

COMPILER OPTIMIZATIONS FOR ARCHITECTURES SUPPORTING
SUPERWORD-LEVEL PARALLELISM

by

Jaewook Shin

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

August 2005

Copyright 2005

Jaewook Shin

Dedication

This dissertation is dedicated to my father Jungsik and mother Bokhee for their patience, prayers and encouragement.

Acknowledgments

I would like to give my special thanks to my advisor, Dr. Mary W. Hall, for her guidance, enthusiasm, and support which made the completion of this dissertation possible. Her insight and suggestions proved to be invaluable in this work. I was fortunate to have Dr. Ulrich Neumann and Dr. Timothy Pinkston as committee members of the dissertation committee.

I would like to thank Dr. Jacqueline Chame, who has been guiding this research together with my advisor. Dr. Saman Amarasinghe and Sam Larsen at MIT deserve a special thanks. Not only their work on SLP has been the basis of this research, but also they provided their implementation and expertise. My office mate Chun Chen was friendly and insightful. He helped me with my math questions. I also have enjoyed working with Dr. Pedro Diniz, Tim Barrett, Heidi Ziegler, Yoonju Lee, Nastaran Baradaran and Spundun Bhatt, who have been supporting me both technically and emotionally.

Finally, I would like to thank my wife Eunlim for her love, support and encouragement without which I wouldn't be writing this at this moment.

Contents

| | |
|---|-------------|
| Dedication | ii |
| Acknowledgments | iii |
| List Of Tables | vii |
| List Of Figures | viii |
| Abstract | xi |
| 1 INTRODUCTION | 1 |
| 1.1 SLP Compilation Technology | 2 |
| 1.2 SLP vs. Vectorizing Compilers | 3 |
| 1.3 Opportunities to Improve SLP | 4 |
| 1.4 Target Applications | 5 |
| 1.5 Target Architectures | 6 |
| 1.5.1 Multimedia Extension: AltiVec | 6 |
| 1.5.2 Processing-In-Memory: DIVA | 7 |
| 1.6 Motivating Examples | 9 |
| 1.6.1 SLP in the Presence of Control Flow | 9 |
| 1.6.2 Superword-Level Locality | 10 |
| 1.7 Contributions | 11 |
| 2 BACKGROUND | 15 |
| 2.1 Data Dependence | 15 |
| 2.2 Data Reuse | 17 |
| 2.3 The SLP algorithm | 18 |
| 2.3.1 Alignment Analysis | 19 |
| 2.3.2 Distance Analysis | 20 |
| 2.3.3 Packing | 21 |
| 2.3.4 Summary | 22 |
| 2.4 Predicate Analysis | 22 |
| 2.4.1 Predicated Execution | 23 |
| 2.4.2 If-conversion: RK-Algorithm | 24 |

| | | |
|----------|--|-----------|
| 2.4.3 | Predicate Hierarchy Graph (PHG) | 25 |
| 2.4.4 | Mutually Exclusive | 26 |
| 2.4.5 | Predicate Covering | 27 |
| 2.4.6 | Predicate CFG Generator | 28 |
| 3 | SUPERWORD-LEVEL PARALLELISM IN THE PRESENCE OF CONTROL FLOW | 29 |
| 3.1 | Overview of the Algorithm | 31 |
| 3.2 | Eliminating Superword Predicates | 35 |
| 3.3 | Unpredicate | 37 |
| 3.4 | Branch-On-Superword-Condition-Code (BOSCC) | 41 |
| 3.4.1 | The Characteristics of BOSCC | 41 |
| 3.4.2 | BOSCC Model | 44 |
| 3.4.3 | Profiling Support to Compute PAFS | 47 |
| 3.4.4 | Identifying BOSCC Predicates | 49 |
| 3.4.5 | Inserting BOSCC Instructions | 51 |
| 4 | SUPERWORD-LEVEL LOCALITY | 52 |
| 4.1 | Background and Motivation | 53 |
| 4.2 | Overview of Superword-Level Locality Algorithm | 57 |
| 4.3 | Modeling Register Requirements & Number of Memory Accesses | 59 |
| 4.3.1 | Computing the Superword Footprint | 60 |
| 4.3.1.1 | Superword Footprint of a Single Reference | 61 |
| 4.3.1.2 | Superword Footprint of a Group of References | 63 |
| 4.3.2 | Registers for Reuse Across Iterations | 69 |
| 4.3.3 | Putting It All Together | 70 |
| 4.4 | Determining Unroll Factors | 72 |
| 4.5 | Code Transformations | 74 |
| 4.5.1 | Index Set Splitting | 76 |
| 4.5.2 | Superword Replacement | 79 |
| 4.5.3 | Packing in Superword Registers | 80 |
| 4.5.4 | An Example: Shifting for Partial Reuse | 83 |
| 5 | CODE GENERATION | 85 |
| 5.1 | Type Size Conversion | 85 |
| 5.2 | Reduction | 87 |
| 5.3 | Alignment Optimization | 88 |
| 5.4 | Prepacking to Optimize Parallelization Overhead | 90 |
| 5.5 | Summary | 93 |
| 6 | EXPERIMENTS | 94 |
| 6.1 | Benchmarks | 95 |
| 6.2 | Implementation | 98 |
| 6.3 | Experimental Methodology | 99 |
| 6.4 | Overall Performance | 101 |
| 6.5 | Packing for Low Parallelization Overhead | 103 |

| | | |
|----------|--|------------|
| 6.6 | SLP in the Presence of Control Flow | 105 |
| 6.6.1 | Branch-On-Superword-Condition-Code (BOSCC) | 107 |
| 6.7 | Superword-Level Locality | 113 |
| 6.8 | Summary | 116 |
| 7 | DIVA AND PIM-SPECIFIC OPTIMIZATIONS | 117 |
| 7.1 | The DIVA ISA | 118 |
| 7.2 | Page-Mode Memory Access | 119 |
| 7.2.1 | Motivation | 120 |
| 7.2.2 | The Page-Mode Memory Access Algorithm | 122 |
| 7.2.3 | Experiments for the Page-Mode Memory Access Algorithm | 126 |
| 7.3 | DIVA-Specific Code Generation | 131 |
| 7.4 | Preliminary Bandwidth Demonstration | 133 |
| 7.5 | Summary | 135 |
| 8 | RELATED WORK | 136 |
| 8.1 | Exploiting SLP in the Presence of Control Flow | 136 |
| 8.2 | Superword-Level Locality | 138 |
| 8.3 | DIVA-specific Optimizations | 139 |
| 8.4 | Summary | 140 |
| 9 | CONCLUSION | 141 |
| 9.1 | Contributions | 141 |
| 9.1.1 | SLP in the Presence of Control Flow | 142 |
| 9.1.2 | Compiler Controlled Caching in Superword Registers | 143 |
| 9.1.3 | Implementation and Evaluation of the Proposed Techniques | 143 |
| 9.1.4 | DIVA-Specific Optimizations | 144 |
| 9.2 | Future Work | 144 |
| | Reference List | 146 |

List Of Tables

| | | |
|-----|--|-----|
| 1.1 | Differences between multimedia extensions and vector architectures. | 4 |
| 2.1 | Transfer functions for the dataflow analysis to find alignments when $a_1n + b_1$ and $a_2m + b_2$ are given. | 19 |
| 2.2 | Transfer functions for distance analysis when two terms $T_1 = (X, c_1, b_1)$ and $T_2 = (X, c_2, b_2)$ of two linear expressions for the same variable X are given. | 20 |
| 4.1 | Number of array accesses under different optimization paths. | 56 |
| 6.1 | Benchmark programs. | 95 |
| 6.2 | Runtime percentage of three functions from UCLA MediaBench. | 96 |
| 6.3 | Input data size. | 97 |
| 7.1 | Memory latency computation. | 121 |
| 7.2 | DIVA simulation parameters. | 126 |
| 7.3 | Benchmark programs. | 127 |
| 7.4 | Experimental environments. | 133 |

List Of Figures

| | | |
|------|--|----|
| 1.1 | Example: Parallelization by the SLP compiler. | 3 |
| 1.2 | AltiVec register file. | 6 |
| 1.3 | DIVA node architecture. | 8 |
| 1.4 | Example: SLP in the presence of control flow. | 10 |
| 1.5 | Motivating example for exploiting superword-level locality. | 11 |
| 2.1 | An example code to show dependence vectors and unroll-and-jam. | 16 |
| 2.2 | An example showing the packing algorithm. | 22 |
| 2.3 | An example showing construction of a PHG. | 26 |
| 3.1 | Overview of the algorithm to exploit SLP in the presence of control flow. | 30 |
| 3.2 | Example illustrating steps of SLP compilation in the presence of control flow. | 32 |
| 3.3 | Merging two superwords using a <code>select</code> instruction. | 33 |
| 3.4 | Merging two superword definitions. | 35 |
| 3.5 | An algorithm to generate <code>select</code> instructions. | 36 |
| 3.6 | Restoring control flow. | 37 |
| 3.7 | Unpredicate algorithm. | 38 |
| 3.7 | Unpredicate algorithm (Continued). | 39 |
| 3.8 | Run time of synthetic kernels. | 42 |
| 3.9 | Automatic instrumentation to compute PAFS in profiling phase. | 46 |
| 3.10 | Algorithm to identify a predicate for instructions. | 48 |
| 3.11 | BOSCC insertion algorithm. | 50 |

| | | |
|------|--|-----|
| 4.1 | Example code for SLL. | 55 |
| 4.2 | Reuse across iterations. | 58 |
| 4.3 | Superword footprint of a single reference. | 62 |
| 4.4 | Superword footprint of a group of references. | 64 |
| 4.4 | Superword footprint of a group of references (Continued). | 65 |
| 4.5 | Code generation example. | 77 |
| 4.5 | Code generation example (Continued). | 78 |
| 4.6 | Operations used for packing in registers. | 82 |
| 4.7 | Shifting. | 83 |
| 5.1 | Parallelization of type size conversions | 86 |
| 5.2 | Parallelization of reduction sum. | 87 |
| 5.3 | Parallelization of unaligned memory references | 88 |
| 5.4 | Parallelization by prepacking | 89 |
| 5.5 | Data dependence graphs for the loop body of Figure 5.4(b) | 91 |
| 5.6 | Multiple packing choices generated by unrolling multiple loops | 92 |
| 6.1 | Implementation. | 98 |
| 6.2 | Experimental flow. | 100 |
| 6.3 | Overall speedup breakdown (large data). | 102 |
| 6.4 | Overall speedup breakdown (small data). | 103 |
| 6.5 | Effect of prepacking. | 104 |
| 6.6 | An SLP-based compiler that supports BOSCC. | 107 |
| 6.7 | Example: BOSCCs generated in EPIC. | 108 |
| 6.8 | Speedups over scalar version for real data. | 110 |
| 6.9 | TM: % taken BOSCCs. | 111 |
| 6.10 | Speedups over scalar version for randomly generated data. | 113 |
| 6.11 | Speedups over MIT-SLP. | 115 |
| 7.1 | The superword data flow. | 118 |

| | | |
|------|--|-----|
| 7.2 | Unroll-and-jam and reordering. | 120 |
| 7.3 | The page-mode memory access algorithm. | 122 |
| 7.4 | Sorting offset addresses. | 126 |
| 7.5 | Experimental flow for page-mode memory access. | 128 |
| 7.6 | SLP versions of VMM and MMM. | 129 |
| 7.7 | Normalized execution time. | 129 |
| 7.8 | Percentage of page-mode accesses. | 131 |
| 7.9 | Speedup breakdown. | 131 |
| 7.10 | Code generation for conditional execution in DIVA. | 132 |
| 7.11 | StreamAdd | 133 |
| 7.12 | Run time of floating point StreamAdd. | 134 |

Abstract

The increasing importance of multimedia applications in embedded and general-purpose computing environments has led to the development of multimedia extensions in most commercial microprocessors. At the core of these extensions is support for *single instruction multiple data* (SIMD) operations on *superwords*, that is, aggregate data objects larger than a machine word.

Several compilers have been developed to generate the SIMD instructions for multimedia extensions automatically. However, most are based on conventional vectorization technology. More recently, a technique called *superword-level parallelization* (SLP) was developed to exploit unique features of multimedia extensions, such as short vectors and single cycle instruction latency. Instead of finding parallelism from loops, SLP finds parallelism between instructions making this approach simple and more robust than the vectorization technique.

We propose a new compiler framework based on SLP where a number of optimizations are performed in a seamless fashion. First, we describe how to extend the concept of SLP in the presence of control flow constructs to increase its applicability. A key insight is that we can use techniques related to optimizations for architectures supporting predicated execution, even for multimedia instruction sets that do not provide hardware predication. Second, we treat the large superword register file as a *compiler-controlled cache*, thus avoiding unnecessary memory accesses by exploiting reuse in superword registers. This approach also targets a research prototype, the DIVA *processor-in-memory*

(PIM) device. We describe DIVA-specific optimizations including a technique to exploit a DRAM memory characteristic automatically.

We implemented the new techniques in a complete compiler that generates SIMD instructions automatically from sequential programs. We describe the evaluation of our implementation on a set of 14 benchmarks. The speedups range from 1.05 to 19.22 over sequential performance.

Chapter 1

INTRODUCTION

The increasing importance of multimedia applications has led to the development of multimedia extensions in most commercial microprocessors. At the core of these extensions is support for short *single instruction multiple data* (SIMD) operations on *superwords*, that is, aggregate data objects larger than a machine word.

Initially, the conventional wisdom was that the appropriate compiler technology for multimedia extensions would borrow heavily from automatic vectorization [64, 15, 19]. More recently, Larsen and Amarasinghe at MIT developed a new technique to parallelize codes specifically targeting multimedia extensions [39]. To make the distinction between the parallelism in multimedia extensions and vector parallelism, they define *superword-level parallelism* (SLP) as fine-grained SIMD parallelism in a superword. The new technique is simple and robust compared to the existing vectorization techniques. Still, there remain open issues for multimedia extensions: how to exploit parallelism across basic block boundaries and how to exploit locality in superword registers. This thesis research was initiated as part of the Data IntensiVe Architecture (DIVA) project [31, 21]. DIVA employs *processing-in-memory* (PIM) technology by combining processing logic and DRAM in a single chip. To exploit high internal memory bandwidth, the DIVA PIM devices support SIMD operations on 256-bit superword registers. Since DIVA is

a new architecture, there exist new compiler optimization opportunities. In this thesis, we describe our approach to address the two open issues and several DIVA-specific optimizations.

The remainder of this chapter is organized as follows. In the next section, we overview the SLP compilation technology. In Section 1.2, we compare this technology with the conventional vectorization techniques. To motivate our approaches in this thesis, we present the remaining opportunities to improve SLP in Section 1.3. Our target applications and target architectures are described in Section 1.4 and Section 1.5, respectively. In Section 1.6, we use simple examples to explain our approaches to the two open issues. The contributions of this thesis are summarized in the last section.

1.1 SLP Compilation Technology

The MIT SLP compiler finds parallelism from a basic block, a block of sequentially executed instructions. Given a basic block, it first identifies *isomorphic* statements which refer to statements with the same corresponding operations. Then, the isomorphic statements are *packed* into a superword statement, that is, collected and replaced by a superword statement, unless dependences prevent doing so. Packing memory references should satisfy further restrictions in order to support hardware requirements; data elements to be referenced are contiguous in memory and the address of the first element is *aligned* to superword boundaries, i.e., its runtime addresses are congruent with respect to superword width. While the steps described so far are enough to exploit parallelism within a basic block, for loop nests, the innermost loop is unrolled to convert loop level parallelism into basic block level parallelism. The unroll amount is determined by dividing superword width by the smallest data type size so that even the operations with the smallest operands can exploit SLP fully when packed into a superword. Figure 1.1 illustrates these steps using an example loop in (a). First, the loop is unrolled by 4 as

| | | |
|---|---|--|
| <pre>for (i=0; i<16; i++) a[i] = b[i] + c[i];</pre> | <pre>for (i=0; i<16; i+=4){ a[i+0] = b[i+0] + c[i+0]; a[i+1] = b[i+1] + c[i+1]; a[i+2] = b[i+2] + c[i+2]; a[i+3] = b[i+3] + c[i+3]; }</pre> | <pre>for (i=0; i<16; i+=4) a[i:i+3] = b[i:i+3] + c[i:i+3];</pre> |
| (a) Original | (b) Unrolled | (c) Parallelized |

Figure 1.1: Example: Parallelization by the SLP compiler.

shown in (b) assuming four array elements fit in a superword register. Then, the four isomorphic statements are packed into a superword statement as shown in (c). Here, `a[i:i+3]` represents four array elements from `a[i]` to `a[i+3]`.

1.2 SLP vs. Vectorizing Compilers

To understand the differences between vector and SLP compilers, we first consider the architectural differences between vector and multimedia extension architectures. While there are many similarities, multimedia extensions are different from vector architectures in several aspects as listed in Table 1.1. For multimedia extensions, strided memory accesses are not supported, vector length is short, instruction latency is roughly one cycle per instruction and memory accesses are usually required to be aligned to superword boundaries. From the compiler’s perspective, these differences mean that superword instructions can mix better with scalar instructions than vector instructions but strided or unaligned memory accesses are more costly. Consequently, as compared to vector architectures, parallelization overhead for multimedia extensions is less a function of vector length and more a function of alignment requirements and cost of packing data elements.

Vectorizing compilers have been used to generate vector instructions automatically for vector supercomputers. A set of loop transformations are applied to expose SIMD parallel operations suitable for vectorization [43]. Because such transformations are not

| | Multimedia extensions | Vector architectures |
|-------------------------|------------------------------|-------------------------------|
| Strided memory access | Not supported | Supported |
| Vector length | ≤ 32 | ≥ 64 |
| Instruction latency | ~ 1 cycle / instruction | ~ 1 cycle / data element |
| Aligned memory accesses | Required | Not required |

Table 1.1: Differences between multimedia extensions and vector architectures.

always applicable, vectorization technology is fragile. Small changes in application code greatly affect the compiler’s ability to recognize vector operations. Compared to vectorization technology, the transformations used in SLP compilers, as discussed in Section 1.1 are quite simple and always applicable. Instead of complex loop transformations, SLP compilers apply only unrolling, scalar renaming, and packing of data for isomorphic statements. For SLP, failure to parallelize a single statement may not affect parallelization of other statements.

1.3 Opportunities to Improve SLP

The SLP compiler developed by Larsen and Amarasinghe only identifies parallelism within a basic block. As a result, the simple and inherently parallel loop in Figure 1.4(a) would not be parallelized. Superword-level parallelization in the presence of control flow is still an open issue. Yet, support for parallelizing control flow is important to multimedia applications. As one data point, control flow appears in key computations in 6 of the 11 codes in the UCLA MediaBench [41], comprising on average over 40% of their execution time.

Parallelizing computation using SLP can stress the memory system, since sometimes compute-bound programs can become memory bound when computation costs are reduced [57]. Thus, an additional optimization opportunity involves reducing the cost of memory accesses. An important feature of all multimedia extension architectures is a register file supporting SIMD operations (*e.g.*, each 128 bit wide in an AltiVec), sometimes

in addition to the scalar register file. A set of 32 such superword registers represents a not insignificant amount of storage close to the processor. Accessing data from superword registers, versus a cache or main memory, has two advantages. The most obvious advantage is lower latency of accesses; even a hit in the L1 cache has at least a 1-cycle latency. Accesses to other caches in the hierarchy or to main memory carry much higher latencies. Another advantage is the elimination of memory access instructions, thus reducing the number of instructions to be issued.

While the two optimization opportunities described above can also be exploited in DIVA, it offers other DIVA-specific compiler optimization opportunities. One such opportunity is to exploit the DIVA ISA features such as conditional execution and permutation instruction. Also, DRAM device characteristics can be exploited to further reduce the memory latency. Since DRAM access time is a large factor in the memory latency of DIVA, this is an opportunity for significant performance gain.

1.4 Target Applications

As described above, multimedia extension architectures are specially designed for multimedia application characteristics such as abundant data parallelism, short iteration counts and small data types [20]. Therefore, these are the primary target applications for our compiler technology. Scientific applications are the main focus of vector architectures, and for the purpose of comparison, they form another important class of target applications for our compiler. Since our locality algorithm uses array subscript expressions in computing register requirement, its main target is array-based loops. However, our extension to exploit SLP in the presence of control flow is effective on some pointer-based applications as well since the SLP algorithm can find alignment and adjacency of pointer-based memory accesses. Nevertheless, applications with regular memory accesses

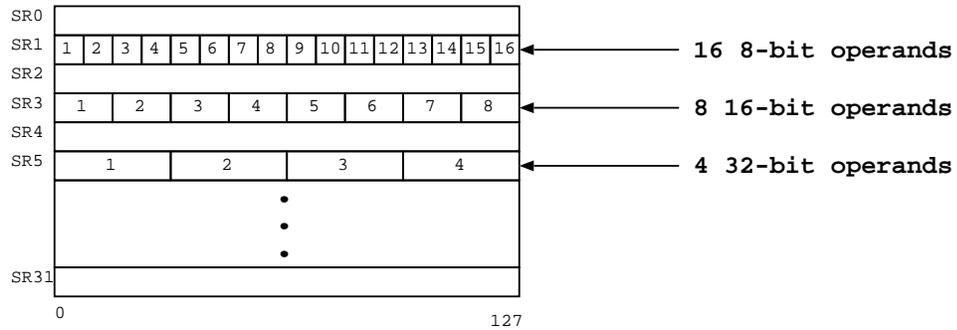


Figure 1.2: AltiVec register file.

of stride 1 are the best candidates for the SLP algorithm to satisfy the underlying hardware requirements. Our locality algorithm is most effective in data-intensive applications with a large amount of data reuse.

1.5 Target Architectures

Our approach described in this thesis can be applied to all architectures supporting SLP. For evaluation purposes, however, our compiler implementation targets two machines: the PowerPC AltiVec and the DIVA processing-in-memory architecture.

1.5.1 Multimedia Extension: AltiVec

While most commercial microprocessors have multimedia extensions, the majority exploit SIMD parallelism within a machine word, called subword parallelism [42, 67, 56]. However, there are a few multimedia extension architectures that have a separate SIMD register file whose width is larger than a machine word, including the Intel SSE and PowerPC AltiVec. Figure 1.2 shows the AltiVec register file. AltiVec has separate 32 128-bit superword registers in addition to the scalar register file. Each superword register can be used as either 16 8-bit operands, 8 16-bit operands, or 4 32-bit operands.

In addition to the common features of multimedia extensions listed in Table 1.1, there are several details of the AltiVec that impact our compiler approach. First, there

are 162 instructions beyond the standard PowerPC ISA [49]. However, some instructions are designed to perform very specialized operations, and not all general operations are supported for all data types. As a result, certain operations cannot be parallelized and should be executed in scalar functional units. Second, AltiVec requires memory accesses to be aligned to superword boundaries by ignoring the last four bits of address operands of memory accesses. Because of this requirement, we need an analysis to find alignments of memory references and additional operations are generated for unaligned superwords in memory. Third, AltiVec does not support data movement between the scalar register file and the superword register file. To move data in a scalar register to a superword register, a scalar data must be written into the memory address range of a superword which is, in turn, loaded into a superword register. Because of these architectural features, automatic generation of the superword instructions by compilers is not easy and sometimes leads to high overhead.

1.5.2 Processing-In-Memory: DIVA

The increasing gap between processor and memory speeds is a well-known problem in computer architecture. As one of the solutions to bridge the gap, processing-in-memory (PIM) is suggested. Because PIM internal processors can be directly connected to the memory banks, the memory bandwidth is dramatically increased (up to 2 orders of magnitude). Latency to on-chip logic is also reduced, down to as little as one-fourth that of a conventional memory system, because internal memory accesses avoid the delays associated with communicating off chip.

The Data-IntensiVe Architecture (DIVA) project is developing a system, from VLSI design through system architecture, systems software, compilers and applications, to take advantage of this technology for applications of growing importance to the high-performance computing community [31, 21]. DIVA combines PIM memory chips with one or more external host processors and a PIM-to-PIM interconnect (see Figure 1.3). To

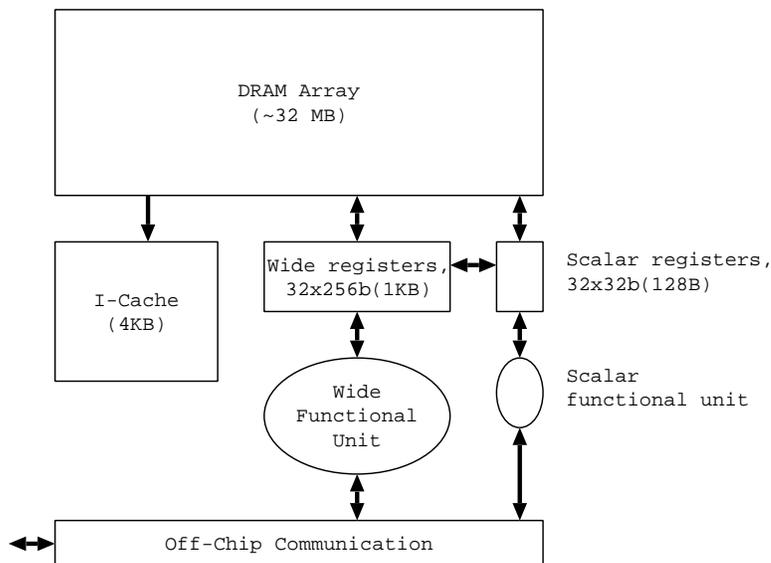


Figure 1.3: DIVA node architecture.

exploit the high data bandwidth effectively, DIVA contains 32 256-bit superword registers in addition to 32 32-bit scalar registers. DIVA contains in-order execution processor cores. Because of short memory latency, data caches are not included in DIVA. However, a small instruction cache is added so that instruction streams do not interfere with data streams [31]. DIVA is a memory coprocessor that requires separate host processors for running the main operating system. As a result, it can be viewed as a standard DRAM to host applications. Due to its low memory latency and high data bandwidth, data intensive applications are the main target applications [14].

In many ways, the DIVA ISA is similar to that of the AltiVec. However, there are several differences. First, DIVA allows data movement between register files. As a result, packing and unpacking scalar values to and from superword registers is cheaper than in the AltiVec. Second, DIVA allows *conditional execution* for most superword instructions. The result of an operation on a field of source registers is committed to the corresponding field of the destination register conditionally depending on the value of the corresponding

bit of the specified condition register. The research of this thesis focuses on developing a DIVA parallelizing compiler that exploits various features of the DIVA processor.

1.6 Motivating Examples

As shown in Section 1.1, the MIT SLP compiler unrolls loops to increase the amount of parallelism within a loop body and packs isomorphic statements. In this section, we use simple examples to illustrate how we exploit the opportunities described in Section 1.3.

1.6.1 SLP in the Presence of Control Flow

In this thesis, we describe how to extend SLP to parallelize computations across basic block boundaries. When a loop body has control flow, unrolling may not increase the basic block size and therefore, may not expose opportunities for SLP parallelization as described by Larsen and Amarasinghe since the MIT SLP compiler parallelizes statements within a basic block.

Consider the example loop in Figure 1.4(a). When the loop is unrolled as in (b), the basic block size does not increase because of the if-statements thus preventing the SLP compiler from parallelizing the loop. However, such a loop can be parallelized as shown in (c). Both the comparison and the statement guarded by the if-statement are parallelized. Then, the old values of `b[i:i+3]` are combined with the new values in `Vtemp` according to the results of the parallel comparison.

In the original scalar code, the statement guarded by the if-statement is bypassed whenever the conditional expression is false. For the parallel version, the instructions in all control flow paths are always executed. In some cases such as when the condition usually evaluates to false, this overhead leads to performance degradation over sequential execution. An optimization that sometimes reduces this overhead is shown in

| | |
|---|--|
| <pre> for (i=0; i<16; i++) if (a[i] != 0) b[i]++; </pre> | <pre> for (i=0; i<16; i+=4){ if (a[i+0] != 0) b[i+0]++; if (a[i+1] != 0) b[i+1]++; if (a[i+2] != 0) b[i+2]++; if (a[i+3] != 0) b[i+3]++; } </pre> |
| (a) Original with control flow | (b) Unrolled |
| <pre> for (i=0; i<16; i+=4){ Vcond = a[i:i+3] != (0, 0, 0, 0); Vtemp = b[i:i+3] + (1, 1, 1, 1); b[i:i+3] = Combine b[i:i+3] and \ Vtemp according to Vcond } </pre> | <pre> for (i=0; i<16; i+=4){ Vcond = a[i:i+3] != (0, 0, 0, 0); branch to L1 if Vcond is all false Vtemp = b[i:i+3] + (1, 1, 1, 1); b[i:i+3] = Combine b[i:i+3] and \ Vtemp according to Vcond L1: } </pre> |
| (c) Parallelized | (d) Overhead reduced |

Figure 1.4: Example: SLP in the presence of control flow.

Figure 1.4(d). We can bypass the parallel code when all fields of the parallel comparison are false.

1.6.2 Superword-Level Locality

Reducing memory references is even more important when computations are parallelized as discussed in Section 1.3. In this section, we use a simple example to illustrate our approach to reduce memory references by storing data in superword registers. In the sequential code shown in Figure 1.5(a), $A[i][j]$ and $A[i-1][j]$ access the same data in memory after 32 iterations. Also $B[j]$ accesses the same memory address after 32 iterations. If we can keep a data element in a register until it is used again, the later memory access can be eliminated. To exploit superword registers, the loop is first parallelized as shown in (b). Assuming four array elements fit in one superword register, the number of memory accesses is reduced by 4X when the j -loop is parallelized. This reduction is the result of accessing four adjacent array elements in one superword memory access. Still,

| | |
|---|--|
| <pre> for(i=1;i<=32;i++) for(j=0;j<32;j++) A[i][j] = A[i-1][j] + B[j]; </pre> <p style="text-align: center;">(a) Original</p> | <pre> for(i=1; i<=32;i++) for(j=0; j<32; j+=4) A[i][j:j+3] = A[i-1][j:j+3]+B[j:j+3]; </pre> <p style="text-align: center;">(b) SLP exploited</p> |
| <pre> for(i=1; i<=32; i+=2) for(j=0; j<32; j+=4) { A[i][j:j+3] = A[i-1][j:j+3]+B[j:j+3]; A[i+1][j:j+3] = A[i][j:j+3]+B[j:j+3]; } </pre> <p style="text-align: center;">(c) Unroll-and-jam applied</p> | <pre> for(i=1;i<=32; i+=2) for(j=0; j<32; j+=4) { SV1 = B[j:j+3]; SV2 = A[i-1][j:j+3] + SV1; A[i+1][j:j+3] = SV2 + SV1; A[i][j:j+3] = SV2; } </pre> <p style="text-align: center;">(d) Memory accesses reduced</p> |

Figure 1.5: Motivating example for exploiting superword-level locality.

we can reduce memory accesses further by keeping a superword written by $A[i][j:j+3]$ in a superword register until read by $A[i-1][j:j+3]$. Since the two superword memory accesses are apart by one iteration of the outer loop (i-loop), we apply unroll-and-jam as shown in (c) so that the two superword memory references access the same data within the same iteration. Most existing compilers fail to remove the redundant memory accesses because they do not allocate registers for array references. For this reason, we replace the redundant superword memory accesses with superword variables. Subsequently, a backend compiler will allocate superword variables to superword registers. The number of memory accesses is consequently reduced as shown in (d). The loop body in (c) has six memory references that are reduced to four in (d) achieving the reduction of memory accesses by 1.5X in addition to the previous reduction of 4X. Overall, a 6X reduction in memory accesses is achieved from (a) to (d).

1.7 Contributions

The contribution of this thesis is the new optimizations for the architectures supporting superword-level parallelism (SLP) as follows.

An algorithm to exploit SLP in the presence of control flow. SLP exploits parallelism within a basic block limiting its applicability. We describe how to extend the concept of SLP in the presence of control flow constructs. A key insight is that we can use techniques related to optimizations for architectures supporting predicated execution, even for multimedia ISAs that do not provide hardware predication. We derive large basic blocks with predicated instructions to which SLP can be applied. We describe how to minimize overheads for superword predicates and re-introduce control flow for scalar operations. We observe speedups on 8 multimedia codes ranging from 1.97 to 15.00 as compared to both sequential execution and SLP alone.

As an optimization on the code parallelized for control flow, we also evaluate the costs and benefits of exploiting branches on the aggregate condition codes associated with the fields of a superword such as the branch-on-any instruction of the AltiVec. *Branch-on-superword-condition-codes* (BOSCC) instructions allow fast detection of aggregate conditions, an optimization opportunity often found in multimedia applications such as image processing and pattern matching. Our experimental results show speedups of up to 1.40 on 8 multimedia kernels when BOSCC instructions are used.

Compiler controlled caching in superword registers. Accessing data from superword registers, versus a cache or main memory, has two advantages, i.e., removing memory access instructions and their latencies. We treat the large superword register file as a compiler-controlled cache, thus avoiding unnecessary memory accesses by exploiting reuse in superword registers. This research is distinguished from previous work on exploiting reuse in scalar registers because it considers not only temporal but also spatial reuse. As compared to optimizations to exploit reuse in

cache, the compiler must also manage replacement, and thus, explicitly name registers in the generated code. In a study on 14 benchmarks, our results show speedups ranging from 1.40 to 8.69 as compared to using the original SLP compiler, and we eliminate the majority of memory accesses.

