Exploiting Superword Level Parallelism with Multimedia Instruction Sets

Samuel Larsen and Saman Amarasinghe
MIT Laboratory for Computer Science
Cambridge, MA 02139
{slarsen,saman}@lcs.mit.edu

Abstract

Increasing focus on multimedia applications has prompted the addition of multimedia extensions to most existing general-purpose microprocessors. This added functionality comes primarily in the addition of short SIMD instructions. Unfortunately, access to these instructions is limited to in-line assembly and library calls. Some researchers have proposed using vector compilers as a 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 paper we introduce the concept of *Superword Level Parallelism(SLP)*, a novel way of viewing parallelism in multimedia applications. We believe SLP is fundamentally different from the loop-level parallelism exploited by traditional vector processing, and therefore warrants a different method for extracting it. We have developed a simple and robust compiler technique 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. Experiments on scientific and multimedia benchmarks have yielded average performance improvements of 84%, and range as high as 253%.

1 Introduction

The recent shift towards computation intensive multimedia workloads has resulted in a flourish of new multimedia extensions to current microprocessors [12, 15, 20, 23, 25, 30]. Many new designs are targeted specifically toward the multimedia domain [5, 13, 16]. This trend is likely to continue as it has been projected that multimedia processing will soon become the main focus of microprocessor design [14].

While different processors vary in the type and number of multimedia instructions offered, at the core of each is a set of short SIMD 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, SIMD instructions can operate on four 32-bit operands. By adding floating point support as well, it is now possible to use these enhancements to perform more general purpose computation.

MIT/LCS Technical Memo,
LCS-TM-601,
November 18, 1999.

It is not surprising that SIMD execution units have appeared in desktop microprocessors. Their simple control, replicated functional units, and absence of multi-ported register files makes 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.

Use of multimedia extensions has been limited since application writers are largely restricted to using in-line assembly routines or specialized library calls. One solution to this inconvenience is to employ vectorization techniques that have been used to parallelize scientific code for vector machines [11, 18, 19]. Since many multimedia applications are vectorizable, this approach promises good results. However, many important multimedia applications are not vectorizable. Furthermore, complicated loop transformation techniques such as loop fission and scalar expansion are required to parallelize loops that are only partially vectorizable [2, 3, 7, 21]. As a result, no commercial compilers implement this functionality. This paper presents a simple and robust 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 supercomputers. We denote this parallelism *Superword Level Parallelism* since parallelism comes in the form of superwords containing packed data. Note that SLP also differs from traditional large scale SIMD parallelism [6, 8, 28]. SIMD supercomputers require large amounts of parallelism in order to achieve speedups, whereas SLP can be profitable when such parallelism is scarce.

In some sense, superword level parallelism is actually a restricted type of ILP. ILP techniques have been very successful in the general purpose computing arena, partly because of their ability to find small amounts of parallelism within basic blocks. Similarly, the amount of parallelism needed to warrant the use of a multimedia operation is much less than the massive amounts of parallelism needed by vector machines. From this perspective, we have been able to develop a general algorithm for detecting SLP.

In the same way that loop unrolling translates loop-level parallelism into ILP, vectorizable loops can be transformed into SLP. Consequently, our method is able to exploit vector parallelism as well. There are many cases in which a vectorizer is unable to take advantage of loop-level parallelism [10]. Vector compilers typically use complicated loop transformation techniques to handle loops that are only partially vectorizable. Our method, however, is not dependent on any of these transformations. As we will demonstrate in the next section, this allows us to easily parallelize loops that present a challenge to vector compilers.

The rest of this paper is organized as follows: Section 2 defines superword level parallelism. Section 3 describes the compiler algorithm that we have implemented to extract superword level parallelism. Section 4 presents results on a variety of multimedia and scientific benchmarks. Section 5 concludes and discusses future directions.

2 Superword Level Parallelism

This section elaborates on the notion of SLP and the means by which it is detected. Throughout this section, we introduce terminology that facilitates the discussion of our algorithm in the next section.

2.1 Detection of Superword Level Parallelism

Superword level parallelism is defined as short SIMD parallelism in which the operands and results of SIMD operations are packed in a storage location. Detection is done through a short, simple analysis in which independent isomorphic statements are located within a basic block. Isomorphic statements are those that contain the same operations in the same order. Such statements can be computed in parallel by a technique we call *statement packing*, an example of which is shown in Figure 1. Here, operands in corresponding positions have been packed into registers and the addition and multiplication operators have been replaced by their SIMD equivalents (shown as $+_{SIMD}$ and $*_{SIMD}$). The results of the computation are themselves packed and may require unpacking depending on how they 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.

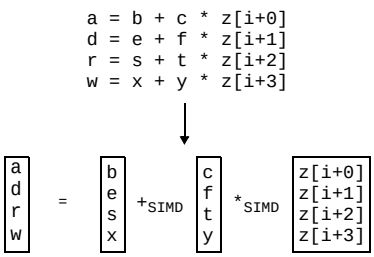

Figure 1: Isomorphic statements that can be packed and operated upon in parallel.

Depending on what operations an architecture provides to facilitate general packing 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 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 [1]. 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 Section 3.

[1] This requires addresses to be aligned on superword boundaries or architectural support for unaligned memory accesses.

2.2 Comparison with Other Forms of Parallelism

2.2.1 Vector Parallelism

The compiler community has generally assumed that exploiting multimedia instructions requires vector parallelization. To provide evidence to the contrary we present two examples, shown in Figures 2 and 3. Although the first example can be made vectorizable after a series of transformations, we know of no vector compilers 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. In comparison, our method is simple and robust, yet still capable of detecting the available parallelism.

