Exploiting Superword Level Parallelism with Multimedia Instruction Sets

by

Samuel Larsen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

Author . . . .
Department of Electrical Engineering and Computer Science
May 2, 2000

Certified by . . . .
Saman Amarasinghe
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Exploiting Superword Level Parallelism with Multimedia Instruction Sets

by

Samuel Larsen

Submitted to the Department of Electrical Engineering and Computer Science
on May 2, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Increasing focus on multimedia applications has prompted the addition of multimedia extensions to most existing general-purpose microprocessors. This added functionality comes primarily with the addition of short SIMD instructions. Unfortunately, access to these instructions is limited to in-line assembly and library calls. Generally, it has been assumed that vector compilers provide the most promising means of exploiting multimedia instructions. Although vectorization technology is well understood, it is inherently complex and fragile. In addition, it is incapable of locating SIMD-style parallelism within a basic block.

In this thesis we introduce the concept of *Superword Level Parallelism (SLP)*, a novel way of viewing parallelism in multimedia and scientific applications. We believe SLP is fundamentally different from the loop level parallelism exploited by traditional vector processing, and therefore demands a new method of extracting it. We have developed a simple and robust compiler for detecting SLP that targets basic blocks rather than loop nests. As with techniques designed to extract ILP, ours is able to exploit parallelism both across loop iterations and within basic blocks. The result is an algorithm that provides excellent performance in several application domains. In our experiments, dynamic instruction counts were reduced by 46%. Speedups ranged from 1.24 to 6.70.

Thesis Supervisor: Saman Amarasinghe
Title: Assistant Professor

Acknowledgments

I want to thank my advisor, Saman Amarasinghe, for initiating this research and for getting his hands dirty with SUIF passes, LaTeX, and takeout. Radu Rugina provided his pointer analysis package and added alignment analysis at the same time he was finishing a paper of his own. Manas Mandal, Kalpesh Gala, Brian Grayson and James Yang at Motorola provided much needed AltiVec development tools and expertise. Many thanks to Matt Deeds for jumping into the SLP project and for writing the multimedia kernels used in this thesis. Finally, I want to thank all the people who read, critiqued and fixed various versions of this thesis: Krste Asanović, Michael Taylor, Derek Bruening, Mike Zhang, Darko Marinov, Matt Frank, Mark Stephenson, Sara Larsen and especially Stephanie Larsen.

This research was funded in part by NSF grant EIA9810173 and DARPA grant DBT63-96-C-0036.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The recent shift toward computation-intensive multimedia workloads has resulted in a variety of new multimedia extensions to current microprocessors [8, 12, 18, 20, 22]. Many new designs are targeted specifically at the multimedia domain [3, 9, 13]. This trend is likely to continue as it has been projected that multimedia processing will soon become the main focus of microprocessor design [10].

While different processors vary in the type and number of multimedia instructions offered, at the core of each is a set of short SIMD (Single Instruction Multiple Data) or superword operations. These instructions operate concurrently on data that are packed in a single register or memory location. In the past, such systems could accommodate only small data types of 8 or 16 bits, making them suitable for a limited set of applications. With the emergence of 128-bit superwords, new architectures are capable of performing four 32-bit operations with a single instruction. By adding floating point support as well, these extensions can now be used to perform more general-purpose computation.

It is not surprising that SIMD execution units have appeared in desktop microprocessors. Their simple control, replicated functional units, and absence of heavily-ported register files make them inherently simple and extremely amenable to scaling. As the number of available transistors increases with advances in semiconductor technology, datapaths are likely to grow even larger.

Today, use of multimedia extensions is difficult since application writers are largely restricted to using in-line assembly routines or specialized library calls. The problem is exacerbated by inconsistencies among different instruction sets. One solution to this inconvenience is to employ vectorization techniques that have been used to parallelize scientific code for vector machines [7, 16, 17]. Since a number of multimedia applications are vectorizable, this approach promises good results. However, many important multimedia applications are difficult to vectorize. Complicated loop transformation techniques such as loop fission and scalar expansion are required to parallelize loops that are only partially vectorizable [2, 5, 19]. Consequently, no commercial compiler currently implements this functionality. This thesis presents a method for extracting SIMD parallelism beyond vectorizable loops.

We believe that short SIMD operations are well suited to exploit a fundamentally different type of parallelism than the vector parallelism associated with traditional vector and SIMD supercomputers. We denote this parallelism *Superword Level Parallelism (SLP)* since it comes in the form of superwords containing packed data. Vector supercomputers require large amounts of parallelism in order to achieve speedups, whereas SLP can be profitable when parallelism is scarce. From this perspective, we have developed a general algorithm for detecting SLP that targets basic blocks rather than loop nests.

In some respects, superword level parallelism is a restricted form of ILP. ILP techniques have been very successful in the general-purpose computing arena, partly because of their ability to find parallelism within basic blocks. In the same way that loop unrolling translates loop level parallelism into ILP, vector parallelism can be transformed into SLP. This realization allows for the parallelization of vectorizable loops using the same basic block analysis. As a result, our algorithm does not require any of the complicated loop transformations typically associated with vectorization. In fact, Chapter 5 will show that vector parallelism alone can be uncovered using a simplified version of the SLP compiler algorithm presented in Chapter 4.

Chapter 2

Superword Level Parallelism

This chapter begins by elaborating on the notion of SLP and the means by which it is detected. Terminology is introduced that facilitates the discussion of our algorithms in Chapters 4 and 5. We then contrast SLP to other forms of parallelism and discuss their interactions. This helps motivate the need for a new compilation technique.

2.1 Description of Superword Level Parallelism

Superword level parallelism is defined as short SIMD parallelism in which the source and result operands of a SIMD operation are packed in a storage location. Detection is done through a short, simple analysis in which independent isomorphic statements are identified within a basic block. Isomorphic statements are those that contain the same operations in the same order. Such statements can be executed in parallel by a technique we call *statement packing*, an example of which is shown in Figure 2-1. Here, source operands in corresponding positions have been packed into registers and the addition and multiplication operators have been replaced by their SIMD counterparts. Since the result of the computation is also packed, unpacking may be required depending on how the data are used in later computations. The performance benefit of statement packing is determined by the speedup gained from parallelization minus the cost of packing and unpacking.

Depending on what operations an architecture provides to facilitate general pack-

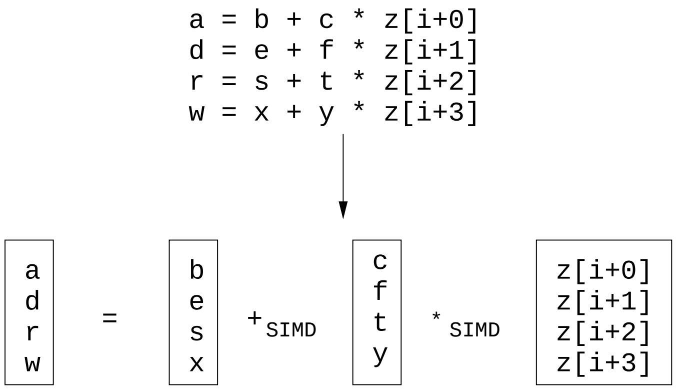

Figure 2-1: Isomorphic statements that can be packed and executed in parallel.

ing and unpacking, this technique can actually result in a performance degradation if packing and unpacking costs are high relative to ALU operations. One of the main objectives of our SLP detection technique is to minimize packing and unpacking by locating cases in which packed data produced as a result of one computation can be used directly as a source in another computation.

Packed statements that contain adjacent memory references among corresponding operands are particularly well suited for SLP execution. This is because operands are effectively pre-packed in memory and require no reshuffling within a register. In addition, an address calculation followed by a load or store need only be executed once instead of individually for each element. The combined effect can lead to a significant performance increase. This is not surprising since vector machines have been successful at exploiting the same phenomenon. In our experiments, instructions eliminated from operating on adjacent memory locations had the greatest impact on speedup. For this reason, locating adjacent memory references forms the basis of our algorithm, discussed in Chapter 4.

2.2 Vector Parallelism

To better explain the differences between superword level parallelism and vector parallelism, we present two short examples, shown in Figures 2-2 and 2-3. Although the first example can be molded into a vectorizable form, we know of no vector compilers

```
for (i=0; i<16; i++) {
  localdiff = ref[i] - curr[i];
  diff += abs(localdiff);
}
```

(a) Original loop.

```
for (i=0; i<16; i++) {
  T[i] = ref[i] - curr[i];
}

for (i=0; i<16; i++) {
  diff += abs(T[i]);
}
```

(b) After scalar expansion and loop fission.

```
for (i=0; i<16; i+=4) {
   localdiff = ref[i+0] - curr[i+0];
   diff += abs(localdiff);

   localdiff = ref[i+1] - curr[i+1];
   diff += abs(localdiff);

   localdiff = ref[i+2] - curr[i+2];
   diff += abs(localdiff);

   localdiff = ref[i+3] - curr[i+3];
   diff += abs(localdiff);
 }
```

(c) Superword level parallelism exposed after unrolling.

```
for (i=0; i<16; i+=4) {
  localdiff0 = ref[i+0] - curr[i+0];
  localdiff1 = ref[i+1] - curr[i+1];
  localdiff2 = ref[i+2] - curr[i+2];
  localdiff3 = ref[i+3] - curr[i+3];

  diff += abs(localdiff0);
  diff += abs(localdiff1);
  diff += abs(localdiff2);
  diff += abs(localdiff3);
}
```

(d) Packable statements grouped together after renaming.

Figure 2-2: Comparison of SLP and vector parallelization techniques.

that can be used to vectorize the second. Furthermore, the transformations required in the first example are unnecessarily complex and may not work in more complicated circumstances. In general, a vector compiler must employ a repertoire of tools in order to parallelize loops on a case by case basis. By comparison, our method is simple and robust, yet still capable of detecting the available parallelism.