Implementation and evaluation of the proposed techniques. The techniques presented in this thesis are fully implemented and evaluated on 14 benchmarks. Our implementation includes additional code generation techniques not supported by the original SLP compiler. The automatically generated parallel C programs are compiled by the backend compiler and run on the PowerPC G4. The overall speedups achieved by our implementation combining all optimizations range from 1.05 to 19.22.

DIVA-specific optimizations. We developed a compiler algorithm and several optimization techniques to exploit a DRAM memory characteristic (*page-mode*) automatically. A page-mode memory access exploits a form of spatial locality, where the data item is in the same row of the memory buffer as the previous access. Thus, access time is reduced because the cost of row selection is eliminated. The algorithm increases frequency of page-mode accesses by reordering data accesses, grouping together accesses to the same memory row. We implemented this algorithm and present speedup results for four multimedia kernels ranging from 1.25 to 2.19 over the SLP algorithm alone for a PIM embedded DRAM device, called DIVA.

The remainder of this thesis is organized as follows. The next chapter provides definitions and background on the existing techniques we build upon in this work. These techniques include the algorithm to exploit SLP and predicate analysis used in our approach to exploit SLP in the presence of control flow. In Chapter 3, we describe our approach to exploit SLP in the presence of control flow. Our technique to exploit superword registers as a compiler-controlled cache is described in Chapter 4. Several optimizations related

to code generation are presented in Chapter 5. The implementation of the techniques in the previous chapters and its evaluation are described in Chapter 6. Chapter 7 describes the DIVA ISA and DIVA-specific optimizations. Related work is described in Chapter 8 followed by our conclusion in Chapter 9.

Chapter 2

BACKGROUND

The techniques presented in this thesis leverage large body of work on analyses for parallelizing compilers. In this chapter, we describe the existing analyses and code transformations used in our approach. Data dependence information described in Section 2.1 is crucial in applying code transformations which are not always legal. Both the superword-level parallelization (SLP) and superword-level locality (SLL) algorithms require data dependence analysis. In Section 2.2, we describe data reuse which is a core concept in the SLL algorithm. In Section 2.3, Larsen and Amarasinghe’s SLP compiler [39] is presented. All our techniques presented in this thesis are based on their SLP compiler. In the last section of this chapter, we describe a predicate analysis necessary for our extension of SLP in the presence of control flow.

2.1 Data Dependence

There exists a data dependence between two instructions if the two instructions access the same data and at least one of them writes to the data. Given two instructions I_1 and I_2 , I_2 cannot be executed before I_1 if there is a dependence from I_1 to I_2 . Three kinds of data dependences can prevent the reordering of the two instructions.

True dependence I_1 writes to a data item which is read by I_2 .

Anti-dependence I_1 reads a data item which is written by I_2 .

| | |
|--|--|
| <pre> for(i=1;i<=32;i++) for(j=0;j<32;j++) A[i][j] = A[i-1][j] + B[j]; </pre> <p style="text-align: center;">(a) Data dependence</p> | <pre> for(i=1;i<=32;i+=2) for(j=0;j<32;j++){ A[i][j] = A[i-1][j] + B[j]; A[i+1][j] = A[i][j] + B[j]; } </pre> <p style="text-align: center;">(b) Unroll-and-jam on i-loop by 2</p> |
|--|--|

Figure 2.1: An example code to show dependence vectors and unroll-and-jam.

Output dependence I_1 writes to a data item which is also written by I_2 .

Input dependence exists between two instructions when the two instructions read the same datum. However, input dependence does not impose ordering constraints among instructions. The *iteration space* of an n -deep loop nest is an n -dimensional polyhedron where a value on each dimension represents the value of the loop index variable of the corresponding loop. Each point in the iteration space represents an iteration of the loop nest whose loop indices are denoted by the position vector of the corresponding point in the iteration space.

Data dependence between two distinct array references in an n -deep loop nest can be represented in a form of *dependence vector*, $d = \langle d_1, d_2, \dots, d_n \rangle$ [4]. A dependence vector captures the vector distance, in terms of the loop iterations, such that the two references may map to the same memory location. Each vector element d_i may be either a constant integer, + (a positive direction where the distance is not fixed), - (a negative direction), or * (the direction and distance are unknown). We refer to a dependence vector as being *lexicographically positive* if the first non-zero d_i is + or a positive integer. A dependence vector is said to be *consistent* if the dependence distance in the iteration space is constant. Figure 2.1(a) shows an example loop nest which contains three array references. There is a true dependence from $A[i][j]$ to $A[i-1][j]$ and the dependence vector is $\langle 1, 0 \rangle$. This means that 1 iteration of i-loop after $A[i][j]$ accesses a data element in memory, $A[i-1][j]$ accesses the same data.

A data dependence is *loop-independent* if the associated pair of instructions access the same data from the same iteration. Otherwise, the data dependence is *loop-carried*. All loop-carried dependence vectors are lexicographically positive. A code transformation preserves the semantics of a program if the ordering constraints imposed by the dependence vectors are not violated.

Data dependences for a set of instructions can be represented by a directed graph called *data dependence graph*. In this graph, a node represents an instruction and a directed edge from one node to another represents a data dependence between the associated instructions.

2.2 Data Reuse

A datum in memory is said to be *reused* if used multiple times. *Reuse distance* is defined as the number of iterations between two uses of the same data. Data dependence and data reuse are similar by nature because both look for the instructions that use the same datum. However, not all data dependences translate to data reuse and vice versa. On one hand, anti-dependence prevents code reordering but is not a reuse opportunity. On the other hand, input dependence does not prevent code reordering but is a reuse opportunity. In an output dependence between two instructions, the datum itself is not reused. However, we consider output dependences as reuse opportunities since we can eliminate the earlier store instruction.

Reuse can be categorized in two different ways. The first concerns whether the same or distinct parts of a datum are reused. If distinct data elements are used from a superword register, it is called *spatial reuse* in the superword register. If the same datum is used repeatedly from a superword register, we call it *temporal reuse*. The other categorization concerns whether two accesses to the same datum are originated from the same or different static instructions. If two dynamic accesses to the same datum are from

one static instruction, it is called *self reuse*, or otherwise *group reuse*. The two orthogonal categorizations of reuse can be combined to generate four more detailed reuse types: *self-spatial*, *self-temporal*, *group-spatial*, and *group-temporal*.

Our analysis is for array references whose array subscript expressions are an *affine* function of loop index variables. In other words, each array subscript expression is a linear function of loop index variables $f(L_1, \dots, L_n) = a_1L_1 + a_2L_2 + \dots + a_nL_n + b$ where a_i and b are constants and L_i are loop index variables. For the array references with non-affine array subscript expressions (e.g., $A[B[i]]$), we make conservative estimations. Two array references with affine array subscript expressions are *uniformly generated* if each array subscript expression of one array reference is different from the corresponding array subscript expression of the other array reference only by a constant term [69]. In Figure 2.1(a), $A[i][j]$ and $A[i-1][j]$ are uniformly generated.

Related data structures are use-definition (UD) chain and definition-use (DU) chain. *Use-definition (UD) chain* is a list of definitions of a variable that reach a particular use of the variable. Similarly, *definition-use (DU) chain* is a list of uses of a variable reached by the same definition. UD-chains and DU-chains are conveniently used in the SLP algorithm described in the next section.

2.3 The SLP algorithm

The SLP algorithm finds SIMD parallelism within a basic block. In Section 1.1, we have presented the SLP algorithm using an example. In this section, we describe the algorithm in detail. In the next section, we present alignment analysis, which is used by the algorithm to find alignment offsets of memory accesses. In Section 2.3.2, we describe distance analysis, which is used to find adjacency between memory references. The information obtained from these two analyses is used in the main algorithm presented in Section 2.3.3.

| | |
|----------|---|
| \sqcap | $a = \gcd(a_1, a_2, b_1 - b_2), b = b_1 \bmod a$ |
| $+$ | $a = \gcd(a_1, a_2), b = (b_1 + b_2) \bmod a$ |
| $-$ | $a = \gcd(a_1, a_2), b = (b_1 - b_2) \bmod a$ |
| \times | $a = \gcd(a_1 a_2, a_1 b_2, a_2 b_1, M), b = b_1 b_2 \bmod a$ |

Table 2.1: Transfer functions for the dataflow analysis to find alignments when $a_1 n + b_1$ and $a_2 m + b_2$ are given.

2.3.1 Alignment Analysis

The architectures supporting SLP either require memory accesses to be aligned or allow unaligned memory accesses but at a higher cost. As a result, finding alignment offsets of memory references is at least a performance issue and sometimes, a correctness problem. Alignment analysis described in this section is used to find an alignment offset of each memory reference. For aligned memory references, the result of this analysis is a constant alignment offset representing all runtime addresses. Other memory references are either known to be unaligned or the alignment offset is unknown for the lack of necessary information. The result of this analysis is used by the SLP compiler to pack only aligned memory references. Consequently, only aligned memory references are parallelized.

To determine the alignment offsets of memory references, an iterative dataflow analysis is used [40]. Variables and constants are associated with a linear expression $an + b$ to represent the set of values they can have. Here, a is a stride, b is an offset, and n is a set of non-negative integers. Initially, a constant D is associated with $Mn + d$ where M is the superword width and $d = D \bmod M$. Since we have a way to allocate array objects to superword boundaries, array base addresses are initialized with $Mn + 0$. All other variables are initialized with \top . To propagate element values, the transfer functions listed in Table 2.1 are used. The *meet* operator (\sqcap) is used to merge control flow. All operations not listed in the table result in $n + 0$ which is equivalent to \perp .

| | |
|----------|--|
| \sqcap | $T_1 == T_2 ? T_1 : \perp$ |
| $+$ | $b_1 == b_2 ? b = b_1, c = c_1 + c_2 : \perp$ |
| $-$ | $b_1 == b_2 ? b = b_1, c = c_1 - c_2 : \perp$ |
| \times | $T_1 \text{ is constant } N ? b = b_1, c = c_2 \times N : \perp$ |

Table 2.2: Transfer functions for distance analysis when two terms $T_1 = (X, c_1, b_1)$ and $T_2 = (X, c_2, b_2)$ of two linear expressions for the same variable X are given.

2.3.2 Distance Analysis

Two memory references must be adjacent with each other to be packed in a superword memory reference. However, the adjacency of two memory references cannot be determined by the alignment analysis. Distance analysis employs an iterative dataflow analysis to find distances in memory address among memory references. The result of this analysis is a partitioning of memory references into groups, in each of which memory references are assigned a constant representing an offset from a reference address. Two memory references in a group are adjacent to each other if the assigned constants are different by the type size of the operand.

Each variable used in address computations is represented by a linear expression of all such variables. Each term in the linear expression is a tuple of three values representing a variable symbol, a coefficient, and a basis indicating the initialization point of the variable. Initially, all variables have only one nonzero term which is the variable itself associated with a coefficient 1 and a basis of zero. In other words, for each variable X , X is initialized with a tuple $(X, 1, 0)$. As the instructions are processed, the linear expression of the destination variable is replaced by the linear expression resulting from evaluating the right hand side of the instruction. Table 2.2 shows transfer functions used in this analysis. After each instruction is processed based on the transfer function, all variables associated with bottom (\perp) are reassigned by the variable itself, a coefficient of 1, and the current instruction number as a new basis.

At the end of the dataflow analysis, the address operands of memory references are associated with a linear equation consisting of variables which cannot be replaced further by other variables. All memory references whose address operands are different by a constant term are grouped together and assigned a unique group ID. Each instruction in a group is annotated with a pair of group ID and the constant term in the linear expression for the address operand. Two memory references in a group are guaranteed to always keep a constant distance during run time.

2.3.3 Packing

The SLP algorithm starts packing instructions from memory references [39]. Two memory references are packed together if both are aligned, adjacent, and their alignment offsets do not cross any superword boundary. Next, UD-chains and DU-chains are followed to pack the instructions defining the source operands or using the destination operands of already packed instructions. The instructions packed at this time inherit the alignment offsets from the instructions packed already. Since there is a limit in number of data elements that can be packed in a superword register, to represent the maximum amount of parallelism we define *superword size* (SWS) as the number of data elements that fit in a superword. The SLP algorithm packs at most SWS instructions for each superword instruction.

As a last step, the algorithm schedules instructions sequentially. At this point, instructions can be either packed into a superword instruction or left as a scalar instruction. In any case, all instructions on which the current instruction is dependent should be scheduled already before the current instruction is scheduled. Sometimes, cyclic dependences prevent further scheduling of the unscheduled instructions. To break the dependence cycles, the first unscheduled packed instructions are unpacked into scalar instructions. Figure 2.2 shows an example code after each step of the algorithm.

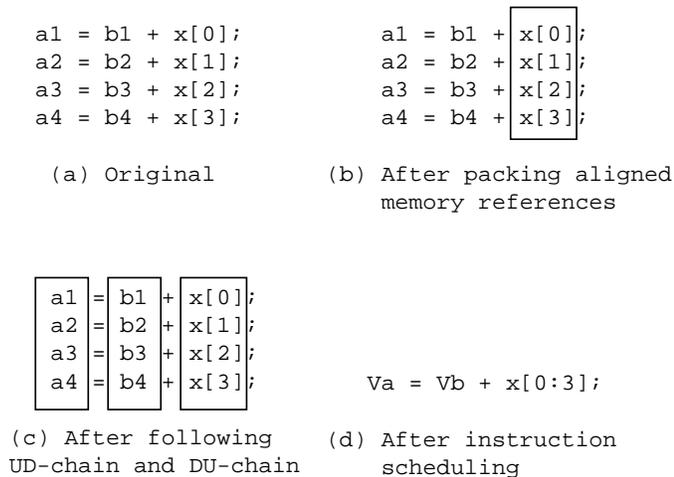


Figure 2.2: An example showing the packing algorithm.

2.3.4 Summary

The SLP algorithm presented in this section works within a basic block. To generate SIMD parallel instructions, the SLP algorithm looks for the isomorphic scalar instructions that can be replaced by a superword instruction. Packing memory references should satisfy further constraints imposed by the target architecture, *i.e.*, alignment to superword boundaries and packed data elements in memory. Alignment analysis and distance analysis provide the information necessary to facilitate the requirements.

2.4 Predicate Analysis

Predicates are introduced when if-conversion is applied to control flow constructs. The predicates guard the execution of the instructions that used to be guarded by conditional statements. However, the analyses based on CFG cannot be used to extract, from a basic block of predicated instructions, the necessary information such as data dependence and reaching definition. Instead, we borrow analyses developed for the architectures supporting predicated execution.

In this section, we describe predicated execution and if-conversion in Section 2.4.1 and Section 2.4.2 respectively. After if-conversion is applied, instructions may be guarded by predicates. For the following passes to extract the necessary information, we use Scott Mahlke’s predicate analysis based on *predicate hierarchy graph* (PHG) [44]. Predicate hierarchy graph is described in Section 2.4.3. *Mutually exclusive* and *predicate covering* are two important relations among predicates described in Section 2.4.4 and Section 2.4.5 respectively. Since our algorithm to restore control flow from a sequence of predicated instruction is based on Mahlke’s *predicate CFG generator*, it is described in Section 2.4.6.

2.4.1 Predicated Execution

Recently, several architectures supporting predicated execution are developed [36]. Since predicated execution is one of the core concepts in our approach, its notation and semantics are described in this subsection. In the architectures supporting predicated execution, instructions are first executed and then the result is committed if the guarding predicate is true, or otherwise nullified. In this thesis, an instruction guarded by a predicate **pred** is denoted as follows.

```
dst = operation; <pred>
```

If **pred** is true, **dst** is updated by the operation’s result. If **pred** is false, however, **dst** remains unchanged.

pset is predicate defining instruction and the syntax is shown below.

```
pT, pF = pset(cond); <pred>
```

pset takes one source operand and two destination operands. The source operand is the result of a previous comparison operation and the two destination operands are predicate variables that can be used to guard the subsequent instructions. A **pset** itself can also be guarded by another predicate just like any other instructions. The semantics of **pset**

is that $pT = \text{cond}$ and $pF = \text{!cond}$ when pred is true. If pred is false, both pT and pF remain unchanged.

2.4.2 If-conversion: RK-Algorithm

If-conversion is a process of removing control flow by introducing predicates to instructions. For if-conversion, we use Park and Schlansker's RK-algorithm [55]. Their algorithm consists of two main functions, R and K. R function associates each node in CFG to a predicate and K function finds the locations to insert predicate defining operations for each predicate. The detailed explanation of R function requires the following definition.

Definition 1 *Let (X, Y, label) be an edge in a CFG such that Y does not postdominate X . The nodes control dependent on this edge are those and only those of the unique path starting (excluding the first) from the immediate postdominator of X to Y in the postdominator tree.*

R function is best described as a partitioning function of basic blocks under a certain equivalence relation. Two basic blocks x and y are in an equivalence class if they are control dependent on the same set of basic blocks. R function is obtained as follows. First, for each basic block b , a set of basic blocks on which b is control dependent are obtained. Then, the basic blocks that are control dependent on the same set of basic blocks are grouped into an equivalence class. A unique predicate variable is assigned to each equivalence class.

A *control dependence set* for a basic block is a set of edges on which the basic block is control dependent. For each predicate, K function defines a control dependence set, on the element edge of which an equivalence class of basic blocks represented by the predicate are control dependent. Predicate defining operations for the associated predicate are generated for each edge in the control dependence set.

Because of the semantics of their predicate defining operations, predicates may not be defined along all possible paths. To always define predicates before being used, the predicates that may have undefined paths are initially set to *false*. This algorithm achieves optimality in terms of the number of predicates being used and the number of predicate defining instructions.

2.4.3 Predicate Hierarchy Graph (PHG)

The predicate analysis starts by building a *predicate hierarchy graph* (PHG) defined as follows.

Definition 2 *A predicate hierarchy graph (PHG) is a directed acyclic graph representing nesting relations among predicates in a predicated basic block.*

A PHG consists of two types of nodes, *predicate nodes* and *condition nodes*, and is constructed as follows. Starting with a single predicate node representing a constant *true*, each instruction is examined in textual order. For each instruction that defines predicates, such as, for example `pT, pF = pset(comp) <pParent>;`, at most one condition node is created. For this example, a condition node for `comp` would be created. An edge is inserted from the predicate node for the predicate guarding the instruction to the condition node just created; for the example, an edge from predicate node `pParent` to condition node `comp` is added. Two predicate nodes for `pT` and `pF` are also created if they do not already exist. They may have been introduced into the PHG by a prior definition, in cases where multiple control flow paths merge. Then, edges are inserted from the condition node to the two predicate nodes; in the example, there would be two edges inserted from condition node `comp` to predicate node `pT` and `pF`, representing the true and false values of a comparison. This process is repeated for each instruction that defines predicates. The resulting PHG permits analysis to reason about the relationship

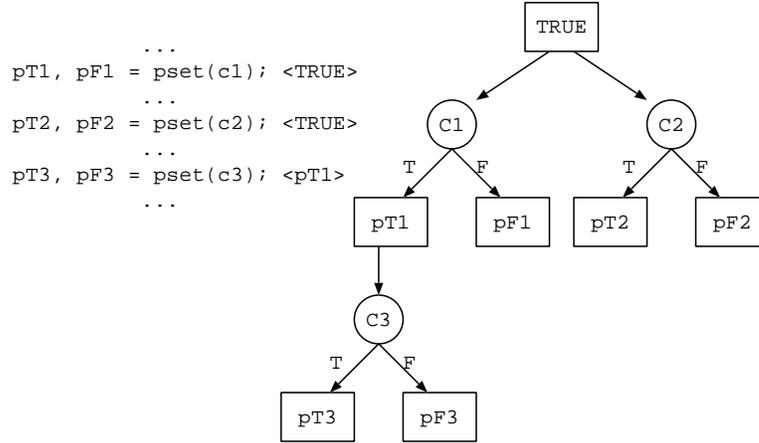


Figure 2.3: An example showing construction of a PHG.

among predicates. Figure 2.3 shows an example of predicated instructions and a PHG built from the code sequence.

2.4.4 Mutually Exclusive

Mutually exclusive relation between two predicates is defined as follows.

Definition 3 *Two predicates $p1$ and $p2$ are mutually exclusive if they are never simultaneously true, i.e., $p1 \wedge p2 = false$.*

The information on mutually exclusive relation is useful in various situations. For example, it can be used to discern dependences more accurately. Consider the following instructions defining a variable. There is no output dependence between them if the two guarding predicates, $p1$ and $p2$, are mutually exclusive.

```

a = 5;   <p1>
a = 7;   <p2>

```

To find whether two given predicates $p1$ and $p2$ are mutually exclusive, the PHG is traversed backward along all paths from the predicate nodes for $p1$ and $p2$. Then, a set of *merge nodes* is obtained by picking the nodes where two backward traversals first meet. $p1$ and $p2$ are mutually exclusive if the two backward traversals from $p1$ and $p2$ merge

from complementary edges at all merge nodes. In the example of Figure 2.3, pT3 and pF1 are mutually exclusive because there is only one merge node which is condition node C1 and the backward traversals merge from two complementary edges.

2.4.5 Predicate Covering

Predicate covering is a relation between a predicate and a set of predicates. This information is used in both restoring control flow and inserting `select` instructions. Predicate covering is defined as follows.

Definition 4 *A predicate p is said to be covered by a set of predicates G if $p = true \Rightarrow \exists p' \in G$ such that $p' = true$.*

Predicate covering relation between a set of predicates G and a predicate p is determined as follows. For each predicate in G , mark the corresponding predicate node in the PHG as covered. Then, apply the above definition repeatedly to propagate predicate covering to adjacent nodes until no further changes can be made. If the predicate node for p in the PHG is marked as covered, p is covered by G .

A related definition, *predicate-covering predecessor* is used to restore control flow and defined as follows.

Definition 5 *An instruction I guarded by a predicate p is a predicate-covering predecessor of a later instruction I' guarded by a predicate p' iff p and p' are not mutually exclusive and neither p nor p' is covered by a set of predicates associated with the instructions between I and I' .*

For a given instruction I associated with a predicate p , its predicate-covering predecessor instructions are obtained by scanning backward the given instruction sequence. An instruction I' associated with a predicate p' , in the instruction sequence, is a predicate-covering predecessor of I if p and p' are not mutually exclusive and p' is not already

marked as covered in the PHG. After placing I' into a predicate-covering predecessor set of I , the node for p' is marked as *covered* in the PHG and the newly covered predicate is propagated in the PHG to mark other covered predicates. This backward scan stops when the predicate node for p is covered.

Based on *mutually exclusive* relation among predicates and the definition of *predicate covering*, reaching definitions can be found as follows.

Definition 6 *A definition d guarded by a predicate p reaches a later use u guarded by a predicate p' in the same basic block if p and p' are not mutually exclusive and neither p nor p' is covered by a set of predicates associated with the instructions, defining the same variable, between d and u .*

2.4.6 Predicate CFG Generator

Given a sequence of predicated instructions, predicate CFG generator is used to restore the embedded control flow. The key idea is to find a set of predicate-covering predecessors of each predicated instruction successively and make connections in the CFG from each of the predicate covering predecessors to the current instruction being processed.

While the original Mahlke's predicate CFG generator scan forward to find predicate-covering successors, we modify it to scan backward. This modification was necessary for our improvement described in the next chapter.

Chapter 3

SUPERWORD-LEVEL PARALLELISM IN THE PRESENCE OF CONTROL FLOW

Many multimedia applications have control flow in their key computations as discussed in Chapter 1. However, the SLP algorithm cannot exploit parallelism when the loop body has control flow because it works only within a basic block. Thus, exploiting SLP in the presence of control flow is an important issue that still needs to be addressed.

In this chapter, we describe our approach employed to address the issue. A key insight is that we can borrow heavily from optimizations developed for architectures supporting wide-issue instruction-level parallelism and predicated execution, such as, for example, the Itanium family of processors [36], *even for architectures such as the AltiVec that do not support predicated execution*. There are two reasons why similar optimization techniques can be used for these two distinct classes of architectures:

- SLP and ILP optimizations operate within basic blocks. Control flow limits the size of basic blocks, and thus limits optimization opportunities. We derive large basic blocks with predicated instructions to which SLP can be applied.
- A commonality in multimedia extension ISAs is what we will call a `select` operation for merging the results of different control flow paths. Based on the value of a boolean superword, individual fields from two different inputs are combined and

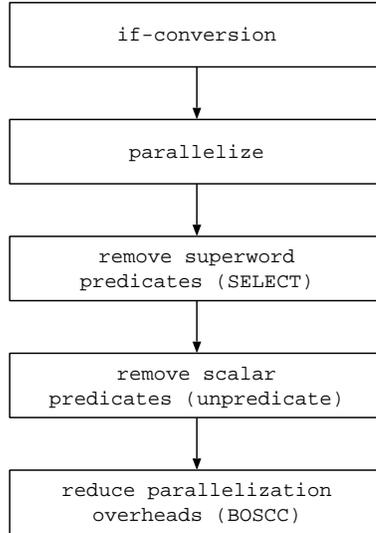


Figure 3.1: Overview of the algorithm to exploit SLP in the presence of control flow.

committed to a final result. Thus, `select` instructions appear similar to predicated instructions, even though the underlying hardware mechanisms to implement the two are very different.

We derive a large basic block of predicated instructions by applying if-conversion. Then, the SLP algorithm is applied to parallelize isomorphic instructions. Following if-conversion and parallelization, the resulting basic block may contain both scalar and superword instructions, and in some cases, the instructions are predicated. The compiler’s job is to remove these predicates. We discuss how superword predicates are removed by inserting `select` operations in Section 3.2 and how scalar predicates are removed through an algorithm we call *unpredicate* in Section 3.3. In Section 3.4, we describe how parallelization overhead can be reduced by introducing *Branch-On-Superword-Condition-Code* (BOSCC). Prior to these descriptions, we outline our approach in the next section.

3.1 Overview of the Algorithm

The algorithm to exploit SLP in the presence of control flow consists of five steps which are applied in sequence as shown in Figure 3.1. Each of the five steps are summarized below. We use an example of Figure 3.2(a) to show the changes made at different steps.

Step 1: Applying if-conversion. As in the original SLP algorithm, first the innermost loop is unrolled by superword size. When there is control flow in the loop body, however, the basic block size is not increased even after unrolling because of the if-statements. To apply if-conversion, we look for the largest acyclic control flow structure of single entry and exit from the innermost loop body. As shown in Figure 3.2(b), the code is unrolled by a factor of four, based on the assumption that the superword register width is sixteen bytes and the array type sizes are four bytes. Next, *if-conversion* using Park and Schlansker’s algorithm [55] is applied to convert control dependences into data dependences. Now, associated with each instruction is a *predicate*, shown in parenthesis at the end of the instruction, that captures the conditions that must be true for the instruction to execute. The `pset` instruction initializes the value of the predicates `pT1` and `pF1` based on the value of the condition represented by `comp1`.

Step 2: Parallelization. After if-conversion, the loop body becomes one basic block of predicated instructions. A modified version of the SLP parallelizer, which packs together isomorphic instructions with their predicates, derives a mix of predicated scalar and superword instructions. This modification includes predicate analysis to find dependences among instructions. The resulting code is shown in Figure 3.2(c). While some instructions are parallelized, several scalar statements remain unparallelized because of data dependences. Since our target architectures do not support predicated execution, both superword and scalar predicates should be removed.

| | |
|---|--|
| <pre> for(i=0; i<1024; i++){ if(fblue[i] != 255){ bblue[i] = fblue[i]; bred[i+1] = bred[i]; } } </pre> | <pre> for(i=0; i<1024; i+=4){ comp1 = fblue[i] != 255; pT1, pF1 = pset(comp1); bblue[i] = fblue[i]; (pT1) bred[i+1] = bred[i]; (pT1) ... } </pre> |
| (a) Original | (b) Unrolled and if-converted |
| <pre> for(i=0; i<1024; i+=4){ vc = fblue[i:i+3] != (255,255,255,255); v_pT, v_pF = v_pset(vc); bblue[i:i+3] = fblue[i:i+3]; (v_pT) pT1, pT2, pT3, pT4 = unpack(v_pT); bred[i+1] = bred[i]; (pT1) bred[i+2] = bred[i+1]; (pT2) bred[i+3] = bred[i+2]; (pT3) bred[i+4] = bred[i+3]; (pT4) } </pre> | <pre> for(i=0; i<1024; i+=4){ vc = fblue[i:i+3] != (255,255,255,255); v_pT, v_pF = v_pset(vc); bblue[i:i+3] = select(bblue[i:i+3], \ fblue[i:i+3], v_pT); pT1, pT2, pT3, pT4 = unpack(v_pT); bred[i+1] = bred[i]; (pT1) bred[i+2] = bred[i+1]; (pT2) bred[i+3] = bred[i+2]; (pT3) bred[i+4] = bred[i+3]; (pT4) } </pre> |
| (c) Parallelized | (d) Select applied |
| <pre> for(i=0; i<1024; i+=4){ vc = fblue[i:i+3] != (255,255,255,255); v_pT, v_pF = v_pset(vc); bblue[i:i+3] = select(bblue[i:i+3], \ fblue[i:i+3], v_pT); pT1, pT2, pT3, pT4 = unpack(v_pT); if(pT1) bred[i+1] = bred[i]; if(pT2) bred[i+2] = bred[i+1]; if(pT3) bred[i+3] = bred[i+2]; if(pT4) bred[i+4] = bred[i+3]; } </pre> | <pre> for(i=0; i<1024; i+=4){ vc = fblue[i:i+3] != (255,255,255,255); v_pT, v_pF = v_pset(vc); branch to L1 if v_pT is all false; bblue[i:i+3] = select(bblue[i:i+3], \ fblue[i:i+3], v_pT); L1: pT1, pT2, pT3, pT4 = unpack(v_pT); if(pT1) bred[i+1] = bred[i]; if(pT2) bred[i+2] = bred[i+1]; if(pT3) bred[i+3] = bred[i+2]; if(pT4) bred[i+4] = bred[i+3]; } </pre> |
| (e) Unpredicated | (f) Overhead reduced |

Figure 3.2: Example illustrating steps of SLP compilation in the presence of control flow.

$$\boxed{3} \boxed{2} \boxed{3} \boxed{2} = \text{SELECT}(\boxed{2} \boxed{2} \boxed{2} \boxed{2}, \boxed{3} \boxed{3} \boxed{3} \boxed{3}, \boxed{1} \boxed{0} \boxed{1} \boxed{0});$$

Figure 3.3: Merging two superwords using a `select` instruction.

Step 3: Eliminating superword predicates. In Figure 3.2(d), we show how a superword `select` operation can be used to select individual fields from two superword definitions according to the value of a superword predicate variable. Concretely, the effect of the `select` operation “`dst = select(src1, src2, mask)`”, is to assign `src2` to `dst` for the fields where the corresponding `mask` bit is 1. Otherwise, `src1` is assigned to `dst`. Figure 3.3 shows this graphically using superwords of 4 scalar elements. Note that the effect of this transformation is to execute both control flow paths and select the value from the one that would have executed in the scalar version of the code. Thus, the parallelization overhead includes the `select` instructions, and the cost of executing both paths. In Section 3.2, we describe how to minimize the number of `select` instructions to reduce this overhead.

Step 4: Eliminating scalar predicates. Next, we restore the control flow for the predicated scalar operations, as shown in Figure 3.2(e). While it is straightforward to insert control flow corresponding to the predicate on the instruction, this strategy could result in an enormous amount of additional branches as compared to the original scalar code. Thus, another important optimization is minimizing the branches, with an attempt to recover as close as possible the control flow of the original scalar code, as described in Section 3.3.

Step 5: Reducing parallelization overheads. The code in Figure 3.2(e) produced by applying the previous four steps can be compiled and run successfully to generate correct results. However, it suffers from the cost of always executing both control flow paths and the extra `select` instruction, which may offset the benefits of parallelism. The code in Figure 3.2(f) takes advantage of a common instruction supported by multimedia

extensions, *branch-on-superword-condition-codes* (BOSCC), which checks the aggregate value of the condition codes associated with each field of a superword predicate. For example, a branch-on-none instruction can be thought of as an *AND* of the condition codes of all fields of a superword, that is, a branch is taken if none of these condition codes is true. The parallelization overheads may be significantly reduced if the expression associated with the BOSCC (`branch`) is false most of the time.

Discussion. The approach described above has been heavily influenced by features of the ISA of the target architectures, as well as the current organization of the SLP compiler, where we treat the SLP pass as a black box and feed it large basic blocks for parallelization. If the target architecture supported *masked* superword operations [62] and predicated scalar execution [55, 35], the code in Figure 3.2(c) would not need any further transformations for SLP. The DIVA ISA supports masked superword operations, but not predicated execution, and the PowerPC AltiVec, the other platform for our work, supports neither. Thus the compiler must eliminate the predicates on scalar instructions by restoring control flow, and for architectures including the AltiVec, replace the predicated superword instructions with `select` instructions that achieve the same effect.

If the architecture combined SLP support and predication, we could adapt recently-developed algorithms by Chuang et. al. to generate *phi-instructions* from the CFG of a scalar code to resolve the *multiple-definition problem* in architectures that support predicated execution [16]. Their phi-instruction is a scalar analog to the superword `select` instruction. While it is possible to use the phi-predicated code as an input to SLP, some scalar phi-instructions would remain and scalar control flow may nevertheless need to be restored in architectures such as the AltiVec.

| | | |
|--|---|---|
| <pre> if (b<0){ a = 1; }else{ a = 0; } .. = a; </pre> | <pre> Vp, Vnp = v_pset(Vb<V0) Va = V1 (Vp) Va = V0 (Vnp) ... = Va </pre> | <pre> Vp, Vnp = Vb < V0 Va1 = V1 Va = select(Va, Va1, Vp) Va2 = V0 Va = select(Va, Va2, Vnp) ... = Va </pre> |
| (a) scalar | (b) parallelized intermediate form | (c) naive generation of select |

Figure 3.4: Merging two superword definitions.