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

(a) Original loop.

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

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

(b) After scalar expansion and loop fission.

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

(c) Superword level parallelism exposed after unrolling.

Figure 2: A comparison between parrallelization techniques.

Figure 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 in order 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 [7]. Loop fission[2] is then used to divide the statements into separate loops [17]. The result of these transformations is shown in Figure 2(b). The first loop is vectorizable, but the second must be executed sequentially.

Figure 2(c) shows the loop from the perspective of SLP. The loop has been unrolled and isomorphic statements have been grouped together. This code rearrangement is legal since it does not violate any dependences (once scalar renaming is performed). Note that SLP analysis is not reliant on any explicit instruction movement, and is shown here only to emphasize the SIMD parallel statements. It is worth mentioning however, that SLP transformations implicitly have an instruction reordering effect when statements are grouped together. As we will explain later, our algorithm only packs statements that can be scheduled to execute in parallel. In the example, the first four statements in the loop body

[2] Also called loop distribution or loop splitting.

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.

```
while (1) {
  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;

  if (dst == end) break;
}
```

Figure 3: An example of a hand-optimized matrix operation that proves unvectorizable.

Figure 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 body. 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.

2.2.2 Loop Level Parallelism

Vector parallelism is a subset of loop level parallelism. Methods for detecting this type of parallelism operate at the level of loop nests. In many cases, complex control flow and loop-carried dependences inhibit vectorization. However, as detailed in the previous subsection, SLP methods can extract parallelism by analyzing an unrolled loop body.

General loop-level parallelism is exploited by multiprocessors. In many cases, parallel loops may not yield any 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 therefore conceivable that loop level parallelism could be exploited orthogonally to superword level parallelism within each processor. Since large amounts of coarse-grain parallelism are required to get good MIMD performance, extracting a small amount of SLP would not detract from existing MIMD parallel performance.

2.2.3 SIMD Parallelism

SIMD parallelism came into prominence with the advent of massively parallel supercomputers such as the Thinking Machines CM-1 and CM-2 [28, 29] and Maspar MP-1 [6, 8]. The association of the term "SIMD" with these types of computers is what led us to utilize the term Superword Level Parallelism when discussing short SIMD parallelism.

These supercomputers were implemented using thousands of small processors which worked synchronously on a single instruction stream. While the cost of massive SIMD parallel execution and near-neighbor communication was cheap, 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 these compilers were successful was even more restrictive than that of vector and MIMD machines.

2.2.4 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 or VLIW 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 more inexpensive and energy efficient solution.

In practice, the majority of ILP is found in the presence of loop-level parallelism. As a result, unrolling the loop multiple times should provide enough parallelism to satisfy both ILP and SLP processor utilization. Therefore, ILP performance should not noticeably degrade after SLP is extracted from a program.

2.3 Key to General Acceptance of SLP

Many of the techniques developed by compiler researchers are not generally accepted in mainstream computing. A good example is the work on loop-level parallelization and vectorization that has continued for more than three decades. However, in a very short period of time ILP compilers have become universal. We believe the following characteristics are critical to the general acceptance of a compiler optimization:

- **Robustness:** If simple source code modifications drastically alter program performance, success becomes dependent on the user's understanding of compiler intricacies. Techniques to uncover loop-level parallelism, for example, are prone to wide fluctuations in performance. A change in one statement of the loop body may result in a vector compiler's sequentialization of the entire loop. In the case of ILP and SLP, failure to parallelize a few statements will not significantly impact aggregate performance. This makes methods for their extraction much more robust.

- **Scalability:** Compiler techniques must be able to handle large programs if they are to gain acceptance in real applications. Some analyses required by loop optimizations do not scale well to large code sizes because of dependence on global program analysis. Since global analysis is not required in ILP and SLP, their complexity grows linearly with program size. This results in smooth scaling to larger applications.

- **Simplicity:** Complex compiler transformations are more prone to bugs than simple analyses. Problems are likely to appear only under very specific conditions, making them difficult to detect. Many time critical projects are compiled without optimizations in order to avoid possible compiler errors. Coarse-grain parallelization and vectorization require involved analyses that are more likely to exhibit this behavior [4]. However, most ILP techniques as well as the SLP techniques presented in Section 3 are extremely simple to understand, implement, and validate. In addition, it is often the case that simplicity leads to faster compilation.

- **Portability:** Optimizations that are dependent on particular features of a source language or programming style will not become universal. Techniques for extracting loop-level

parallelism are limited because they only apply to programs written with loops and arrays. On the other hand, ILP and SLP techniques are applied at the level of basic blocks, making them less dependent on source code characteristics.

- **Effectiveness:** No compiler technique will be used if it does not substantially improve program performance. In Section 4, we will show that our algorithm for detecting SLP is effective in providing remarkable speedups.

We believe SLP compiler techniques have the potential of becoming universally accepted as viable and effective means of extracting SIMD parallelism. As a result, we expect future architectures to place increasing importance on SLP operations.

2.4 Architectural Support for SLP

The compiler algorithm presented in Section 3 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 below:

- Many multimedia instructions are designed for a specific high-level operation. For example, HP's MAX-2 extensions offer matrix transform instructions [20] and SUN's VIS extensions include instructions to compute pixel distances [23]. The complex CISC-like semantics of these instructions make automatic code generation difficult.

- Multimedia hardware has typically been viewed as a coprocessor and has not been designed for general purpose computation [25]. Floating point capabilities, for example, have only recently been added to some architectures. Because only a small portion of the processor die area has been dedicated to SLP hardware, fully optimizing a section of code for SLP will leave most of the processor unused, resulting in poor resource utilization.