Figure 2-2(a) presents the inner loop of the motion estimation algorithm used for MPEG encoding. Vectorization is inhibited by the presence of a loop-carried dependence and a function call within the loop body. To overcome this, a vector compiler can perform a series of transformations to mold the loop into a vectorizable form. The first is scalar expansion, which allocates a new element in a temporary array for each iteration of the loop [5]. Loop fission is then used to divide the statements into separate loops [15]. The result of these transformations is shown in Figure 2-2(b). The first loop is vectorizable, but the second must be executed sequentially.

Figure 2-2(c) shows the loop from the perspective of SLP. After unrolling, the four statements corresponding to the first statement in the original loop can be packed together. The packing process effectively moves packable statements to contiguous positions, as shown in part (d). The code motion is legal because it does not violate any dependences (once scalar renaming is performed). The first four statements in the resulting loop body can be packed and executed in parallel. Their results are then unpacked so they can be used in the sequential computation of the final statements. In the end, this method has the same effect as the transformations used for vector compilation, while only requiring loop unrolling and scalar renaming.

Figure 2-3 shows a code segment that averages the elements of two 16x16 matrices. As is the case with many multimedia kernels, our example has been hand-optimized for a sequential machine. In order to vectorize this loop, a vector compiler would need to reverse the programmer-applied optimizations. Were such methods available, they would involve constructing a *for* loop, restoring the induction variable, and re-rolling the loop. In contrast, locating SLP within the loop body is simple. Since the optimized code is amenable to SLP analysis, hand-optimization has had no detrimental effects on our ability to detect the available parallelism.

```
do {
  dst[0] = (src1[0] + src2[0]) >> 1;
  dst[1] = (src1[1] + src2[1]) >> 1;
  dst[2] = (src1[2] + src2[2]) >> 1;
  dst[3] = (src1[3] + src2[3]) >> 1;

  dst  += 4;
  src1 += 4;
  src2 += 4;
}
while (dst != end);
```

Figure 2-3: Example of an unvectorizable code sequence.

2.3 Loop Level Parallelism

Vector parallelism, exploited by vector computers, is a subset of loop level parallelism. General loop level parallelism is typically exploited by a multiprocessor or MIMD machine. In many cases, parallel loops may not yield performance gains because of fine-grain synchronization or loop-carried communication. It is therefore necessary to find coarse-grain parallel loops when compiling for MIMD machines. Traditionally, a MIMD machine is composed of multiple microprocessors. It is conceivable that loop level parallelism could be exploited orthogonally to superword level parallelism within each processor. Since coarse-grain parallelism is required to get good MIMD performance, extracting SLP should not detract from existing MIMD parallel performance.

2.4 SIMD Parallelism

SIMD parallelism came into prominence with the advent of massively parallel supercomputers such as the Illiac IV [11], and later with the Thinking Machines CM-1 and CM-2 [25, 26] and the Maspar MP-1 [4, 6]. The association of the term “SIMD” with this type of computer is what led us to use “Superword Level Parallelism” when discussing short SIMD operations.

SIMD supercomputers were implemented using thousands of small processors that worked synchronously on a single instruction stream. While the cost of massive SIMD parallel execution and near-neighbor communication was low, distribution of data to these processors was expensive. For this reason, automatic SIMD parallelization centered on solving the data distribution problem [1]. In the end, the class of applications for which SIMD compilers were successful was even more restrictive than that of vector and MIMD machines.

2.5 Instruction Level Parallelism

Superword level parallelism is closely related to ILP. In fact, SLP can be viewed as a subset of instruction level parallelism. Most processors that support SLP also support ILP in the form of superscalar execution. Because of their similarities, methods for locating SLP and ILP may extract the same information. Under circumstances where these types of parallelism completely overlap, SLP execution is preferred because it provides a less expensive and more energy efficient solution.

In practice, the majority of ILP is found in the presence of loops. Therefore, unrolling the loop multiple times may provide enough parallelism to satisfy both ILP and SLP processor utilization. In this situation, ILP performance would not noticeably degrade after SLP is extracted from a program.

Chapter 3

Optimal SLP Extraction

We initially formulated SLP extraction as a graph problem. From there, we derived a set of 0-1 integer linear programming equations that could be used to find the best set of packed statements for a given basic block. Although this technique proved intractable for real benchmarks, we gained valuable insights that helped in the discovery of the heuristic algorithm described in the next chapter.

3.1 The Graph Problem

For any statement in a basic block, there is the possibility for several different packing options. These options can be represented as nodes in a graph. Each node has an associated value that indicates the savings achieved when compared to the sequential alternative. Savings are computed from the type and number of operations within each statement, the number of statements in the packed group, and any necessary packing or unpacking costs. Packing costs will often produce nodes that are assigned negative savings. Such nodes are only profitable when considered in the context of other packed groups. This notion is captured by graph edges. Edges are drawn between two nodes whenever data produced in one node can be used in the other. A value is associated with each edge as well, indicating the packing cost recovered when communicated data are in a useful packed configuration.

An example graph is shown in Figure 3-1. Savings have been omitted since they

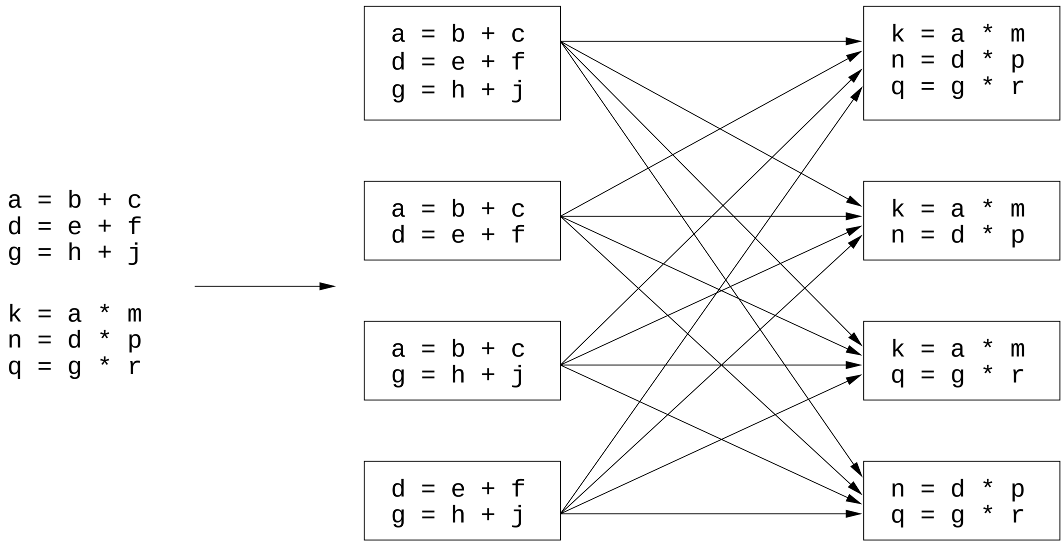

Figure 3-1: Example graph representing packing possibilities.

are architecture-dependent and not pertinent to this discussion. Also, only different statement combinations are shown. In reality, all permutations are possible. This means that a basic block with n isomorphic statements will result in a graph with $n!/(n-k)!$ nodes, where k is the number of operands that can be accommodated on a given superword datapath.

Choosing the set of nodes and edges with the greatest sum value determines the best packing configuration. If the sum is negative, the entire basic block should be left unparallelized. Since many nodes contain duplicate statements, care must be taken to ensure that savings are not counted more than once. We make the simplifying assumption that it is never profitable to execute any statement twice. Therefore, the maximization process is restricted such that only one node can be chosen from any overlapping group.

The example in Figure 3-1 attempts to relate the size and complexity of a graph constructed from even a small basic block. The problem is intensified when:

- Statements are flattened into three-address form, creating an enormous number of statements with common operations.

- Inner loops are unrolled to expose parallelism, increasing the size of basic blocks.

- Small data types allow a large number of statements in a packed group, and therefore more possible statement permutations.

Under these circumstances, the resulting graphs become unmanageable.

3.2 0-1 Integer Linear Programming Solution

Given a graph $G = \langle V, E \rangle$, as described in the previous section, the best possible savings can be calculated using 0-1 integer linear programming as follows:

For a set of nodes:

$$v_1, ..., v_n \in V, \text{ with associated savings } s_{v_1}, ..., s_{v_n} \in Int,$$

we assign a corresponding set of binary variables:

$$x_1, ..., x_n \in \{0, 1\}$$

For a set of edges:

$$e_1, ..., e_m \in E, \text{ with associated savings } s_{e_1}, ..., s_{e_m} \in Int,$$

we assign a corresponding set of binary variables:

$$y_1, ..., y_m \in \{0, 1\}$$

The objective function is then given by:

$$maximize \left(\sum_{i=1}^{n} s_{v_i} \cdot x_i + \sum_{j=1}^{m} s_{e_j} \cdot y_j \right)$$

subject to the following constraints:

$$\forall v_i, v_j \in V \text{ where } i \neq j \text{ and } v_i, v_j \text{ share a common statement, } (x_i + x_j \leq 1)$$

and

Benchmark	Terms in the objective function	Number of Constraints
swim	17,949	752,574
tomcatv	308,450	820,102
mgrid	20	60
su2cor	375,348	347,234,016
apsi	18,095	754,996
hydro2d	83	1,136
turb3d	191,420	10,996,628
applu	11	52

Table 3.1: Linear programming problem size for the most time-instensive basic block in each SPEC95fp benchmark. Input files could not be generated for `fpppp` and `wave5`.

$$\forall e_k \in E \text{ where } e_k \text{ connects } v_i \text{ and } v_j,\ (x_i + x_j - 2y_k \geq 0)$$

This maximizes the savings obtained by summing the values associated with each chosen node and edge. A node or edge is *chosen* when its corresponding binary variable has a value of 1 in the optimal solution. The first set of constraints allows only one node to be chosen from a group of overlapping nodes. The second set of constraints are needed to force the selection of two nodes when the edge between them is chosen.