3.2 Eliminating Superword Predicates

In this section, we show how to remove superword predicates while preserving the semantics of the original program through the use of **select** operations. Figure 3.4(a) shows an example sequential code. After if-conversion and parallelization, the control flow is removed and some instructions are guarded by superword predicates shown in parentheses as in Figure 3.4(b). The first instruction defines a superword predicate Vp and its complement Vnp . A field of Vp is set to true if the result of the comparison is true, and the fields of Vnp are set to the complement of the corresponding fields of Vp . To generate the final code, it is incorrect to simply remove the superword predicates; for example, the first definition of Va would be killed by the second definition. Instead, we rename the second definition and use a **select** instruction to merge their values into one superword variable as shown in Figure 3.4(c).

Figure 3.5 presents the algorithm that generates the minimum number of **select** instructions required to preserve the original program’s behavior. A **select** instruction is required for some but not all definitions of superword variables, as will be discussed below.

Given a parallelized code with instructions guarded by predicates, we first build a predicate hierarchy graph (PHG) as defined in Section 2.4.3 [44]. At this stage, instructions guarded by both scalar predicates and superword predicates can be intermixed.

Algorithm SEL: Given a sequence of predicated instructions IN, remove superword predicates from all superword instructions by generating `select` instructions.

```

Build a predicate hierarchy graph(PHG)
DU-chain and UD-chain are built based on Definition 6 using IN and PHG
for each definition  $d : V = s_1 \text{ op } s_2 (P)$ 
    NeedSelect  $\leftarrow$  false
    for each use  $u \in \text{DU-chain}(d)$ 
        if (  $\exists$  definition  $d_1 \in \text{UD-chain}(u)$  such that  $d_1$  precedes  $d$  in basic block )
            NeedSelect  $\leftarrow$  true
            remove the predicate of  $d_1$ 
        if ( NeedSelect == true )
            rename  $V$  to  $r$  in  $d$  so that  $d : r = s_1 \text{ op } s_2$ 
            remove the predicate  $P$  of  $d$ 
            Insert " $d_{new} : V = \text{select}(V, r, P)$ " after  $d$ 
            Replace  $d$  and  $d_1$  with  $d_{new}$  in UD-chain and DU-chain.

```

Figure 3.5: An algorithm to generate `select` instructions.

For clarity, the reader can assume that the PHG discussed in this section contains only superword predicates. Our implementation actually has separate PHGs for superword and scalar predicates, with connections between the two graphs.

The algorithm relies on both the PHG and UD-chains [2], extended in Definition 6 to consider the effects of predication. Using the PHG and the notion of reaching definition (Definition 6), we build DU-chains for the superword definitions and UD-chains for the corresponding uses as shown in Algorithm SEL of Figure 3.5. Although the PHG involves both scalar and superword predicate variables, only superword variables are included in the DU-chains and UD-chains. To correctly handle *upward exposed uses*, all variables are assumed to be defined on entry of the basic block, and these definitions are included when appropriate in the DU-chains and UD-chains. In this way, the compiler can generate a `select` instruction when there is an upward exposed use.

The main loop of the algorithm SEL examines each instruction in textual order. An instruction with definition d needs a `select` instruction if d reaches at least one use u that is also reached by an earlier definition d_1 . If a definition d is the only definition reaching

| | | |
|--|--|--|
| | | <pre> if(p){ bred[i] = fred; bgreen[i] = fgreen; bblue[i] = fblue; } else{ bred[i] = 100; bgreen[i] = 100; bblue[i] = 100; } </pre> |
| <pre> bred[i] = fred; bred[i] = 100; bgreen[i] = fgreen; bgreen[i] = 100; bblue[i] = fblue; bblue[i] = 100; </pre> | <pre> (p) if(p == 1) bred[i] = fred; (¬p) if(p == 0) bred[i] = 100; (p) if(p == 1) bgreen[i] = fgreen; (¬p) if(p == 0) bgreen[i] = 100; (p) if(p == 1) bblue[i] = fblue; (¬p) if(p == 0) bblue[i] = 100; </pre> | |
| (a) Predicated scalar code | (b) Naive unpredicate applied | (c) Improved |

Figure 3.6: Restoring control flow.

all its reachable uses, it needs not be combined with anything. Figure 3.4(c) illustrates this point. The first `select` instruction is not necessary because no earlier definition reaches any of its uses.

Excluding store instructions, this algorithm generates the minimal number of `select` instructions. Given n definitions to be combined, this algorithm generates $n - 1$ select instructions. The minimality can be proven by reducing the definitions to leaf nodes of a full binary tree.

3.3 Unpredicate

After superword predicates are removed and replaced with `select` instructions, the code may still contain predicated scalar operations. The simplest way of removing scalar predicates is to convert each predicated instruction into an `if`-statement containing one statement, as in the example code in Figure 3.6(b). While correct, the code contains numerous redundant conditional branches, six in this case.

Figure 3.7 presents our algorithm that generates the control flow graph(CFG) representing the improved code as shown in Figure 3.6(c), given input instruction sequence IN. The main algorithm, called UNP, is shown in Figure 3.7(a). In addition to deriving the

Algorithm UNP: Given a sequence of predicated instructions IN, introduce control flow into the instruction sequence after removing predicates.

```

PHG ← Build a predicate hierarchy graph
DG ← Build a data dependence graph
CFG ← new basic block(P0) // root node
for each instruction I ∈ IN in textual order
  B ← {basic block b | ∀ basic block b' ∈ CFG
    (b' is reachable from b in CFG) ⇒
    (∃ an instruction I' ∈ b' such that I is dependent on I')}
  if (B == ∅)
    B ← NBB(CFG, PHG, I, IN)
  else
    Move I in IN to next to the last instruction of the
    earliest basic block in B
  Insert I to end of the earliest basic block b ∈ B
return CFG

```

(a) UNPredicate main

Algorithm NBB: Given an instructions I, predicate hierarchy graph PHG, the current control flow graph CFG, and predicated input code IN, generate a new basic block in CFG.

```

P ← predicate of I
b ← new basic block(P)
B ← PCB(P, PHG, CFG, IN, I)
for each b' ∈ B
  generate an edge from b' to b
return b

```

(b) Create a new basic block

Figure 3.7: Unpredicate algorithm.

Algorithm PCB: Given a predicate P , predicate hierarchy graph PHG , the current control flow graph CFG , predicated input code IN , and an instruction I , return a set of basic blocks that are predecessors of I .

```

RET ← ∅
PHG' ← PHG
I' ← I.previous
while I' ≠ NULL
    P' ← I'.predicate
    if (does_cover(P', P, PHG')) == TRUE
        RET ← RET ∪ I'.block
        PHG' ← mark(PHG', P')
    if (is_covered(PHG', P) == TRUE)
        return RET
    I' ← I'.previous

RET ← RET ∪ {ROOT}
return RET

```

(c) Predicate covering basic blocks

Figure 3.7: Unpredicate algorithm (Continued).

final control flow graph, UNP derives as an intermediate result a reordered instruction sequence IN .

UNP starts by building a predicate hierarchy graph, PHG . The superword predicates have been eliminated and replaced with `select` operations. However, both superword and scalar predicates have be considered to account for the scalar predicates unpacked from the superword predicates. UNP also constructs a data dependence graph for instruction sequence IN , capturing the ordering constraints on the instruction sequence.

Subsequently, UNP initializes the CFG with a root node associated with a constant *true* predicate P_0 . The main loop iterates through the input instruction sequence IN . First, we find a set of existing basic blocks where it is safe to insert the given instruction. An instruction I guarded by predicate P can be inserted in basic block B associated with predicate P' if $P = P'$ and there is no data dependence preventing insertion of I into

B . If the set is not empty, the instruction I is inserted at the end of the earliest such basic block B . Also, in the input instruction sequence IN , I is moved next to instruction I' that is the immediate prior instruction in B . Although we have already processed I , moving it in the instruction sequence will facilitate finding predicate covering basic blocks in Algorithm PCB for subsequent instructions in the stream. If the instruction cannot be inserted into any existing basic block, we create a new basic block B' and I is placed into B' .

When a new basic block B' is created by Algorithm NBB, the predicate covering basic block algorithm (PCB) is used to find a set of predecessors of B' . Whereas Mahlke's *predicate CFG generator* scans forward to find a set of successors, we scan the input instruction sequence backward to find predecessor instructions whose predicates cover the predicate of the given instruction. Since the instructions in the input instruction sequence are processed sequentially, all predecessor instructions chosen must have been inserted already. By keeping a pointer from the inserted instructions to the basic blocks, the predecessor basic blocks for the new basic block are identified. We create a copy PHG' of predicate hierarchy graph PHG so that we may mark covering predicates during the search for the appropriate basic blocks to connect to the new basic block in the intermediate CFG. The function `does_cover(P', P, PHG')` checks if P' covers P in PHG' . If P' is not marked yet in PHG' and P' is not mutually exclusive with P , the function returns *true*. The function `mark(PHG', P')` places a mark on a predicate node P' in PHG' as covered and checks if the predecessor nodes and the successor nodes of P' are also covered as a result of marking. If a node is newly marked, this process is recursively applied to the neighbors of the node. The function `is_covered(PHG', P)` examines PHG' and returns *true* if P is marked as covered.

3.4 Branch-On-Superword-Condition-Code (BOSCC)

Inserting BOSCC instructions is not always profitable. The benefits of BOSCC instructions depend on properties such as the *density* of true or false branches, the number of instructions within a branching construct and the data set size. In the remainder of this section, we describe the tradeoff space in selecting between the two approaches for SLP in the presence of control flow; in one approach, BOSCCs are not used whereas they are used in the other.

The next subsection shows results of a synthetic benchmark to illustrate this tradeoff space, followed by the compiler analysis and code generation techniques used to exploit BOSCC. We assume that parallelization has been performed and `select` instructions are inserted where control flow paths merge, and focus on using BOSCC to reduce the overheads introduced by parallelization of multiple control flow paths. The main components of the algorithm are: a profitability model for BOSCC instructions (Section 3.4.2); a profiling phase for collecting data for the BOSCC model (Section 3.4.3); identifying regions of code and predicates associated with a BOSCC instruction (Section 3.4.4); and code generation for inserting BOSCC instructions (Section 3.4.5).

3.4.1 The Characteristics of BOSCC

To gain insight into the factors influencing the profitability of BOSCC instructions, we performed a series of experiments using the following synthetic benchmark.

```
for(i=0; i<datasize; i++){
    temp = A[i];
    if (temp == B[i])
        C[i] = temp + D[i];
}
```

In this code, whenever the condition `(temp == B[i])` evaluates to false, the code following the conditional is bypassed. Thus, a BOSCC branch is most profitable when

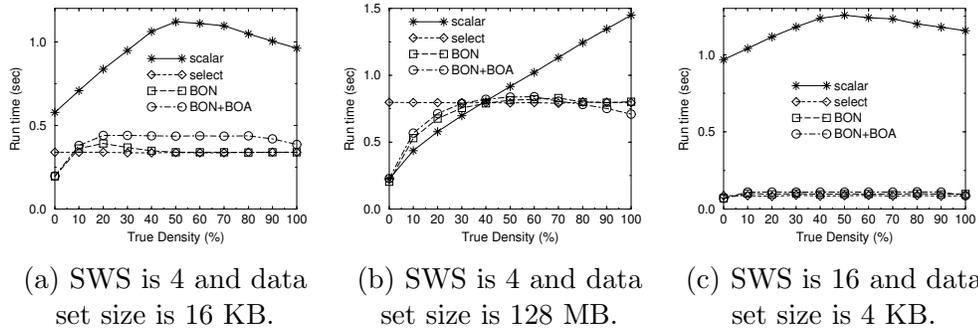


Figure 3.8: Run time of synthetic kernels.

the condition evaluates to false. Profitability therefore depends on the *true density* of the predicate, the frequency of true values for the branch test. We expect that low true densities should correspond to more benefit from BOSCC instructions.

We present the results of a set of experiments in the three graphs from Figure 3.8. In each graph, the horizontal axis corresponds to the true density of the input data set. We used a random number generator to create data sets with true densities from 0% to 100%.

Each graph shows the execution time of four versions of the code, as a function of true density. The *scalar* curve represents the execution time of the original scalar code. The other three versions were hand-coded in C extended with the Motorola-AltiVec programming model. The *select* version corresponds to what would be generated by the default approach in our compiler, as shown in Figure 3.2(e) and described in [60]. The *BON* version was derived by adding a *branch-on-none*(*BON*) instruction to the assembly code of the *select* version to bypass the code guarded by the conditional when the test on all fields evaluates to false, similar to the example in Figure 3.2(f). Finally, the *BON+BOA* version was derived by adding a *branch-on-all*(*BOA*) to the *BON* version. The branch-on-all permits an additional optimization which avoids the select operation; if all fields are known to evaluate to true, then the value of all fields of the corresponding superword of C are the result of the operation guarded by the conditional.

In Figure 3.8(a) and (b), the superword size (SWS) is four, that is, each superword can hold four integer array elements. Therefore the amount of available parallelism in a superword operation is four. Figures 3.8(a) and (b) show the run times of the benchmark for two data sizes: in (a) the data size fits in the L1 cache and in (b) the data size is larger than the L2 cache.

First, we consider the results of Figure 3.8(a). The *scalar* curve is consistently slower than the various parallel versions. It performs best when the true density is either very low or very high. This is because the G4’s branch prediction is most effective when the branching behavior is consistent. In the `select` version, the branch is eliminated and replaced with a merge of fields across the different control flow paths. For this reason, the execution time is the same regardless of the true density. It has the best performance among the four versions for true densities at or above 20%. The performance of the *BON* version is best for true densities near 0% and is the same as the `select` version for true densities above 40%. Interestingly, we see that the slowest performance is at a true density of 16%, also related to branch prediction accuracy. It is lower than 50% because the branch-on-none is taken only when the conditions for all four consecutive scalar comparisons are false. For a superword size of four and true density of D , the probability for all four conditions to be false is $(1 - D)^4$. When two BOSCC instructions are used for the *BON+BOA* version, the overhead of an additional branch overcomes any benefit.

The results of Figure 3.8(b) show how the tradeoff space is affected when the data footprint exceeds the L2 cache size. As the computation becomes memory bound, the benefits of parallelization become less significant. Thus, the performance gap between the *scalar* and parallel versions is reduced. For true densities below 40%, the *scalar* version is actually the fastest. The *BON* version behaves similarly to the *scalar* version for low true densities, while it behaves similarly to the `select` version for higher true densities. The *BON+BOA* version has the best performance for very high true densities.

To evaluate the effects of increasing the amount of available parallelism, Figure 3.8(c) shows the impact of modifying the data type to `char`, thus increasing the superword size to 16. This change increases the performance gap between the scalar version and the other parallel versions for all values of true densities. The various parallel versions exhibit very similar behavior.

From the experiments shown in Figure 3.8, we can summarize the following conclusions. The BOSCC versions incur an overhead due to the addition of branches as compared to the `select` version, and sometimes this overhead makes them unprofitable. For this reason, we have decided in our compiler to use just one BOSCC instruction, comparable to the *BON* version. We have also determined that low true density can be used as one predictor of profitability. In addition, the profitability of the *BON* version over the `select` version increases as the cost of the instructions in the branch body increases. Also, as parallelism increases, the profitable true density range of the *BON* version actually decreases. While not shown in these experiments, a related profitability criteria is how many instructions appear in the code bypassed by the branch; more instructions lead to greater benefit. Finally, the cost of memory access instructions can dwarf the benefits of parallelizing the computation, but the *BON* version performs comparably to the best version for all true densities. In general, while not always the best performing version, the *BON* version has behavior that is comparable to the best version for all of the experiments, whereas both the *scalar* and `select` versions sometimes are much slower than the others. Based on the insights presented in this section, we build a model which can be used to guide the generation of BOSCC instructions only when profitable.

3.4.2 BOSCC Model

The BOSCC model determines the profitability of using a BOSCC instruction to bypass code, allowing the compiler to decide whether or not to generate a BOSCC instruction. The model uses two key properties of the code to determine profitability. The first, *PAFS*

(*percentage of all false superwords*), is the percentage of superword predicates where all fields are false, and indicates how frequently a BOSCC branch is taken. Determining the *PAFS* value associated with a particular superword predicate must be done dynamically, and is computed in a separate profiling phase as discussed in Section 3.4.3. The second, *NBI* (*number of bypassed instructions*), is the number of instructions bypassed when a BOSCC branch is taken, which represents the number of instructions for a single execution of the parallelized code. The *NBI* can be computed statically by the compiler.

The number of instructions of the `select` and BOSCC versions are computed by Equation 3.1 and 3.2 respectively, and a BOSCC instruction is profitable whenever $NI(\text{select}) > NI(\text{BOSCC})$.

$$NI(\text{select}) = NBI \tag{3.1}$$

$$NI(\text{BOSCC}) = NBI + 1 - PAFS \times NBI \tag{3.2}$$

In Equation 3.2, we add an additional instruction for the BOSCC branch, and subtract the number of instructions skipped by the BOSCC branch ($PAFS \times NBI$). In reality, the cost of executing a BOSCC instruction may be higher or lower than that of other instructions depending on how the branch predictor performs. The additional weight of executing BOSCC instructions can be varied to improve the precision of the model, but since it is machine-specific, we omit it here.

Note that this model takes into account the effects discussed in the previous section of the data type size and associated parallelism, as well as the amount of computation bypassed by the BOSCC instruction. However, it ignores locality effects, which must be addressed separately.

To provide intuition as to why parallelization using BOSCC is more profitable than scalar execution of the equivalent code, let us assume that a scalar instruction is mapped to a single equivalent superword instruction and that the run time is computed as the

Algorithm INSTRUMENT: Given a basic block B

```
P ← find superword predicates(B)
if (P == ∅) return
Insert a basic block counter to B
for each superword predicate pred ∈ P
    Insert a counter for pred
```

(a) Algorithm

```
vec = vec_ld(i_0, ptr);
*(_basicblock + 0) = *(_basicblock + 0) + 1;
vec118 = vec_ld(i_0, ptr133);
vec119 = vec_ld(i_0, ptr134);
vec121 = vec_cmpeq(vec, vec120);
vec123 = vec_cmpeq(vec118, vec120);
vec125 = vec_cmpeq(vec119, vec124);
vec126 = vec_and(vec121, vec123);
vec127 = vec_and(vec126, vec125);
vec129 = vec_cmpeq((vector unsigned char)vec127, vec120);
vec130 = vec129;
sel = vec_ld(i_0, ptr135);
vec138 = (vector bool char)vec_splat_u8(0);
instrument = vec_all_eq(vec130, vec138);
if (instrument == 1)
{
*(_superword_predicates + 0) = *(_superword_predicates + 0) + 1;
}
sel = vec_sel(sel, vec, vec130);
```

(b) Example

Figure 3.9: Automatic instrumentation to compute PAFS in profiling phase.

number of executed instructions. In this specific situation, we can have a parallelized code using a BOSCC where each instruction is the superword counterpart of the scalar instruction in the original. The BOSCC can be thought of as the counterpart of the original scalar branch. If the branch body is executed in the scalar version more than once out of SWS iterations, the branch body in the BOSCC version will be executed exactly once for SWS scalar iterations. In this case, the version using BOSCC will run faster than the scalar version because of less loop overhead. If the branch body is not executed in the scalar version for SWS iterations, the branch body in the BOSCC version also will not be executed and will run faster because of less loop overhead.

3.4.3 Profiling Support to Compute PAFS

The *PAFS* value in the previous model is determined using automatic instrumentation in a separate profiling phase ¹. Figure 3.9(a) shows the simple algorithm for inserting instrumentation code. First, for each basic block, all superword predicates are identified. Next, for each basic block that contains superword `select` instructions, we measure the total number of times the block is executed and, for each predicate, the number of BOSCC's taken. To increment the counter only when the superword predicate contains false values in all the fields, we also use a BOSCC instruction. Use of BOSCC expedites the profile run as compared to checking the individual fields in a sequential loop. An example of instrumented code is shown in Figure 3.9(b). The instructions in bold font are added for profiling.

¹While profiling has limitations in deriving dynamic information, particularly when a different input data set is used than was used in the profiling stage, we forgo more elaborate approaches for deriving dynamic information on-the-fly, since issues of deriving dynamic information are orthogonal to the focus of this work. Other approaches could also be used to derive the value of PAFS.

Algorithm ISP(B): Given a basic block B

```
// Initially, all instructions are associated with constant true predicates
for each select instruction I: “dst = select(src1, src2, pred)” ∈ B where dst == src1
// src1 is associated with 'true' value of pred
// src2 is associated with 'false' value of pred
    predicate(I) ← pred;
    IdentifyBranchBody(src2, I, pred);
    IdentifyMemoryAccesses(src1, dst, pred);
```

(a) Identifying superword predicates

Algorithm IdentifyBranchBody(src, I, pred): Given an operand src, an instruction I and a predicate pred

```
rd ← reaching definitions of src;
if (rd is not a single reaching definition ∨ I is not the only use of rd)
    return;
predicate(rd) ← pred;
for each source operand src of rd
    IdentifyBranchBody(src, rd, pred);
```

(b) Identifying branch body

Algorithm IdentifyMemoryAccesses(src, dst, pred):

```
rd ← reaching definitions of src
u ← uses of dst
if (rd is single reaching definition ∧ rd is a load ∧
    u is the only use ∧ u is a store ∧ rd and u access the same address)
    predicate(rd) ← pred
    predicate(u) ← pred
```

(c) Identifying unnecessary memory accesses

Figure 3.10: Algorithm to identify a predicate for instructions.

3.4.4 Identifying BOSCC Predicates

Prior to code generation, the compiler locates predicates associated with select instructions and identifies the set of instructions guarded by each predicate. The third operand of each superword `select` instruction, as shown in Figure 3.3, represents a predicate.

The algorithm to extend these predicates to other instructions is shown in Figure 3.10. Initially, a constant *true* predicate is associated with all instructions. The algorithm in Figure 3.10(a) scans the code to locate `select` instructions. For each `select` instruction whose first source operand and the destination operand are the same, it associates the predicate found in the third source operand with the select instruction, and then follows UD and DU-chains to locate other instructions to which this predicate can be associated. Two sets of instructions are considered, as shown in Figures 3.10(b) and (c).

The goal of the algorithm in Figure 3.10(b) is to identify the set of instructions that are executed only when the predicate evaluates to true. The result of a superword `select` instruction is the first operand (`src1`) when the predicate `pred` contains all false values. We can therefore bypass any instructions that define the value of the second operand `src2` if all the fields of `pred` are false. This set of instructions can be thought of as the branch body from the original program, although it could include an even larger set of instructions. The algorithm `IdentifyBranchBody` then recursively follows the definitions of the variables contributing to the value of `src2`. Those that have a single definition reaching a single use can be guarded by the predicate `pred`, and can be bypassed by the BOSCC instruction. The goal of the algorithm in Figure 3.10(c) is to eliminate unnecessary memory accesses occurring when all fields of `pred` evaluate to false. If a load to `src1` and a store of `dst` occur in the code, the value is not modified between the load and store, and no other instructions depend on this load and store, both memory accesses can be predicated with `pred`. The algorithm in Figure 3.10 guarantees that at most one predicate is associated with each superword instruction.

Algorithm FBR(B): Given a basic block B

```

n ← 0
Region[0] ← new region(NULL)
current ← prev ← NULL
for each instruction I ∈ B
  pred ← predicate(I)
  if (current ≠ pred)
    Region[n].end ← prev
    n++
    Region[n] ← new region(pred)
    Region[n].moved ← false
    Region[n].begin ← I
    current ← pred
  prev ← I
Region[n++].end ← I

for (i=1; i<n; i++)
  for (j=i+1; j<n; j++)
    if (Region[i].predicate ≠ NULL ∧ Region[i].moved == false ∧
        Region[i].predicate == Region[j].predicate)
      if (Region[j] can be moved after Region[i].end)
        move instructions in Region[j] after Region[i].end
        Region[j].moved ← true
      else if (Region[i] can be moved before Region[j].begin)
        move instructions in Region[i] before Region[j].begin
        Region[i].moved ← true
return Region, n

```

(a) Form BOSCC regions

Algorithm Insert-BOSCC(B): Given a basic block B

```

B' ← ISP(B)
R, n ← FBR(B')
for (i=1; i<n; i++)
  if (R[i].moved == false ∧ R[i].predicate ≠ NULL)
    NLselect ← # instructions(R[i])
    NLboscc ← NLselect + 1 - PAFS(R[i]) × NLselect
    if (NLboscc < NLselect)
      Insert boscc(R[i])

```

(b) BOSCC insertion algorithm main

Figure 3.11: BOSCC insertion algorithm.

3.4.5 Inserting BOSCC Instructions

Figure 3.11(b) shows the main algorithm to insert BOSCC instructions. After the predicate for each instruction is identified, instructions with the same predicate are combined into a BOSCC region if there are no intervening dependences. In the algorithm shown in Figure 3.11(a), the initial BOSCC regions are formed by finding consecutive instructions guarded by the same predicate. Then the BOSCC regions associated with the same non-constant predicate are merged if no data dependences with the intervening instructions prevent the code motion. The algorithm first checks if the later region can be moved to the end of the earlier region. If this is not possible because of the data dependences with the intervening instructions, the algorithm checks if the earlier region can be moved before the first instruction of the later region. The goal is to form the largest possible region guarded by a single BOSCC predicate. The number of adjacent instructions guarded by the same predicate provides the value of NBI for the BOSCC model, while the value of $PAFS$ is derived from profiling. If profitable, a BOSCC instruction is inserted just prior to the instructions that form a BOSCC region, and it branches to the instruction immediately following the last instruction of the BOSCC region.

Chapter 4

SUPERWORD-LEVEL LOCALITY

While the most important optimization opportunity for the architectures supporting SLP is to exploit parallelism in the SIMD functional unit, another as important optimization is to exploit memory hierarchy to reduce memory access time. Since parallelization is not as effective when bottleneck is memory accesses, the optimizations targeting memory hierarchy are even more important for the architectures supporting SLP.

A key idea is to notice that a superword register file offers a much larger space than a scalar register file to store frequently used data items. We treat the superword register file as a small compiler-controlled cache. Our approach is distinguished from previous work on increasing reuse in cache [17, 23, 26, 28, 29, 38, 66, 69], in that the compiler must also manage replacement, and thus, explicitly name the registers in the code. As compared to previous work on exploiting reuse in scalar registers [69, 10, 45], the compiler considers not just temporal reuse, but also spatial reuse, for both individual statements and groups of references. Exploiting spatial and group reuse in superword registers requires more complex analysis as compared to exploiting temporal reuse in scalar registers, to determine which accesses map into the same superword.

We develop an algorithm and a set of optimizations to exploit reuse of data in superword registers to eliminate unnecessary memory accesses, which we call *superword-level locality* (SLL). In conjunction with exploiting SLP, the algorithm performs what we call

superword replacement, to replace accesses to contiguous array data with superword temporaries and exploit reuse by replacing accesses to the same superword with the same temporary. Following this code transformation, a separate compilation pass will be able to allocate superword registers corresponding to the superword temporaries. To enhance the effectiveness of superword replacement, it is combined with a loop transformation called *unroll-and-jam*, whereby outer loops in a loop nest are unrolled, and the resulting duplicate inner loop bodies are fused together. Unroll-and-jam reduces the distance between the reuse of the same superword, when reuse is carried by an outer loop, and brings opportunities for superword replacement into the innermost loop body of the transformed loop nest. The optimization algorithm derives appropriate unroll factors for each loop in the nest that attempt to maximize reuse while not exceeding the number of available registers.

The remainder of this chapter is organized into 5 sections. Section 4.1 motivates the problem and introduces terminology used in the remainder of the chapter. Section 4.2 presents an overview of the superword-level locality algorithm. Section 4.3 describes how the algorithm computes the total number of registers required for exploiting reuse and the resulting number of memory accesses. Section 4.4 describes aspects of how the search space is navigated. Section 4.5 presents optimizations to actually achieve this reuse of data in superword registers.

4.1 Background and Motivation

In many cases superword-level parallelism and superword-level locality are complementary optimization goals, since achieving SLP requires each operand to be a set of words packed into a superword, which happens, with no extra cost, when an array reference with spatial reuse is loaded from memory into a superword register. Therefore, in many cases the loop that carries the most superword-level parallelism also carries the most spatial

reuse, and benefits from SLL optimizations. In this chapter, we achieve SLL and SLP somewhat independently, by integrating a set of SLL optimizations into an existing SLP compiler [39]. The remainder of this section motivates the SLL optimizations.

Achieving locality in superword registers differs from locality optimization for scalar registers. To exploit temporal reuse of data in scalar registers, compilers use *scalar replacement* to replace array references by accesses to temporary scalar variables, so that a separate backend register allocator will exploit reuse in registers [10]. In addition, *unroll-and-jam* is used to shorten the distances between reuse of the same array location by unrolling outer loops that carry reuse and fusing the resulting inner loops together [10].

In contrast, a compiler can optimize for superword-level locality in superword registers through a combination of unroll-and-jam and *superword replacement*. These techniques not only exploit temporal reuse of data, but also spatial reuse of nearby elements in the same superword. In fact, even partial reuse of superwords can be exploited by merging the contents of two registers containing superwords that are consecutive in memory (see Section 4.5.4). Thus, as is common in multimedia applications [57], streaming computations with little or no temporal reuse can still benefit from spatial locality at the superword-register level, in addition to the cache level.

While cache optimizations are beyond the scope of this thesis, we observe that the SLL optimizations presented here can be applied to code that has been optimized for caches using well-known optimizations such as unimodular transformations, loop tiling and data prefetching. When combining loop tiling for caches, superword-level parallelism and superword-level locality optimizations, the tile sizes should be large enough for superword-level parallelism, and for unroll-and-jam and superword replacement to be profitable.

These points are illustrated by way of a code example, with the original code shown in Figure 4.1(a). This example shows three optimization paths. Figure 4.1(d) optimizes the code to achieve superword level parallelism. In Figures 4.1(b) and (c), we show how the original program can instead be optimized to exploit reuse in scalar registers, using

```

for(i=0; i<n; i++)
  for (j=0; j<n; j++)
    a[i][j] = a[i-1][j] * b[i] + b[i+1];

```

(a) Original loop nest.

```

for(i=0; i<n; i++)
  for (j=0; j<n; j+=SWS)
    a[i][j:j+SWS-1] = a[i-1][j:j+SWS-1] \
      * b[i] + b[i+1];

```

(d) Superword-level parallelization (j-loop).

```

for(i=0; i<n; i+=2)
  for (j=0; j<n; j++) {
    a[i][j] = a[i-1][j] \
      * b[i] + b[i+1];
    a[i+1][j] = a[i][j] \
      * b[i+1] + b[i+2];
  }

```

(b) Unroll-and-jam on (a) (i-loop).

```

for(i=0; i<n; i+=2)
  for (j=0; j<n; j+= SWS) {
    a[i][j:j+SWS-1] = a[i-1][j:j+SWS-1] \
      * b[i] + b[i+1];
    a[i+1][j:j+SWS-1] = a[i][j:j+SWS-1] \
      * b[i+1] + b[i+2];
  }

```

(e) Unroll-and-jam on (d) (i-loop).

```

tmp1 = b[0];
for(i=0; i<n; i+=2) {
  tmp2 = b[i+1];
  tmp3 = b[i+2];
  for (j=0; j<n; j++) {
    tmp4 = a[i-1][j] \
      * tmp1 + tmp2;
    a[i+1][j] = tmp4 \
      * tmp2 + tmp3;
    a[i][j] = tmp4;
  }
  tmp1 = tmp3;
}

```

(c) Scalar replacement on (b).

```

tmp1[0:SWS-1] = b[0:SWS-1];
stmp1 = tmp1[0];
stmp2 = tmp1[1];
field = 2;
for(i=0; i<n; i+=2) {
  // 'field' denotes an index into 'tmp1'
  // for stmp3
  if(field == 0)
    tmp1[0:SWS-1] = b[i+2:i+SWS+1];
  stmp3 = tmp1[field];
  for (j=0; j<n; j+= SWS) {
    tmp2[0:SWS-1] = a[i-1][j:j+SWS-1] \
      * stmp1 + stmp2;
    a[i+1][j:j+SWS-1] = tmp2[0:SWS-1] \
      * stmp2 + stmp3;
    a[i][j:j+SWS-1] = tmp2[0:SWS-1];
  }
  stmp1 = stmp3;
  stmp2 = tmp1[field+1];
  field = (field+2)%SWS;
}

```

(f) Superword replacement on (e)

Figure 4.1: Example code for SLL.

| | Original Figure 4.1(a) | Scalar register reuse Figure 4.1(c) | SLP only Figure 4.1(d) | SLP and SLL Figure 4.1(f) |
|--------|---------------------------|--|---------------------------|------------------------------|
| Reads | $3n^2$ | $n^2/2 + n$ | $2n^2 + n^2/SWS$ | $(n^2/2 + n)/SWS$ |
| Writes | n^2 | n^2 | n^2/SWS | n^2/SWS |

Table 4.1: Number of array accesses under different optimization paths.

unroll-and-jam and scalar replacement, respectively. In Figures 4.1(e) and (f), we combine these ideas, using unroll-and-jam and superword replacement, respectively, to transform the code in (d) for both superword-level parallelism and superword-level locality.