- Most current multimedia instruction sets were designed with the assumption that data are pre-packed in memory. As a result, data packing and unpacking operations are not well supported. In fact, none of the architectures we have investigated are able to pack four operands into a register in less than three cycles. Data rearrangement is an important operation in our system and with better support SLP performance can be further increased.

Although current architectures were not designed specifically to address superword level parallelism, we believe more support will materialize once the compiler community shows an ability to exploit it. With the introduction of even wider datapaths, SLP will become an important component in the future performance of general purpose computing.

3 SLP Compiler Implementation

Our analysis was implemented with the SUIF compiler infrastructure [32]. The algorithm can be neatly divided into several distinct phases. The following subsections describe these phases in detail. In many situations, simple schemes have been favored over those which might provide better performance. This allows us to develop an extremely simple "proof of concept" that still delivers excellent speedups. Figure 5 presents a running example of the various phases of our algorithm, and Figure 6 lists the pseudo code. Both will be referenced throughout this section.

3.1 Pre-optimization

Several standard transformations are applied to an input program before SLP analysis. The first of these flattens expressions into a slightly modified version of three address form. Since the original structure of arrays is needed in order to identify adjacent references, arrays are not disassembled into address calculations.

Next we apply a series of standard optimizations including constant propagation, copy propagation, dead code elimination, common sub-expression elimination, loop-invariant code motion, and redundant load/store elimination. As discussed earlier, loop unrolling is performed to expose parallelism in vectorizable loops. As a final step, scalar renaming is performed to remove output and anti-dependences since they can inhibit parallelization.

3.2 Loop Unrolling

Loop unrolling is effective in aiding traditional methods for uncovering ILP by exposing more parallelism to the compiler or underlying hardware. It is, in essence, a technique that transforms loop-level parallelism into instruction level parallelism. In the context of superword level parallelism, unrolling translates vectorizable loops into basic blocks with SIMD-parallel statements.

```
for (i=0; i<8; i++) {
  a[i] = b[i] + c[i];
}
```

(a) Before loop unrolling.

```
for (i=0; i<8; 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];
}
```

(b) After loop unrolling.

Figure 4: A simple vectorizable loop before and after loop unrolling.

Figure 4(a) shows a simple vectorizable loop. The unrolled version shown in Figure 4(b) has four isomorphic statements with adjacent memory references. These statements can be packed and operated upon in parallel. Loop unrolling greatly simplifies our analysis since one simple method which identifies parallelism within a basic block is still able to reap the benefits of traditional vectorization.

3.3 Identifying Adjacent Memory References

In general, statements that contain adjacent memory references are automatic candidates for packing. Exceptions occur when true dependences prevent parallelization or high packing costs make it unprofitable. Experiments with realistic packing costs have suggested that statements with adjacent memory references provide the most potential for speedup. As such, our analysis begins by locating these statements, combining them, and adding them to the *packlist*.

Definition 3.1 *Pack* = $\langle a_1, ..., a_n \rangle$ *where* $a_1, ..., a_n$ *are independent isomorphic statements in a basic block.*

Definition 3.2 *PackList* = $\{p_1, ..., p_n\}$ *where* $p_1, ..., p_n \in$ *Pack.*

In this phase of the algorithm, only groups of two statements are constructed. In a later phase, our algorithm merges these groups into larger conglomerates. We will refer to groups of size two as pairs with a left and right statement.

Definition 3.3 *Pair* = $\langle a_{left}, a_{right} \rangle$ *where* a_{left}, a_{right} *are independent isomorphic statements in a basic block.*

Any statement in the original program can occupy at most one left and one right position in the *packlist*. Enforcing this discipline enables later phases to combine pairs such that each statement belongs to a maximum of one packed group. This ensures every statement will be executed only once.

In Figure 5(a), an example sequence of statements is presented. Figure 5(b) shows the results of adjacent memory identification in which two pairs have been added to the *packlist*. Discussion of the *wishlist*, which is also shown in Figure 5(b), is deferred until later.

The pseudo-code for this phase is shown in Figure 6 as FIND_ADJ_REFS.

3.4 Extending the Packlist

Once the *packlist* has been seeded with an initial set of packed statements, more groups can be added. This is done by finding new candidates which can profitably use the current set of packed result data. Def-use chains are followed for each group in the *packlist*. If they lead to fresh packable statements, a new group is created and added to the *packlist*. This process is defined in Figure 6 as EXTEND_PACKLIST, and is shown in the example in parts (c) and (d). In part (c), the def-use chains of the two packed groups are followed to discover new packable statements. For example, statements (2) and (5) can be packed profitably because the result produced by the first group can be used directly as an operand.

The example does not illustrate the fact that several different statement combinations may be possible. This occurs when a result is used in two or more isomorphic statements. Under these circumstances, the most profitable possibility is chosen first. After that, any other groups are assembled in order of their potential savings. This entire process is repeated until nothing new can be added to the *packlist*.

Whenever a pair of packed statements is inserted into the *packlist*, its operands may be produced in the proper packed configuration by other members of the *packlist*. If this is the case, then the savings in packing cost is taken into account. Otherwise, we record the fact that packed operands *could* be used effectively were they to be produced. The *wishlist*, which we have thus far neglected to explain, is the data structure responsible for capturing this information. It is useful in cases where several groups of packed statements use the same packed operands.