3.3 Analysis

We evaluated the system described above on the SPEC95fp benchmark suite. Tests were run using the CPLEX linear programming solver running on a 4-processor Alpha 4100 Cluster with 2Gb of memory. When basic blocks were flattened into three-address form, our system was unable to generate CPLEX input files before exhausting available memory. Without flattening, input files could be generated for eight of the ten benchmarks. Table 3.1 shows input file sizes for the most time-intensive basic blocks.

Of these eight benchmarks, only `mgrid`, `hydro2d` and `applu` were solvable within 24 hours. In an attempt to produce results for the remaining benchmarks, we limited packing choices to sets of eight statements. Each statement's set was determined by its position in the original basic block. Adding this constraint forced the size of each

Benchmark	% Eliminated
swim	64.23%
tomcatv	61.06%
mgrid	22.49%
su2cor	35.91%
wave5	15.34%
apsi	19.75%
hydro2d	18.00%
turb3d	14.82%
applu	19.67%

Table 3.2: Percentage of dynamic instructions eliminated with integer linear programming methods on a hypothetical 256-bit superword datapath. It is assumed that four 64-bit floating point operations can be executed in parallel.

problem to be linearly proportional to the size of the basic block. With this restriction, we were able to generate results for every benchmark except `fpppp`. Table 3.2 lists the number of dynamic instructions eliminated from each benchmark assuming a 256-bit datapath. Results were gathered by instrumenting source code with counters in order to determine the number of times each basic block was executed. These numbers were then multiplied by the number of static instructions in each basic block.

While the SLP extraction methods presented in this chapter proved infeasible, our results allowed us to glean three high-level concepts. First, it was apparent that superword level parallelism was abundant in our benchmark set, we simply needed a viable method of extracting it. Second, statement packing appeared to be more successful when performed on three-address form since packing could be done at the level of subexpressions. Finally, we found that packed statements with adjacent memory references had the biggest potential impact on performance. As a result, the heuristic solution described in the next chapter begins by locating adjacent memory references.

Chapter 4

SLP Compiler Algorithm

This chapter describes the core algorithm developed for extracting superword level parallelism from a basic block. The algorithm can be neatly divided into four phases: *adjacent memory identification*, *PackSet extension*, *combination* and *scheduling*. *Adjacent memory identification* uncovers an initial set of packed statements with references to adjacent memory. *PackSet extension* then constructs new groups based on this initial seed. *Combination* merges all groups into sizes consistent with the superword datapath width. Finally, *scheduling* replaces groups of packed statements with new SIMD operations.

In the discussion of our algorithm, we assume a target architecture without support for unaligned memory accesses. In general, this means that merging operations must be emitted for every wide load and store. These operations combine data from two consecutive aligned segments of memory in order to simulate an unaligned memory access. Alignment analysis attempts to subvert this added cost by statically determining the address alignment of each load and store instruction. When successful, we can tailor packing decisions so that memory accesses never span an alignment boundary. Alignment analysis is described in Chapter 6. For now, we assume that each load and store instruction has been annotated with alignment information when possible.

4.1 Identifying Adjacent Memory References

Because of their obvious impact, statements containing adjacent memory references are the first candidates for packing. We therefore begin our analysis by scanning each basic block to find independent pairs of such statements. Adjacency is determined using both alignment information and array analysis.

In general, duplicate memory operations can introduce several different packing possibilities. Dependences will eliminate many of these possibilities and redundant load elimination will usually remove the rest. In practice, nearly every memory reference is directly adjacent to at most two other references. These correspond to the references that access memory on either side of the reference in question. When located, the first occurrence of each pair is added to the *PackSet*.

Definition 4.1.1 *A Pack is an n-tuple, $\langle s_1, ..., s_n \rangle$, where $s_1, ..., s_n$ are independent isomorphic statements in a basic block.*

Definition 4.1.2 *A PackSet is a set of Packs.*

In this phase of the algorithm, only groups of two statements are constructed. We refer to these as *pairs* with a *left* and *right* element.

Definition 4.1.3 *A Pair is a Pack of size two, where the first statement is considered the left element, and the second statement is considered the right element.*

As an intermediate step, statements are allowed to belong to two groups as long as they occupy a *left* position in one of the groups and a *right* position in the other. Enforcing this discipline here allows the *combination* phase to easily merge groups into larger clusters. These details are discussed in Section 4.3.

Figure 4-1(a) presents an example sequence of statements. Figure 4-1(b) shows the results of *adjacent memory identification* in which two pairs have been added to the *PackSet*. The pseudo code for this phase is shown in Figure 4-2 as find_adj_refs.

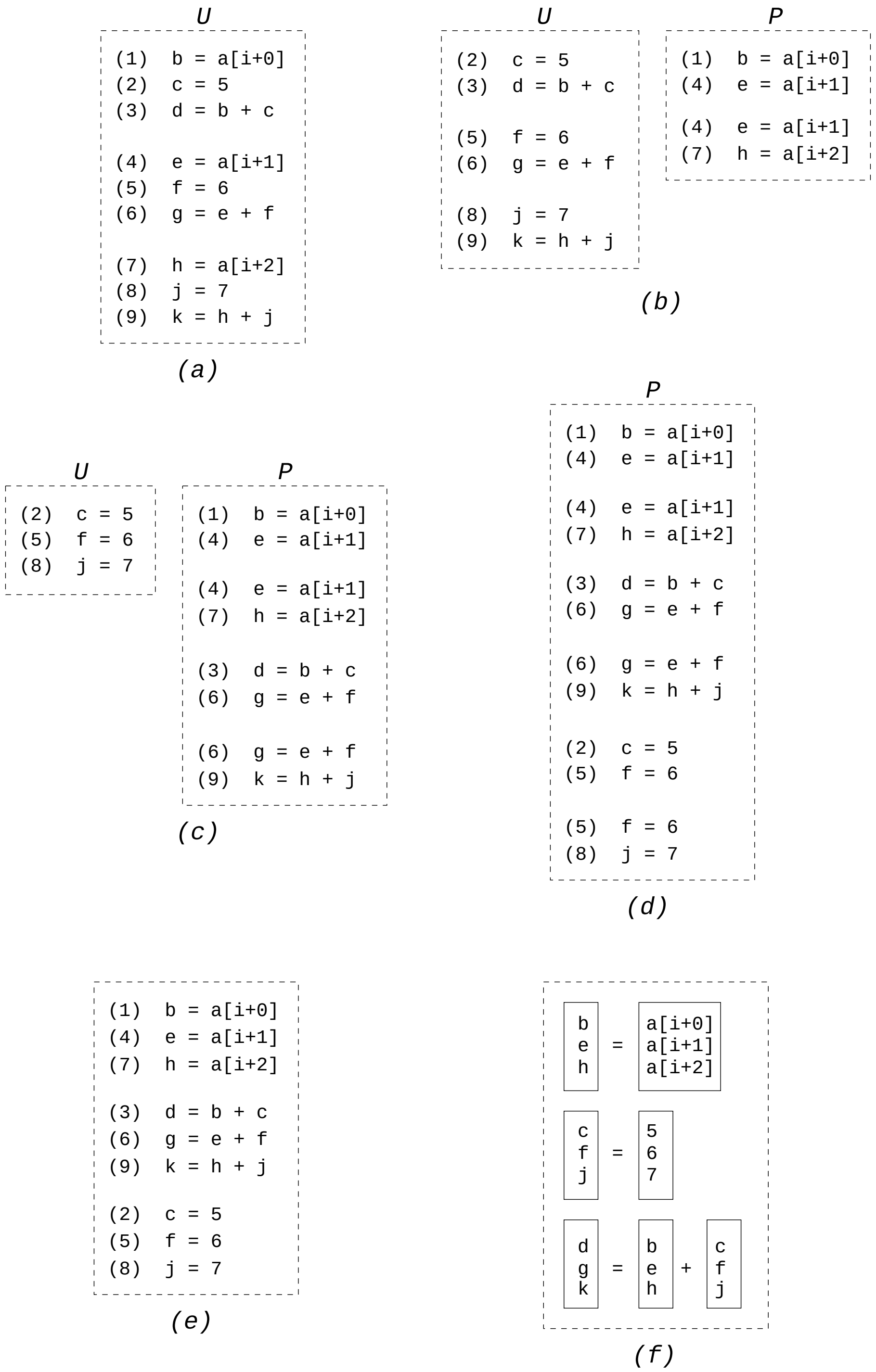

Figure 4-1: Example of SLP analysis. *U* and *P* represent the current set of unpacked and packed statements, respectively. (a) Initial sequence of instructions. (b) Statements with adjacent memory references are paired and added to the *PackSet*. (c) The *PackSet* is extended by following def-use chains of existing entries. (d) The *PackSet* is further extended by following use-def chains. (e) *Combination* merges groups containing the same expression. (f) Each group is scheduled as a new SIMD operation.

```
SLP_extract: BasicBlock B → BasicBlock
    PackSet P ← ∅
    P ← find_adj_refs(B, P)
    P ← extend_packlist(B, P)
    P ← combine_packs(P)
    return schedule(B, [ ], P)


find_adj_refs: BasicBlock B × PackSet P → PackSet
    foreach Stmt s ∈ B do
        foreach Stmt s' ∈ B where s ≠ s' do
            if has_mem_ref(s) ∧ has_mem_ref(s') then
                if adjacent(s, s') then
                    Int align ← get_alignment(s)
                    if stmts_can_pack(B, P, s, s', align) then
                        P ← P ∪ {⟨s, s'⟩}
    return P


extend_packlist: BasicBlock B × PackSet P → PackSet
    repeat
        PackSet P_prev ← P
        foreach Pack p ∈ P do
            P ← follow_use_defs(B, P, p)
            P ← follow_def_uses(B, P, p)
    until P ≡ P_prev
    return P


combine_packs: PackSet P → PackSet
    repeat
        PackSet P_prev ← P
        foreach Pack p = ⟨s_1, ..., s_n⟩ ∈ P do
            foreach Pack p' = ⟨s'_1, ..., s'_m⟩ ∈ P do
                if s_n ≡ s'_1 then
                    P ← P − {p, p'} ∪ {⟨s_1, ..., s_n, s'_2, ..., s'_m⟩}
    until P ≡ P_prev
    return P


schedule: BasicBlock B × BasicBlock B' × PackSet P → BasicBlock
    for i ← 0 to |B| do
        if ∃p = ⟨..., s_i, ...⟩ ∈ P then
            if ∀s ∈ p. deps_scheduled(s, B') then
                foreach Stmt s ∈ p do
                    B ← B − s
                    B' ← B' · s
                return schedule(B, B', P)
        else if deps_scheduled(s_i, B') then
            return schedule(B − s_i, B' · s_i, P)
    if |B| ≠ 0 then
        P ← P − {p} where p = first(B, P)
        return schedule(B, B', P)
    return B'
```