Table 4.1 shows how the three different optimization paths affect the number of array accesses to memory in the final code. The original code has n^2 reads and writes to array a and $2n^2$ reads to array b . Exploiting superword-level parallelism in loop j , as in Figure 4.1(d) reduces the number of reads and writes to array a by a factor of SWS since each load or store operates on SWS contiguous data items; for array b , there is no change since the array is indexed by i rather than j . If instead the code was optimized for scalar register reuse, as in Figure 4.1(c), we can reduce the number of array reads of a down by a factor of 2, and reads of b by a factor of n , with the number of writes remaining the same. By combining superword-level parallelism and superword-level locality as in Figure 4.1(f), we see that the number of reads and writes is further reduced by a factor of SWS . Figure 4.1(f) illustrates some of the challenges in exploiting reuse in superwords. Analysis must identify not just temporal, but also spatial reuse, and for both individual statements and groups of references. The compiler also must generate the appropriate code to exploit this reuse; for example, we select scalar fields of b from the superword, since we are not parallelizing the i loop. The remainder of this chapter describes how the compiler automatically generates code such as is shown in Figure 4.1(f).

4.2 Overview of Superword-Level Locality Algorithm

The superword-level locality algorithm has three main steps, as summarized below. Each step will be described in more detail in the three subsequent sections.

Step 1: Identifying Reuse. The first step of the algorithm is to identify both array references and loops carrying reuse. The array references carrying reuse are the ones for which superword replacement may be applicable. The loops carrying reuse are the ones to which the algorithm will consider applying unroll-and-jam.

Section 2.2 gives a detailed description of data reuse. For the purposes of this algorithm, the relevant dependences carrying reuse are a subset, and are characterized as follows:

1. We consider only true dependences, input dependences, and output dependences.
2. We consider only lexicographically positive dependences.
3. A dependence vector must be consistent, or it must be invariant with respect to one of the loops in the nest.

Applying unroll-and-jam to a loop i with a consistent dependence varying with respect to loop i can create loop-independent dependences in the innermost loop of the unrolled loop body. In the example in Figure 4.1(a), there is a true dependence between references $A[i][j]$ and $A[i-1][j]$ with distance vector $\langle 1, 0 \rangle$. After unroll-and-jam, a loop-independent dependence is created between $A[i][j]$ in the first statement and $A[i][j]$ in the second statement of the loop body, creating a reuse opportunity.

In addition to reuse between copies of a reference created by unrolling, there can be reuse across loop iterations. References with consistent dependences carried by a loop have group reuse which can be exploited by using extra registers to hold the data across iterations. As in previous work [10], our algorithm exploits reuse across iterations of the

| | |
|--|---|
| <pre> for(i=0; i<N; i+=4){ vec1[0:3] = A[i:i+3]; vec2[0:3] = A[i+8:i+11]; ⋮ } </pre> | <pre> tmp[0:3] = A[i:i+3]; vec2[0:3] = A[i+4:i+7]; for(i=0; i<N; i+=4){ vec1[0:3] = tmp[0:3]; tmp[0:3] = vec2[0:3]; vec2[0:3] = A[i+8:i+11]; ⋮ } </pre> |
| (a) Original | (b) After exploiting reuse |

Figure 4.2: Reuse across iterations.

innermost loop only, because exploiting reuse carried by an outer loop could potentially require too many registers to hold the data between uses. Figure 4.2 shows how reuse can be exploited across iterations of the innermost loop by using one register to keep the data that is reused on every two iterations.

For loop-invariant references, unroll-and-jam generates loop-independent dependences between the copies of the reference in the unrolled loop body, since the same location is being referenced by each copy.

Step 2: Determining unroll factors for candidate loops. The algorithm next determines the unroll factors for each candidate loop that carries reuse, as previously described, and for which unroll-and-jam is legal. The optimization goal is as follows.

Optimization Goal: Find unroll factors $\langle X_1, X_2, \dots, X_n \rangle$ for loops 1 to n in an n -deep loop nest such that the number of memory accesses is minimized, subject to the constraint that the number of superword registers required does not exceed what is available.

The algorithm determines the unroll factors $\langle X_1, X_2, \dots, X_n \rangle$ by searching for the combination of unroll factors that satisfies the above optimization goal. To guide the search, the algorithm calculates the total number of registers required for exploiting reuse, which

is the sum of the number of superwords accessed by the references in the loop body after unroll-and-jam is applied, plus the number of registers needed for holding data across iterations of the innermost loop. Section 4.3 describes how the algorithm computes the total number of registers required for exploiting reuse and the resulting number of memory accesses. Section 4.4 describes aspects of how the search space is navigated.

Step 3: Code Transformations - Unroll-and-Jam, Superword Replacement, and Related Optimizations. Once the unroll factors are decided, unroll-and-jam is applied to the loop nest. Array references are replaced with accesses to superword temporaries. As part of code generation, our compiler performs related optimizations to reduce the number of additional memory accesses and register requirements introduced by the SLP passes. These code transformations are the topic of Section 4.5.

4.3 Modeling Register Requirements & Number of Memory Accesses

This section presents the computation of the number of registers required for exploiting data reuse in superword registers and the resulting number of memory accesses, which are the parameters used to guide the search for the combination of unroll amounts to be applied to the loop nest. The next subsection describes how the algorithm computes the *superword footprint*, which represents the number of superwords accessed by the unrolled iterations of the loop nest as a function of the unroll factors. Subsection 4.3.2 presents the computation of the extra registers needed for reusing data across loop iterations. The total number of registers and the corresponding number of memory accesses are computed in subsection 4.3.3.

4.3.1 Computing the Superword Footprint

This section presents the computation of the superword footprint of the references V in a loop nest, $F_L(V)$, after unroll-and-jam is applied to the nest with unroll factors $\langle X_1, X_2, \dots, X_n \rangle$.

The algorithm for computing the superword footprint for a loop nest first partitions the references in the loop into groups of *uniformly generated references* [69](See Section 2.2)¹. Then, for each group of references, it computes the number of superwords accessed in the unrolled loop body. Finally, the total number of superwords is computed as the sum of those of each group of uniformly generated references.

We first discuss how to compute the superword footprint of a single reference as a function of the unroll factors of each unrolled loop. Then we discuss how to compute the superword footprint of a group of uniformly generated references. The superword footprint of a group may be smaller than the sum of the individual footprints, since the same superword may be accessed by two or more copies of the original references when the loops are unrolled.

Our method determines the number of superword registers required to hold the data accessed by the references in the unrolled loop body. However, extra registers may be needed to, for example, align a superword operand which is already kept in superword registers. That is, the computation may require more registers than those needed for storing the data. Therefore, we reserve some scratch registers for manipulating data and compute the number of registers needed just for storing the data accessed in the unrolled loops.

To simplify the presentation, we assume a loop nest of depth n where all array references have array subscripts that are affine functions of a single index variable (SIV

¹We assume that two or more references that access the same array, but are not uniformly generated, access distinct data in memory, which results in a conservative estimate of the number of superwords accessed by the group and of the number of registers required.

subscripts)². We also assume that each p -dimensional array referenced by the loop is defined as $A[s_p][s_{p-1}] \dots [s_1]$, where s_h is the size of dimension h , $1 \leq h \leq p$. Dimension 1 is the lowest dimension of the array, *i.e.*, the dimension in which consecutive elements are in consecutive memory locations. A reference v to array A is then of the form $A[a_p * l_p + b_p][a_{p-1} * l_{p-1} + b_{p-1}] \dots [a_1 * l_1 + b_1]$. Thus, a reference with SIV subscripts has each array dimension h associated with just a single loop index variable in the nest, and the loop index variable associated with h is represented as l_h . We also assume that the arrays are aligned to a superword in memory and that the loops are normalized.

4.3.1.1 Superword Footprint of a Single Reference

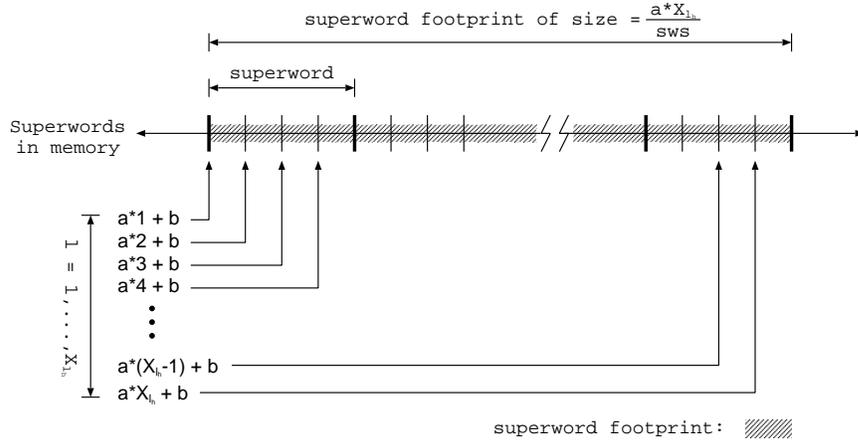
For each reference v with array subscripts $a_h * l_h + b$, where h is the array dimension and l_h is the loop index variable appearing in subscript h , the number of superwords accessed by all copies of v when l_h is unrolled by X_{l_h} is given by the *superword footprint* of v in l_h , or $F_{l_h}(v)$.

When dimension h is the lowest array dimension ($h = 1$), the superword footprint is given by Equation (4.1). Equation (4.1a) corresponds to the footprint of a loop-invariant reference. Equation (4.1b) corresponds to the footprint of a reference with self-spatial reuse within a superword, as illustrated in Figure 4.3(a), and (4.1c) holds when the reference has no spatial reuse.

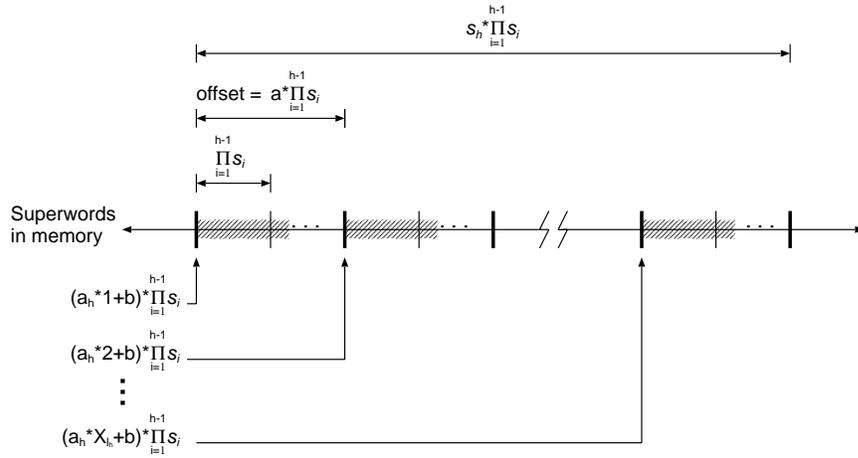
$$F_{l_h}(v) = \begin{cases} 1 & \text{(a) if } a_h = 0 \\ \left\lceil \frac{X_{l_h} * a_h}{SWS} \right\rceil & \text{(b) if } a_h < SWS \\ X_{l_h} & \text{(c) if } a_h \geq SWS \end{cases} \quad (4.1)$$

When h is one of the higher dimensions, $1 < h \leq p$, and loop l_h is unrolled, the offset between the footprints of each copy of v is $a_h * \prod_{i=1}^{h-1} s_i$, where s_i is the size of the i^{th}

²Our current implementation can handle affine SIV subscripts and certain affine MIV subscripts.



(a) $h = 1$ and $a_h < SWS$



(b) $h \neq 1$

Figure 4.3: Superword footprint of a single reference.

array dimension, as shown in Figure 4.3(b). Assuming that the size of the lowest array dimension (s_1) is larger than SWS , which is usually the case in practice for realistic array dimensions, each copy of v in the unrolled loop body corresponds to a separate footprint, as shown in Figure 4.3(b). Therefore the size of the footprint of v in l_h is the sum of the X_{l_h} disjoint footprints, and is recursively defined by Equation (4.2), where $F_{l_1}(v)$ is computed as in Equation (4.1).

$$\begin{aligned}
F_{l_h}(v) &= X_{l_h} * F_{l_{h-1}}(v) \\
&= \left(\prod_{i=2}^h X_{l_i} \right) * F_{l_1}(v)
\end{aligned} \tag{4.2}$$

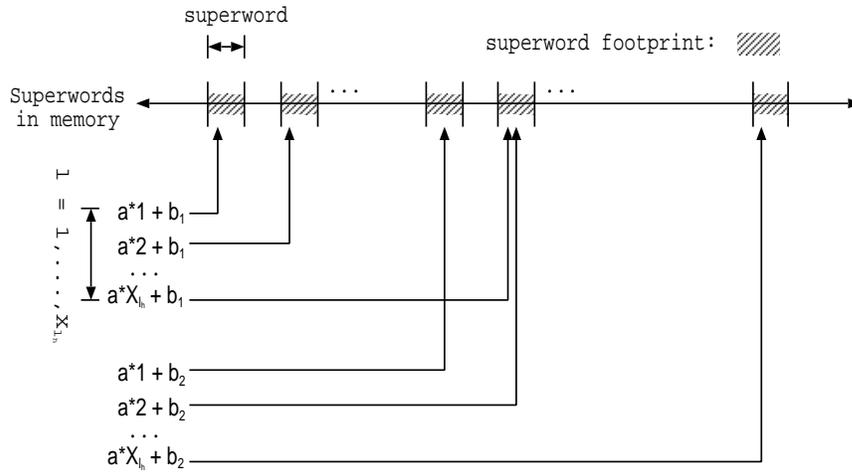
For a single reference, the number of superword registers required to keep the superword footprint given by Equation (4.1) and the number of scalar registers that would be required if the same unroll factors were used differ only when $a_h < SWS$, that is, when spatial reuse can be exploited in superword registers. For a group of uniformly generated references the analysis must also consider group reuse, as discussed next.

4.3.1.2 Superword Footprint of a Group of References

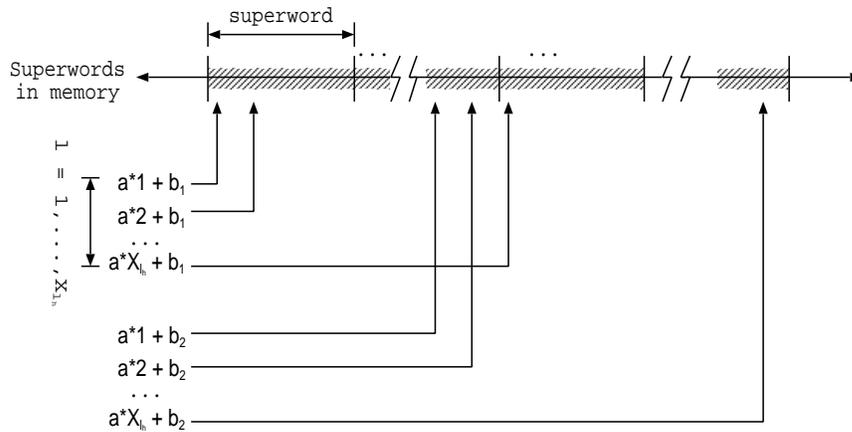
The number of superwords accessed by a group of uniformly generated references $V = \{v_1, v_2, \dots, v_m\}$ when loop l_h is unrolled by X_{l_h} is the superword footprint of the group, $F_{l_h}(V)$. The superword footprint of a group consists of the union of the footprints of the individual references, as some of the reference footprints may overlap, depending on the distance between the constant terms in the array subscripts.

The footprints of two uniformly generated references may overlap in dimension h only if they overlap in all dimensions higher than h . For example, the footprints of references $A[2i][j+2]$ and $[2i+1][j]$ do not overlap in the highest (row) dimension, since the first reference accesses the even-numbered rows of the array and the second accesses the odd-numbered rows. Therefore the footprints cannot overlap in the lowest (column) dimension. On the other hand, the footprints of $A[2i][j+2]$ and $A[2i+4][j]$ overlap in the row dimension for iterations $i_1, i_2, 1 \leq i_1, i_2 \leq X_i$, such that $2i_1 = 2i_2 + 4$. For the iterations of i in which the footprints overlap in the row dimension, the footprints may overlap in the column dimension if there exist iterations $j_1, j_2, 1 \leq j_1, j_2 \leq X_j$, such that $j_1 + 2 = j_2$.

$$F_{l_h}(v_1, v_2) = \begin{cases} X_{l_h} + (b_2 - b_1)/a_h & \text{(a) if } a_h \geq SWS \text{ and } (b_2 - b_1) < a_h * X_{l_h} \\ & \text{and } (b_2 - b_1) \bmod a_h = 0 \\ \lceil (a_h * X_{l_h} + b_2 - b_1) / SWS \rceil & \text{(b) if } a_h < SWS \text{ and } (b_2 - b_1) < a_h * X_{l_h} \\ F_{l_h}(v_1) + F_{l_h}(v_2) & \text{(c) otherwise} \end{cases} \quad (4.3)$$



(a) $a_h \geq sws$ and $(b_2 - b_1) < a_h * X_{l_h}$
and $(b_2 - b_1) \bmod a_h = 0$



(b) $a_h < sws$ and $(b_2 - b_1) < a_h * X_{l_h}$

Figure 4.4: Superword footprint of a group of references.

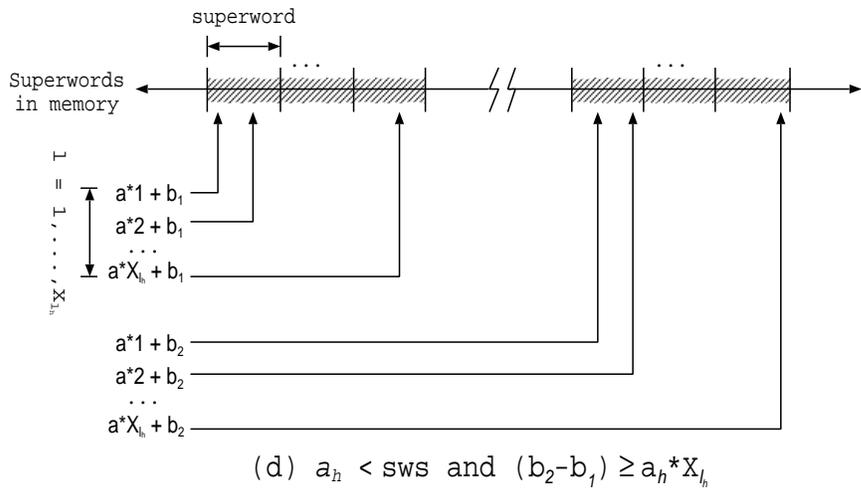
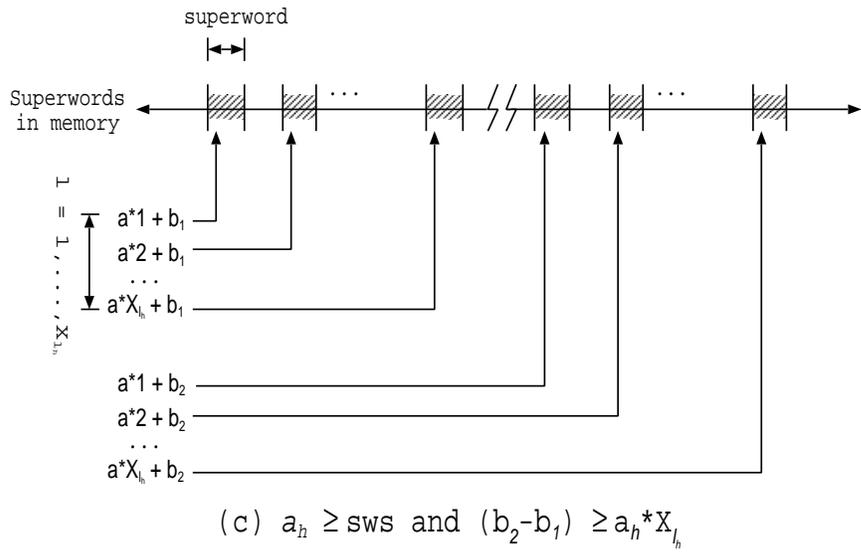


Figure 4.4: Superword footprint of a group of references (Continued).

The superword footprint $F_L(V)$ of a group V , following unroll-and-jam, is computed as follows. First, the array dimensions with array subscripts that are a function of any of the unrolled loops are identified. Then, for each such dimension h , from highest to lowest dimension, the footprint is computed assuming that the footprints of the references in the group overlap in the higher dimensions. For each dimension $h > 1$, the algorithm partitions references into subsets such that each subset corresponds to a disjoint footprint in dimension h . Then, for each subset, the algorithm recursively computes the footprint in dimension $h - 1$, as we now describe.

Dimension h is the lowest dimension ($h = 1$). We first compute the group footprint of two array references, and then we extend it for m references. The footprint of group $V = \{v_1, v_2\}$, where references v_1 and v_2 have lowest dimension subscripts $a_h * l_h + b_1$ and $a_h * l_h + b_2$ such that $b_1 \leq b_2$, when loop l_h is unrolled by X_{l_h} is given by Equation (4.3) in Figure 4.4. Equations (4.3a) and (4.3b) apply when the two footprints overlap, that is, when $(b_2 - b_1) < a_h * X_{l_h}$, as shown in Figures 4.4(a) and (b). When the footprints do not overlap, the group footprint is the sum of the individual footprints, as in Equation (4.3c), with examples in Figures 4.4(c) and (d).

In Figure 4.4(a), the references have no self-spatial reuse, that is, $a_h \geq SWS$, and each individual footprint is a set of X_{l_h} superwords. The footprints overlap if $(b_2 - b_1)$ is evenly divided by a_h and there exists an integer value k , $1 \leq k \leq X_{l_h}$, such that $k = 1 + (b_2 - b_1) / a_h$. This case corresponds to Equation (4.3a), which computes the group footprint precisely when the two references have group-temporal reuse. In Figure 4.4(b), both references have self-spatial reuse within a superword, that is, $a_h < SWS$. The corresponding footprint size is given by Equation (4.3b). In Figure 4.4(c), v_1 has no self-spatial reuse and each copy of v_1 in the unrolled loop body accesses a distinct superword, and the same is true for v_2 . In Figure 4.4(d) both v_1 and v_2 have self-spatial reuse.

The footprint of a group $V = \{v_1, v_2, \dots, v_m\}$, with array subscripts $a_1 * l_1 + b_i$ such that $1 \leq i \leq m$ and $b_1 \leq b_2 \leq \dots \leq b_m$, is computed by first partitioning V into subgroups with disjoint footprints in the lowest dimension, as follows. A subgroup $V_i = \{v_{i_{min}}, v_{i_{min}+1}, \dots, v_{i_{max}}\}$ is defined by lowest dimension subscripts $a_1 * l_1 + b_j$, where $\forall j, \quad i_{min} < j \leq i_{max}$,

$$\begin{aligned}
& (b_{j-1} \leq b_j) \wedge \\
& (b_j - b_{j-1} < a_1 * X_{l_1}) \wedge \\
& (b_{i_{min}} = b_1 \vee b_{i_{min}} - b_{i_{min}-1} \geq a_1 * X_{l_1}) \wedge \\
& (b_{i_{max}} = b_m \vee b_{i_{max}+1} - b_{i_{max}} \geq a_1 * X_{l_1})
\end{aligned} \tag{4.4}$$

Then the group footprint V is computed as the sum of the disjoint footprints of sets V_i , as in (4.5).

$$F_{l_h}(V) = \sum_i F_{l_h}(V_i) \tag{4.5}$$

The footprint of each subgroup V_i is computed by extending Equation (4.3) to $m > 2$ references. For example, when the references in V have self-spatial reuse, as in Equation (4.3b) ($a_1 < SWS$), each subgroup V_i has a footprint consisting of contiguous superwords, since $b_j - b_{j-1} < a_1 * X_{l_1}$ for all j such that $i_{min} < j \leq i_{max}$. The footprint of V_i consists of the union of the individual footprints, with size given by Equation (4.6).

$$\begin{aligned}
F_{l_h}(V_i) &= F_{l_h}(\{v_{i_{min}}, \dots, v_{i_{max}}\}) \\
&= \left\lceil \frac{a_1 * X_{l_1} + b_{i_{max}} - b_{i_{min}}}{SWS} \right\rceil
\end{aligned} \tag{4.6}$$

For example, if $SWS = 4$ and $X = 4$, group $V = \{A[i], A[i + 2], A[i + 5], A[i + 12], A[i + 14]\}$ can be partitioned into two subgroups $V_1 = \{A[i], A[i + 2], A[i + 5]\}$ and $V_2 = \{A[i + 12], A[i + 14]\}$ with disjoint superword footprints. Since the references have

self-spatial reuse, each individual footprint and the footprint of each subgroup is a set of contiguous superwords. The total number of superwords accessed by the references in V is the sum of the disjoint footprints of sets V_1 and V_2 , as in (4.7).

$$\begin{aligned}
F_{l_1}(V) &= F_{l_1}(V_1) + F_{l_1}(V_2) \\
&= \left\lceil \frac{1 * 4 + 5 - 0}{4} \right\rceil + \left\lceil \frac{1 * 4 + 14 - 12}{4} \right\rceil \\
&= 5
\end{aligned} \tag{4.7}$$

Dimension h is not the lowest dimension ($h \neq 1$). When h is one of the higher dimensions, the superword footprint of $V = \{v_1, v_2, \dots, v_m\}$ in loop l_h is again the union of the individual footprints.

From Section 4.3.1.1, the footprint of each reference v_i in the unrolled loop body consists of a set of X_{l_h} disjoint footprints (each footprint corresponding to a copy of v_i created by unrolling), and the offset between each pair of consecutive footprints is $a_h * \prod_{i=1}^{h-1} s_i$, where s_i is the size of dimension i .

Therefore the footprints of different references in the group may overlap, depending on the values of a_h , b_j and the unroll factor X_{l_h} . The footprints of two uniformly generated references v_1 and v_2 overlap in dimension h if there exists an integer value k , $1 \leq k \leq X_{l_h}$ that satisfies Condition (4.8):

$$a_h * k + b_1 = a_h + b_2. \tag{4.8}$$

that is, if $(b_2 - b_1) \% a_h = 0$ and $(b_2 - b_1) / a_h + 1 \leq X_{l_h}$. Furthermore, if there exists k satisfying the above condition, the footprints of the last $X_{l_h} - k + 1$ copies of v_1 in the unrolled loop body overlap with those of the first $X_{l_h} - k + 1$ copies of v_2 . The footprint of $\{v_1, v_2\}$ is then given by Equation (4.9).

$$\begin{aligned}
F_{l_h}(v_1, v_2) &= (k - 1) * F_{l_{h-1}}(v_1) \\
&+ (X_{l_h} - k + 1) * F_{l_{h-1}}(v_1, v_2) \\
&+ (k - 1) * F_{l_{h-1}}(v_2)
\end{aligned} \tag{4.9}$$

To compute the size of the entire footprint of V in l_h , our algorithm partitions V into subsets $V_i = \{v_{i_{min}}, \dots, v_{i_{max}}\}$ such that, for any j , $i_{min} < j \leq i_{max}$, the pair $\{v_{j-1}, v_j\}$ satisfies Condition (4.8). The footprint of V_i is the union of the footprints of its reference set and is computed by extending Equation (4.9) to more than two references.

4.3.2 Registers for Reuse Across Iterations

In addition to superword registers for exploiting reuse in the body of the transformed loop nest, extra superword registers may be required for exploiting reuse across iterations of the innermost loop for references with group-temporal reuse carried by the innermost loop n of the transformed loop nest.

To compute the number of registers needed to exploit group-temporal reuse across iterations of loop n , the algorithm examines groups of references that have consistent dependences carried by n ³. Assume that unroll-and-jam has been applied to outer loops in a nest. After subsequently unrolling the innermost loop, extra registers are required if the reuse distance between references prior to unrolling loop n is larger than the unroll amount, *i.e.*, if $d_n > X_n$, as in Figure 4.2, where $d_n = 8$ and $X_n = 4$.

Let $C = \{v_1, v_2, \dots, v_m\}$ be a set of references that is a subset of a uniformly generated set, and, prior to unrolling the innermost loop resulting from unroll-and-jam by X_n , each pair $\langle v_i, v_{i+1} \rangle$ in C has a consistent dependence $d^i = \langle 0, 0, \dots, d_n^i \rangle$, $d_n^i > 0$. Also, assume that the array subscript of the lowest dimension of each reference v_i in C is of the form

³Note that such references, if their lowest dimension varies with n , may also have group-spatial reuse across loop iterations. However, our algorithm focuses on exploiting group-temporal reuse across iterations, since most of the group-spatial reuse is achieved within the body of the unrolled loop.

$a_i * n + b_i$, and that $b_1 \leq b_2 \leq \dots \leq b_m$. Unrolling loop n generates X_n copies of each original reference v_i in the body of the transformed loop nest.

When d_n^i is a multiple of the unroll factor X_n , each pair of copies of references $\langle v_i, v_{i+1} \rangle$ will reuse data after $\frac{d_n^i}{X_n}$ iterations. When d_n^i is not a multiple of X_n , some copies of a reference will reuse data after $\left\lceil \frac{d_n^i}{X_n} \right\rceil - 1$ iterations of n , while others will have a reuse distance of $\left\lceil \frac{d_n^i}{X_n} \right\rceil$ requiring one more register per copy. Thus, each pair of copies of references $\langle v_i, v_{i+1} \rangle$ requires at most $\left\lceil \frac{d_n^i}{X_n} \right\rceil - 1$ additional superword registers to keep the data across iterations of the innermost loop.

The number of registers required to exploit reuse across iterations of n by all pairs of copies is the number of registers required for each pair times the number of registers required to keep the superword footprint of reference v_i in the transformed loop nest:

$$R_A(v_i, v_{i+1}) = \left(\left\lceil \frac{d_n^i}{X_n} \right\rceil - 1 \right) \times F_L(v_i) \quad (4.10)$$

Equation (4.10) may overestimate the number of registers if the footprint component ($F_L(v_i)$) overestimates registers, or for certain copies of references if d_n^i is not a multiple of X_n .

The total number of registers required for exploiting reuse across iterations for set C with leading reference v_1 is given by:

$$R_A(C) = \sum_{1 \leq i < m} \left(\left(\left\lceil \frac{d_n^i}{X_n} \right\rceil - 1 \right) \times F_L(v_1) \right) \quad (4.11)$$

4.3.3 Putting It All Together

Subsections 4.3.1 and 4.3.2 describe the computation of the number of registers required to exploit reuse in the body of the innermost loop (superword footprint) and across iterations of the innermost loop, assuming that unroll-and-jam has been applied the loop nest. This section presents the computation of the total number of registers required and

the total number of memory accesses in the innermost loop of the transformed loop nest, which are the metrics used to prune and guide the search for unroll factors described in Section 4.2.

The total number of registers required to exploit reuse is the sum of the superword footprint of the references in the innermost loop of the transformed loop nest and the number of registers needed for exploiting reuse across iterations of the same innermost loop.

The superword footprint of the references, $F_L(V)$, is computed as in subsection 4.3.1. The total number of extra registers required for exploiting reuse across iterations of the innermost loop is computed as in subsection 4.3.2, for each set C of loop-variant references with consistent dependences carried by the innermost loop.

The total number of superword registers required is then:

$$R(V) = F_L(V) + \sum_C R_A(C) \quad (4.12)$$

The total number of memory accesses in the innermost loop of the transformed loop nest is the sum of the memory accesses of each group C of references that are variant with the innermost loop n and have consistent dependences carried by n . For each group C , the number of memory accesses is given by the superword footprint of the leading reference of the group, v_1^c :

$$M(C) = F_L(v_1^c) \quad (4.13)$$

The total number of memory accesses is then:

$$M(V) = \sum_c F_L(v_1^c) \quad (4.14)$$

4.4 Determining Unroll Factors

As previously stated, the goal of the search algorithm is to identify the unroll factors for the loops in the loop nest such that the number of memory accesses is minimized, without exceeding available registers. Thus, we must consider an n -dimensional search space, where each dimension has the number of elements corresponding to the iteration count of the loop. A full global search of this search space is prohibitively expensive, especially for deep loop nests or large loop bounds. Thus, we use a number of strategies for pruning the search space.

First, we eliminate from the search loops that do not carry reuse or for which unroll-and-jam is not safe. Further, we rely on the observation that the number of registers required monotonically increases with the unroll factor of a loop, assuming that all other unroll factors are fixed. Thus, we need not search beyond the unroll factors that exceed available registers. This latter point significantly prunes the search space in that the number of registers is usually fairly small (*e.g.*, 32 superword registers on the AltiVec), so that the search is concentrated on fairly small unroll factors. These pruning strategies are used in our current implementation, and at least for the programs in this study, are quite effective at making the search practical.