Definition 3.4 *Wish* = $\langle \langle a_1, a_2 \rangle, n \rangle$ *where* a_1, a_2 *are independent isomorphic statements in a basic block, and* n *is the number of groups able to use* $\langle a_1, a_2 \rangle$ *as an operand.*

Definition 3.5 *WishList* = $\{w_1, ..., w_n\}$ *where* $w_1, ..., w_n \in$ *Wish.*

In Figure 5, we have shown *wishlist* entries as packed operands with a reference count. Entries are added to the *wishlist* if they are not already produced by a group in the *packlist*. When a *wishlist* entry is referenced a sufficient number of times (twice in our example), it can be moved to the *packlist*. This process has been folded into the transition from parts (d) to (e) in Figure 5. *packlist* and *wishlist* insertion are handled by the functions ADD_TO_PACKLIST and ADD_TO_WISHLIST.

3.5 Combination

Once all profitable pairs have been chosen, they can be combined into larger groups. Two pairs can be combined when the left statement of one is the same as the right statement of the other. In general, any two groups of arbitrary size can be combined when they share the same statement on an edge. In fact, groups *must* be combined in this fashion in order to prevent a statement from appearing in more than one group. This process, provided by the COMBINE_PACKS function, checks all groups against one another, and repeats until all possible combinations have been made. Figure 5(e) shows the result of our example after combination.

It is possible for two groups to contain identical statements in positions other than their edges. In this case, it may be difficult or even impossible to combine them in any sensible fashion. When this scenario arises, the less profitable of the two groups is simply discarded. Since this occurs rarely in practice, dropping one of the groups has little impact on performance.

3.6 Datapath Matching

After completion of the combination phase, we have constructed a list of SLP statements that could be packed profitably. Unfortunately, the number of bits needed to represent any particular group may exceed the superword size of the underlying hardware. As a result, large groups must be broken into a size consistent with the capabilities of the architecture. The simple approach we take is to iteratively separate the first *n* statements whose cumulative width is less than or equal to the datapath of the machine. This process is repeated until all groups are of an acceptable size. Any resulting groups of size one are eliminated. The function MATCH_DATAPATH provides this functionality. In Figure 5(f), we have assumed that the (unspecified) types of each statement allow packing of at most two statements.

The reader might notice that this procedure could break groups at sub-optimal points. It is possible that packed data used efficiently as operands before datapath matching are separated in such a way as to make them useless. We have not seen any of this pathological behavior in our experiments thus far, but plan to develop better separation techniques that would overcome this deficiency should the need arise.

3.7 Cycle Checking

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 7. 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. In general, cycles of arbitrary size can exist.

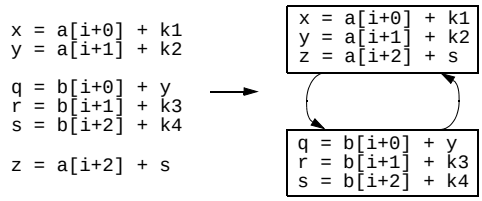

Figure 7: Example of a dependence between groups of packed statements.

To ensure correctness, our algorithm eliminates any groups belonging to a cycle. To detect these cycles, we use Tarjan's algorithm for strongly connected components [27]. This algorithm is well documented elsewhere, so its reproduction is omitted from our pseudo-code.