Figure 4-2: Pseudo code for the SLP extraction algorithm. Helper functions are listed in Figure 4-3

stmts_can_pack: BasicBlock B × PackSet P × Stmt s × Stmt s' × Int $align$ → Boolean
  if isomorphic(s, s') **then**
    if independent(s, s') **then**
      if $\forall\langle t, t'\rangle \in P.t \neq s$ **then**
        if $\forall\langle t, t'\rangle \in P.t' \neq s'$ **then**
          Int $align_s \leftarrow$ get_alignment(s)
          Int $align_{s'} \leftarrow$ get_alignment(s')
          if $align_s \equiv \top \vee align_s \equiv align$ **then**
            if $align_{s'} \equiv \top \vee align_{s'} \equiv align+$data_size($s'$) **then**
              return true
  return false

follow_use_defs: BasicBlock B × PackSet P × Pack p → PackSet
      where $p = \langle s, s'\rangle$, $s = [\,\mathtt{x_0} := \mathtt{f}(\mathtt{x_1}, ..., \mathtt{x_m})\,]$, $s' = [\,\mathtt{x'_0} := \mathtt{f}(\mathtt{x'_1}, ..., \mathtt{x'_m})\,]$
  Int $align \leftarrow$ get_alignment(s)
  for $j \leftarrow 1$ **to** m **do**
    if $\exists t \in B.t = [\,\mathtt{x_j} := ...\,] \wedge \exists t' \in B.t' = [\,\mathtt{x'_j} := ...\,]$ **then**
      if stmts_can_pack($B, P, t, t', align$)
        if est_savings ($\langle t, t'\rangle, P) \geq 0$ **then**
          $P \leftarrow P \cup \{\langle t, t'\rangle\}$
          set_alignment($s, s', align$)
  return P

follow_def_uses: BasicBlock B × PackSet P × Pack p → PackSet
      where $p = \langle s, s'\rangle$, $s = [\,\mathtt{x_0} := \mathtt{f}(\mathtt{x_1}, ..., \mathtt{x_m})\,]$, $s' = [\,\mathtt{x'_0} := \mathtt{f}(\mathtt{x'_1}, ..., \mathtt{x'_m})\,]$
  Int $align \leftarrow$ get_alignment(s)
  Int $savings \leftarrow -1$
  foreach Stmt $t \in B$ **where** $t = [\,... := \mathtt{g}(..., \mathtt{x_0}, ...)\,]$ **do**
    foreach Stmt $t' \in B$ **where** $t \neq t' = [\,... := \mathtt{h}(..., \mathtt{x'_0}, ...)\,]$ **do**
      if stmts_can_pack($B, P, t, t', align$) **then**
        if est_savings($\langle t, t'\rangle, P) > savings$ **then**
          $savings \leftarrow$ est_savings($\langle t, t'\rangle, P$)
          Stmt $u \leftarrow t$
          Stmt $u' \leftarrow t'$
  if $savings \geq 0$ **then**
    $P \leftarrow P \cup \{\langle u, u'\rangle\}$
    set_alignment(u, u')
  return P

Figure 4-3: Pseudo code for the SLP extraction helper functions. Only key procedures are shown. Omitted functions include: 1) has_mem_ref, which returns true if a statement accesses memory, 2) adjacent, which checks adjacency between two memory references, 3) get_alignment, which retrieves alignment information, 4) set_alignment, which sets alignment information when it is not already set, 5) deps_scheduled, which returns true when, for a given statement, all statements upon which it is dependent have been scheduled, 6) first, which returns the *PackSet* member containing the earliest unscheduled statement, 7) est_savings, which estimates the savings of a potential group, 8) isomorphic, which checks for statement isomorphism, and 9) independent, which returns true when two statements are independent.

4.2 Extending the PackSet

Once the *PackSet* has been seeded with an initial set of packed statements, more groups can be added by finding new candidates that can either:

- Produce needed source operands in packed form, or
- Use existing packed data as source operands.

This is accomplished by following def-use and use-def chains of existing *PackSet* entries. If these chains lead to fresh packable statements, a new group is created and added to the *PackSet*. For two statements to be packable, they must meet the following criteria:

- The statements are isomorphic.
- The statements are independent.
- The left statement is not already packed in a *left* position.
- The right statement is not already packed in a *right* position.
- Alignment information is consistent.
- Execution time of the new parallel operation is estimated to be less than the sequential version.

The analysis computes an estimated speedup of each potential SIMD instruction based on a cost model for each instruction added and removed. This includes any packing or unpacking that must be performed in conjunction with the new instruction. If the proper packed operand data already exist in the *PackSet*, then packing cost is set to zero.

As new groups are added to the *PackSet*, alignment information is propagated from existing groups via use-def or def-use chains. Once set, a statement's alignment determines which position it will occupy in the datapath during its computation. For

```
(1)  b = a[i+0]
(2)  c = a[i+1]

(3)  d = b - e
(4)  f = c - g

(5)  h = b - j
(6)  k = c - m
```

Figure 4-4: Multiple packing possibilites resulting from many uses of a single definition.

this reason, a statement can have only one alignment. New groups are created only if their alignment requirements are consistent with those already in place.

When definitions have multiple uses, there is the potential for many different packing possibilities. An example of this scenario is shown in Figure 4-4. Here, statements (1) and (2) would be added to the *PackSet* after *adjacent memory identification*. Following def-use chains from these two statements leads to several different packing possibilities: $\langle(3),(4)\rangle$, $\langle(5),(6)\rangle$, $\langle(3),(6)\rangle$, and $\langle(5),(4)\rangle$. When this situation arises, the cost model is used to estimate the most profitable possibilities based on what is currently packed. These groups are added to the *PackSet* in order of their estimated profitability as long as there are no conflicts with existing *PackSet* entries.

In the example of Figure 4-1, part (c) shows new groups that are added after following def-use chains of the two existing *PackSet* entries. Part (d) introduces new groups discovered by following use-def chains. The pseudo code for this phase is listed as extend_packset in Figure 4-2.

4.3 Combination

Once all profitable pairs have been chosen, they can be combined into larger groups. Two groups can be combined when the *left* statement of one is the same as the *right* statement of the other. In fact, groups must be combined in this fashion in order to prevent a statement from appearing in more than one group in the final *PackSet*. This process, provided by the combine_packs routine, checks all groups against one

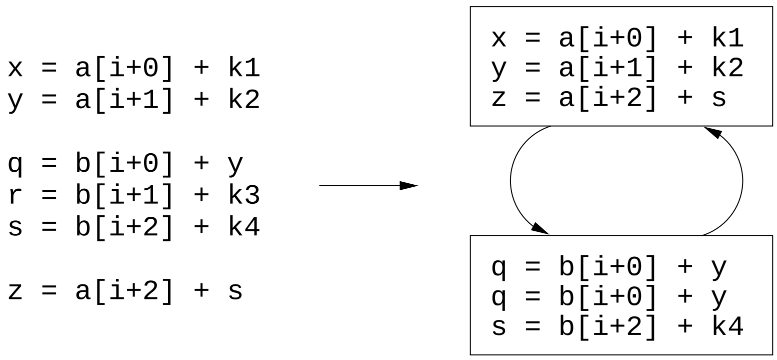

Figure 4-5: Example of a dependence between groups of packed statements.

another and repeats until all possible combinations have been made. Figure 4-1(e) shows the result of our example after combination.

Since *adjacent memory identification* uses alignment information, it will never create pairs of memory accesses that cross an alignment boundary. All packed statements are aligned based on this initial seed. As a result, *combination* will never produce a group that spans an alignment boundary. Combined groups are therefore guaranteed to be less than or equal to the superword datapath size.

4.4 Scheduling

Dependence analysis before packing ensures that statements within a group can be executed safely in parallel. However, it may be the case that executing two groups produces a dependence violation. An example of this is shown in Figure 4-5. Here, dependence edges are drawn between groups if a statement in one group is dependent on a statement in the other. As long as there are no cycles in this dependence graph, all groups can be scheduled such that no violations occur. However, a cycle indicates that the set of chosen groups is invalid and at least one group will need to be eliminated. Although experimental data has shown this case to be extremely rare, care must be taken to ensure correctness.

The scheduling phase begins by scheduling statements based on their order in the original basic block. Each statement is scheduled as soon as all statements on which it is dependent have been scheduled. For groups of packed statements, this property must be satisfied for each statement in the group. If scheduling is ever inhibited by the presence of a cycle, the group containing the earliest unscheduled statement is split apart. Scheduling continues until all statements have been scheduled.

Whenever a group of packed statements is scheduled, a new SIMD operation is emitted instead. If this new operation requires operand packing or reshuffling, the necessary operations are scheduled first. Similarly, if any statements require unpacking of their source data, the required steps are taken. Since our analysis operates at the level of basic blocks, it is assumed that all data are unpacked upon entry to the block. For this reason, all variables that are live on exit are unpacked at the end of each basic block.