Further pruning is possible by making the additional observation that for each unrolled loop l , the amount of reuse of an array reference with reuse carried by l increases with the unroll factor X_l . Therefore reuse, like the register requirement calculation, is a monotonic, non-decreasing function of the unroll factor for each loop, given that the unroll factor of all other loops is fixed. Thus, within each dimension, holding all other unroll factors constant, binary search can be used rather than searching all points. We can also increase unroll factors by amounts corresponding to the superword size without much loss of precision, rather than considering each possible unroll factor, since the register requirements increase stepwise as a function of superword size. Additional pruning techniques that take into

account the hardware’s capability to take advantage of the results of optimization have been used in prior work [10, 63].

Our implementation navigates the search space from innermost loop to outermost loop, for the applicable loops in the nest, varying the unroll factor of one loop while keeping the unroll factors of all other loops fixed. Within a dimension of the search space, the lowest number of memory accesses will be derived at the largest unroll factor that meets the register constraint. However, lower unroll factors may also have the same estimate of memory accesses (because reuse is monotonically non-decreasing), so we identify the lowest unroll factor with the equivalent estimate of memory accesses. Then, the implementation considers the next applicable outer loop and the applicable inner loops nested inside it, and in a particular dimension, each time it reaches the largest unroll factor that meets the register constraint, it compares the estimated number of memory accesses to the lowest estimate so far to determine if a better solution has been found. The final result of the algorithm is the unroll factors corresponding to the best solution. As a subtle point, when unroll-and-jam is applied from outermost to innermost loop, unrolling the inner loop does not affect data access patterns or reuse distance. For this reason, inner loop unrolling is not performed in earlier work [10]. In our context, however, because of the relationship between superword-level parallelism and superword replacement, inner loop unrolling exposes opportunities for superword loads and stores and thus can impact the analysis of register requirements. Nevertheless, when reuse is exploited across iterations of the innermost loop body as described in Section 4.3.2, it is not necessary to unroll the innermost loop beyond the superword size to achieve the goal of considering register requirements in conjunction with superword-level parallelism. Note, however, that smaller unroll factors for the innermost loop may be selected, if an unroll-and-jam of an outer loop carries more parallelism and reuse.

Although this search should theoretically find the optimal solution, according to our optimization criteria, in fact the solution is not guaranteed to result in the fewest number

of memory accesses, for a number of reasons. First, in a few cases as noted, the register requirement analysis defined in the previous section must conservatively approximate. Second, it is difficult to estimate the register requirements used to hold temporaries, so we conservatively approximate this as well. Third, there is a tradeoff between using extra registers to hold values across iterations, as discussed in Section 4.3.2, versus using them to actually exploit reuse within the transformed innermost loop body. In fact, in general the algorithm does not take into consideration the amount of reuse resulting from performing superword replacement on specific references; replacing some references has more impact on decreasing memory accesses than others.

4.5 Code Transformations

The previous two sections have described how the compiler analyzes the code to identify reuse, register requirements and the unroll factors leading towards the lowest number of memory accesses. In this section, we describe how these analyses are used in transforming the code to achieve the desired result.

In the previous section, we showed how consideration of superwords instead of scalar variables greatly increases the complexity of determining the number of registers and memory accesses associated with exploiting reuse under different unroll amounts. In this section, we further discuss the increased complexity of code generation when performing superword replacement instead of scalar replacement. The chief source of code generation complexity is the need for superword objects to be properly *aligned*, as in the following examples.

When performing memory operations, the architecture may actually require that an access be aligned at superword boundaries. For example, the AltiVec ignores the last four bits of an address when performing a superword load or store. In such an architecture, when an access is not aligned at a superword boundary, the compiler or programmer

must read/write two adjacent superwords. A series of additional instructions *packs* the two superwords for reads or *unpacks* a superword into its corresponding two superwords for writes. Even on architectures that support memory accesses not aligned at superword boundaries, such as Intel's SSE, there is a performance penalty on unaligned accesses because the hardware must perform this realignment.

To perform an arithmetic or logical operation on two superword registers, the fields of the two operands must also be aligned. For example, to add the third and fourth fields of one superword register to the first and second fields of another, one of the registers must be shifted by two fields. Consider also the following example:

```
for i = 1, n
    c[i] = a[2i] + b[i]
```

The access to **a** has a stride of 2, while the access to **b** has a unit stride. Thus, the compiler or programmer must first pack the even elements of **a** into a superword register before adding them to the elements of **b**. A third example occurs when exploiting partial reuse of a superword where data in a register must be aligned to accommodate the next operation.

In the SLP compiler, the default solution to alignment involves packing data through memory. The SLP compiler allocates superword variables by declaring them using a special `vector` type designation, which is interpreted by the backend compiler to align the beginning of the variable to a superword boundary in memory. The start of each dimension of an array of such objects should also be aligned, by padding if necessary. Under these assumptions, the SLP compiler can detect when operations are unaligned. Unaligned data is packed into an aligned superword in memory before being loaded into a superword register, and is unpacked before storing back to memory ⁴.

⁴For architectures that support copying between scalar and superword register files, such as Intel's SSE and DIVA, this packing can be performed more efficiently through register copies.

In summary, alignment is a key consideration in code generation, and the overhead of performing alignment operations can be quite high. Further, alignment operations may require a number of additional superword registers, and in some cases, may result in additional accesses to memory not accounted for by the model in the previous section. In this section, we show how to achieve the number of registers derived by our model through a set of code transformations, presented in the order in which they are performed by our compiler. In addition to superword replacement, described in Section 4.5.2, we also describe how index set splitting is used to align accesses to the beginning of an iteration in Section 4.5.1, and how our compiler eliminates additional memory accesses resulting from packing through memory for alignment in Section 4.5.3. We illustrate how these transformations collaborate with each other by way of an example in Figure 4.5, which is a simplified FIR filter.

4.5.1 Index Set Splitting

A simple way to reduce the need for alignment operations, when applicable, is to perform index set splitting on loops. For example, in Figure 4.5(b), the initial access to `out[1]` refers to the second field of a superword, assuming `out[0]` is aligned at a superword boundary. Through index set splitting, the portion of the loop from line 4-6 will always perform aligned accesses. This transformation is always safe, and is profitable whenever it increases the number of aligned memory accesses.

We assume index set splitting is performed prior to the SLP compiler. The loop is transformed so that accesses corresponding to a particular reference in the main loop body are aligned to superword boundaries. If there are multiple references and different choices for index set splitting are needed to align specific references, we select a representative reference that, if aligned through index set splitting, will also maximize alignment for other references. The reference selected must have unit stride within the innermost loop.

```

1)  for (i = 1; i < 64; i++)
2)      out[i] = 0.0;
3)
4)  for (i = 256; i < 320; i++)
5)      for (j = 0; j < 256; j++)
6)          out[i-256] = out[i-256] + in[i-j] * coe[j];

```

(a) Original

```

1)  for (i = 1; i < 4; i++)
2)      out[i] = 0.0;
3)
4)  for (i = 4; i < 64; i++)
5)      out[i] = 0.0;
6)
7)  for (i = 256; i < 320; i++)
8)      for (j = 0; j < 256; j++)
9)          out[i - 256] = out[i - 256] + in[i - j] * coe[j];

```

(b) After index set splitting

```

1)  for (i = 1; i < 4; i++)
2)      out[i] = 0.0;
3)
4)  for (i = 4; i < 64; i += 4){
5)      out[i + 0] = 0.0;
6)      out[i + 1] = 0.0;
7)      out[i + 2] = 0.0;
8)      out[i + 3] = 0.0;
9)  }
10) for (i = 256; i < 320; i += 8)
11)     for (j = 0; j < 256; j += 8){
12)         out[i + 0 - 256] = out[i + 0 - 256] + in[i + 0 - (j + 0)] * coe[j + 0];
13)         out[i + 0 - 256] = out[i + 0 - 256] + in[i + 0 - (j + 1)] * coe[j + 1];
14)         ⋮
15)         out[i + 7 - 256] = out[i + 7 - 256] + in[i + 7 - (j + 7)] * coe[j + 7];
16)     }

```

(c) After unroll-and-jam

Figure 4.5: Code generation example.

| | |
|--|---|
| <pre> ⋮ 1) flat1 = *((float *)&vec0 + 3); 2) flat2 = *((float *)&vec1 + 0); 3) flat3 = *((float *)&vec1 + 1); 4) flat4 = *((float *)&vec1 + 2); 5) *((float *)&vec2 + 0) = flat1; 6) *((float *)&vec2 + 1) = flat2; 7) *((float *)&vec2 + 2) = flat3; 8) *((float *)&vec2 + 3) = flat4; 9) vec4 = vec_add(vec3, vec2); 10) vec_st(vec4, i * 4 + 0, (float *)&out[-63]); 11) vec5 = vec_ld(i * 4, (float *)&out[-63]); 12) flat5 = *((float *)&vec6 + 2); 13) flat6 = *((float *)&vec7 + 2); 14) *((float *)&vec8 + 0) = flat5; 15) *((float *)&vec8 + 1) = flat6; ⋮ </pre> | <pre> ⋮ 1) flat1 = *((float *)&vec0 + 3); 2) flat2 = *((float *)&vec1 + 0); 3) flat3 = *((float *)&vec1 + 1); 4) flat4 = *((float *)&vec1 + 2); 5) *((float *)&vec2 + 0) = flat1; 6) *((float *)&vec2 + 1) = flat2; 7) *((float *)&vec2 + 2) = flat3; 8) *((float *)&vec2 + 3) = flat4; 9) vec4 = vec_add(vec3, vec2); 10) flat5 = *((float *)&vec6 + 2); 11) flat6 = *((float *)&vec7 + 2); 12) *((float *)&vec8 + 0) = flat5; 13) *((float *)&vec8 + 1) = flat6; ⋮ </pre> |
| (d) After SLP compilation | (e) After superword replacement |

```

      ⋮
1) temp1 = replicate(vec0, 3);
2) temp2 = replicate(vec1, 0);
3) temp3 = replicate(vec1, 1);
4) temp4 = replicate(vec1, 2);
5) vec2 = shift_and_load(temp1, temp1, 4);
6) vec2 = shift_and_load(vec2, temp2, 4);
7) vec2 = shift_and_load(vec2, temp3, 4);
8) vec2 = shift_and_load(vec2, temp4, 4);
9) vec4 = vec_add(vec3, vec2);
10) temp1 = replicate(vec6, 2);
11) temp2 = replicate(vec7, 2);
11) vec8 = shift_and_load(temp1, temp1, 4);
13) vec8 = shift_and_load(vec8, temp2, 12);
      ⋮

```

(f) After packing in registers

Figure 4.5: Code generation example (Continued).

Let i be the loop index variable for the innermost loop, and lb and ub are the lower and upper bounds for i . To derive the loop bounds for the copies of the innermost loop resulting from index set splitting, we begin with the starting address, $addr$, of the reference when $i = lb$, where $addr = base + offset$. Here, $base$ refers to the beginning of the lowest dimension of the selected array, and $offset$ is the offset within that dimension (Recall that the beginning of each dimension is aligned at superword boundaries.).

The lower bound ($split$) of the main loop body is computed by the following equation.

$$split = \begin{cases} lb & \text{if } offset \bmod SWS = 0 \\ lb + SWS - (offset \bmod SWS) & \text{if } offset \bmod SWS \neq 0 \end{cases} \quad (4.15)$$

If lb is constant, $split$ can be computed at compile time. Otherwise, it is computed at run time. In the example in Figure 4.5, $offset$ for `out[1]` is 1, so if $SWS = 4$, then $split = 4$.

4.5.2 Superword Replacement

Superword replacement removes redundant loads and stores of superword variables, using superword temporaries instead. We assume that this code transformation will be followed by register allocation that places these variables in registers. For example, in Figure 4.5(d) and (e), the store and load at statements 10 and 11 can both be eliminated, and `vec4` can be used in place of `vec5` in subsequent statements. Superword replacement is also affected by alignment, in that we detect redundant loads and stores by identifying distinct memory operations that refer to the same aligned superword, even if the addresses are not identical.

The compiler recognizes opportunities for superword replacement by determining that addresses and offsets for different memory accesses fit within the same superword, and verifies that there are no intervening kills to the memory locations. The current implementation uses *value numbering* [52] to detect such opportunities. Value numbering is a

well-known compiler technique for detecting redundant computation, but it is sensitive to operand and operator ordering. To increase the success of value numbering, we first preprocess the code so that memory access operations are rewritten into a canonical form, constant folding has been applied to simplify addresses, and alignment is taken into account. As earlier stated, all memory accesses are aligned at superword boundaries, so if an unaligned address appears in a memory access, the resulting access will be aligned to the preceding superword boundary. The preprocessing performs this alignment in software so that redundant accesses will be identified by value numbering.

The current implementation of superword replacement is more restrictive than what was presented in Section 4.2. Value numbering operates on a basic block at a time so we cannot exploit reuse across iterations of the unrolled loop body. This is because we are performing this transformation after the SLP compiler has flattened the loop structure to `gotos` and labels. The dependence information used to perform the register requirement analysis cannot easily be reconstructed from such low-level code. In an implementation where SLP and SLL are more tightly integrated, it should be possible to perform superword replacement as a byproduct of the analysis in Section 4.2.

4.5.3 Packing in Superword Registers

As previously described, packing in memory is performed to align superword objects. Memory packing moves data elements from a set of locations in memory (*sources*) to a superword location (*destination*) so that the destination superword contains contiguous data, aligned to a superword boundary or to another operand. For example, in Figure 4.5(e), superword variables `vec0` and `vec1` are the sources and superword variable `vec2` is the destination for memory packing in lines 1-8.

Our implementation performs a transformation we call *register packing* to optimize memory packing operations. A series of memory loads and stores for scalar variables are replaced by superword operations on registers, as shown in Figure 4.5(f). We identify a

destination as a superword data type that is the target of a series of scalar store instructions into its fields, such as `vec2` in the example. The corresponding sources are identified by finding preceding loads of these scalar variables. If the inputs to these loads are fields of superword data types, then these superwords are the sources. In the example, `flat1` is stored into a field of `vec2`, and there is a preceding load of `flat1` that copies a field of source `vec0`. Once we find such a pattern, we verify the safety of this transformation by guaranteeing that there are no intervening modifications or uses of either the scalar variables or destination superwords between loading the scalar variables and completion of storing into the destination. We also verify that the destination statements ultimately produce contiguous data in the superword. We define *source* and *destination indices* as the fields in the source and destination superword variables, respectively. For example, the source index of `vec0` is 3 in line 1 of the example.

Once the compiler identifies sources and destinations, it transforms the code to replace memory accesses with operations on superword registers. The register packing transformation takes advantage of two instructions that are common in multimedia extension architectures. *Replicate* replicates one element of a source register to all elements of a temporary output register (Figure 4.6(a)). *Shift-and-load* takes two input registers. The first input register is a temporary, and is shifted left by the number of bytes specified by the third argument. The same number of fields is taken from the second input register, which is a temporary derived from a source superword, to fill the output temporary register (Figure 4.6(b)). Simply stated, we are shifting each source element into the destination superword, in order, so that the final result is a destination superword that corresponds to contiguous aligned data.

The steps of the register packing transformation are as follows.

1. We sort the destination statements in increasing order of their destination indices.

We then sort the source statements to correspond to the ordering of the destination

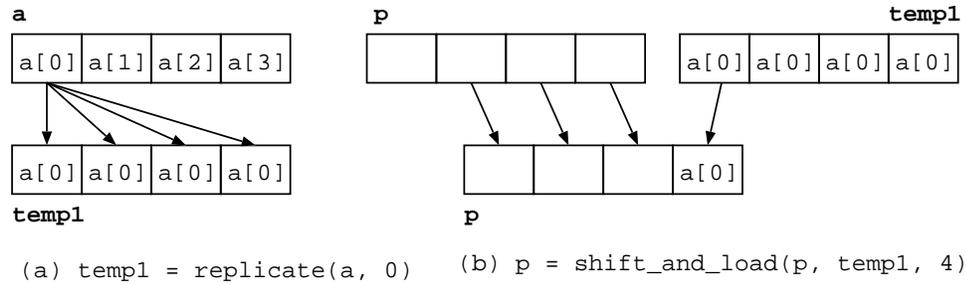


Figure 4.6: Operations used for packing in registers.

statements, so that, for example, the scalar variable associated with the first source statement is the same as the scalar variable associated with the first destination statement.

2. For each source statement, in sorted order, we generate a `replicate` statement whose two inputs are the source superword and the source index, and the output is a superword temporary. For example, as in Figure 4.5(f), we have replaced line 1 of Figure 4.5(e) with `temp1 = replicate(vec0, 3)`.
3. We replace each destination statement, in sorted order, with a `shift_and_load` operation. The first input is the destination superword. The second input is the temporary generated by the `replicate` of the corresponding source statement. The third argument, the shift amount, usually involves shifting by a single superword field. For the last destination field, the shift amount is the difference, in bytes, between the *SWS* and the last destination field. For completely filled destination superwords, it will also be just a single field. For example, in lines 1-8 of Figure 4.5(e), the destination superword is completely filled, so the shift amount is always a single 4-byte field. In lines 10-13, however, only the first two fields are filled, so the shift amount of the last destination statement is a total of 12 bytes.
4. Source statements are deleted if the scalar variables are not live beyond the corresponding destination statements.

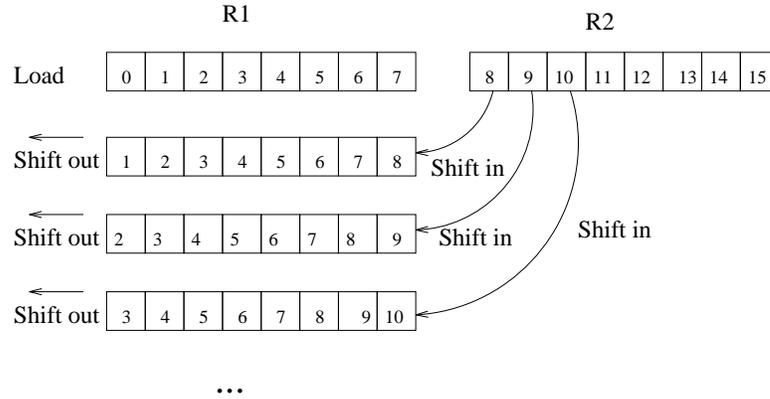


Figure 4.7: Shifting.

4.5.4 An Example: Shifting for Partial Reuse

In addition to the three optimization opportunities described in this section, we discovered a new optimization opportunity, called *shifting*, for reducing memory accesses. In shifting, data in superword registers are partially reused. *Partial spatial reuse* of superwords occurs when distinct loop iterations access data in consecutive superwords in memory, partially reusing the data in one or both superwords, as shown by the example in Figure 4.5(a), and illustrated graphically in Figure 4.7. In this example, as before assuming that $SWS = 4$, array reference $in[i - j]$ has partial spatial reuse in loop i . For a fixed value of i and j , the data accessed in iteration $\langle i, j \rangle$ consists of the last three words of the superword accessed in iteration $\langle i - 1, j \rangle$, plus the first word of the next superword in memory. This type of reuse can be exploited by shifting the first word out of the superword, and shifting in the next word, as in Figure 4.7. As partially shown in Figure 4.5(c) and (f), only four superwords need to be loaded for the data accessed in the 64 copies of $in[i - j]$ in the loop body, after shifting is applied. Before shifting, $in[i - j]$ had to be loaded from memory (and possibly aligned) for each of the four copies of $in[i - j]$ in the loop body.

This shifting opportunity arises frequently in both signal and image processing applications, where one object is compared to a subcomponent of another object, such as the example in Figure 4.5(a). We detect these opportunities through the analysis described

in Section 4.2. The optimization shown in Figure 4.7 falls out from the combination of unroll-and-jam, alignment operations generated by the SLP compiler, superword replacement and register packing.

Chapter 5

CODE GENERATION

In addition to the optimizations described in Chapters 3 and 4, this chapter describes several optimizations and their associated code generation requirements to exploit SLP for full multimedia applications. They are the techniques to parallelize *type size conversion*, *reduction* and *unaligned memory references* and a new packing algorithm called *prepacking*. *Type size conversion* is a common feature of multimedia applications, particularly to promote small data types before or after arithmetic operations. A *reduction* operation is a computation of a sum, product, maximum, or other commutative and associative operation over a set of data elements. A memory reference is *unaligned* if at least one pair of its run time addresses are not congruent with each other modulo superword width. Finally, when a memory reference can be packed with multiple other memory references, the first packing opportunity encountered by the SLP algorithm may not be the best choice. For each of these four cases, we describe our extension in the next four sections. In the last section, we summarize this chapter.

5.1 Type Size Conversion

Type size conversion is a common feature of multimedia applications, particularly to promote small data types before or after arithmetic operations. Type size conversion is

| | |
|--|--|
| <pre>int in[1024]; short sh[1024]; for (i=0; i<1024; i++) in[i] = (int)sh[i];</pre> | <pre>int in[1024]; short sh[1024]; for (i=0; i<1024; i+=8) in[i:i+3], in[i+4:i+7] = typesize_up(sh[i:i+7]);</pre> |
| (a) Original | (b) Our approach |

Figure 5.1: Parallelization of type size conversions

more difficult on superwords than scalar data types, due to alignment issues, instruction set limitations and the impact on parallelization.

We extend the SLP compiler to perform type conversions in parallel. On AltiVec, the available instructions supporting type size conversion convert to fields that are half or double the size of the source operand. Type size conversions of a factor larger than two must be broken into multiple conversions. The alignment offset of the destination variable is adjusted from that of the source variables. Predicate variables also require type conversions so that they match the size of the destination variable of the instruction being guarded. To represent parallel type size conversion operations, we define the following parallel macros.

```
dst1, dst2 = typesize_up(src)
dst        = typesize_down(src1, src2)
```

The macro `typesize_up` doubles the type size of the data fields in `src` by assigning the higher half to `dst1` and the lower half to `dst2`. The macro `typesize_down` concatenates two superword operands `src1` and `src2`, reduces the data field size by half and assigns the result to `dst`. These high-level parallel macros are replaced by a few AltiVec instructions during code generation. For *signed* operands, different AltiVec instructions are generated for the macro `typesize_up` from *unsigned* operands.

Figure 5.1(c) shows the code generated by our approach for type size conversion. After eight short integers are loaded into a superword register, type size conversion is performed in parallel using a few AltiVec instructions, represented by `typesize_up`. Finally, the two

| | |
|---|--|
| <pre> for (i=0; i<16; i++) sum = sum + a[i]; </pre> <p>(a) Original</p> | <pre> sumV = pack(0, 0, 0, 0); for (i=0; i<16; i+=4) sumV = sumV + a[i:i+3]; sum1, sum2, sum3, sum4 = unpack(sumV); sum = sum + sum1; sum = sum + sum2; sum = sum + sum3; sum = sum + sum4; </pre> |
| <pre> for (i=0; i<16; i+=4){ sum = sum + a[i]; sum = sum + a[i+1]; sum = sum + a[i+2]; sum = sum + a[i+3]; } </pre> <p>(b) Unrolled</p> | <pre> sumV = pack(0, 0, 0, 0); for (i=0; i<16; i+=4) sumV = sumV + a[i:i+3]; sum1, sum2, sum3, sum4 = unpack(sumV); sum = sum + sum1; sum = sum + sum2; sum = sum + sum3; sum = sum + sum4; </pre> <p>(c) Reduction optimization</p> |

Figure 5.2: Parallelization of reduction sum.

superwords are stored in memory. Among the 14 benchmarks used in the next chapter, two (`MPEG2-dist1` and `EPIC-unquantize`) have type size conversions.

5.2 Reduction

A *reduction* operation is a computation of a sum, product, maximum, or other commutative and associative operation over a set of data elements. From the compiler’s perspective, a reduction occurs when a location is updated on each iteration of a loop, where a commutative and associative operation is applied to that location’s previous contents and some data value. In this case, it is safe to reorder the operations. However, reduction variables have dependences, so the compiler must transform the code to obtain parallel code from a sequential code. Figure 5.2(a) shows a loop containing a reduction sum operation. When this loop is unrolled as shown in (b), scalar data dependences prevent packing the isomorphic statements.

We extend the SLP algorithm to support reductions in a way similar to the standard code generation for reductions in multiprocessors. We create as many private copies of the reduction variable as will fit in a superword. The private copies are packed into one

| | |
|---|---|
| <pre> for (i=0; i<1024; i++) for (j=0; j<256; j++) t = inp[i+j]; </pre> | <pre> for (i=0; i<1024; i++) for (j=0; j<256; j+=4){ tV1 = load(&inp[i+j]); tV2 = load(&inp[i+j+4]); permV = perm_vec(&inp[i+j:i+j+3]); tV = permute(tV1, tV2, permV); } </pre> |
| (a) Original | (c) Code generation |
| <pre> for (i=0; i<1024; i++) for (j=0; j<256; j+=4) tV = inp[i+j:i+j+3]; </pre> | |
| (b) Parallelized | |

Figure 5.3: Parallelization of unaligned memory references

superword and reduction operations are performed in parallel when the loop is parallelized.

Figure 5.2(c) shows the code after the reduction optimization is applied to the loop in (a). Assuming *superword size* is 4, the four private copies are created and initialized by zero in above the parallelized loop, below which a sequential add operation for each private copy accumulates into the global variable. Note that `pack` and `unpack` instructions are moved outside the loop.

Private copies of a reduction variable are initialized with the *identity* of the associated operation. For reduction `sum` in the above example, the private copies are initialized with zero. For *reduction max / min*, private copies are initialized by the reduction variable itself. Of the 14 benchmarks used in the experiments, four (`TM`, `MAX`, `MPEG2-dist1` and `GSM-Calculation`) have reduction operations.

5.3 Alignment Optimization

In Chapter 2, we described alignment analysis that finds constant offsets with respect to superword width for the run time addresses of each memory reference. A memory reference is *unaligned* if at least one pair of its run time addresses are not congruent with each other modulo superword width. In this section, we describe our approach to parallelize

| | |
|--|--|
| <pre> for (y=2; y<768; y++) for (x=0; x<1024; x++) e[y][x] = u[y][x] - u[y-2][x] + u[y][x+1] - u[y-2][x+1] + u[y][x+2] - u[y-2][x+2]; </pre> <p style="text-align: center;">(a) Original</p> <pre> for (y=2; y<768; y++) for (x=0; x<1024; x+=4){ e[y][x+0] = sum(u[y][x+0:x+2]) - sum(u[y-2][x+0:x+2]); e[y][x+1] = sum(u[y][x+1:x+3]) - sum(u[y-2][x+1:x+3]); e[y][x+2] = sum(u[y][x+2:x+4]) - sum(u[y-2][x+2:x+4]); e[y][x+3] = sum(u[y][x+3:x+5]) - sum(u[y-2][x+3:x+5]); } </pre> <p style="text-align: center;">(c) Parallelized by the MIT SLP compiler</p> | <pre> for (y=2; y<768; y++) for (x=0; x<1024; x+=4){ e[y][x+0] = u[y][x+0] - u[y-2][x+0] + u[y][x+1] - u[y-2][x+1] + u[y][x+2] - u[y-2][x+2]; e[y][x+1] = u[y][x+1] - u[y-2][x+1] + u[y][x+2] - u[y-2][x+2] + u[y][x+3] - u[y-2][x+3]; : e[y][x+3] = u[y][x+3] - u[y-2][x+3] + u[y][x+4] - u[y-2][x+4] + u[y][x+5] - u[y-2][x+5]; } </pre> <p style="text-align: center;">(b) Unrolled</p> <pre> for (y=2; y<768; y++) for (x=0; x<1024; x+=4) e[y][x+0:x+3] = u[y][x+0:x+3] - u[y-2][x+0:x+3] + u[y][x+1:x+4] - u[y-2][x+1:x+4] + u[y][x+2:x+5] - u[y-2][x+2:x+5]; </pre> <p style="text-align: center;">(d) Parallelized by prepacking</p> |
|--|--|

Figure 5.4: Parallelization by prepacking

unaligned memory references. In the SLP algorithm, two memory references are packed if they are adjacent to each other, access a constant offset with respect to superword width, and they are not separated by any superword boundary. Thus, unaligned memory references are not parallelized.

We loosen the requirements for packing memory references so that two memory references can be packed only if they are adjacent. Figure 5.3(a) shows an array reference `inp[i+j]` whose alignment offset varies with respect to superword width. With our extension, the array reference can be parallelized as shown in (b). While parallelized, the memory offset of `inp[i+j:i+j+3]` in (c) varies during run time with respect to superword width. For such *unaligned* superword memory references, we generate code such

that a desired superword is obtained dynamically from two aligned superword memory accesses. Figure 5.3(c) shows the code generated from (b). After two adjacent superwords are loaded by aligned memory accesses and a permutation vector is generated from the address, the desired superword is obtained from the two superwords using a `permute` instruction. In general, the address of a superword memory reference can be one of *aligned to zero offset*, *aligned to non-zero offset* or *unaligned*. Depending on the kind of alignment, our implementation generates a simple aligned load, a static alignment with two loads, or a dynamic alignment for an unknown alignment.

5.4 Prepacking to Optimize Parallelization Overhead

The SLP algorithm packs isomorphic scalar instructions into superword instructions. The way in which an SLP compiler packs instructions governs the parallelism that can be exploited and the amount of parallelization overhead. The packing policy in the original SLP algorithm is very simple; two memory references are packed in the first chance where they satisfy the three conditions, that is, they are adjacent to each other, access a constant offset with respect to superword width, and they are not separated by any superword boundary. This packing policy is quite effective in many common cases. However, when a memory reference can be packed with multiple other memory references, the first packing opportunity encountered by the SLP algorithm may not be the best choice. Figure 5.4(a) shows an example loop nest used to illustrate this point. The original loop nest contains adjacent memory references even before unrolling is applied. When the original SLP algorithm is applied to the unrolled loop body shown in (b), it packs the adjacent memory references in the same statement instead of packing them with their unrolled copies, resulting in the code shown in (c). In the first statement of Figure 5.4(b), three array references `u[y][x+0]`, `u[y][x+1]` and `u[y][x+2]` are packed into a parallel memory reference `u[y][x+0:x+2]` in (c). Since the three array elements should be added into a

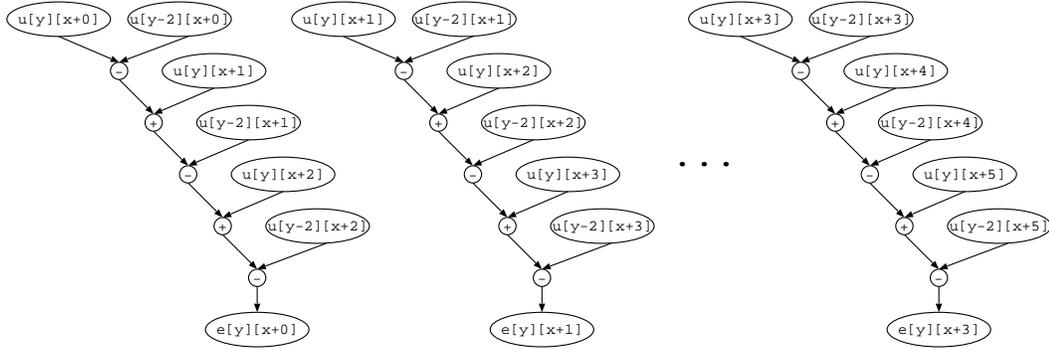


Figure 5.5: Data dependence graphs for the loop body of Figure 5.4(b)

scalar datum, a high level operation `sum` consists of unpacking the three array elements and adding them in scalar mode. The code in (c) is inefficient not only because it involves scalar additions but also because it contains additional memory accesses necessary for unpacking data elements from a superword register to scalar registers. To make a good choice when there are multiple statements with which a given statement can be packed, we need a basic block-level view that can be used to compare the costs of different packing possibilities.

We developed an algorithm that packs isomorphic data dependence graphs instead of isomorphic statements. By this algorithm, we prefer parallelizing isomorphic statements from independent data dependence graphs to the ones from the same data dependence graph. From the unrolled loop body, we first build data dependence graphs. The data dependence graphs for the code in Figure 5.4(b) are shown in Figure 5.5. Next, the isomorphic scalar data dependence graphs are packed into a parallel data dependence graph, where the nodes represent parallel operations and operands. For two independent data dependence graphs to be packed together, they must be *isomorphic* [18] and in addition, each pair of corresponding nodes should have the same operation. For memory reference nodes, there is an additional requirement; the two memory references should be adjacent. Figure 5.4(d) shows the parallel code generated by our approach, where all operations are performed in parallel mode. In our current implementation, this packing

| | |
|---|--|
| <pre> for (i=0; i<1024; i++) for (j=0; j<256; j++) temp[j] = inp[i+j]*fl[j]; </pre> | <pre> for (i=0; i<1024; i+=4) for (j=0; j<256; j+=4){ temp[j+0] = inp[i+j+0]*fl[j+0]; temp[j+1] = inp[i+j+1]*fl[j+1]; temp[j+2] = inp[i+j+2]*fl[j+2]; temp[j+3] = inp[i+j+3]*fl[j+3]; temp[j+0] = inp[i+j+1]*fl[j+0]; temp[j+1] = inp[i+j+2]*fl[j+1]; temp[j+2] = inp[i+j+3]*fl[j+2]; temp[j+3] = inp[i+j+4]*fl[j+3]; temp[j+0] = inp[i+j+2]*fl[j+0]; temp[j+1] = inp[i+j+3]*fl[j+1]; temp[j+2] = inp[i+j+4]*fl[j+2]; temp[j+3] = inp[i+j+5]*fl[j+3]; temp[j+0] = inp[i+j+3]*fl[j+0]; temp[j+1] = inp[i+j+4]*fl[j+1]; temp[j+2] = inp[i+j+5]*fl[j+2]; temp[j+3] = inp[i+j+6]*fl[j+3]; } </pre> |
| (a) Original | (b) Unrolled |

Figure 5.6: Multiple packing choices generated by unrolling multiple loops

algorithm is applied conservatively only when this algorithm is surely profitable over the default strategy. Thus, our new packing algorithm is applied before we apply the original packing algorithm so that we can apply the original packing algorithm to the remaining scalar instructions. Because of this order of application, the new packing algorithm is called *prepacking*.