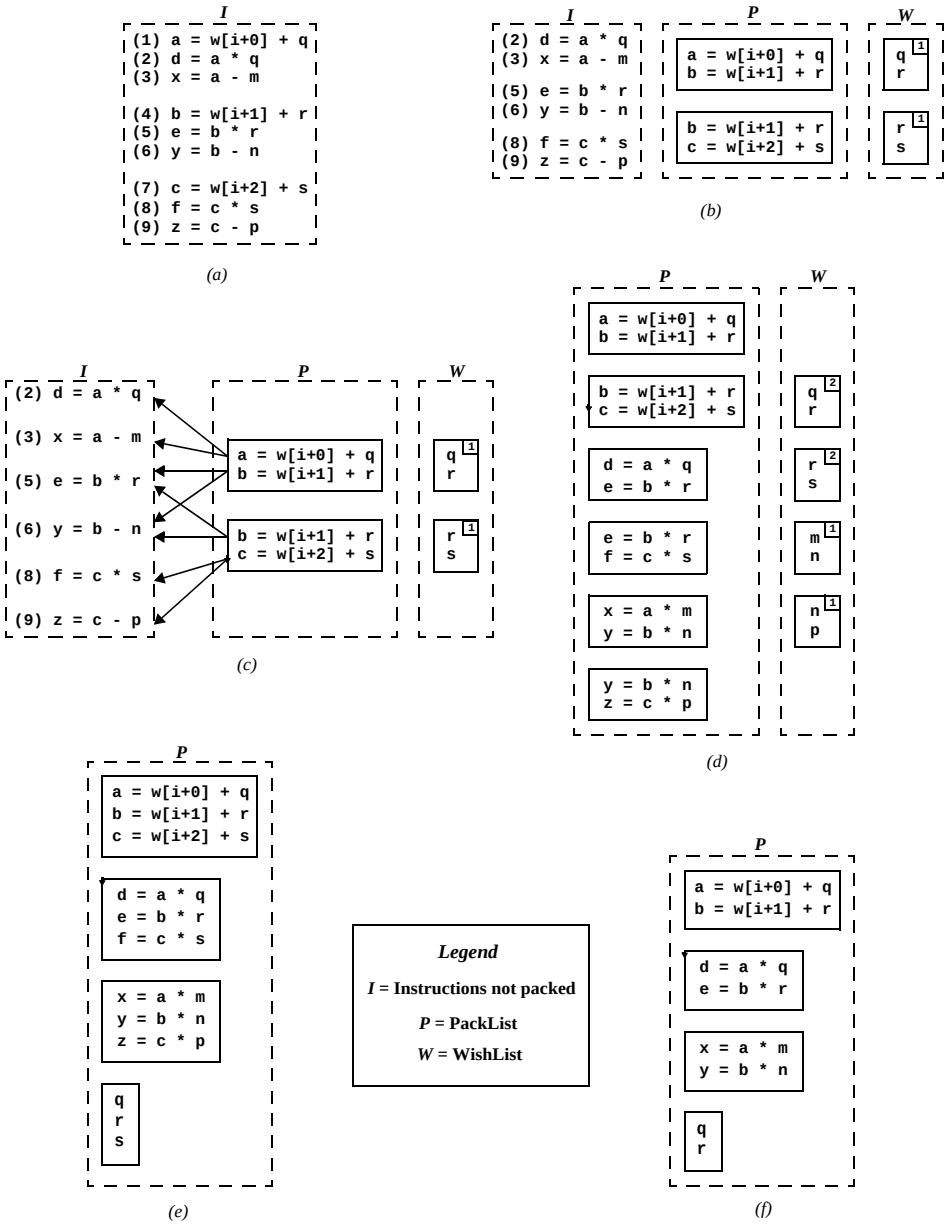

Figure 5: Various phases of our algorithm. (a) Initial sequence of instructions. (b) After statements with adjacent memory references have been packed. (c) Searching of def-use chains. (d) Packed statements resulting from *packlist* extension. (e) After combination. (f) After datapath matching to an architecture with two-way parallelism.

```
global BasicBlock K
global WishList W
global PackList P
global PackList D


SIMD_parallelize:
    FIND_ADJ_REFS()
    EXTEND_PACKLIST()
    COMBINE_PACKS()
    MATCH_DATAPATH()
    CALC_FINAL_PROFIT()


find_adj_refs:
    foreach Stmt a ∈ K do
      foreach Stmt b ∈ K where a ≠ b do
        if STMTS_CAN_PACK(a, b) then
          if HAVE_ADJ_REFS(a, b) then
            ADD_TO_PACKLIST(a, b)


stmts_can_pack: Stmt a × Stmt b → boolean
    if ⟨a, c⟩ ∉ P where c ∈ K then
      if ⟨d, b⟩ ∉ P where d ∈ K then
        if ISOMORPHIC(a, b) then
          if not DEPENDENT(a, b) then
            return true
    return false


add_to_packlist: Stmt a × Stmt b where
            a = [ ... = f(x_1, ..., x_m) ] ,
            b = [ ... = f(y_1, ..., y_m) ]
    if ⟨⟨a, b⟩, n⟩ ∈ W where n ∈ Int then
      W ← W − ⟨⟨a, b⟩, n⟩
    P ← P ∪ ⟨a, b⟩
    for j ← 1 to m do
      Stmt c ← [ x_j = g(...) ]
      Stmt d ← [ y_j = h(...) ]
      if c ≠ φ ∧ d ≠ φ then
        if STMTS_CAN_PACK(c, d) then
          ADD_TO_WISHLIST(c, d)


add_to_wishlist: Stmt a × Stmt b
    if ⟨⟨a, b⟩, n⟩ ∈ W where n ∈ Int then
      W ← W − ⟨⟨a, b⟩, n⟩ ∪ ⟨⟨a, b⟩, n + 1⟩
      n ← n + 1
    else
      W ← W ∪ ⟨⟨a, b⟩, 1⟩
      n ← 1
    if CALC_PROFIT(⟨a, ..., b⟩, n, P) ≥ 0 then
      W ← W − ⟨⟨a, b⟩, n⟩
      ADD_TO_PACKLIST(⟨a, b⟩)


calc_final_profit: → Int
    Int profit = 0
    foreach Pack A ∈ D do
      profit ← profit+ CALC_PROFIT(A, 0, D)
    return profit


extend_packlist:
    repeat
      PackList P_prev ← P
      foreach Pack ⟨a, b⟩ ∈ P where
          a = [ x_0 = f(...) ] ,
          b = [ y_0 = f(...) ] do
        repeat
          profit ← 0
          foreach Stmt c ← [ ... = g(..., x_0, ...) ] ∈ K do
            foreach Stmt d ← [ ... = h(..., y_0, ...) ] ∈ K do
              if STMTS_CAN_PACK(c, d) then
                if CALC_PROFIT(⟨c, d⟩, 0, P) > profit then
                  profit ← CALC_PROFIT(⟨c, d⟩, 0, P)
                  l_best ← c
                  r_best ← d
          if profit ≥ 0 then
            ADD_TO_PACKLIST(l_best , r_best)
        until profit ≡ 0
    until P ≡ P_prev


combine_packs:
    repeat
      PackList P_prev ← P
      foreach Pack A ← ⟨a_1, ..., a_n⟩ ∈ P do
        foreach Pack B ← ⟨b_1, ..., b_m⟩ ∈ P do
          if A ≠ B then
            if a_1 ≡ b_m ∧ a_i ≠ b_j where 1 < i ≤ n, 1 ≤ j < m then
              P ← P − A − B ∪ ⟨b_1, ..., b_m, a_2, ..., a_n⟩
            else if a_i ≡ b_j where 1 ≤ i ≤ n, 1 ≤ j ≤ m then
              if CALC_PROFIT(A, 0, P) > CALC_PROFIT(B, 0, P) then
                P ← P − B
              else
                P ← P − A
    until P ≡ P_prev


match_datapath:
    foreach Pack ⟨a_1, ..., a_n⟩ ∈ P do
      sum ← 0
      i ← 1
      for j ← 1 to n do
        if sum+ DATA_WIDTH(a_j ) > max_dw then
          if i < j − 1 then
            D ← D ∪ ⟨a_i, ..., a_{j−1}⟩
          sum ← 0
          i ← j
        sum ← sum+ DATA_WIDTH(a_j )
      if i < j then
        D ← D ∪ ⟨a_i, ..., a_n⟩


calc_profit: Pack ⟨a_1, ..., a_n⟩ × Int w × PackList L → Int where
            a_1 = [ ... = f(x_1^1, ..., x_m^1) ] , ...,
            a_n = [ ... = f(x_1^n, ..., x_m^n) ]
    Int profit = (NUM_OPS(a_1) ∗ (n − 1)) + (w∗ PACK_COST(n))
    for j ← 1 to m
      if ARE_ADJ_REFS(x_j^1, ..., x_j^n) then
        profit ← profit+ ADJ_PROFIT(x_j^1, ..., x_j^n)
      else if ⟨x_j^1, ..., x_j^n⟩ ∉ L then
        profit ← profit− PACK_COST(n)
    return profit
```