Scheduling is provided by the `schedule` routine in Figure 4-2. In the example of Figure 4-1, the result of scheduling is shown in part (f). At the completion of this phase, a new basic block has been constructed wherever parallelization was successful. These blocks contain SIMD instructions in place of packed isomorphic statements. As we will show in Chapter 7, the algorithm can be used to achieve speedups on a microprocessor with multimedia extensions.

Chapter 5

A Simple Vectorizing Compiler

The SLP concepts presented in Chapter 4 lead to an elegant implementation of a vectorizing compiler. Vector parallelism is characterized by the execution of multiple iterations of an instruction using a single vector operation. This same computation can be uncovered with unrolling by limiting packing decisions to unrolled versions of the same statement. With this technique, each statement has only one possible grouping, which means that no searching is required. Instead, every statement can be packed automatically with its siblings if they are found to be independent. The profitability of each group can then be evaluated in the context of the entire set of packed data. Any groups that are deemed unprofitable can be dropped in favor of their sequential counterparts. The pseudo code for the vector extraction algorithm is shown in Figure 5-1. The schedule routine is omitted since it is identical to the one shown in Figure 4-2.

While not as aggressive as the SLP algorithm, this technique shares many of the same desirable properties. First, the analysis itself is extremely simple and robust. Second, partially vectorizable loops can be parallelized without complicated loop transformations. Most importantly, this analysis is able to achieve good results on scientific and multimedia benchmarks.

The drawback to this method is that it may not be applicable to long vector architectures. Since the unroll factor must be consistent with the vector size, unrolling may produce basic blocks that overwhelm the analysis and the code generator. As

```
vector_parallelize: BasicBlock B → BasicBlock
    PackSet P ← ∅
    P ← find_all_packs(B, P)
    P ← eliminate_unprofitable_packs(P)
    return schedule(B, [ ], P)


find_all_packs: BasicBlock B × PackSet P → PackSet
    foreach Stmt s ∈ B do
        if ∀p ∈ P.s ∉ p then
            Pack p ← [s]
            foreach Stmt s′ ∈ B where s′ ≠ s do
                if stmts_are_packable(s, s′) then
                    p ← p · s′
            if |p| > 1 then
                P ← P ∪ {p}
    return P


stmts_are_packable: Stmt s × Stmt s′ → Boolean
    if same_orig_stmt(s, s′) then
        if independent(s, s′) then
            return true
    return false


eliminate_unprofitable_packs: PackSet P → PackSet
    repeat
        PackSet P′ ← P
        foreach Pack p ∈ P do
            if est_savings(p, P) < 0 then
                P ← P − {p}
    until P ≡ P′
    return P
```

Figure 5-1: Pseudo code for the vector extraction algorithm. Procedures that are identical to those in Figures 4-2 and 4-3 are omitted. same_orig_stmt returns true if two statements are unrolled versions of the same original statement.

such, this method is mainly applicable to architectures with short vectors.

In Chapter 7, we will provide data that compare this approach to the algorithm described in Chapter 4. These results demonstrate that superword level parallelism is a superset of vector parallelism. Experiments on the SPEC95fp benchmark suite show that 20% of dynamic instruction savings are from non-vectorizable code sequences.

Chapter 6

SLP Compiler Implementation

Our compiler was built and tested within the SUIF compiler infrastructure [27]. Figure 6-1 shows the basic steps and their ordering. First, loop unrolling is used to transform vector parallelism into SLP. Next, redundant load elimination is applied in order to reduce the number of statements containing adjacent memory references. After this, all multidimensional arrays are padded in the lowest dimension. Padding improves the effectiveness of alignment analysis which attempts to determine the address alignment of each load and store instruction. Alignment analysis is needed for compiling to architectures that do not support unaligned memory accesses. As a final step before SLP extraction, the intermediate representation is transformed into a low level form and a series of standard dataflow optimizations is applied. Finally, superword level parallelization is performed and a C representation is produced for use on a macro-extended C compiler. The following sections describe each of these steps.

6.1 Loop Unrolling

Loop unrolling is performed early since it is most easily done at a high level. As discussed, it is used to transform vector parallelism into basic blocks with superword level parallelism. In order to ensure full utilization of the superword datapath in the presence of a vectorizable loop, the unroll factor must be customized to the data

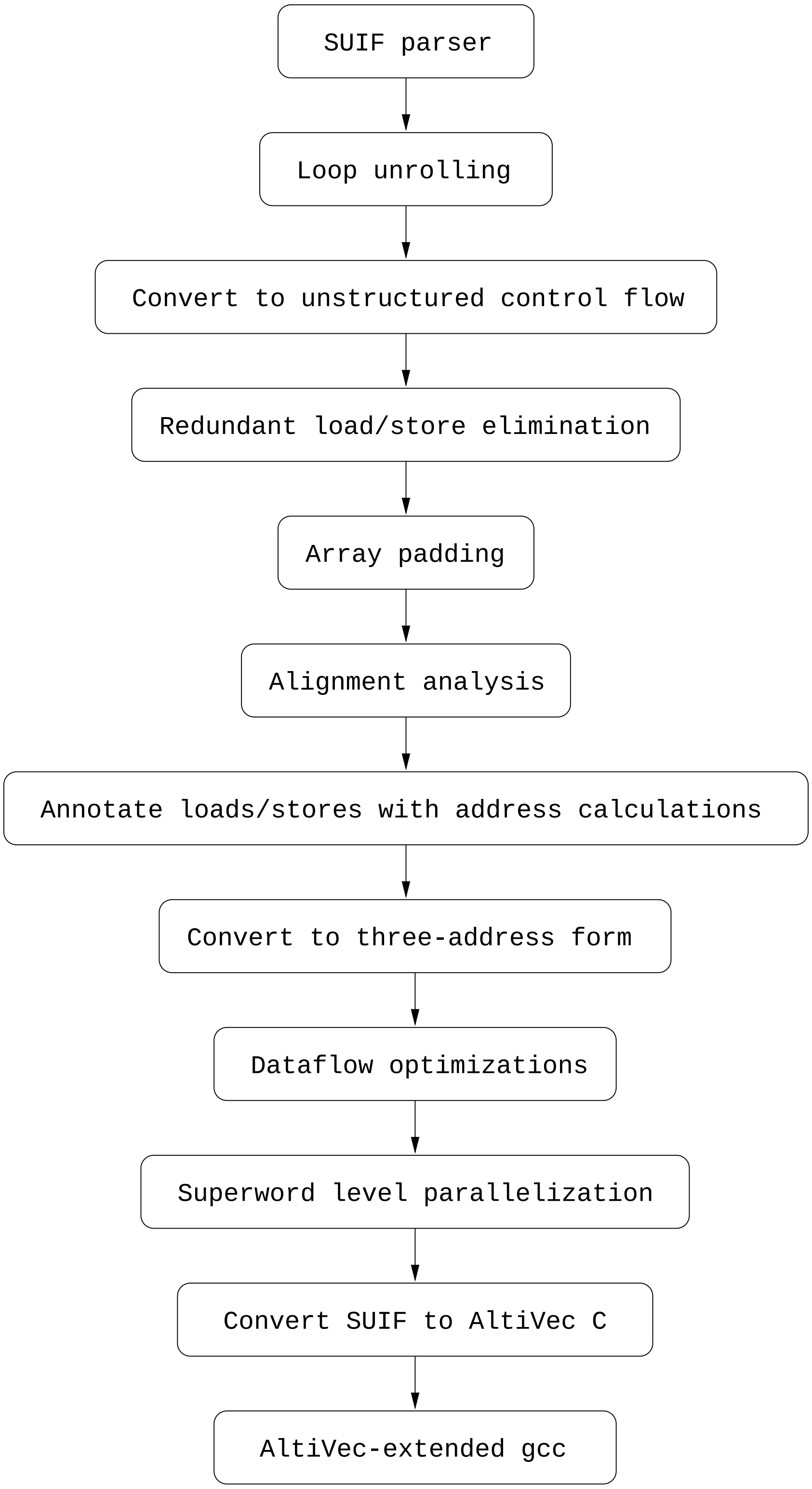

Figure 6-1: Compiler flow.

sizes used within the loop. For example, a vectorizable loop containing 16-bit values should be unrolled 8 times for a 128-bit datapath. Our system currently unrolls loops based on the smallest data type present. After unrolling, all high-level control flow is dismantled since the remaining passes operate on a standard control flow graph.

6.2 Redundant load elimination

Redundant load elimination removes unnecessary memory fetches by assigning the first in a series of redundant loads to a temporary variable. The temporary is then used in place of each subsequent redundant load. For FORTRAN sources, we limit the analysis to array references since they constitute the majority of memory references. Removing redundant loads is therefore a matter of identifying identical array accesses. This is accomplished using SUIF's built-in dependence library. For C sources, we use a form of partial redundancy elimination [14] augmented with pointer analysis [23], which allows for the elimination of partially redundant loads. In addition to being a generally useful optimization, redundant load elimination is particularly helpful in SLP analysis. As was discussed in Chapter 4, it reduces the number of packing possibilities in *adjacent memory identification*.

6.3 Array Padding

Array padding is used to improve the effectiveness of alignment analysis. Given an index into the lower order dimension of a multidimensional array, the corresponding access will be consistently aligned on the same boundary only if the lower order dimension is a multiple of the superword size. For this reason, all multidimensional arrays are padded in their lowest dimension.

6.4 Alignment Analysis

Alignment analysis determines the alignment of memory accesses with respect to a certain superword datapath width. For architectures that do not support unaligned memory accesses, alignment analysis can greatly improve the performance of our system. Without it, memory accesses are assumed to be unaligned and the proper merging code must be emitted for every wide load and store.

One situation in which merging overhead can be amortized is when a contiguous block of memory is accessed within a loop. In this situation, overhead can be reduced to one additional merge operation per load or store by using data from previous iterations.