While prepacking is effective when the original loop body contains adjacent memory references as shown in Figure 5.4(a), similar situations are often generated by our superword-level locality (SLL) algorithm when multiple loops are unrolled. For example, when both loops in Figure 5.6(a) are unrolled as shown in (b), array references to `inp` have multiple packing choices. This type of partial temporal reuse opportunities are common in multimedia applications.

5.5 Summary

SLP is a new technique that provides new optimization opportunities. In addition to the two techniques described in the previous two chapters, we also developed other optimizations that can be used to enhance the performance further. While common in many multimedia applications, type size conversion, reduction and unaligned memory references are not parallelized by the original SLP algorithm. Also, the simple packing policy of the original SLP algorithm is powerful in many common cases, but suffers when there are multiple choices for combining an object with others into a superword. In this chapter, we presented algorithms that can be used to generate efficient parallel code in such cases. All of these extensions working together are essential to obtain the results in the next chapter.

Chapter 6

EXPERIMENTS

Chapters 3, 4, and 5 introduced techniques to exploit superword-level parallelism in the presence of control flow, locality in superword registers, and code generation techniques to support these optimizations. These techniques are applicable to both multimedia extension architectures and a processing-in-memory architecture, DIVA. We have implemented the techniques in the SUIF compiler [32] and evaluated the implementation on 14 benchmarks. In this chapter, we describe the implementation and the experimental evaluation.

This chapter is organized as follows. The next section describes the benchmarks and their input data sets. Implementation and experimental methodology are described in Sections 6.2 and 6.3, respectively. Section 6.4 presents an experimental evaluation of the performance of the benchmarks when all of our techniques are applied. Since this performance is the result of multiple techniques, we also perform separate experiments to identify the benefits of each individual technique. The benefits of packing data dependence graphs, exploiting SLP in the presence of control flow, and exploiting superword-level locality are discussed in Sections 6.5, 6.6, and 6.7 respectively.

| Name | Description | Data Width | # lines |
|-----------------|---|-----------------------------------|---------|
| VMM | Vector-matrix multiply | 32-bit float | 60 |
| FIR | Finite impulse response filter | 32-bit float | 66 |
| YUV | RGB to YUV conversion | 16-bit integer | 110 |
| MMM | Matrix-matrix multiply | 32-bit float | 76 |
| Chroma | Chroma keying of two images | 8-bit character | 106 |
| Sobel | Sobel edge detection | 16-bit integer | 128 |
| TM | Template matching | 32-bit integer | 85 |
| Max | Max value search | 32-bit float | 90 |
| TR | Shortest path search | 32-bit integer | 94 |
| swim | Shallow water model | 32-bit float | 429 |
| tomcatv | Mesh generation | 32-bit float | 197 |
| MPEG2-dist1 | MPEG2 encoder (dist1 function) | 8-bit character 32-bit integer | 157 |
| EPIC-unquantize | EPIC(Efficient Pyramid Image Coder) (unquantize_image of unepic) | 16-bit integer 32-bit integer | 85 |
| GSM-Calculation | GSM encoder (Calculation_of_the_LTP_parameters) | 16-bit integer 32-bit integer | 204 |

Table 6.1: Benchmark programs.

6.1 Benchmarks

We use the set of 14 benchmarks shown in Table 6.1 to evaluate our compiler implementation, representing multimedia and scientific applications. The first nine are kernels consisting of a few loop nests. `VMM` and `MMM` are important kernels in scientific applications, `FIR` is frequently used in digital signal processing, and `YUV` performs conversion between different color encoding systems. `Chroma`, also known as blue screening, merges two images so that an object in a foreground image appears with the other image as a background. `Sobel` detects edges from a gray scale image by performing convolutions with two 3 by 3 pixel areas. `TM` is a representative kernel of an application performing image convolution between two images: a template and an input data image. `Max` is a kernel that looks for the maximum value. Since it is extracted from `tomcatv`, its input data is also collected by running the same application. `TR` is a core computation of the Floyd-Warshall’s shortest path algorithm [18]. The last five are benchmark programs.

| Benchmark | Runtime(%) |
|-----------------|------------|
| MPEG2-dist1 | 55 |
| EPIC-unquantize | 25 |
| GSM-Calculation | 49 |

Table 6.2: Runtime percentage of three functions from UCLA MediaBench.

`Swim` and `tomcatv` are SpecFP applications written in Fortran. `Swim` is a weather prediction program based on the shallow water model [59], and `tomcatv` is a mesh generation program. `MPEG2-dist1`, `EPIC-unquantize` and `GSM-Calculation` are complete functions from the three applications in the UCLA MediaBench [41]. Table 6.2 shows the percentage of each application’s execution time of the baseline code spent in these functions, measured on the platform described in Section 6.3. Each function takes up the largest fraction of the overall runtime in the application. `MPEG2-dist1` computes total absolute difference between two blocks of video frames to convert uncompressed video frames into MPEG-1 and MPEG-2 video coded bitstream sequences. EPIC (Efficient Pyramid Image Coder) is an image data compression utility designed to allow extremely fast decoding at the expense of slower encoding. In EPIC, `EPIC-unquantize` restores the quantized values to decompress the compressed images. GSM is a European standard for mobile communications. In GSM encoder, `GSM-Calculation` computes the long term predictor gain and the long term predictor lag for the long term analysis filter.

Table 6.3 shows the input data sizes for the benchmarks. For the last 8 benchmarks, two different input sizes are used. Large sizes represent the standard inputs provided with the applications whose data footprints are much larger than the L1 cache size. Smaller input sizes that fit in the L1 data cache are also evaluated to help isolate the potential gains of increased parallelism from the effects of the memory behavior of the benchmarks.

| Name | Input Size |
|-----------------|--|
| VMM | 512 elements |
| FIR | 256 filter, 1M signal |
| YUV | 32K elements |
| MMM | 512 elements |
| swim | Specfp95 reference input |
| tomcatv | Specfp95 reference input |
| Chroma | Large: 400×431 color image(1 MB) Small: 48×48 color image(12 KB) |
| Sobel | Large: 1024×768 gray scale image(3 MB) Small: 1024×4 gray scale image(16 KB) |
| TM | Large: 64×64 image, 72 32×32 templates(1.4 MB) Small: 16×64 image, 1 16×32 templates(10 KB) |
| Max | Large: $2 \ 100 \times 256 \times 256$ (52 MB) Small: $2 \ 8 \times 256$ (16 KB) |
| TR | Large: $2 \ 1024 \times 1024$ (8 MB) Small: $2 \ 16 \times 16$ (2 KB) |
| MPEG2-dist1 | Large: data blocks for the first 1000 calls (11 MB) Small: data blocks for the first 2 calls(22 KB) |
| EPIC-unquantize | Large: reference input (393 KB) Small: first 4 calls (6 KB) |
| GSM-Calculation | Large: reference input (1.1 MB) Small: first 50 calls (16 KB) |

Table 6.3: Input data size.

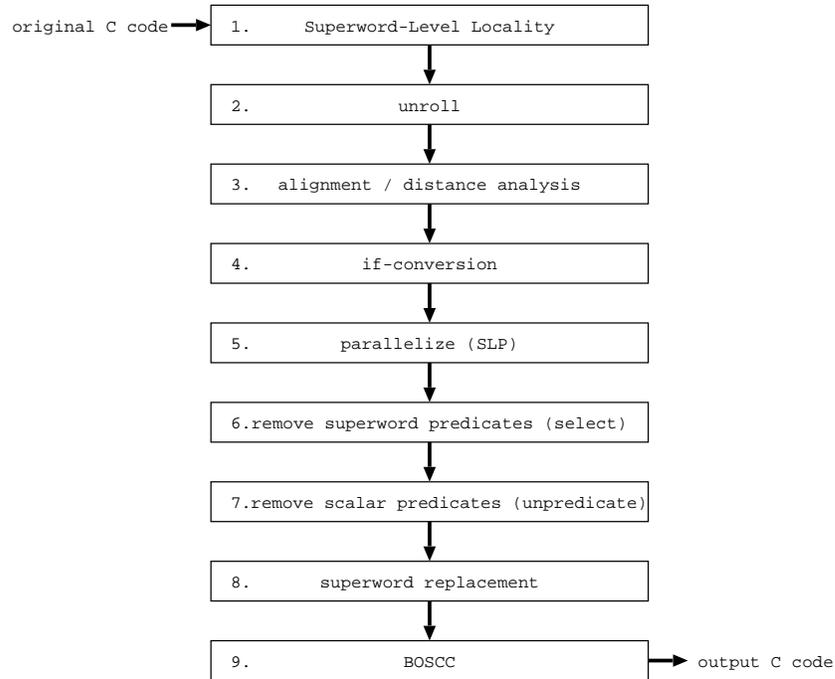


Figure 6.1: Implementation.

6.2 Implementation

Figure 6.1 illustrates the compiler implementation. The input to the system is a C program, which is then optimized by the SUIF passes in Figure 6.1. *Superword-Level Locality* (SLL) determines unroll factors based on the algorithm described in Chapter 4. *Unroll* performs loop unrolling. The unroll factors are either provided by the previous SLL pass or computed by dividing superword width by the smallest data type size. As in [40], *Alignment / distance analysis* determines whether memory references are aligned to superword boundaries and are adjacent to each other in memory. *If-conversion* is applied right before parallelization and results in code for which instructions are predicated. The next three passes can recognize predicates and use the predicate analysis described in Section 2.4. We extend *parallelize (SLP)* so that predicate operands are packed in the same way as the other operands. As described in Chapter 3, predicates are removed by *remove superword predicates (select)* and *remove scalar predicates (unpredicate)*. Then,

redundant superword memory references are eliminated by *superword replacement*. Finally, *BOSCC* instructions are generated wherever profitable according to the model described in Section 3.4. Among the nine passes, *unroll*, *alignment / distance analysis*, and *parallelize (SLP)* are taken from the original SLP compiler developed by Larsen and Amarasinghe [39] and modified to support our extensions.

This ordering of passes was selected primarily for implementation convenience, since we were building on the existing SLP compiler implementation. The SLP passes operate on the code at a low level, where it is difficult to reconstruct the loop structure and array access expressions. Thus, superword-level locality analysis is applied prior to SLP, rather than afterward, as suggested by the examples in Figure 1.5. Superword replacement must follow SLP, which is the reason the components of the SLL algorithm are performed on either side of SLP. Note that both the SLP and SLL passes employ loop unrolling, but for different reasons. The *unroll* pass unrolls the innermost loop of a loop nest to convert loop-level parallelism into basic block-level parallelism. The SLL pass performs unroll-and-jam to expose locality in basic blocks. However, the loop that carries the most spatial locality at the superword-level is often the loop that carries the most superword-level parallelism. Therefore, it is a reasonable choice to use the SLL pass to expose both parallelism and locality in the loop body while suppressing the unrolling originally performed by SLP. The code generation techniques described in Chapter 5 are implemented by extending *parallelize (SLP)* except for the *reduction* transformation, which is incorporated into the *unroll* pass to rename the unrolled copies of the reduction variable.

6.3 Experimental Methodology

Figure 6.2 illustrates the experimental flow. We evaluate six different versions of the codes: *Baseline*, *MIT-SLP*, *SLP+SLL₀*, *SLP-CF-S*, *SLP-CF-S+B*, and *SLP-CF+SLL₁*. *Baseline* is the original C or Fortran program that is the input to the compiler. *MIT-SLP* is compiled

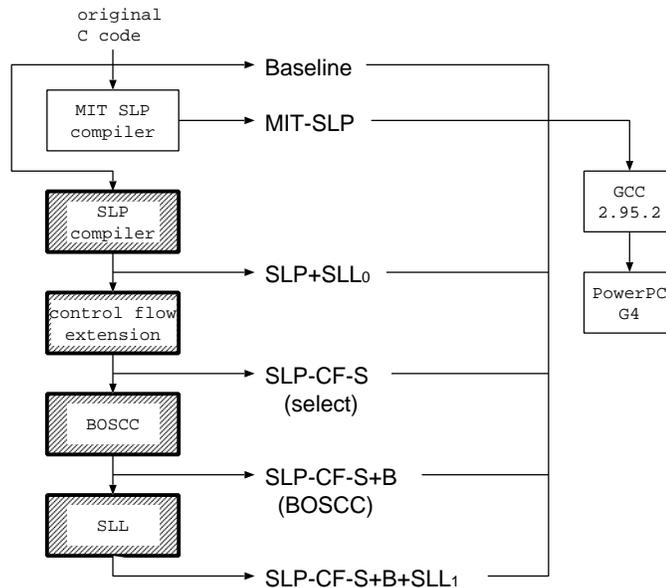


Figure 6.2: Experimental flow.

by the original MIT SLP compiler [39] represented by the three passes 2, 3 and 5 in Figure 6.1. SLP+SLL₀ incorporates *superword replacement* (pass 8 in Figure 6.1) and *packing in superword registers* described in Chapter 4 as well as the code generation techniques described in Chapter 5, which are incorporated into passes 2 and 8. SLP-CF-S exploits SLP in the presence of control flow, represented by passes 4, 6 and 7 in Figure 6.1, in addition to all the optimizations exploited by SLP+SLL₀. Similarly, SLP-CF-S+B exploits BOSCC (pass 9) in addition to all optimizations exploited by SLP-CF-S. SLP-CF+SLL₁ exploits the unroll factors determined by SLL (pass 1) in addition to all the optimizations applied to SLP-CF-S+B.

Each output version is an optimized C program, augmented with special superword data types and operations [50]. The resulting code is compiled by a GCC (version 2.95.2) backend which has been modified to support superword data types and operations for the PowerPC AltiVec [61]. The optimized programs are executed on a 533 MHz Macintosh PowerPC G4, which has a superword register file with 32 128-bit registers, a 32 KByte

L1 cache and a 1 MByte L2 cache. All programs are compiled by the extended GCC backend with optimization flag `-O3`.

6.4 Overall Performance

Figure 6.3 shows the speedups of the five versions with respect to **Baseline**. Each bar represents the corresponding version with the same name in Figure 6.2. For 8 of the 14 programs, MIT-SLP performs worse than **Baseline** because of some overhead introduced by the SUIF compiler passes leading up to SLP, particularly its code transformations related to decomposing program constructs. This overhead is not inherent to the SLP approach, and we believe it could be eliminated with tuning of the SUIF passes. Nevertheless, since it is not identifying parallelism across basic block boundaries, the best results we could hope for from the SLP compiler is no change from the sequential performance unless there is parallelism within the basic block. While the reduction sum operation in **GSM** can be parallelized, it appeared as a data dependence to the original SLP compiler remaining unparallelized. The speedups range from 0.61 to 5.15. When our code generation techniques and two SLL optimizations ¹ are applied, **SLP+SLL₀** speeds up dramatically for the first four kernels. However, the other 10 benchmarks are not improved much. Other than **GSM**, we observe that the **SLP+SLL₀** results, for the eight benchmarks with control flow, do not speed up at all over sequential execution, and for **Max** show a significant degradation. The main reason for this is that **SLP+SLL₀** is unable to exploit *any* parallelism in the presence of control flow. The analyses and transformations in **SLP-CF-S** are crucial to exploiting superword-level parallelism in these codes.

SLP-CF-S, exploiting SLP across basic block boundaries, yields a speedup compared to **SLP+SLL₀** for the eight benchmarks with control flow while there are almost no changes for the first six benchmarks. When BOSCC instructions are exploited in addition,

¹Superword replacement and packing in superword registers

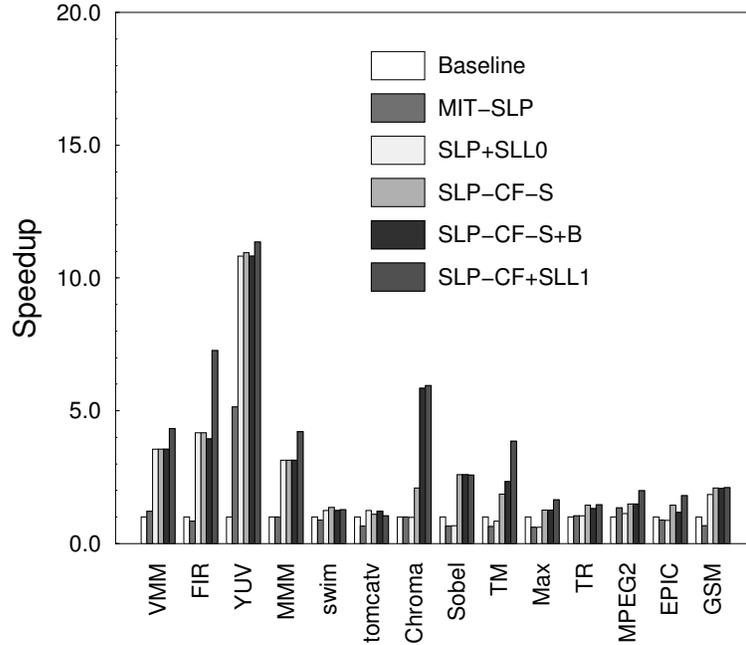


Figure 6.3: Overall speedup breakdown (large data).

SLP-CF-S+B achieves further speedups for **Chroma** and **TM**. The last bar, representing SLP-CF+SLL₁, shows additional improvements in Figure 6.3 for seven of the 14 benchmarks depending on the amount of data reuse. Overall, when all techniques are combined, Figure 6.3 shows the speedups ranging from 1.05 to 11.36.

Cache effects can limit the performance benefits of parallelization for memory-bound computations. To demonstrate the potential of parallelization, Figure 6.4 shows the same graph for the eight benchmarks with control flow using small data set sizes. The speedups for seven of eight benchmarks improve, in the case of **Chroma** from 5.95 to 19.22. The overall speedups range from 2.18 to 19.22. From these results, we can see that cache optimizations are even more valuable when codes are parallelized. Since cache optimizations are usually applicable for multimedia codes, optimizations such as prefetching and tiling should be used in conjunction with parallelization. In the next three sections, we present

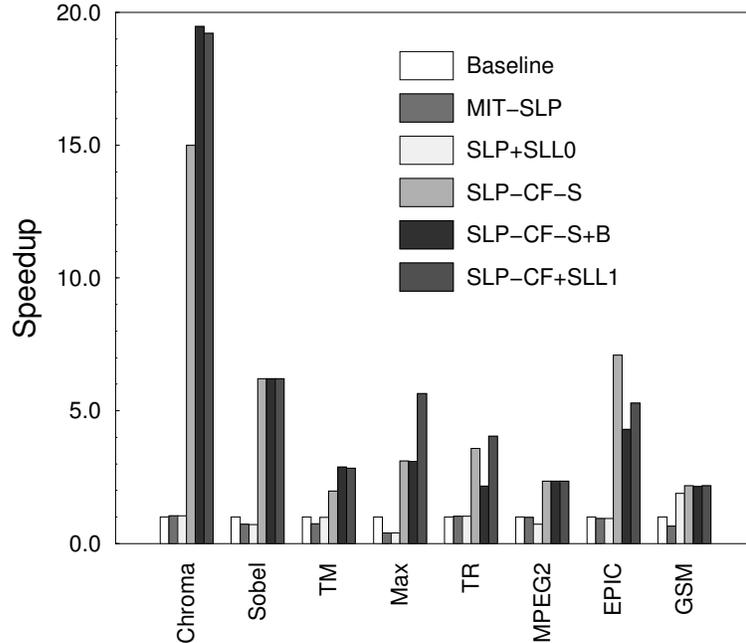


Figure 6.4: Overall speedup breakdown (small data).

the isolated benefits of exploiting prepacking, SLP in the presence of control flow, and superword-level locality respectively.

6.5 Packing for Low Parallelization Overhead

Section 5.4 describes a technique called *prepacking* that leads to better packing decisions in terms of overall parallelization overhead. For prepacking, isomorphic data dependence graphs are packed instead of isomorphic instructions. To evaluate the effects of prepacking, Figure 6.5 compares the performance of three versions. **Baseline** and **SLP-CF+SLL₁** are the same as in Figure 6.3. **NO-PREPACK** represents the version compiled without prepacking. When prepacking is not in use, the original packing algorithm is used [39]. For nine out of 14 benchmarks, the two packing algorithms result in roughly the same performance. For the other five benchmarks, however, prepacking achieves improvements. For both **NO-PREPACK** and **SLP-CF+SLL₁**, the main loop body of FIR is almost completely

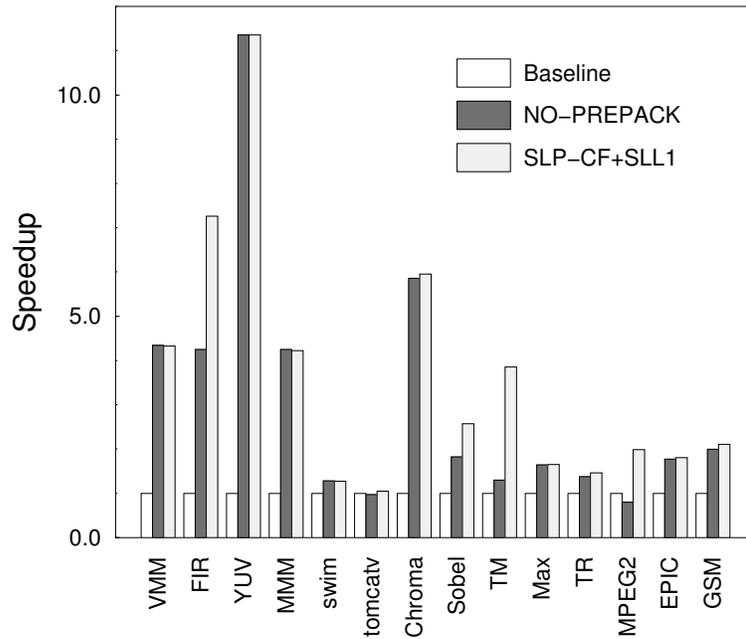


Figure 6.5: Effect of prepacking.

parallelized. However, the number of C statements in the parallelized main loop body has decreased from 308 in NO-PREPACK to 182 in SLP-CF+SLL₁ because the instructions necessary to shuffle data elements are reduced. Similarly, the number of superword instructions has shrunk in Sobel and GSM. For TM, the main loop of NO-PREPACK has 28 independent BOSCC regions, each of which containing two to four superword instructions to by pass. For SLP-CF+SLL₁, it has only four BOSCC regions containing from 16 to 21 superword instructions. By packing data dependence graphs rather than individual instructions, large number of instructions are packed at once resulting in more instructions guarded by each superword predicate. In this case, better packing decisions contribute to not only low parallelization overhead but also bigger BOSCC regions for each superword predicate making the BOSCCs more beneficial. MPEG2 is of special interest because it is parallelized only when prepacking is used. This is because all memory references

are unaligned in `MPEG2`. Since the original SLP algorithm packs only aligned memory references, prepacking is essential in parallelizing `MPEG2`.

6.6 SLP in the Presence of Control Flow

To evaluate the benefits of supporting SLP in the presence of control flow, this section focuses on the eight benchmarks containing at least one conditional statement in a loop body parallelized by the compiler in Figure 6.3 and Figure 6.4. For each of the benchmarks, we compare the speedups of two versions, `SLP+SLL0` and `SLP-CF-S`, using two different data set sizes.

For the large data set sizes of Figure 6.3, the speedups achieved by `SLP-CF-S` range from 1.25 to 2.59 for the eight benchmarks over `Baseline`, with an average of 1.78. Most benchmarks show significantly increased speedups for the smaller input sizes, ranging from 1.97 to 15, with an average of 5.18. These results suggest that exploiting cache optimizations and SLP in the presence of control flow together may result in much better performance for large data sets.

The `SLP-CF-S` versions of `Chroma`, `Sobel`, and `EPIC-unquantize` effectively exploit the parallelism available in these benchmarks, yielding speedups of more than 6.21. In particular, the 15 speedup on `Chroma` is because the data type size of the operands is 8 bits, which results in 16 operations on 8-bit objects per superword operation. `TM`, `Max`, `TR`, `MPEG2-dist1` and `GSM-Calculation` show more modest speedups. `MPEG2-dist1`, `TM` and `GSM-Calculation` have a reduction. In `MPEG2-dist1`, the initialization and finalization of the reduction remain inside the loop body since the reduction variable is used as the test for loop exit. `Sobel` and `TM` show a performance loss due to unaligned memory accesses. We also observe that for the provided input data set size, `TM` has a very low number of true values for the branch parallelized by `SLP-CF-S`. While in sequential execution the code would branch around the core computation, in `SLP-CF-S` it must perform the

computation on every iteration and merge with prior results using a *select* operation. This additional computation over sequential execution reduces the benefits of parallelization. The computation in `GSM-Calculation` is not fully parallelized due to a scalar dependence, but a set of statements between the control flow constructs, representing a loop that was manually unrolled, is parallelized by both `SLP+SLL0` and `SLP-CF-S`. Even though the code within the control flow construct is not parallelized, the use of predication allowed our compiler to exploit parallelism across what would have been multiple basic blocks, resulting in a slightly higher speedup for `SLP-CF-S`.

The `SLP-CF-S` approach presented in this section has demonstrated fairly significant speedups on eight multimedia benchmarks for which the SLP compiler was unable to exploit parallelism. The performance gain for superword-level parallelization in the presence of control flow depends on a number of factors, related to both the underlying architecture and the input data set. The AltiVec ISA does not support a full set of general operations for all possible types. As examples, 32-bit integer multiplication, unpacking unsigned integers and division are not directly supported in the ISA, requiring additional instructions. For 16-bit multiplies, `vec_mule` and `vec_mulo` multiply even or odd numbered elements respectively in superword registers, producing two superwords to promote the results to 32 bits. These even and odd multiplications shuffle the data elements breaking the spatial adjacency of data elements, requiring additional instructions to reorganize the results. Bitwise selection causes another problem in conjunction with the inconsistency of scalar boolean values and superword boolean values. In some cases, the SLP compiler may pack scalar boolean variables into a superword. Since the result of a scalar comparison is either 0 or 1 instead of a vector of all 0s or all 1s, the superword `select` can be incorrect if scalar boolean variables are packed into a superword and used in `selects`.

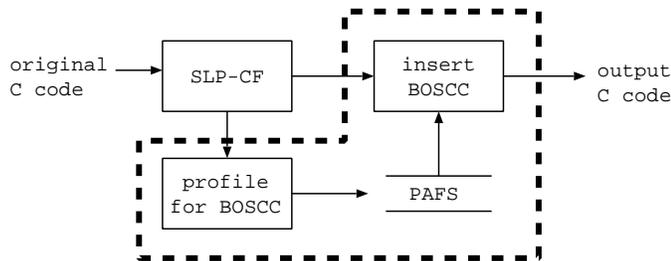


Figure 6.6: An SLP-based compiler that supports BOSCC.

As discussed in [62], different instruction set features supporting conditionals impact performance. In the AltiVec, the general mechanism of *select* operations requires executing instructions along all control flow paths and merging the results. When compared to sequential execution, where branches around code constructs may reduce the operation count, there is a tradeoff between parallelism and code with fewer branches versus less overall computation. In examples such as TM where the number of branches taken is large, this can limit performance improvement. To reduce parallelization overhead in such cases, we can bypass parallel codes using a special instruction, described in the next subsection.

6.6.1 Branch-On-Superword-Condition-Code (BOSCC)

We use BOSCC instructions to reduce parallelization overhead in the presence of control flow as described in Section 3.4. In this subsection, we isolate the benefits of using BOSCC instructions and investigate its characteristics. Figure 6.6 shows our implementation inside the thick dashed box, which is based on SLP-CF incorporating the SLP compiler and our control flow extension. Since our profitability model of BOSCC instructions relies on profile information, the implementation runs in two phases. In the first run, it generates instrumented code which is then compiled by an AltiVec-extended GCC and linked to a library that supports the generation of a PAFS² file. In the second run, the

²See Section 3.4.2.

| | |
|---|--|
| <pre> boscc = vec_any_ne(v4, vzero); if (boscc == 1) { v5 = vec_sub(v6, v7); v8 = vec_cts(v5, 0); v1 = vec_sel(v1, v8, v4); } boscc1 = vec_any_ne(v9, vzero); if (boscc1 == 1) { v10 = vec_cmplt(v11, v12); v2 = vec_nor(v10, v3); v13 = vec_sel(v13, v10, v9); v3 = vec_sel(v3, v2, v9); } </pre> | <pre> boscc = vec_any_ne(v4, vzero); if (boscc == 1) { v5 = vec_sub(v6, v7); v8 = vec_cts(v5, 0); v1 = vec_sel(v1, v8, v4); } v10 = vec_cmplt(v11, v12); v2 = vec_nor(v10, v3); v13 = vec_sel(v13, v10, v9); v3 = vec_sel(v3, v2, v9); </pre> |
| (a) BOSCC-N | (b) BOSCC-M |

Figure 6.7: Example: BOSCCs generated in EPIC.

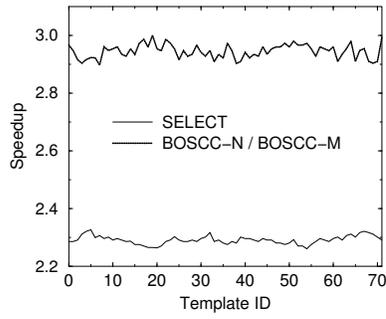
predicates in the source code are annotated with PAFS values produced in the profiling run. Based on the PAFS values, our BOSCC model determines the profitability of each BOSCC instruction.

Figure 6.8 shows speedup curves for the eight benchmarks with control flow in Table 6.1. Each graph shows the speedups of three parallel versions of a benchmark, **SELECT**, **BOSCC-N** and **BOSCC-M**, with respect to the sequential version of the benchmark. **SELECT** is the same as **SLP-CF-S** in Figure 6.2 and the **BOSCC-N** (*Naive BOSCC*) version is derived by inserting a BOSCC instruction in all possible BOSCC regions. In the **BOSCC-M** (*Model-based BOSCC*) version, the model described in Section 3.4.2 is used to evaluate the profitability of inserting BOSCC instructions. Figure 6.7 shows an example code taken from the parallelized EPIC code. The code segment contains two BOSCC regions of consecutive instructions shown in bold; three instructions in the first region and four instructions in the second. One BOSCC is generated for each of the two superword predicate **v4** and **v9** in **BOSCC-N** shown in Figure 6.7(a) whereas in **BOSCC-M** shown in Figure 6.7(b), the second BOSCC is not generated. While **BOSCC-N** has generated BOSCC

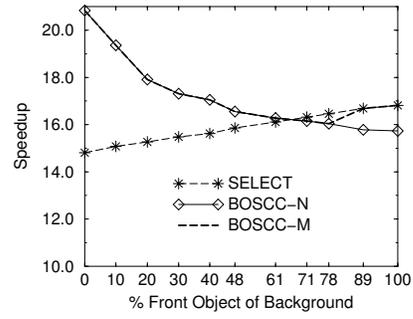
instructions without considering PAFS values, **BOSCC-M** has generated a **BOSCC** instruction only for **v4** in this example because the PAFS values for **v4** and **v9** are 82 % and 0 % respectively.

Figure 6.8(a) shows the speedups of **TM** for each of the 72 templates of the kernel's input data set, for versions **SELECT**, **BOSCC-N** and **BOSCC-M**. The speedup of **BOSCC-N** varies with the input data sets, since the true density varies from template to template. The **BOSCC-M** version also has a **BOSCC** instruction for all templates, and therefore the speedups are the same as those of **BOSCC-N**. Figure 6.9 shows that the speedup curve of the **BOSCC** versions closely matches the percentage of taken **BOSCC** branches of each template. Although not shown in the figure, the speedups of **SELECT** follow the inverse of the percentage of taken **BOSCC** branches, because the run time of the sequential baseline is affected by the PAFS while that of **SELECT** is not.