Figure 6: Pseudo code for the SLP extraction algorithm.

3.8 Savings Calculation

With the completion of cycle checking, a set of valid packable groups has been chosen. This phase completes by calculating the estimated speedup based on the packed statements and their packing/unpacking costs. This step is used solely as a method to gauge our success on a given benchmark, and has no effect on the results themselves. CALC_FINAL_PROFIT is responsible for computing the speedup from statement packing.

3.9 Discussion

Below we discuss some of the choices made in developing our algorithm. These ideas were omitted earlier in order to focus on the details of our method.

- One approach to locating isomorphic statements is to compare them based on the structure inherited from the source code, where statements can be arbitrarily long. This method is attractive because implicit temporaries are always in the proper packed configuration and never require packing or unpacking. A major drawback, however, is that statements must match exactly at the source code level in order to be considered for packing. This behavior is unacceptable since statements that are close in structure may still benefit from parallelization. Using three address form overcomes this disadvantage. In addition, it is more amenable to traditional compiler optimizations, allowing us to perform SLP analysis on an optimized representation.

- Once source code has been flattened into a low-level representation, a large basic block could contain an inordinate number of isomorphic statements. Searching the space for the best arrangement becomes difficult. This gives rise to our motivation for seeding the analysis with statements containing adjacent memory references. After performing redundant load/store elimination, statements have few packing options and our task is greatly simplified.

- When cycles are detected among packed statements, our solution is to discard all groups involved in the cycle. A better solution would be to discard only enough groups to break the cycle. Since our experiments show this situation to be rare, we decided not to focus our attention on developing an optimal cycle breaking algorithm. If our current method shows a performance degradation in later tests, we will improve the cycle checking phase.

- A final point concerns the scope over which our analysis operates. As discussed, our algorithm matches statements within a basic block. One of the attractive aspects of superword level parallelism is that it can be uncovered on a local scale. Nonetheless, one can imagine more complicated approaches that search for isomorphic statements over a greater range of the program. In fact, SLP could be used as a form of predication in which data could simply be discarded if they were to become invalid due to control flow. We use loop unrolling as a method of increasing basic block size, but for the most part we have chosen the simplest approach.

4 Results

4.1 Experimental Methodology

Of the popular multimedia instruction sets available in commercial microprocessors, we believe the AltiVec instruction set best matches the compilation technique described in this paper [30]. AltiVec defines 128-bit floating point and fixed point 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. In addition, AltiVec offers a powerful vector permute instruction in which packed elements of two input sources can be arbitrarily permuted into a destination register. The use of this instruction in conjunction with a few dedicated registers to hold common permute masks can provide a relatively inexpensive packing and unpacking mechanism.

At the time of this writing, the G4 processor from Motorola has just become available. As such, we have been unable to test our compiler's performance on a real machine. This is currently work in progress and we expect to be running hardware based performance tests in the near future. For now, we are limited to less accurate approximations.

Our algorithm requires knowledge of packing and unpacking costs in order to determine which groups of packed statements can execute more efficiently than their sequential versions. We are currently categorizing these costs in terms of the number of instructions required to perform a given pack or unpack operation. On a processor supporting AltiVec, packing *n* values into a single register requires *n-1* instructions if the necessary permute masks are stored in dedicated registers. We use this formula to compute costs.

Results are reported as the percentage of dynamic instructions eliminated from the original source program after parallelization. To the extent that execution time is directly proportional to dynamic instruction count, speedup can be computed from this percentage. We have provided both perspectives in our graphs. The performance calculations require two pieces of information, the first being static instruction count. Since we are unable to generate native machine code at this time, counting the instructions in the intermediate format is a good approximation. The second piece of information needed is the number of times each basic block is executed. Our system includes functionality to automatically instrument source code so that this information can be easily determined.

The speedups we report are ignorant of instruction latencies, cache performance, register pressure, and any other nuances encountered when executing on a real machine. In some of these cases, SLP parallelization may provide further benefits that are not reflected in our results. For example, packing data into registers effectively provides more registers. Since register storage is better utilized, SLP parallelization may result in fewer costly memory spills.

It should be emphasized that we are not measuring any effects of traditional superscalar execution such as instruction reordering and branch speculation. For this reason, our speedup numbers essentially compare a single issue, in-order processor against one with a wider SLP datapath. As we discussed earlier, ILP execution units may or may not provide cumulative speedups.