Alignment analysis, however, can completely remove this overhead. For FORTRAN sources, a simple interprocedural analysis can determine alignment information in a single pass. This analysis is flow-insensitive, context-insensitive, and visits the call graph in breadth-first order. For C sources, we use an enhanced pointer analysis package developed by Rugina and Rinard [23]. Since this pass also provides location set information, we can consider dependences more carefully when combining packing candidates.

6.5 Flattening

SLP analysis is most useful when performed on a three-address representation. This way, the algorithm has full flexibility in choosing which operations to pack. If isomorphic statements are instead matched by the tree structure inherited from the source code, long expressions must be identical in order to parallelize. On the other hand, identifying adjacent memory references is much easier if address calculations maintain their original form. We therefore annotate each load and store instruction with this information before flattening.

6.6 Dataflow Optimizations

After flattening, several standard optimizations are applied to an input program. This ensures that parallelism is not extracted from computation that would otherwise be eliminated. Optimizations include constant propagation, copy propagation, dead code elimination, common sub-expression elimination, and loop-invariant code motion. As a final step, scalar renaming is performed to remove output and anti-dependences since they can inhibit parallelization.

6.7 Superword Level Parallelization

After optimization, the SLP algorithm is applied. When parallelization is successful, packed statements are replaced by new SIMD instructions. Ideally, we would then interface to an architecture-specific backend in order to generate machine code. However, we have opted for the simpler method of emitting C code with multimedia macros inserted for use on a macro-extended C compiler. While this solution provides less optimal results, leveraging existing compilation technology allows us to concentrate on the SLP algorithm itself rather than on architectural specifics.

Chapter 7

Results

This chapter presents potential performance gains for SLP compiler techniques and substantiates them using a Motorola MPC7400 microprocessor with the AltiVec instruction set. All results were gathered using the compiler algorithms described in Chapters 3, 4 and 5.

7.1 Benchmarks

We measure the success of our SLP algorithm on both scientific and multimedia applications. For scientific codes, we use the SPEC95fp benchmark suite. Our multimedia benchmarks are provided by the kernels listed in Table 7.1. The source code for these kernels is listed in Appendix A.

Name	Description	Datatype
FIR	Finite impulse response filter	32-bit float
IIR	Infinite impulse response filter	32-bit float
VMM	Vector-matrix multiply	32-bit float
MMM	Matrix-matrix multiply	32-bit float
YUV	RGB to YUV conversion	16-bit integer

Table 7.1: Multimedia kernels used to evaluate the effectiveness of SLP analysis.

Benchmark	128 bits	256 bits	512 bits	1024 bits
swim	61.59%	64.45%	73.44%	77.17%
tomcatv	40.91%	61.28%	69.50%	73.85%
mgrid	43.49%	55.13%	60.51%	61.52%
su2cor	33.99%	48.73%	56.06%	59.63%
wave5	26.69%	37.25%	41.97%	43.87%
apsi	24.19%	29.93%	31.32%	29.85%
hydro2d	18.53%	26.17%	28.88%	30.80%
turb3d	21.16%	24.76%	21.55%	15.13%
applu	15.54%	22.56%	10.29%	0.01%
fpppp	4.22%	8.14%	8.27%	8.27%
FIR	38.72%	45.37%	48.56%	49.84%
IIR	51.83%	60.59%	64.77%	66.45%
VMM	36.92%	43.37%	46.63%	51.90%
MMM	61.75%	73.63%	79.76%	82.86%
YUV	87.21%	93.59%	96.79%	98.36%

Table 7.2: Percentage of dynamic instructions eliminated using the SLP heuristic for a variety of hypothetical datapath widths.

7.2 SLP Availability

To evaluate the availability of superword level parallelism in our benchmarks, we calculated the percentage of dynamic instructions eliminated from a sequential program after parallelization. All instructions were counted equally, including SIMD operations. When packing was required, we assumed that *n-1* instructions were needed to pack n values into a single SIMD register. These values were also used for unpacking costs. Measurements were obtained by instrumenting source code with counters in order to determine the number of times each basic block was executed. These quantities were then multiplied by the number of static SUIF instructions in each basic block. The following subsections present results for the three extraction techniques.

7.2.1 SLP Heuristic

Performance tests using the SLP extraction algorithm of Chapter 4 were made for both sets of benchmarks. The results for a variety of hypothetical datapath widths are shown in Table 7.2 and Figure 7-1. It is assumed that each datapath can accommodate SIMD versions of any standard data type. For example, a datapath of 512 bits can perform eight 64-bit floating point operations in parallel. To uncover the maximum amount of superword level parallelism available, we compiled each benchmark without

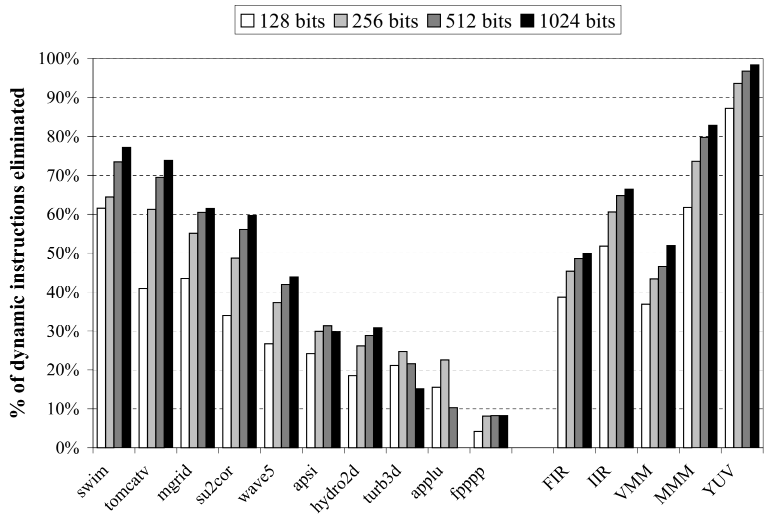

Figure 7-1: Percentage of dynamic instructions eliminated using the SLP heuristic for a variety of hypothetical datapath widths.

alignment constraints. This allowed for a maximum degree of freedom when making packing decisions.

For the multimedia benchmarks, YUV greatly outperforms the other kernels. This is because it operates on 16-bit values and is entirely vectorizable. The remaining kernels are partially vectorizable and still exhibit large performance gains.

For the SPEC95fp benchmark suite, some of the applications exhibit a performance degradation as the datapath width is increased. This is due to the large unroll factor required to fill a wide datapath. If the dynamic iteration counts for these loops are smaller than the unroll factor, the unrolled loop is never executed. For turb3d and applu, the optimal unroll factor is four. A 256-bit datapath is therefore sufficient since it can accommodate four 64-bit operations. In fpppp, the most time-intensive loop is already unrolled by a factor of three. A 192-bit datapath can support the available parallelism in this situation.

7.2.2 Heuristic vs. Linear Programming Methods

Table 7.3 compares the linear programming method of Chapter 3 to the SLP heuristic algorithm presented in Chapter 4. Interestingly, the heuristic approach performs much better than the integer linear programming methods. This is due to the shortcuts that

Benchmark	Heuristic	Linear Programming
swim	64.45%	64.23%
tomcatv	61.28%	61.06%
mgrid	55.13%	22.49%
su2cor	48.73%	35.91%
wave5	37.25%	15.34%
apsi	29.93%	19.75%
hydro2d	26.17%	18.00%
turb3d	24.76%	14.82%
applu	22.56%	19.67%
fpppp	8.14%	-

Table 7.3: Percentage of dynamic instructions eliminated using the SLP heuristic and integer linear programming methods on a 256-bit datapath.

Benchmark	SLP	Vector
swim	64.45%	62.29%
tomcatv	61.28%	56.87%
mgrid	55.13%	34.29%
su2cor	48.73%	44.20%
wave5	37.25%	28.73%
apsi	29.93%	15.89%
hydro2d	26.17%	22.91%
turb3d	24.76%	20.35%
applu	22.56%	14.67%
fpppp	8.14%	0.00%
FIR	45.37%	73.63%
IIR	60.59%	43.63%
VMM	43.37%	60.59%
MMM	73.63%	45.37%
YUV	93.59%	93.59%

Table 7.4: Percentage of dynamic instructions eliminated using SLP and vector parallelization on a 256-bit datapath.

were required to produce a solvable linear programming problem. Specifically, none of the benchmarks could be analyzed in three-address form. More importantly, most of the benchmarks were tested with a limitation on the number of packing permutations attempted. This did not allow for a complete search of all packing possibilities.

7.2.3 SLP vs. Vector Extraction

In Table 7.4 and Figure 7-2 we compare the SLP algorithm to the vectorization technique described in Chapter 5. For the multimedia benchmarks, both methods perform identically. However, there are many cases in the scientific applications for which the SLP algorithm is able to find additional packing opportunities. In fact,

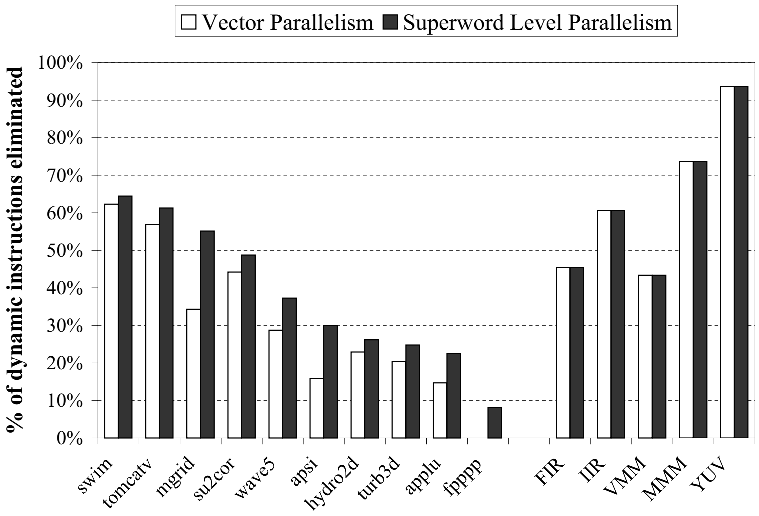

Figure 7-2: Percentage of dynamic instructions eliminated using SLP and vector parallelization on a 256-bit datapath.