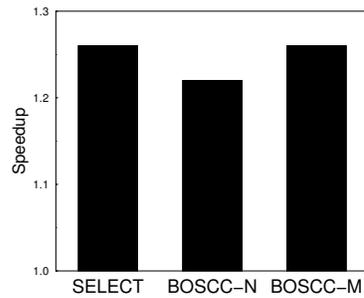
The speedups of the parallel versions of **Chroma** are shown in Figure 6.8(b). The horizontal axis corresponds to the ratio between the sizes of the foreground object and the background image in the input data set (both the size and shape of the foreground object affect the true density of the input data). Since in **Chroma** a **BOSCC** branch is taken when all pixels in a superword are outside the foreground object, the speedups corresponding to smaller foreground objects are larger, as expected. In **SELECT**, the runtime does not vary with the true densities, but there is a small speedup due to the fact that in the sequential version the body of the conditional is executed more often as the true density increases. **BOSCC-M** follows the better of the **SELECT** and **BOSCC-N** speedups for most input data sets. The few exceptions are caused by a simplification in our model, where we assume that the cost of executing a **BOSCC** instruction is the same as any other instruction. In general, branch instructions cost more than arithmetic and logical instructions as the percentage of the taken **BOSCCs** approaches 50 %. The **BOSCC** model makes the right decisions around 0 % and 100 % but it tends to make



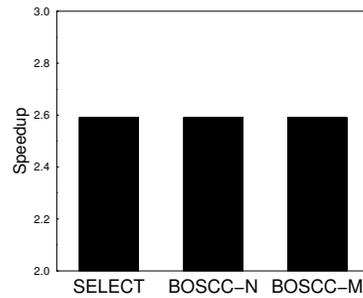
(a) TM



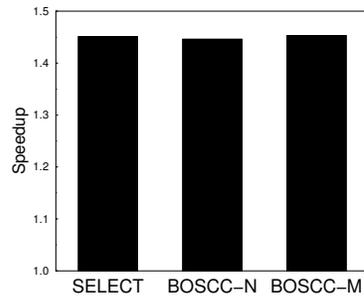
(b) Chroma



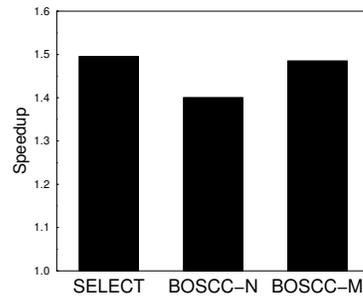
(c) Max



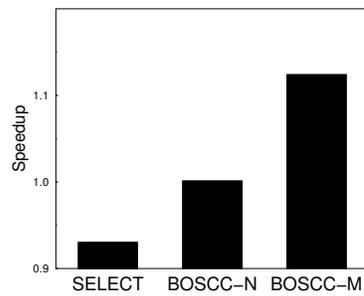
(d) Sobel



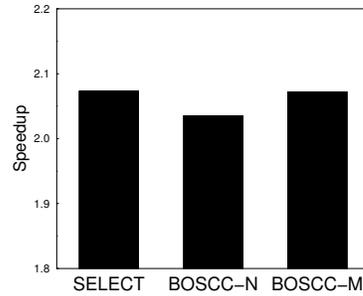
(e) TR



(f) MPEG2-dist1



(g) EPIC-unquantize



(h) GSM-Calculation

Figure 6.8: Speedups over scalar version for real data.

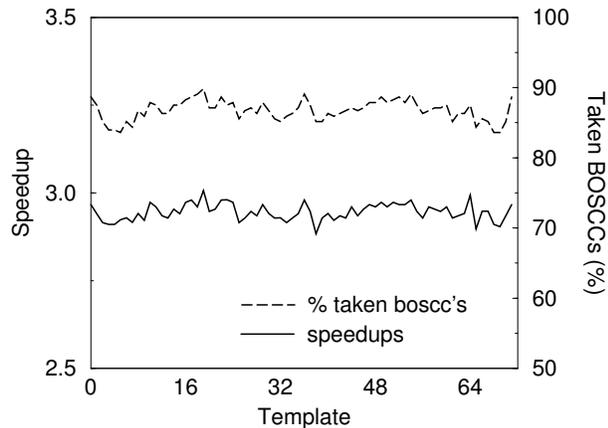


Figure 6.9: TM: % taken BOSCCs.

wrong decisions in between the two ends when the performance margin between the two versions with and without BOSCC is small.

The speedups of `Max` are 1.26 for `SELECT` and 1.22 for `BOSCC-N`, as shown in Figure 6.8(c). In `BOSCC-N`, each BOSCC body contains a single instruction.

```
max = select(max, new_value, compare);
```

We expected GCC to generate a BOSCC instruction for the region associated with the `select` instruction. However, the GCC version we use generates code such that the `select` instruction is always executed and a new `copy` instruction is added after the BOSCC, possibly because the destination variable (`max`) is live across the iterations of the innermost loop. Thus `BOSCC-N` has two extra instructions, a BOSCC instruction and an extra copy instruction, resulting in a slow down with respect to `SELECT`. When this problem is corrected manually at the assembly level by removing the copy instruction and moving the BOSCC ahead of the select instruction, the new `BOSCC-N` performs better than `SELECT`.

For the `BOSCC-N` version of `Sobel`, a BOSCC instruction is generated for four BOSCC regions containing 2, 2, 1, and 1 instructions, respectively, yielding the same performance as the `SELECT` version. The PAFS for each BOSCC region are 17 %, 4 %, 2 % and 82

% respectively. Since the PAFS values are either high (82 %) or low (17 %, 4 % and 2 %), the cost of BOSCC instructions is reduced. Also, large memory latencies play a role in this result by overlapping with the BOSCC latency. If we reduce the memory latencies by using small data set, BOSCC-N slows down by 10 % with respect to SELECT. No BOSCC instructions are generated for the BOSCC-M version. The speedups of the parallel versions with respect to the sequential baseline are 2.59 for all three versions, as shown in Figure 6.8(d).

For TR, BOSCC-N performs slightly worse than SELECT, as shown Figure 6.8(e), again because the only BOSCC region in the kernel contains a single instruction. In addition, since the BOSCC instruction is never taken, the hardware branch predictor performs well.

The BOSCC-N version of MPEG2-dist1, shown in Figure 6.8(f), has 16 BOSCC instructions, generated for 4 basic blocks. Each BOSCC region consists of two instructions, and the PAFS ranges from 30 to 40% for all BOSCCs increasing their costs. Thus the BOSCC-M version does not have BOSCC instructions.

EPIC-unquantize, shown in Figure 6.8(g) is interesting because the BOSCC-M version outperforms both SELECT and BOSCC-N. While BOSCC-N has seven BOSCC instructions, BOSCC-M has only four BOSCCs, associated to the four BOSCC regions with the highest number of instructions and PAFS. As a result, while SELECT performs worse than the baseline and BOSCC-N achieves a negligible improvement, the BOSCC-M version speeds up by 1.12.

As discussed in Section 6.6, the parallelized main loop of GSM-Calculation, shown in Figure 6.8(h), does not have any `select` instruction because no instructions guarded by conditional statements in the sequential code are parallelized. However, six BOSCC instructions are generated in BOSCC-N for another loop nest. Since the PAFS values for all BOSCC regions are less than or equal to 10 %, BOSCC-N slowed down compared to SELECT. Because of the same reason, BOSCC-M does not have any BOSCC instruction generated and the performance is the same as SELECT.

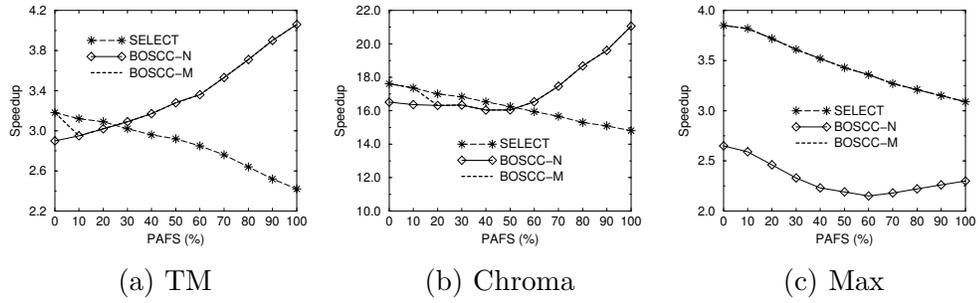


Figure 6.10: Speedups over scalar version for randomly generated data.

To further investigate how the performance of the BOSCC-M versions varies with the input data set, we used a random number generator to derive synthetic data sets with PAFS from 0% to 100% for **TM**, **Chroma** and **Max**. Figure 6.10 shows the speedups of the **SELECT**, **BOSCC-N** and **BOSCC-M** parallel versions of these three kernels with the synthetic data sets. For all three kernels, the speedup of **SELECT** decreases as the PAFS increases, because the sequential version performs better when the scalar branches are taken more often. In general, **BOSCC-N** runs increasingly faster than the sequential version as the PAFS increases. This is because the **BOSCC-N** versions skip superword instructions, each of which corresponds to SWS^3 scalar instructions. Mild slopes in the lower half of the PAFS range are due to the branch prediction mechanism of the machine. Finally, **BOSCC-M** usually performs as well as the better of the two other versions except for a small range of PAFS values, again due to our model’s simple assumption for the cost of a branch.

6.7 Superword-Level Locality

The SLL algorithm described in Chapter 4 use compiler-controlled caching in superword registers to reduce memory accesses. In Section 6.2, we described an implementation that incorporates superword-level locality optimizations into an existing compiler exploiting

³See Section 2.3.3.

superword-level parallelism [39]. Now, we describe the experimental evaluation that helps to isolate and analyze the benefits of the SLL algorithm.

Figure 6.11 shows how the reductions in memory accesses translates into speedups over MIT-SLP, which represents the original MIT SLP compiler. To isolate the benefits of individual components of our implementation, we measure the performance of the code at several stages of the optimization process. The first bar, normalized to 1, represents MIT-SLP. The second bar, called `Unroll+SLP-CF`, shows the results of running the first code transformation of the SLL algorithm, described in Section 4.2, which performs unroll-and-jam on the loop nest to expose opportunities for superword reuse, and following up with SLP. This bar isolates the impact of unrolling, since it is not until after the SLP pass that this reuse is actually exploited. Also, because it is reordering the iteration space to bring reuse closer together in time, this version also obtains locality benefits in the data cache. Thus, this bar provides the cache locality benefits of unroll-and-jam, which can be compared against the additional improvements from superword register locality. From this bar and on, we use the compiler extended with our techniques, represented by `SLP-CF-S+B` in Figure 6.2, instead of the original SLP compiler. By doing so, we can make the performance gain achieved by our extensions explicit as compared to MIT-SLP. The third bar, representing `Unroll+SLP-CF+SWR`, shows the speedups after superword replacement is additionally applied. Finally, `Unroll+SLP-CF+SWR+RP` shows the additional improvement due to packing in superword registers, described in Section 4.5.3.

Overall, we see that in combination, applications achieve speedups between 1.40 and 8.69 over the original SLP compiler alone, with an average of 3.40. As compared to the default unroll amount, the `Unroll+SLP-CF` versions achieve huge performance gains for most benchmarks by exploiting the unroll amounts determined by the SLL algorithm in addition to the code generation techniques of Chapter 5. Investigation of the low speedups in

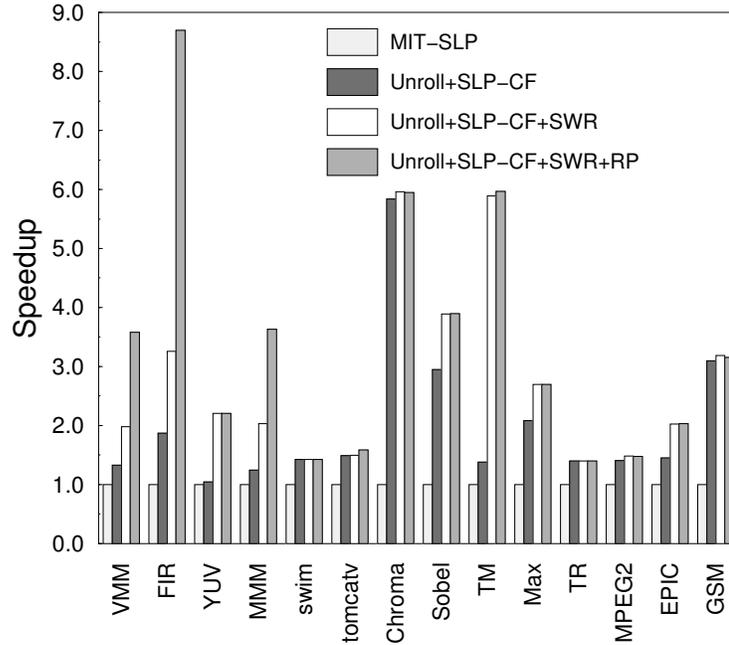


Figure 6.11: Speedups over MIT-SLP.

MMM and TM revealed that both had a severe register spilling. The register allocation algorithm in the GCC backend compiler is not optimal and tends to make worse register allocations for the bigger basic blocks. Although the number of superword registers used by the generated C codes is less than the available superword registers, the register spills occur because of the register allocation algorithm. We expect that optimal register allocators can eliminate the unnecessary register spilling [27]. When redundant memory references are removed by superword replacement for MMM and TM, register spilling also decreases achieving large speedups over MIT-SLP. For eight benchmarks, `Unroll+SLP-CF+SWR` shows significant improvements. Further speedups are achieved for three benchmarks when packing in superword registers is applied in `Unroll+SLP-CF+SWR+RP`. The other benchmarks do not have the opportunities for packing in superword registers. Consideration of `tomcatv` and `swim` shows that both programs have little temporal reuse, although there is a small amount of spatial reuse that is exploited by our approach, particularly in

`tomcatv`. We also observe additional superword-level parallelism due to index set splitting, motivated by the need to create a steady-state loop where the data is aligned to a superword boundary.

In summary, the SLL techniques presented in Chapter 4 dramatically reduce the number of memory accesses and yield significant performance improvements across these 14 programs. Thus, this section has demonstrated the value of exploiting locality in superword registers in architectures that support superword-level parallelism such as the `AltiVec`.

6.8 Summary

In this chapter, we presented the implementation of the techniques described in Chapter 3, 4, and 5. In evaluation of the implementation on 14 benchmarks, speedups ranged from 1.05 to 19.22 over the sequential input programs. To identify the factors contributing to the overall performance improvement, further experiments were performed focusing on individual techniques. Our extension to exploit SLP in the presence of control flow enabled speedups of 1.97 to 15 over the sequential input programs on 8 benchmarks. This is a dramatic improvement, considering without this extension no performance improvement was observed for 7 of 14 benchmarks. We also evaluated our BOSCC-based algorithm to reduce parallelization overhead in the presence of control flow. On three out of eight benchmarks, BOSCC instructions have been used to achieve further speedups. Moreover, the profitability model to insert BOSCC instructions closely estimates the actual profit. The implementation of the SLL algorithm is also evaluated on the 14 benchmarks. Comparing to the original SLP compiler, our implementation achieves speedups from 1.40 to 8.69 removing a majority of memory references.

Chapter 7

DIVA AND PIM-SPECIFIC OPTIMIZATIONS

DIVA is a Processing-In-Memory (PIM) embedded DRAM device that supports superword-level parallelism (SLP). Thus the two algorithms described in Chapter 3 and Chapter 4 are also applicable to DIVA. In this chapter, we focus on DIVA and PIM-specific issues and optimizations. One such optimization is to exploit a DRAM memory characteristic, called *page-mode*, automatically. A page-mode memory access exploits a form of spatial locality, where the data item is in the same row of the memory buffer as the previous access. Memory access time is reduced because the cost of row selection is eliminated. The algorithm increases frequency of page-mode accesses by reordering data accesses, grouping together accesses to the same memory row.

The DIVA architecture is described briefly in Section 1.5.2. In the next section, we describe the *instruction set architecture* (ISA) features specific to the DIVA processor. In Section 7.2, we introduce a compiler optimization that exploits page-mode memory accesses in DIVA and present the experimental results on four data intensive kernels. This experiment is separately described from those in Chapter 6 because it is performed on a DIVA simulator instead of the PowerPC G4. In Section 7.3, we discuss code generation issues specific to DIVA ISA. In Section 7.4, we present a preliminary experimental result on a prototype DIVA system. Section 7.5 summarizes this chapter.

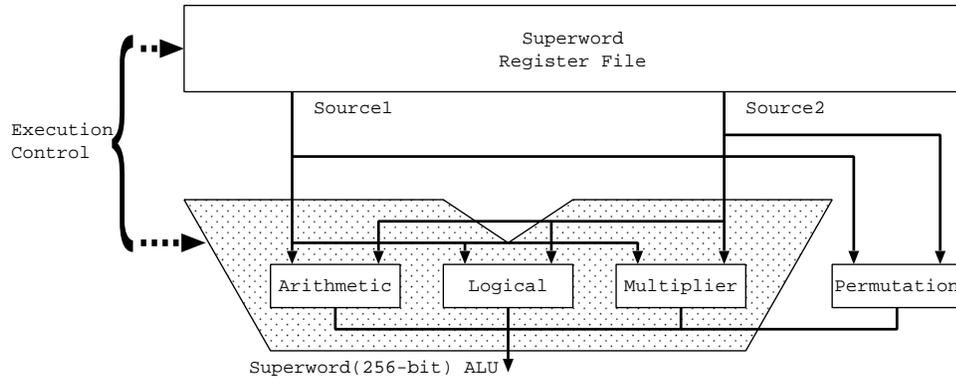


Figure 7.1: The superword data flow.

7.1 The DIVA ISA

DIVA supports a wide range of superword instructions for superword datapath in addition to ordinary scalar instructions. The intent of the superword datapath is to process objects aggregated within a row of the local memory array by operating on 256 bits in a single processor cycle. This fine-grained parallelism offers additional opportunity for exploiting the increased processor-memory bandwidth available in a PIM. The superword functional unit can perform bit-level operations, such as simple pattern matching, or higher-order computations such as searches and reduction operations.

The superword data flow is shown in Figure 7.1 and has several features to distinguish it from the other multimedia extension architectures. First is the ability to support conditional execution of instructions on sub-fields within a superword, depending on the state of local condition codes [9]. Although similar designs support some type of conditional operation, the DIVA superword functional unit provides a much richer functionality through the ability to specify conditional execution in almost every superword instruction and the use of global condition code information in selection decisions. Second, even for applications where the superword operations are not applicable, the superword datapath can be used to accelerate memory access time and communication. Contiguous data required for the scalar or floating point datapaths can be loaded into (stored from) a superword

register, and transferred directly to (from) the other register files at a small fraction of the scalar memory access latency. Third, because there is no data cache, exploiting the large capacity of the superword register file (1 KB) as described in Chapter 4 is even more important. Finally, the superword datapath is integrated into the communication mechanism, transferring data to/from the local communication buffer; this allows entire communication packets to be read or written in only one operation.

Conditional execution, direct transfers to/from other register files (only in SSE), integration with communication, as well as the ability to access main memory at very low latency, distinguish the DIVA superword capabilities from multimedia ISA extensions such as SSE and AltiVec, as well as subword parallelism approaches such as MAX [42].

7.2 Page-Mode Memory Access

Accessing a data within a DRAM macro consists of two steps. First, the entire row containing the data is copied into the DRAM open-row buffer. Then, the desired data is accessed from the buffer. This mode of DRAM accesses requiring both row and column accesses is called *random-mode*. However, most DRAM modules support an efficient *page-mode* access, where a memory access to a location currently in the DRAM open-row buffer fetches the data directly from that buffer, eliminating the cost of fetching the row from the DRAM array. To fully exploit lower latency page-mode accesses, the user or the compiler must reorganize the computation so that accesses to a same memory row are grouped together, and there are no intervening accesses to other rows.

Exposing opportunities for grouping accesses to a same array may require transformations such as unroll-and-jam, to bring accesses issued in distinct loop iterations to the body of the transformed loop, and statement reordering, to group the memory accesses.

| | |
|---|---|
| <pre> for(i=0;i<n;i++){ for(j=0;j<m;j++){ load A[j][i] load B[i] : } } </pre> | <pre> for(i=0;i<n;i+=4){ for(j=0;j<m;j++){ load A[j][i] load A[j][i+1] load A[j][i+2] load A[j][i+3] load B[i] load B[i+1] load B[i+2] load B[i+3] : } } </pre> |
| (a) Original | (b) After unroll-and-jam and reordering |

Figure 7.2: Unroll-and-jam and reordering.

Recent research has proposed to exploit page-mode accesses through manual code transformations [51, 47, 14]. This section presents a compiler algorithm for exploiting page mode automatically.

Although the proposed compiler algorithm is applicable to other embedded DRAM systems, we describe the algorithm from the viewpoint of DIVA. In Chapter 4, we presented an algorithm for exploiting locality in superword registers. In this section, we show that with a similar approach we can also exploit spatial locality in the page of a DRAM memory array.

The remainder of this section is organized as follows. Section 7.2.1 motivates our approach using a simple example. Section 7.2.2 introduces our algorithm for exploiting page-mode memory accesses. Section 7.2.3 presents experimental results on a set of four multimedia kernels.

7.2.1 Motivation

Figure 7.2 illustrates the benefits of page-mode accesses using a simple loop nest with two array references. Assuming that the sizes of arrays *A* and *B* are larger than the DRAM’s

| Ref. | Loop j | Loop i |
|---------|--------------------------------|----------------------------|
| A[j][i] | $m * \text{RMLatency}$ | $n * m * \text{RMLatency}$ |
| B[i] | $m * \text{RMLatency}$ | $n * m * \text{RMLatency}$ |
| Total | $2 * n * m * \text{RMLatency}$ | |

(a) Original

| Ref. | Loop j | Loop i' |
|-----------|--|--------------------------------------|
| A[j][i] | $m * \text{RMLatency}$ | $\frac{n}{4} * m * \text{RMLatency}$ |
| A[j][i+1] | $m * \text{PMLatency}$ | $\frac{n}{4} * m * \text{PMLatency}$ |
| A[j][i+2] | $m * \text{PMLatency}$ | $\frac{n}{4} * m * \text{PMLatency}$ |
| A[j][i+3] | $m * \text{PMLatency}$ | $\frac{n}{4} * m * \text{PMLatency}$ |
| B[i] | $m * \text{RMLatency}$ | $\frac{n}{4} * m * \text{RMLatency}$ |
| B[i+1] | $m * \text{PMLatency}$ | $\frac{n}{4} * m * \text{PMLatency}$ |
| B[i+2] | $m * \text{PMLatency}$ | $\frac{n}{4} * m * \text{PMLatency}$ |
| B[i+3] | $m * \text{PMLatency}$ | $\frac{n}{4} * m * \text{PMLatency}$ |
| Total | $\frac{n}{2} * m * \text{RMLatency} + \frac{3n}{2} * m * \text{PMLatency}$ | |

(b) After unroll-and-jam and reordering

Table 7.1: Memory latency computation.

open-row buffer, all array references in Figure 7.2(a) are in random-mode, since reference $B[i]$ displaces the DRAM row containing $A[j][i]$ from the open-row buffer and vice-versa.

For the same number of memory accesses in this loop nest, we can increase the page-mode memory accesses by applying a series of code transformations, as shown in Figure 7.2(b). First, unroll-and-jam is used to create opportunities for page-mode accesses by moving array references from successive loop iterations of the outer loop into the body of the transformed inner loop. In the example, unroll-and-jam is used to unroll the outer i loop and fuse together the resulting inner j loop bodies. Next, accesses to the same memory page in the loop body may be grouped together by reordering the memory accesses in the transformed loop body, if the reordering does not violate data dependences. In Figure 7.2(b), where the i loop is unrolled by a factor of 4, references to the same array (A or B) in the body of the transformed loop are grouped together. This results in

| |
|--------------------------------------|
| 1. Select a loop to unroll |
| 2. Control register pressure |
| 3. Align the loop to page boundaries |
| 4. Unroll-and-jam |
| 5. Reorder memory accesses |

Figure 7.3: The page-mode memory access algorithm.

page-mode accesses for all references in the loop body, except leading references $A[j][i]$ and $B[i]$, which are in random mode.

Table 7.1 shows the total memory access cost for the code in Figures 7.2(a) and (b), if we assume that accesses are not going through cache. Assuming that random-mode latency is three times the page-mode latency as in [33], loop (a) has a total latency cost of $6 * n * m * \text{PMLatency}$, while (b) has a cost of $3 * n * m * \text{PMLatency}$, a factor of 2 difference in overall memory latency.

This example shows the potential for improving performance in embedded DRAM devices through the above code transformations. To expose opportunities for page-mode accesses by applying unroll-and-jam and memory access reordering, a compiler algorithm must: (1) determine the safety of these code transformations and select a loop for which unrolling is profitable; (2) select an unroll factor that increases page-mode accesses while not causing register spilling; and, (3) transform the code to reorder the memory accesses. In the next subsection we present our compiler algorithm for exploiting page-mode accesses, which includes these three steps.

7.2.2 The Page-Mode Memory Access Algorithm

In this subsection, we introduce a compiler algorithm for exploiting *page-mode memory accesses*. Our algorithm is applicable to loop nests with array references in the loop body, where the array subscript expressions are affine functions of the loop index variables. Only array accesses are reordered by the algorithm, since it is difficult to determine whether

two scalar accesses are on the same memory page. For presentation purposes, we make some simplifying assumptions as follows.

1. Array objects are aligned at memory page boundaries.
2. The lowest dimension sizes of array objects are multiples of a memory page size.
3. The compiler backend does not change the memory access order generated by the algorithm.

Some of these assumptions can be removed by modifying the compiler backend (1,3) or by padding array objects (2).

The algorithm presented in this subsection unrolls a single loop in a loop nest, since in practice unrolling more than one loop could create register pressure and instruction cache misses. A set of heuristics is used to select which loop to unroll and its unroll amount. These heuristics result in a fast algorithm that is effective for the benchmarks presented in Section 7.2.3.

In Chapter 4, we present an algorithm for exploiting superword-level locality (SLL) which uses unroll-and-jam to expose data reuse, and unrolls multiple loops in a nest. However, the SLL algorithm cannot be used as is to exploit page-mode memory accesses. Assuming that the SLL algorithm has been applied a priori and focusing on the goal of exploiting page-mode allow much simpler algorithm which is computationally cheaper as well.

Figure 7.3 illustrates the steps of the algorithm, which are described in the remainder of this subsection. The first step selects which loop to unroll, after determining the safety of the code transformations (unroll-and-jam and statement reordering). The second steps selects an unroll factor that increases page-mode accesses while not causing register spilling. The last three steps apply the code transformations to the loop nest.

Selecting a Loop To Unroll The first step of the algorithm selects a loop to unroll, based on the number of random-mode memory accesses of the loop nest after applying unroll-and-jam. The algorithm uses data dependence information to determine the safety of unroll-and-jam.

For each loop l in the loop nest, the algorithm computes the unroll amount X_l and its corresponding number of random-mode accesses R_l , such that R_l is the smallest number of random-mode memory accesses if l is selected to be unrolled (assuming that references to a same memory page can be grouped together). Then the algorithm compares the number of random-mode accesses of each loop in the nest and selects the loop with the smallest R_l . For each loop l , the smallest unroll amount that minimizes R_l is computed as in Equation 7.1.

$$X_l = \frac{P}{\min_{a \in A}(T(a) * C(a, l))} \quad (7.1)$$

where P is the memory page size, A is the set of array references in the loop nest which are loop-variant with l in the lowest dimension, a is an array reference in A , $T(a)$ is the type size of a and $C(a, l)$ is the coefficient of the index variable l in the lowest-dimension subscript of a .

After computing the unroll amounts, the algorithm computes the corresponding number of random-mode memory accesses R_l , with the goal of selecting the loop with smallest R_l . For each loop l , the number of random-mode accesses R_l is computed as the number of distinct pages in the *memory-page footprint* of A , $F_l(A, X_l)$ (assuming that the algorithm can group together references to a same page). In Chapter 4, we present the computation of the *superword footprint* of a set of array references in a loop nest, which consists of the number of distinct superwords accessed by the references, a function of the unroll amounts. The memory-page footprint can be computed in a similar way to that of the superword footprint.

Controlling Register Pressure After selecting a loop l to unroll, the algorithm adjusts the unroll amount of the selected loop to avoid register pressure and register spilling, which could offset the benefits of unroll-and-jam.

In Chapter 4, we presented the computation of the number of registers required to keep the data accessed by the references in the loop nest after applying transformations for increasing locality in the superword register file. Here we adopt a similar approach to compute the number of superword registers required for the given unroll factor.

The total number of registers required (TNR) to keep the data accessed in the loop nest is computed as the sum of the number of registers required for each group of uniformly generated references. If the total number of registers is larger than the number of registers available, the algorithm adjusts the unroll amount X_l , by dividing it by the ratio of TNR and the number of available registers $NREG$.

$$X_l = \left\lfloor \frac{X_l}{\left\lceil \frac{TNR}{NREG} \right\rceil} \right\rfloor \quad (7.2)$$

Since the smallest type size is used in Equation 7.1, all references that have spatial reuse carried by loop l can exploit spatial reuse fully at the memory page level.

Aligning the Loop To Page Boundaries If the starting addresses of the memory accesses in the unrolled loop body are not aligned to a page boundary, each set of memory accesses to a same array will have one additional random-mode access per iteration. In Chapter 4, we applied *index set splitting* to reduce the need for alignment operations. Here, we apply the same transformation to reduce these unnecessary random-mode accesses. To determine the split points, we use Equation 4.15 except that *superword size* (SWS) is replaced by $\frac{P}{T}$ where P is the memory-page size and T is the type size of a representative array reference.

| | |
|---|---|
| <pre> for(i=32; i<N; i+=64){ load A[i + 0] (RMA) load A[i + 32] (RMA) load A[i + 8] (RMA) load A[i + 40] (RMA) load A[i + 16] (RMA) load A[i + 48] (RMA) load A[i + 24] (RMA) load A[i + 56] (RMA) ... } </pre> | <pre> for(i=32; i<N; i+=64){ load A[i + 0] (RMA) load A[i + 8] load A[i + 16] load A[i + 24] load A[i + 32] (RMA) load A[i + 40] load A[i + 48] load A[i + 56] ... } </pre> |
| (a) Unsorted | (b) Sorted |

Figure 7.4: Sorting offset addresses.

| Parameters | Value | Unit |
|---------------------|-------|--------|
| Random-mode latency | 12 | Cycles |
| Page-mode latency | 4 | Cycles |
| Page size | 256 | Bytes |

Table 7.2: DIVA simulation parameters.

Reordering Memory Accesses Finally, the reordering step hoists loads to the top of the loop body and sinks stores to the bottom. While being hoisted / sunk, the loads / stores to the same array are grouped together and sorted by their offset addresses. When there are unaligned array references even after aligning the loop, sorting the offset addresses can reduce the number of random-mode accesses. Figure 7.4 shows an example where the page size includes 64 elements of array **A**. All eight memory accesses are in random mode before sorting. After sorting the offset addresses, only two random-mode accesses remain.

7.2.3 Experiments for the Page-Mode Memory Access Algorithm

Two prototypes of the DIVA PIM chip have been fabricated recently [21], but the complete DIVA system is not available for our experiments at the time of this writing. Therefore,

| Name | Description | Input Size |
|------|--------------------------------|-----------------------|
| VMM | Vector-matrix multiply | 64 elements |
| MMM | Matrix-matrix multiply | 64 elements |
| YUV | RGB to YUV conversion | 32K elements |
| FIR | Finite impulse response filter | 256 filter, 1K signal |

Table 7.3: Benchmark programs.

we used a cycle-accurate DIVA simulator (DSIM) [21], which is modified from RSIM [53]. Table 7.2 shows the simulation parameters for the memory system which closely match those of the IBM Cu-11 embedded DRAM macro [33]. In general, there can be multiple DRAM macros and multiple open pages in a single chip, but for our experiments we assume that only one memory page is open at any given time.

We implemented the bulk of the algorithm presented in the previous subsection, and integrated it into the Stanford SUIF compiler. The input to the modified SUIF compiler is a C program, and the output is a DIVA-extended C program which, in turn, is translated by the DIVA GCC backend.

Table 7.3 shows the four kernels used to evaluate the effectiveness of the algorithm, a subset of the kernels from Chapter 6. Figure 7.5 shows the experimental flow. The main algorithm involves selecting unroll factors, performing unroll-and-jam and memory access reordering, and is represented by the hashed rectangles in Figure 7.5.

In Chapter 4, we selected unroll factors for unroll-and-jam that maximize reuse in superword registers; here, we use the unroll factors determined by the algorithm in Section 7.2.2, which are likely to be larger than in Chapter 4. In some sense, the optimizations for page-mode memory accesses are complementary to exploiting SLP and locality in superword registers, and the page-mode optimizations are difficult to isolate in our compiler. In fact, because the SLP and SLL optimizations reduce the number of memory accesses, we will see less benefit from the page-mode optimizations than if considered in isolation.

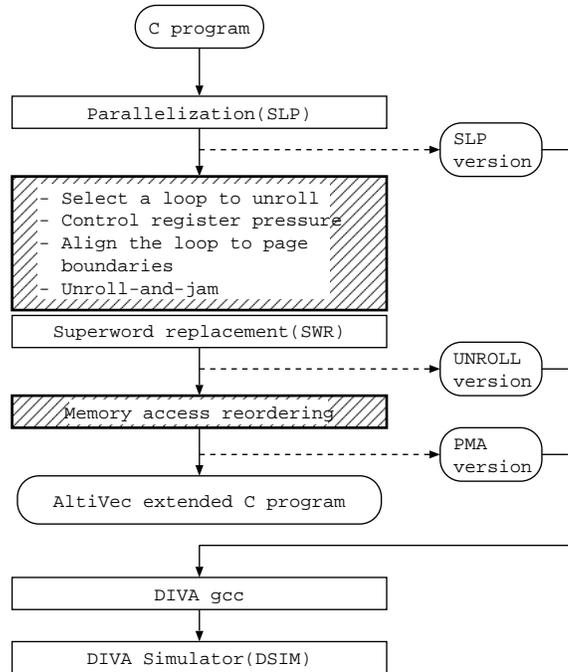


Figure 7.5: Experimental flow for page-mode memory access.

We use as our baseline the SLP version of the code with no unrolling beyond what is required to exploit parallelization of the innermost loop. The UNROLL version includes unroll-and-jam, where the loop selected by the algorithm in Section 7.2.2 is unrolled by the chosen amount, and inner loop bodies are fused together. As compared to the baseline version, this version isolates the benefits of unroll-and-jam and superword replacement in terms of reduced memory accesses and less loop overhead. The PMA version reflects the performance improvements due to memory access reordering, yielding the full benefit of the optimizations for page-mode accesses.

