mm_subs_epi16( x_tmp38[1], x_tmp26[1] ); 
_x_tmp41[0] = 
__mm_subs_epi16( x_tmp39[0], x_tmp32[0] ); 
_x_tmp41[1] = 
__mm_subs_epi16( x_tmp39[1], x_tmp32[1] ); 
_x_tmp42[0] = 
__mm_adds_epi16( x_tmp41[0], x_tmp35[0] ); 
_x_tmp42[1] = 
__mm_adds_epi16( x_tmp41[1], x_tmp35[1] ); 
_x_tmp43[0] = 
__mm_subs_epi16( x_tmp40[0], x_tmp7[0] ); 
_x_tmp43[1] = 
__mm_subs_epi16( x_tmp40[1], x_tmp7[1] ); 
_x_tmp44[0] = 
__mm_adds_epi16( x_tmp43[0], x_tmp13[0] ); 
_x_tmp44[1] = 
__mm_adds_epi16( x_tmp43[1], x_tmp13[1] ); 
_x_tmp45[0] = 
__mm_subs_epi16( x_tmp42[0], x_tmp13[0] ); 
_x_tmp45[1] = 
__mm_subs_epi16( x_tmp42[1], x_tmp13[1] ); 
_x_tmp46[0] = 
__mm_adds_epi16( x_tmp45[0], x_tmp17[0] ); 
_x_tmp46[1] = 
__mm_adds_epi16( x_tmp45[1], x_tmp17[1] ); 
_x_tmp47[0] = 
```c
_mm_adds_epi16( x_tmp44[0], x_tmp29[0] );
_x_tmp47[1] =
_mm_adds_epi16( x_tmp44[1], x_tmp29[1] );
x_tmp48[0] =
__ia32_pcmpltsw( x_tmp46[0], x_tmp49[0] );
x_tmp48[1] =
__ia32_pcmpltsw( x_tmp46[1], x_tmp50[1] );
x_tmp51[0] =
_mm_adds_epi16( x_tmp47[0], x_tmp17[0] );
x_tmp51[1] =
_mm_adds_epi16( x_tmp47[1], x_tmp17[1] );
x_tmp52[0] = __ia32_pnegatew( x_tmp46[0] );
x_tmp52[1] = __ia32_pnegatew( x_tmp46[1] );
x_tmp53[0] =
pselectw( x_tmp48[0], x_tmp52[0], x_tmp46[0] );
x_tmp53[1] =
pselectw( x_tmp48[1], x_tmp52[1], x_tmp46[1] );
x_tmp54[0] =
__ia32_pcmpltsw( x_tmp51[0], x_tmp55[0] );
x_tmp54[1] =
__ia32_pcmpltsw( x_tmp51[1], x_tmp56[1] );
x_tmp57[0] = __ia32_pnegatew( x_tmp51[0] );
x_tmp57[1] = __ia32_pnegatew( x_tmp51[1] );
x_tmp58[0] =
pselectw( x_tmp54[0], x_tmp57[0], x_tmp51[0] );
x_tmp58[1] =
pselectw( x_tmp54[1], x_tmp57[1], x_tmp51[1] );
x_tmp59[0] =
_mm_adds_epi16( x_tmp58[0], x_tmp53[0] );
x_tmp59[1] =
_mm_adds_epi16( x_tmp58[1], x_tmp53[1] );
x_tmp60[0] =
__mm_cmpgt_epi16( x_tmp59[0], x_tmp61[0] );
x_tmp60[1] =
__mm_cmpgt_epi16( x_tmp59[1], x_tmp62[1] );
x_tmp63[0] =
pselectw( x_tmp60[0], x_tmp64[0], x_tmp59[0] );
x_tmp63[1] =
pselectw( x_tmp60[1], x_tmp65[1], x_tmp59[1] );
x_tmp67 =
__denote_short2uchar( x_tmp63[0], x_tmp63[1] );
_mm_storeu_si128(&x_tmp66[+(x_tmp1)/16], x_tmp67);
```
REFERENCES


VITA

Lewis Baunstark, Jr. was born on July 18, 1975, in Augusta, Georgia. This was just early enough to see the first Star Wars movie (confusing titled “Episode IV”) before it left the theatre for good. Unfortunately, such a memorable experience at such a young age meant he soon developed a passion for technology (because when you pushed buttons, stuff happened), which led to a desire to be an inventor, which eventually morphed into engineering, which finally landed him in the unlikely role of PhD. (Author’s note: Thanks a lot, Mr. Lucas. I could have been a rock star.)

Lewis received his BS in Electrical and Computer Engineering from Tennessee Technological in 1998, his MS in the same from Georgia Institute of Technology in 2001, and finally his PhD, also from Georgia Tech, in 2004 (just in time to view the sixth and final Star Wars movie, inappropriately titled “Episode III”). He has a habit of starting new hobbies every couple of years without properly retiring the old ones, so he enjoys (or has enjoyed) kayaking, Kung-Fu, Tai-Chi, reading sci-fi novels, watching sci-fi movies, playing piano, collecting Star Wars memorabilia, griping about politics, listening to everything from rock to jazz to weird imported video game soundtracks, and, most of all, spending time with friends and family. Particularly if food is involved.