It is important to note that the reduction in dynamic instruction count has direct impact on instruction cache performance and energy consumption. Reduced instruction count is recognized as a major benefit of vector and SIMD computers. SLP processors provide a similar feature.

4.2 Benchmarks

We measure our success on two benchmark suites, the first being the SPEC95fp benchmarks [9]. Most of the applications in this suite are either vectorizable or partially vectorizable. Our results show that our SLP algorithm is indeed able to exploit vector parallelism.

The second benchmark suite for which we present results is the UTDSP benchmark kernels [22]. These kernels represent core operations common to many multimedia applications. The UTDSP suite offers two problem sizes for each kernel. In all cases, we

Benchmarks	Data path size					
	256 bits		128 bits		64 bits	
bench	%saved	speedup	%saved	speedup	%saved	speedup
swim	71.66%	3.529	68.96%	3.222	48.73%	1.950
tomcatv	66.41%	2.977	47.12%	1.891	10.60%	1.119
mgrid	59.22%	2.452	41.70%	1.715	1.91%	1.019
su2cor	44.57%	1.804	34.68%	1.531	10.80%	1.121
hydro2d	25.65%	1.345	19.21%	1.238	5.10%	1.054
apsi	25.44%	1.341	21.52%	1.274	7.06%	1.076
wave5	23.54%	1.308	19.30%	1.237	5.56%	1.059
applu	21.87%	1.280	16.26%	1.194	6.36%	1.068
turb3d	21.46%	1.273	17.14%	1.207	2.75%	1.028
fpppp*	15.53%	1.184	10.24%	1.114	0.00%	1.000
fir	58.08%	2.385	50.88%	2.036	39.82%	1.662
lmsfir	52.99%	2.127	49.58%	1.983	38.90%	1.637
latnrm	49.96%	1.998	49.80%	1.992	37.55%	1.601
fft	40.90%	1.692	40.90%	1.692	32.32%	1.478
iir	33.09%	1.495	33.09%	1.495	29.12%	1.411
mult	20.71%	1.261	19.19%	1.237	14.07%	1.164

Table 1: Instruction savings and speedup for the SPEC95fp benchmark suite and UTDSP kernels. Note that the current heuristics required compiler directives to seed the analysis for fpppp.

have reported numbers for the larger configuration. Since the kernels have been written in the C source language, we have added declarations indicating that all arrays are non-overlapping. This is necessary because array aliasing can inhibit parallelization since dependence analysis must be overly conservative. In the future, we plan to incorporate pointer analysis in order to alleviate this necessity.

4.3 Experimental Results

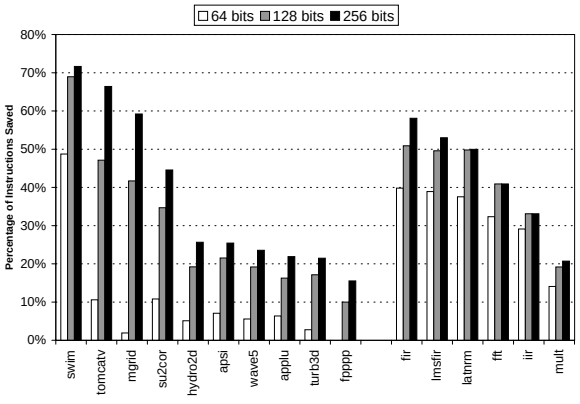

Figure 8: Percentage of instructions eliminated with superword level parallelism.

Table 1 presents our results on a variety of datapath widths. Figures 8 and 9 present these numbers graphically in the form of instructions eliminated and speedup, respectively. For simplicity, we have assumed that all datapaths are able to perform both double and single precision floating point operations. The SPEC95fp benchmarks perform poorly at a data width of 64 bits since all but `swim` require 64-bit data types. With a 128-bit datapath, it is possible to pack two operands and therefore provide a significant performance improvement. Extending the datapath to 256 bits provides even better results when extra parallelism is available. On this datapath, the SPECratio improvement for the entire SPEC95fp suite is 1.71.

`fpppp` is the worst performing benchmark since the dominating loop contains very few array references. As a result, our algorithm is unable to find a good starting point from which to grow the *packlist*. However, careful inspection of the loop body reveals a moderate amount of superword level parallelism. In order to detect the available parallelism, we have added compiler hints in order to provide our analysis with a starting point. We are currently working on methods of seeding besides adjacent memory references so that we are able to automatically detect SLP in purely scalar code.

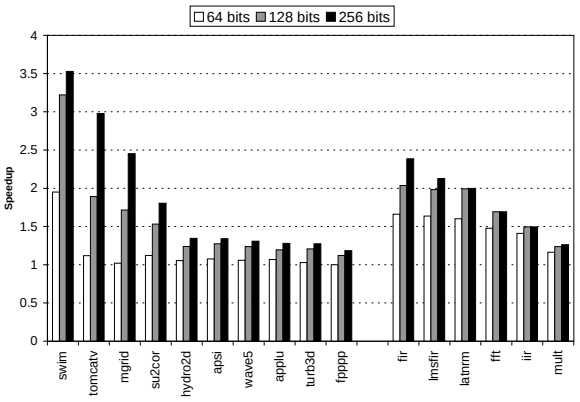

Figure 9: Speedup obtained by superword level parallelism.

The UTDSP benchmarks employ 32-bit data types and therefore perform well on a 64-bit datapath. Several of these benchmarks benefit from pairs of adjacent memory references within the loop body. Unrolling yields no extra parallelism and provides limited improvement when extending the datapath to 128 and 256 bits.