In these experiments, we used optimization level `-O1` for the DIVA GCC backend rather than a higher level of optimization. This was required to avoid reordering of memory accesses in subsequent optimization passes, which occurs at higher levels of optimization.

For all programs but YUV, the algorithm was able to unroll the selected loop by the unroll factor determined by Equation 7.1. For YUV, which references six distinct arrays,

```

for(i = 0; i < 64; i++)
  for(j = 0; j < 64; j++)
    for(k = 0; k < 64; k += 8){
      load C[i][j]
      load B[i][k]
      load A[j][k]
      ...
      store C[i][j]
    }

```

(a) VMM

```

for(i = 0; i < 64; i++)
  for(j = 0; j < 64; j += 8)
    for(k = 0; k < 64; k++){
      load C[i][j]
      load A[i][k]
      load B[k][j]
      ...
      store C[i][j]
    }

```

(b) MMM

Figure 7.6: SLP versions of VMM and MMM.

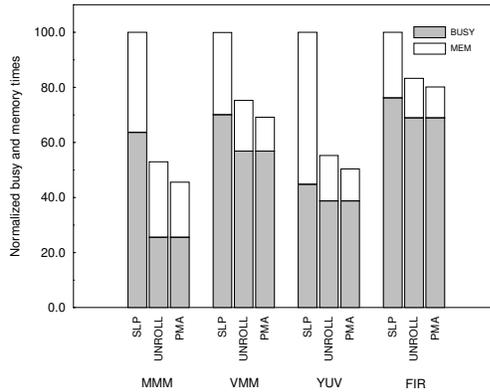


Figure 7.7: Normalized execution time.

this unroll factor was too large and resulted in register spilling. The algorithm reduced the unroll amount by half and the register spilling was eliminated.

We first consider how the optimizations for exploiting page-mode memory accesses impact memory stall time. Figure 7.7 shows the normalized execution times broken down into processor busy time and memory stall time, derived from simulation. The UNROLL version sees a significant reduction in both processor busy time (9% to 60%) and memory stall time (25% to 71%). The primary reason for this is that superword replacement has eliminated a large number of memory accesses, which not only reduces memory stall time,

but also reduce processor busy time by eliminating address calculation and instruction issue associated with the eliminated memory accesses. Further, reduction in loop control overhead also reduces processor busy time. For all programs, the PMA version further reduces memory stall time by 21% to 33%. As compared to the UNROLL version, we have not eliminated any instructions, but rather have converted random-mode accesses to page-mode accesses.

Next we consider in Figure 7.8 the percentage of all memory accesses that are in page-mode. The percentages of page-mode accesses ranges from 25% to 37% for the baseline version of the programs. We see a decrease in page-mode accesses as a percentage of memory accesses for most programs for the UNROLL version, ranging from 6% to 32%. This effect is because superword replacement has removed a large number of page-mode memory accesses, and the remainder tend to be in random mode. For example, in the VMM loop shown in Figure 7.6(a) after SLP, references to $C[i][j]$ in the k -loop are loop-invariant after unrolling, and are usually removed, but were page-mode accesses in the SLP version due to the preceding store to the same location. In MMM, the page-mode percentage actually increases for the UNROLL version, as can be seen in Figure 7.6(b). References to $A[i][k]$ are random-mode accesses, and are eliminated by superword replacement. For the PMA version, which reflects the same number of memory accesses as the UNROLL version, the percentages of page-mode accesses range from 63% to 87%.

These results show that our algorithm has been successful at increasing the percentage of page-mode accesses and reducing the memory stall time. We now see how the approach impacts the overall performance. Figure 7.9 shows the speedups for the SLP, UNROLL and PMA versions of Figure 7.5. Overall speedups as compared to the SLP baseline range from 1.25 to 2.19. Most of this speedup comes from the 1.19 to 1.89 improvement from unroll-and-jam and superword replacement, as can be seen from the UNROLL version. The speedup of the PMA version over the UNROLL version ranges from 1.04 to 1.16.

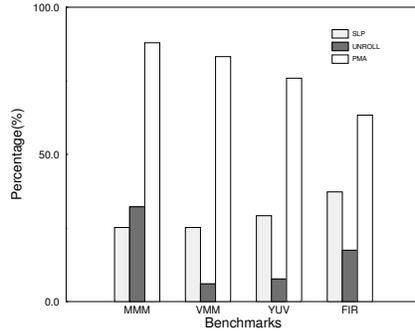


Figure 7.8: Percentage of page-mode accesses.

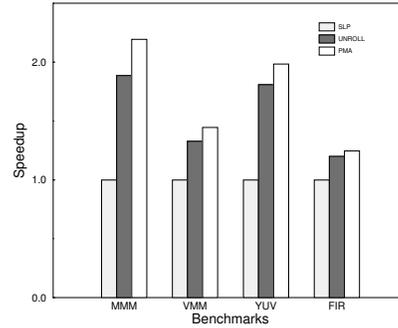


Figure 7.9: Speedup breakdown.

7.3 DIVA-Specific Code Generation

We described the DIVA ISA features in Section 7.1. In this section, we consider issues in generating code for DIVA.

Although DIVA does not support predicated execution, almost all superword instructions can be executed conditionally. In Chapter 3, we described removing superword predicates by inserting *select* instructions. The same goal can be achieved by using conditional execution. Given a predicated superword instruction, a special instruction is inserted to move the predicate to the *mask* register, which is referenced by the subsequent conditional execution. Figure 7.10 illustrates this using an example shown in (a). For comparison, we also show the code in (b), generated by the *select* algorithm described in Chapter 3. In (c), each predicated superword instruction is replaced with a sequence of two instructions, that is, one for setting the mask register and the other for conditional execution. Here, we observe an optimization opportunity where the later *mask*-setting instruction is redundant if we can recognize that the two superword predicate values are identical. The more instructions with the same predicate are collected, the more such *mask*-setting instructions can be eliminated, leading to a larger performance benefit. In

```

for(i=0; i<1024; i++){
  if(c[i] != 1){
    a[i] = c[i];
    b[i] = d[i];
  }
}

```

(a) Original

```

for(i=0; i<1024; i+=4){
  v_comp = c[i:i+3] != (1,1,1,1);
  v_pT, v_pF = v_pset(v_comp);
  a[i:i+3] = select(a[i:i+3], c[i:i+3], v_pT)
  b[i:i+3] = select(b[i:i+3], d[i:i+3], v_pT)
}

```

(b) *Select* instructions inserted

```

for(i=0; i<1024; i+=4){
  v_comp = c[i:i+3] != (1,1,1,1);
  v_pT, v_pF = v_pset(v_comp);
  move to mask register (v_pT);
  a[i:i+3] = cond_store(c[i:i+3]);
  move to mask register (v_pT);
  b[i:i+3] = cond_store(d[i:i+3]);
}

```

(c) Conditional execution

```

for(i=0; i<1024; i+=4){
  v_comp = c[i:i+3] != (1,1,1,1);
  v_pT, v_pF = v_pset(v_comp);
  move to mask register (v_pT);
  a[i:i+3] = cond_store(c[i:i+3]);
  b[i:i+3] = cond_store(d[i:i+3]);
}

```

(d) Optimized conditional execution

Figure 7.10: Code generation for conditional execution in DIVA.

Section 3.4, we described an algorithm that forms a largest region of superword instructions guarded by the same superword predicate. The similar algorithm can be used for this optimization.

As discussed in Section 7.1, the support for data transfer between different register files allows an optimization, by which scalar memory latencies are reduced further. In this optimization, we increase the number of instructions to reduce the latencies of scalar memory accesses. For example, replacing one scalar memory access with a pair of a superword memory access and a copy instruction will not be profitable whereas it may be profitable if the same optimization is applied for two scalar memory accesses. Thus, a code generation issue is to find the right number of scalar memory accesses for this optimization to be profitable.

Since Altivec supports the general permutation instruction, one field of a superword register can be moved to any field of another register. The movement of the data fields is guided by a permutation vector, that can be generated from an address dynamically as

```

float a[], b[], c[];

for (i=0; i<DATASIZE; i++)
    a[i] = b[i] + c[i];

```

Figure 7.11: StreamAdd

| Processors | Clock (MHz) | Operating System | Compiler (Optimization) |
|------------|-------------|------------------|---------------------------|
| DIVA | 140 | DIVA O/S | icc 8.0 (-O3) |
| Itanium2 | 900 | Linux | gcc 2.95.3 for DIVA (-O2) |

Table 7.4: Experimental environments.

discussed in Chapter 5. DIVA allows accessing pre-arranged permutation vectors using an index in a scalar register in addition to general permutation. While these permutation instructions can be used for alignment operations and parallel reduction operations, their versatility entices further exploration in code generation techniques for applications such as *matrix transpose* and *sorting*.

7.4 Preliminary Bandwidth Demonstration

The DIVA processor described in Chapter 1 is fabricated and being integrated into a complete system. Currently the second prototype of the DIVA chip is up and running in an Itanium2 server. In this section, we present a preliminary performance result demonstrating the data bandwidth of the DIVA processor.

Figure 7.11 shows a kernel, called *StreamAdd*, which is used to measure performance for this section. Since there is no data reuse in this computation, and very little computation to hide memory latency, it is a useful benchmark for stressing memory subsystem of architectures.

In this experiment, we compare the StreamAdd run times on DIVA to those on an Itanium2 processor. Table 7.4 shows the experimental settings for the two processors.

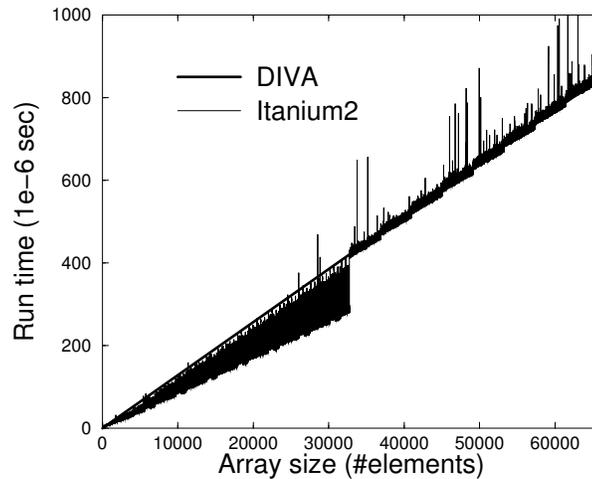


Figure 7.12: Run time of floating point StreamAdd.

The DIVA code was compiled with the DIVA compiler, which has a separate optimizing compiler and a backend compiler. Based on the algorithms in Chapter 3 and Chapter 4, the optimizing compiler parallelizes the code. The backend compiler is ported for DIVA from GCC 2.95.3 extended to support the PowerPC AltiVec.

The StreamAdd performance results are shown in Figure 7.12. The X-axis represents the data set sizes in number of array elements, and the Y-axis represents execution time in micro seconds. There are two curves in the graph. The thick straight line labeled DIVA shows deterministic performance increasing linearly as the problem size increases. This is expected because DIVA does not have data cache. The Itanium2 result shows the performance that varies as the problem size increases reflecting its more complicated memory hierarchy. It is better for smaller problem sizes, but as the problem sizes get larger, the way in which the system allocates memory leads to worse performance. Over all, in this experiment we observe that the single DIVA execution time is comparable to that of the Itanium2 execution time.

7.5 Summary

In this chapter, the issues specific to DIVA and PIM architectures are described. We presented a compiler algorithm that reduces random-mode memory accesses. In an experimental evaluation of the algorithm on a cycle-accurate DIVA simulator, we obtain speedups ranging from 1.25 to 2.19 over the parallel baseline for four multimedia kernels. In addition, we presented a preliminary experimental result demonstrating data bandwidth on a prototype DIVA system. We observe that the performance of a single DIVA processor is comparable to that of the Itanium2 processor.

Chapter 8

RELATED WORK

In this chapter, we examine previous work related to each of our approaches and distinguish our research. Previous work related to our control flow extension, superword-level locality algorithm and a DIVA-specific optimization is described in Sections 8.1, 8.2 and 8.3, respectively. In the last section, we summarize this chapter.

8.1 Exploiting SLP in the Presence of Control Flow

Some prior work has described automatic parallelization for multimedia extensions [39, 37, 64, 15, 42, 8]. Two distinct approaches are used, that is, SLP [39, 37] and an adaptation of vectorization [64, 15, 42, 8]. Extending vectorization techniques for conditionals has been addressed [8, 64], but there is no prior work describing how to parallelize conditionals using an SLP approach.

If-conversion is described in [4, 3]. Ferrante and Mace describe restoring control flow back from if-converted code [24]. However, their main focus is in generating a sequential code from parallel intermediate representations. More recently, Park and Schlansker describe an if-conversion algorithm that is optimal in terms of the number of predicates used and the number of predicate defining instructions [55], which is the algorithm we use in our compiler. Vectorizing compilers targeting multimedia extensions should have a mechanism corresponding to our *unpredicate* unless if-conversion is applied selectively

only to the statements that will be parallelized. Mahlke describes a *predicate CFG generator* which restores the original control flow from a predicated hyperblock code [44]. We use his algorithm in the *unpredicate* algorithm when an instruction cannot be inserted into an existing basic block.

Concepts similar to the *select* instruction have been described elsewhere [22, 62, 49]. Bik and et. al. used a technique called *bit masking* to combine definitions. However, their method is limited to singly nested conditional statements [8]. Chuang et. al. directly generate *phi-instructions* from the CFG of a scalar code to address *multiple-definition problem* in architectures supporting predicated execution [16]. A phi-instruction is a scalar analog of the superword *select* instruction described in Chapter 3. Their approach is related to ours in that Park and Schlansker’s algorithm is also used to derive predicates for the phi-instructions. While phi-predication could be run as a pre-pass to SLP, the code resulting from SLP would potentially contain remaining scalar predicated instructions. In an architecture such as the AltiVec, efficient code generation of the predicated scalar instructions would require an algorithm akin to the unpredicate pass described here. Using phi-predication as opposed to full predication to parallelize conditionals in the SLP compiler is a topic of future research.

Branch-on-superword-condition-code (BOSCC) is supported in the AltiVec G4 [50], DIVA [31, 21], and other architectures [7, 6]. The `movemask` instruction in Pentium can also be used for a similar purpose to BOSCC [34]. However, no prior work describes generating BOSCC instructions automatically to reduce parallelization overhead of conditionals. A vector flag population count instruction [46] can be used to change the control flow similar to BOSCC instructions in vectorized programs. However, the probability of taken BOSCCs decreases exponentially with the vector length, and the long vector length of vector machines reduces the chances for the profitability of BOSCC instructions dramatically.

8.2 Superword-Level Locality

For well over a decade, a significant body of research has been devoted to code transformations to improve cache locality, most of it targeting loop nests with regular data access patterns [25, 12, 70, 71]. Loop optimizations for improving data locality, such as tiling, interchanging and skewing, focus on reducing cache capacity misses. Of particular relevance to this thesis are approaches to tiling for cache to exploit temporal and spatial reuse; the bulk of this work examines how to select tile sizes that eliminate both capacity misses and conflict misses, tuned to the problem and cache sizes [13, 17, 23, 26, 28, 29, 38, 66, 69, 58]. The key difference between our work and that of tiling for caches is that interference is not an issue in registers. Therefore, models that consider conflict misses are not appropriate. Further, our code generation strategy must explicitly manage reuse in registers.

There has been much less attention paid to tiling and other code transformations to exploit reuse in registers, where conflict misses do not occur, but registers must be explicitly named and managed. A few approaches examine mapping array variables to scalar registers [69, 11, 45]. Most closely related to ours is the work by Carr and Kennedy, which uses scalar replacement and unroll-and-jam to exploit scalar register reuse [10]. Like our approach, in deriving the unroll factors, they use a model to count the number of registers required for a potential unrolling to avoid register pressure, and they replace array accesses, which would result in memory accesses, with accesses to temporaries that will be put in registers by the backend compiler. Their search for an unroll factor is constrained by register pressure and another metric called *balance* that matches memory access time to floating point computation time. Our approach is distinguished from all these others in that the model for register requirements must take spatial locality into account, we replace array accesses with superwords rather than scalars, and we also consider the optimizations in light of superword parallelism.

There are several recent compilation systems developed for superword-level parallelism [39, 64, 15, 19, 5]. Most, including also commercial compilers [68, 48], are based on vectorization technology [64, 19]. In contrast, Larsen and Amarasinghe devised a superword-level parallelization system for multimedia extensions [39]. None of these approaches exploit reuse in the superword register file.

8.3 DIVA-specific Optimizations

Previous research has identified the benefits of exploiting page-mode DRAM accesses [51, 47, 14, 54, 30]. Moyer modeled memory systems analytically and developed a compiler technique called *access ordering* that reorders memory accesses to better utilize the memory system [51]. McKee et al. described Stream Memory Controller (SMC) whose access ordering circuitry attempts to maximize memory system performance based on the device characteristics [47]. Their compiler is used to detect streams but access ordering and instruction issue is determined by the hardware. Chame et al. manually optimized an application for the DIVA system [14] by applying loop unrolling and memory access reordering to increase the number of page-mode accesses.

Panda et al. have developed a series of techniques to exploit page-mode DRAM access in high-level synthesis [54]. Their techniques include scalar variable clustering, memory access reordering, hoisting and loop transformations. While their ASIC design was able to exploit page-mode memory access, they do not describe an algorithm for automatic code generation. Grun et al. have optimized a set of benchmarks to better utilize efficient memory access modes for their IP library based Design Space Exploration [30]. However, their focus was on accurate timing models of the hardware system description.

Our research on exploiting page-mode memory access is distinguished from previous research as the design and implementation of a compiler algorithm to exploit page-mode automatically. Although the experiments are performed for a PIM-based system [31],

this compiler framework is applicable to embedded-DRAM systems and can also be used as a preprocessor for high-level synthesis.

8.4 Summary

This chapter described previous work related to our approaches described in this thesis. Our SIMD parallelization in the presence of control flow is distinguished by its applicability to arbitrary acyclic control flow graphs and the two optimizations to reduce parallelization overheads. The superword-level locality algorithm is the first approach that exploits superword register files as a compiler-controlled cache. Our page-mode algorithm for embedded DRAM devices is the first compiler approach that exploits page-mode memory accesses automatically.

Chapter 9

CONCLUSION

Multimedia extension architectures have been around for the last decade. Yet, compilers that automatically map sequential applications to exploit the SIMD parallelism for such architectures are relatively new. Although multimedia extensions are different from conventional vector processors in many aspects, most existing commercial / research compilers are based on the technique targeting loop-level parallelism used for conventional vector machines. More recently, a new approach that exploits *superword-level parallelism* (SLP) is suggested specifically targeting multimedia extension architectures [39]. This thesis has extended the SLP compiler approach by addressing two important open issues: how to exploit SLP in the presence of control flow and how to use superword register files as a compiler controlled cache. For DIVA, which is a processing-in-memory architecture, we have described a DIVA-specific optimization that exploits a faster DRAM access mode, called *page-mode*, automatically. In the next section, we describe our contributions by summarizing each technique and in Section 9.2, we describe our future work.

9.1 Contributions

This thesis makes the following contributions.

9.1.1 SLP in the Presence of Control Flow

Control flow is common in the core computation of multimedia applications. However, the SLP compiler cannot exploit parallelism across basic block boundaries. This thesis has extended the SLP compiler for exploiting SLP in the presence of control flow. A key insight is that we can use techniques related to optimizations for architectures supporting predicated execution, even for multimedia ISAs that do not provide hardware predication. We derive large basic blocks with predicated instructions to which SLP can be applied. After parallelization, the basic block can be a mix of predicated scalar and superword instructions. Since our target architectures do not support predicated execution, both superword and scalar predicates must be removed. We describe how to minimize the overheads for removing superword predicates and re-introduce efficient control flow for scalar predicated instructions. In addition, we have discussed other extensions to SLP to address common features of real multimedia codes. We have presented automatically generated performance results on 14 multimedia codes to demonstrate the power of this approach. We observe speedups ranging from 1.09 to 15.00 as compared to sequential execution.

As an optimization on the code parallelized for control flow, we also evaluate the costs and benefits of exploiting branches on the aggregate condition codes associated with the fields of a superword such as the branch-on-any instruction of the AltiVec. *Branch-on-superword-condition-codes* (BOSCC) instructions allow fast detection of aggregate conditions to bypass a parallel code segment, an optimization opportunity often found in multimedia applications such as image processing and pattern matching. Our experimental results show speedups of up to 1.40 on 8 multimedia kernels when BOSCC instructions are used as compared to the versions not using them.

9.1.2 Compiler Controlled Caching in Superword Registers

Parallelization is not as effective when bottleneck is memory accesses. Thus optimizations targeting memory hierarchy are even more important for the architectures supporting SLP. This thesis has described a compiler algorithm that exploits these superword register files as a compiler controlled cache to avoid unnecessary memory accesses. Accessing data from superword registers, versus a cache or main memory, has two advantages, i.e., removing memory access instructions and their latencies. This research is distinguished from previous work on exploiting reuse in scalar registers because it considers not only temporal but also spatial reuse. As compared to optimizations to exploit reuse in cache, the compiler must also manage replacement, and thus, explicitly name registers in the generated code. We have presented a set of results derived automatically on 14 benchmarks. Our results show speedups ranging from 1.40 to 8.69 as compared to using the original SLP compiler.

9.1.3 Implementation and Evaluation of the Proposed Techniques

The proposed algorithms to exploit both SLP in the presence of control flow and locality in superword registers have been fully implemented into a compiler by extending the original SLP compiler. Our extension also includes additional code generation techniques described in Chapter 5. We have described our implementation for a target architecture, the PowerPC AltiVec. The automatically generated parallel C programs are compiled by the backend compiler and run on the PowerPC G4. The overall speedups achieved by the compiler implementation range from 1.05 to 19.22. Since these speedups are the results of multiple techniques, we also have presented experimental results isolating the benefits of individual techniques.

9.1.4 DIVA-Specific Optimizations

Since DIVA is a new architecture, there exist new compiler optimization opportunities. This thesis has described a compiler algorithm and several optimization techniques to exploit a DRAM memory characteristic (*page-mode*) automatically. A page-mode memory access exploits a form of spatial locality, where the data item is in the same row of the memory buffer as the previous access. Thus, access time is reduced because the cost of row selection is eliminated. The algorithm increases frequency of page-mode accesses by reordering data accesses, grouping together accesses to the same memory row. We implemented this algorithm and presented speedup results for four multimedia kernels for a PIM embedded DRAM device, DIVA. The speedups achieved by exploiting page-mode memory access alone range from 1.04 to 1.16, resulting in overall speedups ranging from 1.25 to 2.19 when combined with optimizations targeting superword-level parallelism and locality as compared to SLP. These results show that there is a benefit in exploiting page-mode memory access in embedded systems, where the DRAM access time dominates the memory latency seen by the processor. Furthermore, our results show that for embedded systems with support for superword-level parallelism [65, 9, 31], optimizations for exploiting the DRAM’s page-mode accesses are complementary to optimizations for superword-level parallelism and superword-level locality. In addition, we presented a preliminary experimental result demonstrating data bandwidth on a prototype DIVA system. We observe that the performance of a single DIVA processor is comparable to that of the Itanium2 processor.

9.2 Future Work

In the course of this research, we encountered several open issues and future directions for this work described as follows.

Parallelization for architectures supporting SLP involves a certain overhead because of the architectural features and limitations. For example in AltiVec, superword memory accesses are required to be aligned to superword boundaries, not all operations are supported for all operand types, and data movements between register files are not directly supported. To get around these requirements and limitations, usually additional instructions are generated. We plan to expand this research by developing a cost model for parallelization so that codes are not parallelized when doing so may generate adverse effect.

Also, we plan to expand this research in the context of DIVA. Although we already have run several applications on DIVA, at the point of this writing, running applications on the DIVA system is not as easy as in commercial product systems. We expect to be able to run more applications on DIVA in the near future and compare the results with those on the AltiVec. Exploiting DIVA specific ISA features is also left as a future work.

Most of our current benchmark programs are selected from multimedia and scientific application domains. While we desire to include more applications from those two domains, we also plan to apply our techniques to the ones in other domains such as data intensive search algorithms in artificial intelligence. Traditionally, artificial intelligence applications are not considered suitable for SIMD parallelization. However, we see that their requirements for high data bandwidth and large volume of computation are well matched by the features of the DIVA processor. Currently, we are working on mapping a link discovery algorithm [1] to the DIVA processor.

Reference List

- [1] Jafar Adibi. Link discovery via a mutual information model: From graphs to ordered lists. In *DIMACS Workshop on Applications of Order Theory to Homeland Defense and Computer Security*, DIMACS Center, CoRE Building, Rutgers University, NJ, September 2004.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [3] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Annual Symposium on Principles of Programming Languages*, pages 177–189, Austin, Texas, USA, 1983.
- [4] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
- [5] Krste Asanovic and James Beck. T0 engineering data. UC Berkeley CS technical report UCB/CSD-97-930.
- [6] Krste Asanovic, James Beck, Tim Callahan, Jerry Feldman, Bertrand Irissou, Brian Kingsbury, Phil Kohn, John Lazzaro, Nelson Morgan, David Stoutamire, and John Wawrzynek. CNS-1 architecture specification: A connectionist network supercomputer. Technical Report TR-93-021, International Computer Science Institute, April 1993.
- [7] Mladen Berekovic, Hans-Joachim Stolberg, and Peter Pirsch. Implementing the MPEG-4 AS profile for streaming video on a SOC multimedia processor. In *3rd Workshop on Media and Streaming Processors*, Austin, Texas, December 2001.
- [8] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, April 2002.
- [9] Jay B. Brockman, Peter M. Kogge, Vincent Freeh, Shannon K. Kuntz, and Thomas Sterling. Microservers: A new memory semantics for massively parallel computing. In *ACM International Conference on Supercomputing*, June 1999.
- [10] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1994.

- [11] Steve Carr and Ken Kennedy. Scalar replacement in the presence of conditional control flow. *Software—Practice and Experience*, 24(1):51–77, 1994.
- [12] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, October 1994.
- [13] Jacqueline Chame and Sungdo Moon. A tile selection algorithm for data locality and cache interference. In *International Conference on Supercomputing*, pages 492–499, 1999.
- [14] Jacqueline Chame, Jaewook Shin, and Mary Hall. Compiler transformations for exploiting bandwidth in PIM-based systems. In *The 27th Annual International Symposium on Computer Architecture, Workshop on Solving the Memory Wall Problem*, June 11, 2000, Vancouver, British Columbia, Canada.
- [15] Gerald Cheong and Monica S. Lam. An optimizer for multimedia instruction sets. In *The Second SUIF Compiler Workshop*, Stanford University, USA, August 1997.
- [16] Weihaw Chuang, Brad Calder, and Jeanne Ferrante. Phi-predication for light-weight if-conversion. In *International Symposium on Code Generation and Optimization*, pages 179–190, San Francisco, California, 2003.
- [17] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *The SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
- [18] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. McGraw Hill, 2nd edition, 1990.
- [19] Derek J. DeVries. A vectorizing suif compiler: Implementation and performance. Master’s thesis, University of Toronto, 1997.
- [20] Keith Diefendorff and Pradeep K. Dubey. How multimedia workloads will change processor design. *Computer*, 30(9):43–45, 1997.
- [21] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steel, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the 16th ACM International Conference on Supercomputing*, pages 26–37, June 2002.
- [22] Jeff Draper, Jeff Sondeen, and Chang Woo Kang. Implementation of a 256-bit wideword processor for the data-intensive architecture (DIVA) processing-in-memory (PIM) chip. In *28th European Solid-State Circuits Conference*, Florence, Italy, September 2002.
- [23] Karim Esseghir. Improving data locality for caches. Master’s thesis, Dept. of Computer Science, Rice University, September 1993.

- [24] Jeanne Ferrante and Mary Mace. On linearizing parallel code. In *Annual Symposium on Principles of Programming Languages*, pages 179–190, New Orleans, Louisiana, United States, 1985.
- [25] Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash. On estimating and enhancing cache effectiveness. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 328–343, Santa Clara, California, August 1991.
- [26] Christine Fricker, Olivier Temam, and William Jalby. Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply. *ACM Transactions on Programming Languages and Systems*, 17(4):561–575, July 1995.
- [27] Changqing Fu and Kent Wilken. A faster optimal register allocator. In *ACM/IEEE international symposium on Microarchitecture*, pages 245–256, Istanbul, Turkey, November 2002.
- [28] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.
- [29] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, California, October 1998.
- [30] Peter Grun, Nikil D. Dutt, and Alexandru Nicolau. Memory aware compilation through accurate timing extraction. In *Design Automation Conference*, pages 316–321, 2000.
- [31] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoorv Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *ACM International Conference on Supercomputing*, November 1999.
- [32] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, December 1996.
- [33] IBM. *IBM Cu-11 embedded DRAM macro datasheet*, March 2002. <http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/>.
- [34] Intel. *Intel Architecture Software Developer’s Manual, Volume 2: Instruction Set Reference*, 1999. Order Number 243191.
- [35] Intel. *Intel(R) Itanium Architecture Software Developer’s Manual*, October 2002. 24531904.pdf.
- [36] Intel. *Intel(R) Itanium(R)2 Processor Reference Manual*, April 2003. 25111002.pdf.

- [37] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.
- [38] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimization of blocked algorithms. *ACM SIGPLAN Notices*, 26(4):63–74, 1991.
- [39] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC Canada, June 2000.
- [40] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Increasing and detecting memory address congruence. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2002.
- [41] Chunho Lee, Miodrag Potkonjak¹, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.
- [42] Ruby Lee. Subword parallelism with MAX2. *ACM/IEEE international symposium on Microarchitecture*, 16(4):51–59, August 1996.
- [43] Glenn Luecke and Waqar Haque. Evaluation of fortran vector compilers and preprocessors. *Software Practice and Experience*, 21(9), September 1991.
- [44] Scott A. Mahlke. *Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois, Urbana IL, September 1996.
- [45] Agustin Fernandez Marta Jimenez, Jose M. Llaberia and Enric Moranco. Index set splitting to exploit data locality at the register level. Technical Report UPC-DAC-1996-49, Universitat politecnica de Catalunya, 1996.
- [46] David Martin. Vector extensions to the MIPS-IV instruction set architecture (The V-IRAM Architecture Manual), March 2000.
- [47] Sally A. McKee, William A. Wulf, James H. Aylor, Robert H. Klenke, Maximo H. Salinas, Sung I. Hong, and Dee A. B. Weikle. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49(11):1255–1271, 2000.
- [48] Metrowerks. *CodeWarrior version 7.0 data sheet*, 2001. <http://www.metrowerks.com/pdf/mac7.pdf>.
- [49] Motorola. *AltiVec Technology Programming Environments Manual, Rev. 0.1*, November 1998. ftp://www.motorola.com/SPS/PowerPC/teksupport/teklibrary/manuals/altivec_pem.pdf.
- [50] Motorola. *AltiVec Technology Programming Interface Manual*, June 1999. <http://www.motorola.com/brdata/PDFDB/docs/ALTIVECPIM.pdf>.
- [51] Steven A. Moyer. *Access Ordering and Effective Memory Bandwidth*. PhD thesis, University of Virginia, 1993.

- [52] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 340 Pine St. Sixth Floor, San Francisco, CA 94104-3205, USA, 1997.
- [53] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. Rsim reference manual. version 1.0. Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, July 1997.
- [54] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Exploiting off-chip memory access modes in high-level synthesis. *IEEE Transactions on CAD*, February 1998.
- [55] Joseph C. H. Park and Mike Schlansker. On predicated execution, May 1991. Software and Systems Laboratory, HPL-91-58.
- [56] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel mmx for multimedia pcs. *Communications of the ACM*, 40(1):24–38, 1997.
- [57] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *International Symposium on Computer Architecture*, May 1999.
- [58] Gabriel Rivera and Chau-Wen Tseng. A comparison of compiler tiling algorithms. In *the 8th International Conference on Compiler Construction (CC'99), Amsterdam, The Netherlands*, March 1999.
- [59] Robert Sadourny. The dynamics of finite difference models of the shallow water equations. *Journal of the Atmospheric Sciences*, 32(4):680–689, 1975.
- [60] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. Superword-level parallelism in the presence of control flow. In *International Symposium on Code Generation and Optimization*, March 2005.
- [61] SIMDtech. Altiivec documents archive, 2005. <http://www.simdtech.org/altivec/documents/>.
- [62] James E. Smith, Greg Faanes, and Rabin Sugumar. Vector instruction set support for conditional operations. In *International Symposium on Computer Architecture*. ACM, 2000.
- [63] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in fpga-based systems. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.
- [64] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 2000.
- [65] Thomas Sterling. An introduction to the Gilgamesh PIM architecture. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par*, volume 2150 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 2001.

- [66] Olivier Temam, Elana D. Granston, and William Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *ACM International Conference on Supercomputing*, Portland, OR, November 1993.
- [67] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. Vis speeds new media processing. *IEEE Micro*, 16(4):10–20, August 1996.
- [68] Veridian. *VAST/Altivec Features*, June 2001. http://www.psrvc.com/altivec_feat.html.
- [69] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, 1992.
- [70] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991.
- [71] Michael J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, Reno, Nevada, November 1989.