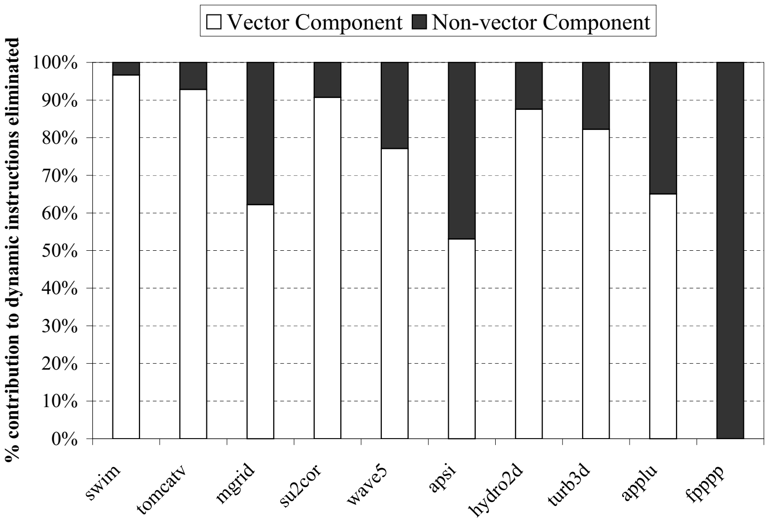

Figure 7-3: Contribution of vectorizable and non-vectorizable code sequences in total SLP savings for the SPEC95fp benchmark suite.

Benchmark	Speedup
swim	1.24
tomcatv	1.57
FIR	1.26
IIR	1.41
VMM	1.70
MMM	1.79
YUV	6.70

Table 7.5: Speedup on an MPC7400 processor using SLP compilation.

20% of the dynamic instruction savings are from non-vectorizable code sequences. In Figure 7-3, we show the available vector parallelism as a subset of the available superword level parallelism.

7.3 SLP Performance

To test the performance of our SLP algorithm in a real environment, we targeted our compilation system to the AltiVec [21] instruction set. Of the popular multimedia extensions available in commercial microprocessors, we believe AltiVec best matches the compilation technique described in this thesis. AltiVec defines 128-bit floating point and integer SIMD operations and provides a complementary set of 32 general-purpose registers. It also defines load and store instructions capable of moving a full 128 bits of data.

Our compiler automatically generates C code with AltiVec macros inserted where parallelization is successful. We then use an extended gcc compiler to generate machine code. This compiler was provided by Motorola and supports the AltiVec ABI (application binary interface). Due to the experimental nature of the AltiVec compiler extensions, it was necessary to compile all benchmarks without optimization. Base measurements were made by compiling the unparallelized version for execution on the MPC7400 superscalar unit. In both cases, the same set of SUIF optimizations and the same gcc backend were used. Since AltiVec does not support unaligned memory accesses, all benchmarks were compiled with alignment constraints in place.

Table 7.5 and Figure 7-4 present performance comparisons on a 450MHz G4 Pow-

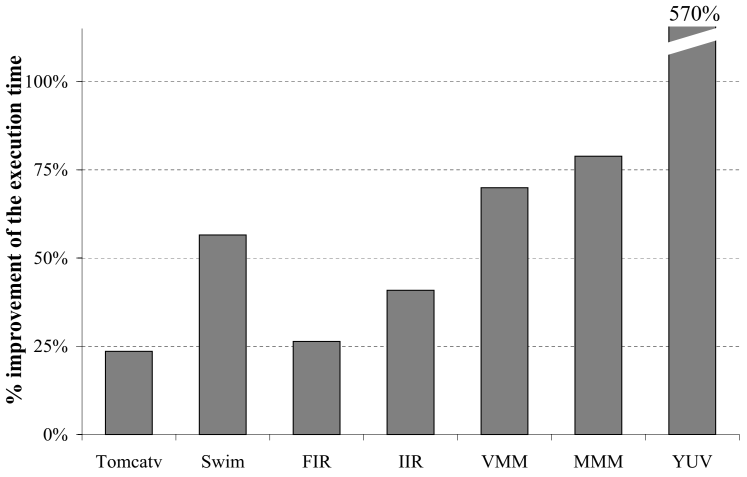

Figure 7-4: Percentage improvement of execution time on an MPC7400 processor using SLP compilation.

erMac workstation. Most of the SPEC95fp benchmarks require double precision floating point support to operate correctly. Since this is not supported by AltiVec, we were unable to compile vectorized versions for all but two of the benchmarks. `swim` utilizes single precision floating point operations, and the SPEC92fp version of `tomcatv` provides a result similar to the 64-bit version.

Our compiler currently assumes that all packed operations are executed on the AltiVec unit and all sequential operations are performed on the superscalar unit. Operations to pack and unpack data are therefore required to go through memory since AltiVec provides no instructions to move data between register files. Despite this high cost, our compiler is still able to exploit superword level parallelism and provide speedups.

Chapter 8

Architectural Support for SLP

The compiler algorithm presented in Chapter 4 was inspired by the multimedia extensions in modern processors. However, several limitations make it difficult to fully realize the potential provided by SLP analysis. We list some of these limitations below:

- Many multimedia instructions are designed for a specific high-level operation. For example, HP's MAX-2 extensions offer matrix transform instructions [18] and SUN's VIS extensions include instructions to compute pixel distances [20]. The complex CISC-like semantics of these instructions make automatic code generation difficult.

- SLP hardware is typically viewed as a multimedia engine alone and is not designed for general-purpose computation. Floating point capabilities, for example, have only recently been added to some architectures. Furthermore, even the most advanced multimedia extensions lack certain fundamental operations such as 32-bit integer multiplication and division [21].

- In current architectures, data sets are usually considered to belong exclusively to either multimedia or superscalar hardware. This design philosophy is portrayed in the lack of inter register file move operations in the AltiVec instruction set. If SLP compilation techniques can show a need for a better coupling between these two units, future architectures may provide the necessary support.

- Most current multimedia instruction sets are designed with the assumption that data are always stored in the proper packed configuration. As a result, data packing and unpacking instructions are generally not well supported. This important operation is useful to our system. With better support, SLP performance can be further increased.

- Although our system is capable of compiling for machines that do not support unaligned memory accesses, the algorithm is potentially more effective without this constraint. Architectures supplying efficient unaligned load and store instructions might improve the performance of SLP analysis.

The first three points discuss simple processor modifications that we hope will be incorporated into future multimedia instruction sets as they mature. The last two points address difficult issues. Solving them in either hardware or software is not trivial. More research is required to determine the best approach.

Chapter 9

Conclusion

In this thesis we introduced superword level parallelism, the notion of viewing parallelism from the perspective of partitioned operations on packed superwords. We showed that SLP can be exploited with a simple and robust compiler implementation that exhibits speedups ranging from 1.24 to 6.70 on a set of scientific and multimedia benchmarks.

We also showed that SLP concepts lead to an elegant implementation of a vectorizing compiler. By comparing the performance of this compiler to the more general SLP algorithm, we demonstrated that vector parallelism is a subset of superword level parallelism.

Our current compiler implementation is still in its infancy. While successful, we believe its effectiveness can be improved. By extending SLP analysis beyond basic blocks, more packing opportunities could be found. Furthermore, SLP could offer a form of predication, in which unfilled slots of a wide operation could be filled with speculative computation. If data are invalidated due to control flow, they could simply be discarded.

Recent research has shown that compiler analysis can significantly reduce the size of data types needed to store program variables [24]. Incorporating this analysis into our own has the potential of drastically improving performance by increasing the number of operands that can be packed and executed in parallel.

Today, most desktop processors are equipped with multimedia extensions. Nonuniformities in the different instruction sets, exacerbated by a lack of compiler support, has left these extensions underutilized. We have shown that SLP compilation is not only possible, but also applicable to a wider class of application domains. As such, we believe SLP compilation techniques have the potential to become an integral part of general-purpose computing in the near future.

Bibliography

[1] E. Albert, K. Knobe, J. Lukas, and G. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[2] J. R. Allen and K. Kennedy. PFC: A Program to Convert Fortran to Parallel Form. In K. Hwang, editor, *Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, Silver Spring, MD, 1984.

[3] Krste Asanović, James Beck, Bertrand Irissou, Brian E. D. Kingsbury, Nelson Morgan, and John Wawrzynek. The T0 Vector Microprocessor. In *Proceedings of Hot Chips VII*, August 1995.

[4] T. Blank. The MasPar MP-1 Architecture. In *Proceedings of the 1990 Spring COMPCON*, San Francisco, CA, February 1990.

[5] D. Callahan and P. Havlak. Scalar expansion in PFC: Modifications for Parallelization. Supercomputer Software Newsletter 5, Dept. of Computer Science, Rice University, October 1986.

[6] P. Christy. Software to Support Massively Parallel Computing on the MasPar MP-1. In *Proceedings of the 1990 Spring COMPCON*, San Francisco, CA, February 1990.

[7] Derek J. DeVries. A Vectorizing SUIF Compiler: Implementation and Performance. Master's thesis, University of Toronto, June 1997.

[8] Keith Diefendorff. Pentium III = Pentium II + SSE. *Microprocessor Report*, 13(3):1,6–11, March 1999.

[9] Keith Diefendorff. Sony's Emotionally Charged Chip. *Microprocessor Report*, 13(5):1,6–11, April 1999.

[10] Keith Diefendorff and Pradeep K. Dubey. How Multimedia Workloads Will Change Processor Design. *IEEE Computer*, 30(9):43–45, September 1997.