Figure 10 contrasts a standard 256-bit machine with one that can pack and unpack with no cost. The small performance differences indicate that almost no packing and unpacking operations are performed. The savings in instruction count is due largely to operating on adjacent memory locations. Any operation on scalars

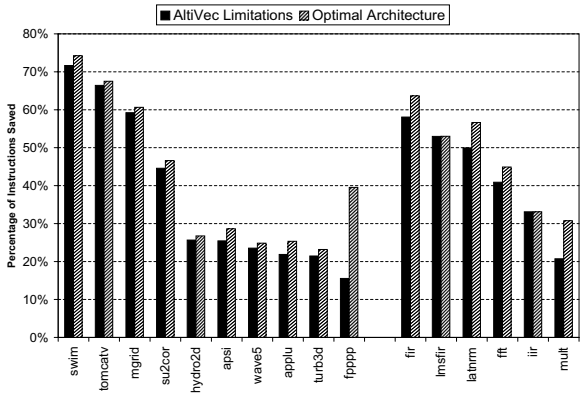

Figure 10: SLP with packing/unpacking costs consistent with AltiVec vs. zero packing/unpacking cost.

requires little packing since most operands are produced in the proper packed configuration by earlier operations. `fpppp` is the only benchmark with ample superword level parallelism in the form of scalars. As seen by the graph, it would greatly benefit from inexpensive packing and unpacking operations.

5 Conclusion

In this paper we introduced superword level parallelism, the idea of viewing parallelism from the perspective of partitioned operations on packed superwords. We compared this view with more traditional vector parallelism and argued that SLP is needed to effectively exploit the capabilities of multimedia instruction sets. This led to a simple and robust compiler implementation that was able to exhibit excellent performance on a number of benchmarks. In the best case we achieved a speedup of 3.53 for a 256-bit architecture executing `swim`. The entire SPEC95fp benchmark suite showed a SPECratio improvement of 1.71.

Our compiler implementation for the algorithm described in Section 3 is still in its infancy. We plan to make many improvements in the near future. For example, extending the scope of our analysis beyond basic blocks provides packing opportunities across branches. SLP can also offer a form of predication, in which speculated computation can simply be discarded if it is invalidated due to control flow.

Currently, we use program information to determine the data size needed for each SLP operation. Recent research shows that compiler analysis can significantly reduce the size of data types needed to store program variables [26]. Incorporating this analysis has the potential of drastically improving SLP performance by increasing the number of operands that can be packed in a register.

An SLP-aware register allocator can operate after packing decisions are made, effectively creating a larger register file and reducing costly memory spills. Similarly, calling conventions that pass packed data as arguments will reduce the number of stack operations. Memory allocation schemes that align data in a manner consistent with the needs of an SLP compiler lead to the possibility of more parallel loads and stores.

We are optimistic that superword level parallelism techniques will boost performance in next generation microprocessors. Current superscalar techniques are providing diminishing returns since they are unable to scale with the amount of available silicon. New architectures are emerging in an attempt to address this issue. Examples include chip multiprocessors [24] and RAW architectures [31]. It will take time for these architectures and their associated compiler technology to mature. SLP execution, on the other hand, has already begun to appear in general purpose microprocessors. With the ability to automatically exploit this parallelism, such processors are well equipped to provide increasing performance.

SLP architectures offer the potential for large speedups and significant power savings. Their simple replicated designs, coupled with the absence of multi-ported register files, means they will scale well with future technologies. It is now the responsibility of the compiler community to demonstrate the benefits of automatically extracting SLP from general purpose programs. This paper takes an important first step in that direction.

6 Acknowledgments

The authors wish to thank Krste Asanović for his input on vector processing, as well as the RAW group members who provided feedback during the writing of this paper.

References

[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] Randy Allen and Steve Johnson. Compiling c for vectorization, parallelization and inline expansion. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.

[4] Saman Amarasinghe. Parallelizing Compiler Techniques Based on Linear Inequalities. In *Ph.D Thesis, Stanford University. Also appears as Techical Report CSL-TR-97-714*, Jan 1997.

[5] 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.

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

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

[8] 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.

[9] Standard Performance Evaluation Corporation. Digital Equipment Corporation AlphaServer 8400 5/350 SPEC CFP95 Results. *spec newsletter*, 8(1):39, April 1996.

[10] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, IL, August 1986.

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

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

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

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

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

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

[17] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, August 1993.

[18] 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.

[19] 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.

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

[21] Glenn Luecke and Waqar Haque. Evaluation of fortran vector compilers and preprocessors. *Software—Practice and Experience*, 21(9), September 1991.

[22] M. A. R. Saghir and P. Chow and C. G. Lee. A Comparison of VLIW and Traditional DSP Architectures for Compiled Code. In *Proceedings of the 31st Annual International Symposium on MicroArchitecutre*, Dallas, TX, December 1998.

[23] 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.

[24] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Cambridge, Massachusetts, October 1–5, 1996.

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

[26] Mark Stephenson, Jonathon Babb, and Saman Amarasinghe. Bitwidth Analysis with Application to Silicon Compilation. Submitted for publication to the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation.

[27] Robert E. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

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

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

[30] Jon Tyler, Jeff Lent, Anh Mather, and Huy Van Nguyen. AltiVec(tm): Bringing Vector Technology to the PowerPC(tm) Processor Family. In *Proceedings of the 1999 IEEE International Performance Computing and Communications Conference*, Phoenix, AZ, February 1999.

[31] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997. Also available as MIT-LCS-TR-709.

[32] 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.