[11] G. H. Barnes, R. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The Illiac IV Computer. *IEEE Transactions on Computers*, C(17):746–757, August 1968.

[12] Linley Gwennap. AltiVec Vectorizes PowerPC. *Microprocessor Report*, 12(6):1,6–9, May 1998.

[13] Craig Hansen. MicroUnity's MediaProcessor Architecture. *IEEE Micro*, 16(4):34–41, Aug 1996.

[14] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy Code Motion. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 224–234, San Francisco, CA, July 1992.

[15] D.J. Kuck, R.H. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM Symposium on Priciples of Programming Languages*, pages 207–218, Williamsburg, VA, Jan 1981.

[16] Corina G. Lee and Derek J. DeVries. Initial Results on the Performance and Cost of Vector Microprocessors. In *Proceedings of the 30th Annual International Symposium on MicroArchitecutre*, pages 171–182, Research Triangle Park, USA, December 1997.

[17] Corina G. Lee and Mark G. Stoodley. Simple Vector Microprocessors for Multimedia Applications. In *Proceedings of the 31st Annual International Symposium on MicroArchitecutre*, pages 25–36, Dallas, TX, December 1998.

[18] Ruby Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, Aug 1996.

[19] Glenn Luecke and Waqar Haque. Evaluation of Fortran Vector Compilers and Preprocessors. *Software—Practice and Experience*, 21(9), September 1991.

[20] Marc Tremblay and Michael O'Connor and Venkatesh Narayanan and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, Aug 1996.

[21] Motorola. *AltiVec Technology Programming Environments Manual*, November 1998.

[22] Alex Peleg and Uri Weiser. MMX Technology Extension to Intel Architecture. *IEEE Micro*, 16(4):42–50, Aug 1996.

[23] Radu Rugina and Martin Rinard. Pointer Analysis for Multithreaded Programs. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[24] Mark Stephenson, Jonathon Babb, and Saman Amarasinghe. Bitwidth Analysis with Application to Silicon Compilation. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, BC, June 2000.

[25] Thinking Machines Corporation, Cambridge, MA. *Connection Machine CM-2 Technical Summary*, April 1987.

[26] Thinking Machines Corporation, Cambridge, MA. *Connection Machine CM-200 Technical Summary*, June 1991.

[27] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

Appendix A

Multimedia Kernels

```
#define M_SIZE 256

float A[M_SIZE*M_SIZE];
float B[M_SIZE*M_SIZE];
float C[M_SIZE*M_SIZE];


void vectorMultiply(float A[M_SIZE*M_SIZE],
                    float B[M_SIZE],
                    float C[M_SIZE]) {
  int i,j;

  for (i=0; i<M_SIZE; i++) {
    C[i] = 0.0;
    for (j=0; j<M_SIZE; j++) {
      C[i] = C[i] + B[j] * A[i*M_SIZE+j];
    }
  }
}


int main() {
  int i, j;

  for (j=0; j<9; j++) {
    for (i=0; i<M_SIZE; i++) {
      vectorMultiply(A, &B[i*M_SIZE], &C[i*M_SIZE]);
    }
  }
}
```

Figure A-1: VMM: Vector-matrix multiply.

```c
#define M_SIZE 256

float A[M_SIZE*M_SIZE];
float B[M_SIZE*M_SIZE];
float C[M_SIZE*M_SIZE];


/* Performs C = A(B^T) as a 2D matrix multiply */
/* This is done using the straight forward implementation which takes
 * O(n^3) time.  It is not Strausen's alg which runs in O(n^lg7) time.  */

void matrixMultiply(float A[M_SIZE*M_SIZE],
                    float B[M_SIZE*M_SIZE],
                    float C[M_SIZE*M_SIZE]) {
  float* v1;
  float* v2;
  float prod[M_SIZE];
  float sum;

  int i,j,k;
  for (i=0; i<M_SIZE; i++) {
    for (j=0; j<M_SIZE; j++) {
      sum = 0.0;
      for (k=0; k<M_SIZE; k++) {
        prod[k] = A[i*M_SIZE+k] * B[j*M_SIZE+k];
      }
      for (k=0; k<M_SIZE; k++) {
        sum = sum + prod[k];
      }
      C[i*M_SIZE+j] = sum;
    }
  }
}


/*
 * Transposes A in place.
 */
void matrixTranspose(float A[M_SIZE*M_SIZE]) {
  int i,j;
  float t;

  for (i=0; i<M_SIZE; i++) {
    for (j=i; j<M_SIZE; j++) {
      t = A[i*M_SIZE+j];
      A[i*M_SIZE+j] = A[j*M_SIZE+i];
      A[j*M_SIZE+i] = t;
    }
  }
}


int main() {
  int i;

  for (i=0; i<10; i++) {
    matrixTranspose(B);
    matrixMultiply(A,B,C);
  }
}
```

Figure A-2: MMM: Matrix-matrix multiply.

```
/*
 * Code based on FIR/IIR documentation found at:
 * http://www-svr.eng.cam.ac.uk/~ajr/SpeechAnalysis/node13.html
 */

#define FILTER_LENGTH 256
#define SIGNAL_LENGTH 1024

float input_data[SIGNAL_LENGTH];
float output_data[SIGNAL_LENGTH];
float filter1[FILTER_LENGTH];


/*
 * Applys an FIR filter to input to produce output.  filter is
 * an array of coefficients to calculate a weighted sum of input
 * to get output.  Note that for efficiency of calculation
 * the order of the terms in filter is the reverse of what
 * one might expect.
 *
 */

void applyFIR(float input[SIGNAL_LENGTH],
              float filter[FILTER_LENGTH],
              float output[SIGNAL_LENGTH]) {
  int i,j;

  for (i=0; i<FILTER_LENGTH; i++) {
    output[i] = 0.0;
    for (j=FILTER_LENGTH-i-1; j<=FILTER_LENGTH-1; j++) {
      output[i] = output[i] + input[1+i-FILTER_LENGTH+j]*filter[j];
    }
  }
  for (i=FILTER_LENGTH; i<SIGNAL_LENGTH; i++) {
    output[i] = 0.0;
    for (j=0; j<FILTER_LENGTH; j++) {
      output[i] = output[i] + input[1+i-FILTER_LENGTH+j]*filter[j];
    }
  }
}


int main() {
  int i;

  for (i=0; i<300; i++) {
    applyFIR(input_data, filter1, output_data);
  }
}
```

Figure A-3: FIR: Finite impulse response filter.

```
/*
 * Code based on FIR/IIR documentation found at:
 * http://www-svr.eng.cam.ac.uk/~ajr/SpeechAnalysis/node13.html
 */

#define FILTER_LENGTH 256
#define SIGNAL_LENGTH 1024

float input_data[SIGNAL_LENGTH];
float output_data[SIGNAL_LENGTH];
float filter1[FILTER_LENGTH];
float filter2[FILTER_LENGTH];


/*
 * Note that the filters are backwards.
 */

void applyIIR(float input[SIGNAL_LENGTH],
              float inFilter[FILTER_LENGTH],
              float outFilter[FILTER_LENGTH],
              float output[FILTER_LENGTH]) {
  int i,j;

  for (i=0; i<FILTER_LENGTH; i++) {
    output[i] = 0.0;
    for (j=0; j<=i; j++) {
      output[i] = output[i] +
        input[j-FILTER_LENGTH+1+i]*inFilter[j] +
        output[j-FILTER_LENGTH+1+i]*outFilter[j];
    }
  }
  for (i=FILTER_LENGTH; i<SIGNAL_LENGTH; i++) {
    output[i]=0.0;
    for (j=0; j<FILTER_LENGTH; j++) {
      output[i] = output[i] +
        input[j-FILTER_LENGTH+1+i]*inFilter[j] +
        output[j-FILTER_LENGTH+1+i]*outFilter[j];
    }
  }
}


int main() {
  int i;
  filter2[0] = 0.0;  /* Must be zero */

  for (i=0; i<300; i++) {
    applyIIR(input_data, filter1, filter2, output_data);
  }
}
```

Figure A-4: IIR: Infinite impulse response filter.

```
/*
 * YUV equations taken from
 * http://www.cse.msu.edu/~cbowen/docs/yuvtorgb.html
 */

#define VECTOR_SIZE 4096

short int R[VECTOR_SIZE];
short int G[VECTOR_SIZE];
short int B[VECTOR_SIZE];
short int Y[VECTOR_SIZE];
short int U[VECTOR_SIZE];
short int V[VECTOR_SIZE];


void convertRGBtoYUV() {
  int i;
  for (i=0; i<VECTOR_SIZE; i++) {
    Y[i] = (R[i]*77 + G[i]*150 + B[i]*29);
    U[i] = (R[i]*-43 + G[i]*-85 + B[i]*128 + 32767);
    V[i] = (R[i]*128 + G[i]*-107 + B[i]*-21 + 32767);
    Y[i] = Y[i] + 256;
    U[i] = U[i] + 256;
    V[i] = V[i] + 256;
    Y[i] = Y[i] >> 8;
    U[i] = U[i] >> 8;
    V[i] = V[i] >> 8;
  }
}


void convertYUVtoRGB() {
  int i;
  for (i=0; i<VECTOR_SIZE; i++) {
    Y[i] = Y[i] << 8;
    R[i] = (Y[i]+(360*(V[i]-128)));
    G[i] = (Y[i]-(88*(U[i]-128) - (184*(V[i]-128))));
    B[i] = (Y[i]+(455*(U[i]-128)));
    R[i] = R[i] + 256;
    G[i] = G[i] + 256;
    B[i] = B[i] + 256;
    R[i] = R[i] >> 8;
    G[i] = G[i] >> 8;
    B[i] = B[i] >> 8;
  }
}


int main() {
  int i;

  for (i=0; i<1000; i++) {
    convertRGBtoYUV();
    convertYUVtoRGB();
  }
}
```

Figure A-5: YUV: RGB to YUV conversion.